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

V1.5, 17.03.2025

Intro

Das Sperren und das Wiederfreischalten von Interrupts ist ein übliches Szenario in Embedded SW. Der Cortex-M bietet dafür geeignete Mechanismen, doch ganz so trivial ist das Thema wegen potentiell auftretender Race Conditions nicht und daher werfen wir hier einen tieferen Blick in ein auf den Cortex-M optimiertes Verfahren.

Scenario

In einem Code Segment, das kann eine normale Funktion, eine Interrupt Service Routine oder eine Task in einem RTOS Kontext sein, sollen alle Interrupt Behandlungen temporär unterdrückt werden, um sie zu einem späteren Zeitpunkt wieder frei zu schalten.

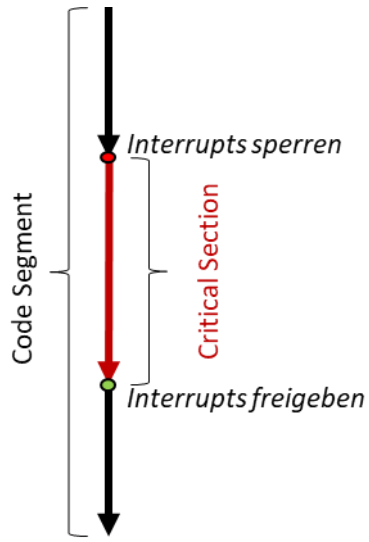

Bild 1: Critical Section

Wie in Bild 1 dargestellt bildet das Code Segment zwischen dem Sperren und dem Wiederfreischalten der Interrupts eine s.g. Critical Section, die nicht durch Interrupts oder Taskwechsel unterbrochen werden kann.

Wenn nun mehrere solche Critical Sections durch Interrupts verschiedener Prioritäten[*)] oder durch ein Multitasking Betriebssystem semi-parallel ablaufen, d.h. zeitlich versetzt ineinander verschachtelt ablaufen, dann muss man dafür sorgen, dass am Ende einer Critical Section die Interrupts nur dann wieder freigeschaltet werden, wenn sie vor der Critical Section nicht ohnehin schon gesperrt waren.

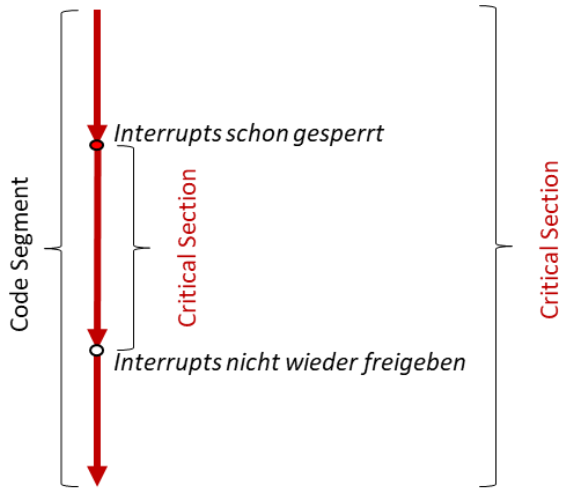

Bild 2: Verschachtelte Critical Section

Bild 2 zeigt ein soches Beispiel, das z.B. daraus entstehen kann, dass innerhalb einer Critical Section eine Funktion aufgerufen wird, die ihrerseits eine Critical Section ausführen möchte. Nested Critical Sections machen Sinn, da diese Funktion auch von ausserhalb einer Critcal Section aufgerufen werden könnte.

Race Conditions

Beim Sperren der Interrupts muß man sich also z.B. in einer lokalen Variablen merken, ob die Interrupts schon gesperrt waren. Am Ende der Critical Section gibt man die Interrupts gemäß des in der lokalen Variablen gespeicherten Zustand vor dem Sperren, wieder frei, oder eben nicht.

Da das Lesen und Merken des Zustands und das Sperren der Interrupts nicht in einem einzigen ununterbrechbaren Befehl möglich sind, können hier Race Conditions auftreten. Dies geschieht, wenn der Programmfluß z.B. durch einen höher prioren Interrupt genau zwischen dem Lesen und dem Sperren unterbrochen wird und die Interrupts innerhalb dieser Unterbrechung in der höher prioren Interrupt Service Routine gespserrt werden. Dann hätte man z.B. gelesen, dass die Interrupts nicht gesperrt sind und gibt sie am Ende der Critical Section wieder frei. In der Unterbrechung sind aber die Interrupts ebenfalls gesperrt worden, man würde daher die gerade beschriebene Freigabe am Ende der Critical Section fälschlicherweise durchführen und damit das Systemverhalten verändern und Fehlerzstände verursachen. So geht es also nicht!

Übliches Mittel, um soche Race Conditions zu vermeiden sind durch Semaphore oder Mutexes realisierte Critical Sections. Der Cortex-M bietet wunderbare exclusive Primitives LDREX, STREX, … mit deren Hilfe Semaphore und Mutexes realisierbar sind, ohne Interrupts sperren zu müssen. Doch genau in unserem Szenario schiessen wir hier mit Kanonen auf Spatzen. Warum?

Lösungsvorschlag

Wir können ein primitives Counting Semaphor nutzen, dass nicht zugriffsgeschützt werden muss, solange wir nur innerhalb der Critical Section darauf zugreifen. D.h. wir können einfach eine simple Variable verwenden, um darin die Anzahl der Sperrungen und Freigaben zu zählen. Eine Zugriffsverletzung ist durch die Critical Section ja ohnehin ausgeschlossen, das ist praktisch.

Wir verwenden also nicht das in der HW realisierte PRIMASK Bit um uns zu merken, ob die Interrupts gesperrt waren, sondern einen einfachen Zähler. **Sperren = Inkrementieren**, **Freigeben = Dekrementieren** und nur dann, wenn der Zähler Null erreicht, wird das PRIMASK Bit in der HW gelöscht und damit die Interrupts wieder freigegeben. In dem Code Schnipsel in Bild 3 sehen wir, dass wir vor dem Inkrementieren natürlich das PRIMASK Bit in der

HW setzen müssen und starten damit die Critical Section, die den Zähler automatisch mit schützt.

Natürlich müssen alle Code Segmente, die Interrupts sperren wollen, den gleichen Zähler verwenden.

```
1     #include "stm32f10x.h"
2
3     static volatile uint32_t diasble_counter = 0;
4
5     void disable_Interrupts( void )
6     {
7           __disable_irq();
8           // START OF CRITICAL SECTION
9           disable_counter++;
10    }
11
12    void resume_Interrupts( void )
13    {
14          //may only be called from within a critical section
15          disable_counter--;
16          if( disable_counter == 0UL )
17          {
18                __enable_irq();
19                // END OF CRITICAL SECTION
20          }
21    }
```

Bild 3: Code Beispiel

Das lässt sich mit einer globalen Variablen erreichen. Die Clean Code Regeln lehren uns globale Variablen, wo immer möglich, zu vermeiden und das Prinzip der Kapselung zu nutzen. Wir packen also die Funktionen zum Sperren und Freigeben der Interrupts in ein eigenes C oder C++ Modul und können den Zähler dann als static Variable in diesem Modul realisieren. Das Schlüsselwort `volatile` stellt sicher, dass der Compiler Code erzeugt, der den Inhalt des Zählers nicht in einem Core Register zwischen-speichert und dadurch immer den aktuellen Wert aus dem Speicher verwendet. In Bild 3 sehen wir eine mögliche Realisierung eines solchen Moduls.

Natürlich sollte man den Code noch gegen fehlerhafte Aufrufe absichern. Wenn es ausreichend ist den fehlerhaften Aufruf der `resume_Interrupts()`Funktion aus einer nicht Critical Section heraus zu Überleben, ohne eine Fehlerbehandlung einzuleiten, dann kann man den kompletten Funktionsbody in eine Bedingung setzen, die nur ausgeführt wird, wenn der Zähler auch wirklich noch größer Null ist.

Das verhindert den Unterlauf des Zählers von Null auf den maximalen uint32_t Wert. Alternativ kann man dem Dekrementieren (Zeile 15 in Bild 3) ein assert voranstellen.

```
assert(__get_PRIMASK() != 0);
```

Die `__get_PRIMASK()` Funktion liefert Null zurück, wenn die Interrupts enabled sind. Der assert Makro führt ein exit aus, falls der Wert nicht der Erwartung entspricht. Neben dem Unterlaufen kann der Zähler auch beim Inkrementieren überlaufen. Wir haben hier einen 32-bit Zähler verwendet, somit zählt ein Überlauf nicht zu den wahrscheinlichen Szenarien, doch schnell wird beim Speichersparen ein 8-bit Zähler draus und dann ist es gut, wenn eine Assertion uns vor Überraschungen schützt. Die folgende zusätzliche assert-Anweisung nach dem Inkrementieren erledigt das.
Danke an Herrn Hofbauer für den Verbesserungsvorschlag.

```
assert(disable_counter != 0);
```

Zum Abschluss hier noch ein paar kleine Hinweise:

- Damit die assert-Zeile fehlerfrei übersetzt werden kann, muss man oben in dem Modul die Zeile `#include <assert.h>` einfügen.
- Der Assert Makro lässt sich durch das globale Define NDEBUG ausblenden. Im Produktiv-Code möchte man ja nicht unbedingt ein exit() ausführen. Dann wird der Macro zur Compilezeit einfach ignoriert und kostet weder Speicher noch Rechenzeit. Das globale Define wird bei den meisten Compilern in den Compilersettings oder Kommandozeile gesetzt.
- *) Der Reset, der Non-Maskable Interrupt und auch der Hardfaulthandler bleiben auch bei ausgeschalteten Interrupts aktiviert, da sie negative Prioritäten und damit die höchsten Prioritäten im System haben. Im Non-Maskable Interrupt und im Hardfaulthandler sollten daher die Interrupts nicht disabled werden, warum auch, sie kommen wegen der niedrigeren Priorität ohnehin nicht durch.
- Das Setzen des PRIMASK-bits z.B. durch `__disable_irq()` unterbindet die Behandlung von allen Interrupts (außer Reset, Non-Maskable Interrupt und Hardfault). Die individuellen Pending-Bits der Interrupts werden aber weiterhin gesetzt, wenn der jeweilige Interrupt anliegt. Diese Pending-Bits bleiben solange gesetzt, bis sie entweder durch das Behandeln des Interrupts durch die HW automatisch gelöscht werden, oder durch Setzen des dazu passenden Clear-Pending-Bits durch SW wieder gelöscht werden. Nach dem Wiederfreigeben von Interrupts werden daher alle während der Critical Section aufgetretene Interrupts gemäß ihrer Priorität abgehandelt. Einen HW-Zähler für die Häufigkeit des Auftretens eines Interrupts während der Critical Section gibt es nicht, es ist eben nur ein einziges Bit pro Interrupt.
- Nach dem Reset ist das PRIMASK Bit gelöscht und die Interrupts freigegeben. Die Individuellen Enable Bits der Interrupts sind nach dem Reset nicht enabled, es können also nur der Hardfault, der NMI und der Reset auftreten.
- Es ist für die Funktion des `disable_counter` essentiell, dass alle Sperrungen und Wiederfreigaben von Interrupts über die neuen Funktionen erfolgen, die den `disable_counter` verwenden. Damit ist sichergestellt, dass vor dem ersten Aufruf der `disable_Interrupts()` Funktion die Interrupts freigegeben sind, d.h. das PRIMASK Bit noch gelöscht ist. Auch hier, *danke an Herrn Hofbauer für den Hinweis.*
- Die Include Anweisung in Bild 3 Zeile 1 ist natürlich für den jeweiligen, von Ihnen verwendeten Controller anzupassen. Das Prinzip funktioniert für ALLE Cortex-M, vom M0 bis zum M85.
- In naher Zukunft werden ich einen vergleichbaren kleine Artikel zu dem hier kurz angerissenen Thema der Exclusive Primitives LDREX, STREX, CLREX und deren CMSIS C Varianten verfassen. Bei Interesse dürfen Sie Sich gerne über info@tnms.de an mich wenden.

Viel Spaß mit den tollen Cortex-Mx-ern 😊.

Hier noch die zwei Codebeispiele für criticalsection.h und criticalsection.c

```
1   /*****************************************************************************//**
2   * @file criticalsection.h
3   * @version V1.1
4   * @date 14th of Feb 2025
5   * @author Remo Markgraf
6   * Copyright (C) 2025 Remo Markgraf info@tnms.de All rights reserved.
7   * @warning
8   * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS,
9   * IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF
10  * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT TO
11  * THIS SOFTWARE. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY CLAIM,
12  * DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
13  * OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
14  * USE OR OTHER DEALINGS IN THE SOFTWARE.
15  *****************************************************************************/
16  #ifndef CRICICALSECTION_H_
17  #define CRICICALSECTION_H_
18
19  #ifdef __cplusplus
20  extern "C" {
21  #endif
22
23  #include "stm32f10x.h"
24
25  //prototypes
26  void disable_Interrupts( void );
27  void resume_Interrupts( void );
28
29
30  #ifdef __cplusplus
31  }
32  #endif
33
34  #endif //CRICICALSECTION_H_
35
```

```c
/*****************************************************************************//**
* @file criticalsection.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 <assert.h>
#include "criticalsection.h"

static volatile uint32_t diable_counter = 0;

void disable_Interrupts( void )
{
	__disable_irq();
	// START OF CRITICAL SECTION
	diable_counter++;
}

void resume_Interrupts( void )
{
	//may only be called from within a critical section
	assert(__get_PRIMASK() != 0);

	diable_counter--;
	if( diable_counter == 0UL )
	{
		__enable_irq();
		// END OF ALL CRITICAL SECTIONS
		assert(__get_PRIMASK() == 0);
	}
}

```

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