

Building Bare-Metal ARM Systems with GNU

Part 7: Interrupt Locking and Unlocking

Miro Samek

In this part of this article series I describe the interrupt locking and unlocking policy for ARM that will be safe for both IRQ and FIQ interrupt handlers as well as the task level code (the code called from `main()`). You would probably never think that interrupt locking could deserve the whole article. But the ARM architecture somehow manages to make it amazingly complex. The recommended reading for this part includes: ARM Technical Support Note “What happens if an interrupt occurs as it is being disabled?” [1], and Atmel Application Note “Disabling Interrupts at Processor Level” [2].

1. Problem Description

In the simple foreground/background architecture without any underlying multitasking OS or kernel (bare metal) the foreground (ISRs) communicates with the background (the `main()` loop) by means of shared variables. The background code has the responsibility of protecting these shared variables from corruption by the asynchronously firing ISRs. The only mutual exclusion mechanism available in this simple architecture is to briefly lock interrupts before accessing the shared resource and to unlock the interrupts after releasing the resource. The section of code executing atomically between locking and unlocking interrupts is often called the **critical section** or critical region. Of course, you should keep the time spend inside each critical section to the minimum, so that you don't extend the interrupt latency of the system.

Critical sections are necessary not just in the task-level code callable from the `main()` loop. If nesting of IRQ interrupts is allowed (which I assume in this article) critical sections are necessary also inside the IRQ interrupts.

Finally, the same critical sections used in the IRQs handlers might be also used inside the FIQ handler, simply because of the coding convenience. Even though the FIQ handler runs with interrupts locked at all times, so it does not really need to use a critical section (see part 6 of this article), experience shows that programmers can all too easily forget that FIQ requires a completely different interrupt locking policy than all other interrupt handlers. A problem would arise if interrupts were inadvertently unlocked inside the FIQ handler upon the exit from a critical section. Please note that a critical section can be buried inside a deeply nested function call chain.

In summary, a real-life, bare-metal ARM project requires a universal interrupt locking and unlocking policy that would be safe to use from the task-level, nested IRQ handlers, and the FIQ handler.

2. The Policy of Saving and Restoring Interrupt Status

The interrupt locking policy that has all the properties required in this case is the policy of saving and restoring the interrupt status. This policy is common, and is used for example inside the VxWorks RTOS (see the function pair `intLock()/intUnlock()` documented at www.slac.stanford.edu/exp/glast/flight/sw/vxdocs/vxworks/ref/intArchLib.html#intLock) The following code snippet shows how you use this type of critical section:

```
void your_function() {
(1)   ARM_INTERRUPT_KEY_TYPE int_lock_key;
      .
(2)   ARM_INTERRUPT_LOCK(int_lock_key); /* enter the critical section */
      /* access the shared resource */
(3)   ARM_INTERRUPT_UNLOCK(int_lock_key); /* leave the critical section */
      .
}
```

```
}
```

First, you need to declare a temporary variable (typically a stack or register variable) that will hold the interrupt status throughout the duration of the critical section (1). For portability, the type of the interrupt status is declared as a macro `ARM_INT_KEY_TYPE`, which along with the macros for locking and unlocking interrupts, is defined in the file `arm_exc.h` included in the code accompanying this article. Next, you lock interrupts by means of another macro `ARM_INT_LOCK()`, which saves the interrupt status in the `int_lock_key` variable (2). Finally, you unlock the interrupts by means of the macro `ARM_INT_UNLOCK()`, which restores the interrupt lock to the state before the matching `ARM_INT_LOCK()` has been called (3).

This policy allows nesting critical sections, because the interrupt state is preserved across the critical section in a temporary variable. In other words, upon the exit from a critical section the interrupts are actually unlocked in the `ARM_INT_UNLOCK()` macro only if they were unlocked before the invocation of the matching `ARM_INT_LOCK()` macro. Conversely, interrupts will remain locked after the `ARM_INT_UNLOCK()` macro if they were locked before the matching `ARM_INT_LOCK()` macro.

3. Critical Section Implementation with GNU gcc

The macros `ARM_INT_KEY_TYPE`, `ARM_INT_LOCK()`, and `ARM_INT_UNLOCK()` are defined in the `arm_exc.h` header file provided in the code accompanying this article and shown in Listing 1

```
(1) #define ARM_INT_KEY_TYPE      int
(2) #ifdef __thumb__
(3)     #define ARM_INT_LOCK(key_) ((key_) = ARM_int_lock_SYS())
(4)     #define ARM_INT_UNLOCK(key_) (ARM_int_unlock_SYS(key_))
(5)     ARM_INT_KEY_TYPE ARM_int_lock_SYS(void);
(6)     void ARM_int_unlock_SYS(ARM_INT_KEY_TYPE key_);
(7) #else
(8)     #define ARM_INT_LOCK(key_) do { \
(9)         asm("MRS %0,cpsr" : "=r" (key_)); \
(10)        asm("MSR cpsr_c, #(0x1F | 0x80 | 0x40)"); \
(11)    } while (0)
(12)    #define ARM_INT_UNLOCK(key_) asm("MSR cpsr_c,%0" : : "r" (key_))
    #endif
```

Listing 1 Implementation of the critical for ARM with GNU gcc

The main points of the implementation are as follows:

(1) The `ARM_INT_KEY_TYPE` macro represents the type of the interrupt lock status preserved across the critical section. In the case of the ARM processor, the interrupt lock key is the value of the CPSR register (an `int` is 32-bit wide in ARM gcc).

(2) GNU gcc for ARM pre-defines the macro `__thumb__` when it is invoked with the `-mthumb` compiler option to compile the code in the 16-bit Thumb mode.

(3) The Thumb instruction set does not include the MSR/MRS instructions, which are the only way to manipulate the interrupt bits in the CPSR register. Therefore the `ARM_INT_LOCK()` macro invokes an ARM function `ARM_int_lock_SYS()`, which locks the interrupts and returns the value of the CPSR before locking both the IRQ and FIQ in the CPSR. The gcc compiler and linker add the appropriate

Thumb-to-ARM and ARM-to-Thumb call “veneers” automatically thanks to the `-mthumb-interwork` compiler option (see part 4 of this article series).

(4) The `ARM_INT_UNLOCK()` macro is defined as a call to the ARM function `ARM_int_unlock_SYS()`, which restores the CPSR from the interrupt lock key argument. Both `ARM_int_lock_SYS()` and `ARM_int_unlock_SYS()` are defined in the assembly module `arm_exc.s`. The assembly code used inside these two functions is actually identical as the inline assembly used when the code is compiled to ARM (see highlights for Listing 1(9-11)).

(5-6) The C-prototypes of the assembly functions must be provided for the C compiler. (NOTE: for the C++ version, the prototypes must be defined with the extern “C” linkage specification.)

(7) If the ARM instruction set is used (the GNU `gcc` is invoked with the compiler option `-marm`), the critical section can be defined more optimally without the overhead of a function call for each entry and exit from a critical section.

(8,11) The macro `ARM_INT_LOCK()` is defined as the `do {...} while (0)` compound statement to guarantee syntactically-correct parsing of the macro in every context (including the dangling-else case).

(9) The GNU `gcc` supports the C assembler instructions with C expression operands. This feature is used to pass the C argument (`key_`) to the assembly instruction `MSR`, which saves the state of the CPSR into the provided register argument.

(10) The interrupts are disabled by setting the IRQ and FIQ bits simultaneously in the CPSR register. The most compact load-immediate form of the `MSR` instruction is used which does not clobber any registers.

NOTE: The `MSR` instruction has a side effect of also setting the ARM mode to `SYSTEM`. This is not a problem in this case, because all C-code, including the C-level IRQ and FIQ handlers, execute in the `SYSTEM` mode and no other ARM mode is ever visible to the C-level code (see part 6 of this article).

(12) The macro `ARM_INT_UNLOCK()` restores the CPSR from the interrupt lock key argument (`key_`) passed to the macro. Again, the most efficient store-immediate version of the `MRS` instruction is used, which does not clobber any additional registers.

4. Discussion of the Critical Section Implementation

Various application notes available online (e.g., [1,2]) provide more elaborate implementations of interrupt locking and unlocking for ARM. In particular they use read-modify-write to the CPSR rather than load-immediate to CPSR. In this section I discuss how the simple critical section implementation shown in Listing 1 addresses the potential problems identified in the various application notes.

When the IRQ line of the ARM processor is asserted, and the I bit (bit `CPSR[7]`) is cleared, the core ends the instruction currently in progress, and then starts the IRQ sequence, which performs the following actions (“ARM Architecture Reference Manual, 2nd Edition”, Section 2.6.6 [3]):

- `R14_irq` = address of next instruction to be executed + 4
- `SPSR_irq` = CPSR
- `CPSR[4:0]` = 0b10010 (enter IRQ mode)
- `CPSR[7]` = 1, NOTE: `CPSR[6]` is unchanged
- `PC` = 0x00000018

The ARM Technical Note “What happens if an interrupt occurs as it is being disabled?” [1] points out two potential problems. Problem 1 is related to using a particular function as an IRQ handler and as a regular subroutine called outside of the IRQ scope. Such a dual-purpose function must inspect the SPSR_irq register to detect in which context the handler function is called, which can be problematic. This scenario is impossible in the interrupt handling policy discussed in this article series, because there is no possibility of dual-purpose functions since all C-level IRQ handlers are always called in the SYSTEM mode, where the application programmer has no access to the SPSR register. Problem 2 described in the ARM Technical Note [1] is more applicable to this article and relates to the situation when both IRQ and FIQ are disabled simultaneously, which is actually the case in the implementation of the critical section (see Listing 1(10)). If the IRQ is received during the CPSR write, the ARM7TDMI core sets the I and F bits in both CPSR **and** in SPSR_irq (Saved Program Status Register), and the interrupt **is** entered. If the IRQ handler never explicitly re-enables the FIQ, the fast interrupt will be disabled for the execution time of the IRQ handler and even beyond, until the exit from the critical section. Such situation represents a priority inversion and can extend the FIQ latency beyond the acceptable limit. One of the workarounds recommended in the ARM Note is to explicitly enable FIQ early in the IRQ handler. This is exactly done in the ARM_irq assembler “wrapper”, which I will discuss in detail in the next part (part 8) of this article.

For completeness, this discussion should mention the Atmel Application Note “Disabling Interrupts at Processor Level” [2], which describes another potential problem that might occur when the IRQ or FIQ interrupt is recognized exactly at the time that it is being disabled. The problem addressed in the Atmel Application Note [2] arises when the IRQ or FIQ handler manipulates the I or F bits in the SPSR register, which might lead to enabling interrupts upon the exit from the interrupt right at the beginning of a critical section. This scenario is not applicable in the interrupt handling policy used in this article, because the both the ARM_irq and ARM_fiq “wrapper” functions in assembly never change any bits in the SPSR, which corresponds to the Workaround 1 described in the Atmel Application Note [2].

5. Coming Up Next

In the next part of this article I'll describe the interrupt “wrapper” functions ARM_irq and ARM_fiq in assembly (the ARM_irq and ARM_fiq functions have been introduced in part 6).

6. References

[1] ARM Technical Support Note “What happens if an interrupt occurs as it is being disabled?”, available online at <http://www.arm.com/support/faqip/3677.html>

[2] Atmel Application Note “Disabling Interrupts at Processor Level”, available online at http://www.atmel.com/dyn/resources/prod_documents/DOC1156.PDF

[3] Seal, David, editor, “ARM Architecture Reference Manual Second Edition”, Addison Wesley, 2000.

Miro Samek is the President of [Quantum Leaps, LLC](http://www.quantum-leaps.com), the provider of lightweight, state machine-based, event-driven frameworks for embedded systems. He is the author of “Practical Statecharts in C/C++” (CMP Books, 2002), has written numerous articles for magazines, and is a regular speaker at the Embedded Systems Conferences. He welcomes contact at miro@quantum-leaps.com.