STM32 same while loop code but compiled to different assembly code

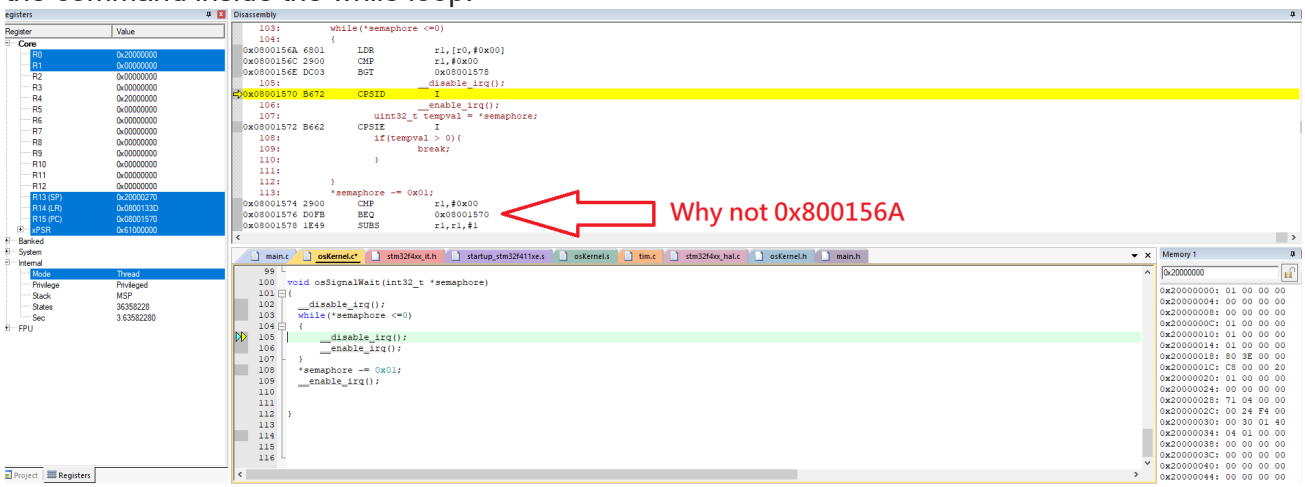
Asked today Active today Viewed 66 times

2

I'm learning RTOS on stm32F411RE board (Cortex-M4). I use MDK uVision v5. I encounter a problem of C code **while loop**. The code in the following is exactly the same in my project and the instructor's project (on Udemy), however, after compiling both project (on my PC), the assembly code look's different. I want to ask what makes this different. Thank you.

```
void osSignalWait(int32_t *semaphore)
{
    __disable_irq();
    while(*semaphore <=0)
    {
            __disable_irq();
            __enable_irq();
    }
    *semaphore -= 0x01;
    __enable_irq();
}
```

In the debug view (see image), if the condition does not match, it does not go to load the real value **LDR r1,[r0, #0x00]** and then do the comparison. Instead, it compares and goes to execute the command inside the while loop.

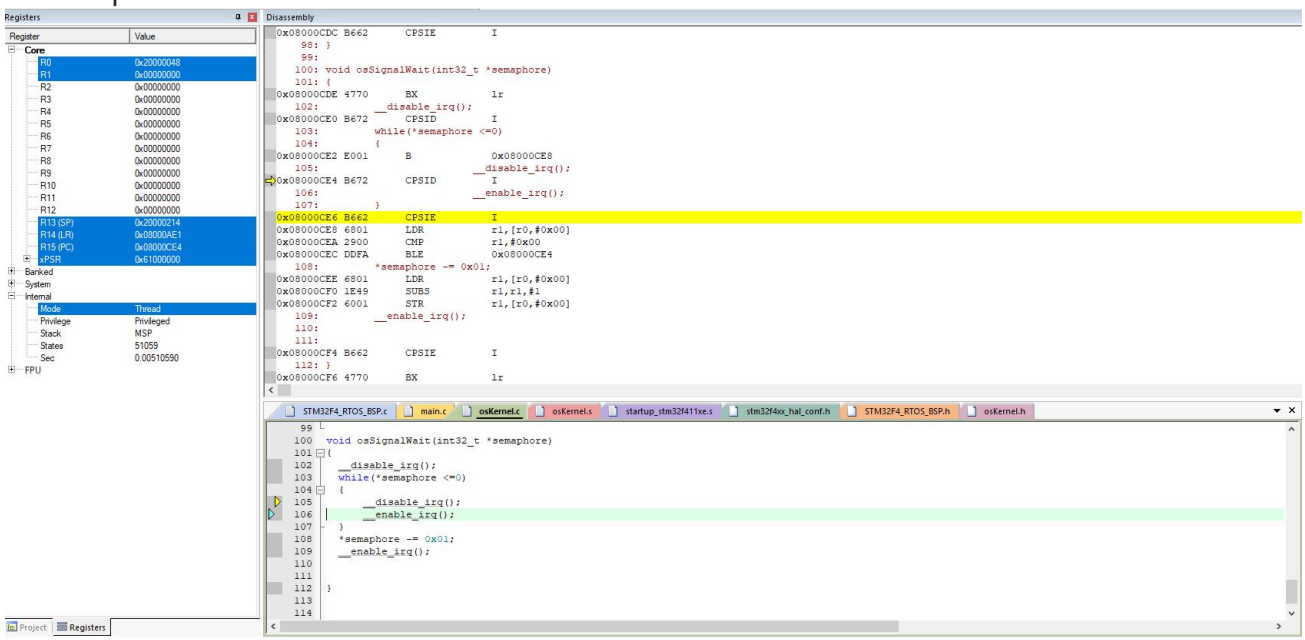


My code compiled below

```
   100: void osSignalWait(int32_t *semaphore)
   101: {
0x08001566 4770      BX             lr
   102:         __disable_irq();
   103:         while(*semaphore <=0)
   104:         {
0x08001568 B672      CPSID          I
   101: {
   102:         __disable_irq();
   103:         while(*semaphore <=0)
   104:         {
0x0800156A 6801      LDR            r1,[r0,#0x00]
0x0800156C E001      B              0x08001572
   105:                         __disable_irq();
```

```
0x0800156E B672      CPSID         I
   106:                          __enable_irq();
   107:          }
   108:          *semaphore -= 0x01;
0x08001570 B662      CPSIE         I
0x08001572 2900      CMP           r1,#0x00
0x08001574 DDFB      BLE           0x0800156E
0x08001576 1E49      SUBS          r1,r1,#1
   109:          __enable_irq();
0x08001578 6001      STR           r1,[r0,#0x00]
0x0800157A B662      CPSIE         I
   110: }
```

If I compile the instructor's (on Udemy) code (on my PC using his project), the assembly code look's different (with exactly the same while loop code). It would load the real value again and do the comparison.



Instructor's code compiled below (Compiled on my PC)

```
100: void osSignalWait(int32_t *semaphore)
   101: {
0x08000CDE 4770      BX            lr
   102:          __disable_irq();
0x08000CE0 B672      CPSID         I
   103:          while(*semaphore <=0)
   104:          {
0x08000CE2 E001      B             0x08000CE8
   105:                          __disable_irq();
0x08000CE4 B672      CPSID         I
   106:                          __enable_irq();
   107:          }
0x08000CE6 B662      CPSIE         I
0x08000CE8 6801      LDR           r1,[r0,#0x00]
0x08000CEA 2900      CMP           r1,#0x00
0x08000CEC DDFA      BLE           0x08000CE4
   108:          *semaphore -= 0x01;
0x08000CEE 6801      LDR           r1,[r0,#0x00]
0x08000CF0 1E49      SUBS          r1,r1,#1
0x08000CF2 6001      STR           r1,[r0,#0x00]
   109:          __enable_irq();
   110:
   111:
```

```
0x08000CF4 B662      CPSIE        I
   112: }
```

c assembly while-loop stm32 cortex-m

edited 30 mins ago

asked 2 hours ago

Dung-Yi

23 ● 4

New contributor

2 Different compiler version? Different compiler options? – Paul Ogilvie 2 hours ago

Did you ask your instructor? – Jabberwocky 2 hours ago

Hi, because the instructor never reply to students, I have to ask the question here. – Dung-Yi 2 hours ago

@PaulOgilvie Which compiler option do I need to look into? Thank you – Dung-Yi 2 hours ago

OK, so the generated assembly code is not the same. Is this a problem? Does *your* code work as expected? If it's not and if the code works, then go on and assume your instructor uses a different version of the compiler and/or uses different compiler settings. Which settings depends on your compiler, mainly optimisation settings, but who knows... If your instructor doesn't answer, you're out of luck. – Jabberwocky 2 hours ago

@Jabberwocky Thank you. My code doesn't work as expected (If I change the Program Counter Register during the debug mode to the LDR r1,[r0, #0x00] address, the code would work!! I want to know how to make the while loop check the real value (in my assembly code, it doesn't check it) – Dung-Yi 2 hours ago

1 @Dung-Yi in the instructor's code image you didn't show the very first line of the function. Yes this matters for us as we can't assume anything. – Jabberwocky 2 hours ago

1 FWIW: I think the body of the while loop should be `__enable_irq(); __disable_irq();` in *that* order. – Chris Hall 2 hours ago

@Jabberwocky Thank you so much. I've updated the image. Please advise. Thank you – Dung-Yi 1 hour ago

2 Answers

2

Since you aren't telling the compiler `semaphore` can change during the execution of this function, your compiler has decided to optimize your code and load the value of semaphore only once and use its copy in the while loop, then only write the result in the end. As it is written now, there's no reason for the compiler to assume this could be harmful.

To notify the compiler a variable *can* change outside the function, during the execution of that function, please use the `volatile` keyword, see:
https://en.cppreference.com/w/c/language/volatile

In that case, your code would become:

```
void osSignalWait(volatile int32_t *semaphore)
{
    __disable_irq();
    while(*semaphore <=0)
    {
        __disable_irq();
        __enable_irq();
    }
    *semaphore -= 0x01;
    __enable_irq();
}
```

By the way, calling `__disable_irq` twice (once before the while loop, then at the start inside the loop) then `__enable_irq` seems a bit wonky, don't you mean enable (and do something) then disable within the while loop?

edited 2 hours ago

answered 2 hours ago

Elijan9

875 ● 2 ● 13 ● 16

Thank you so much. It works and solves my problem in my project. But the instructor's doesn't use volatile, how can the debugger knows that. (The instructor's code works perfectly) – Dung-Yi 2 hours ago

Without optimizations on, the compiler may or may not do the load/store for each iteration. A different compiler or different compiler options may produce a different result. – Elijan9 2 hours ago

1 @Dung-Yi: the *debugger* doesn't know that. It only reloads because it must have been compiled without optimization, so the compiler treated *everything* kind of like `volatile`. See Why does clang produce inefficient asm with -O0 (for this simple floating point sum)?. Or specifically for that bug in your and your instructor's code: MCU programming - C++ O2 optimization breaks while loop / Multithreading program stuck in optimized mode but runs normally in -O0 – Peter Cordes 1 hour ago

1 And BTW, the portable C choice for communication between cores / threads is `_Atomic int32_t *` (with memory_order_relaxed). (If you're disabling IRQs, you can use a separate atomic load and atomic store, not `-=`). @Dung-Yi: Hmm, I think your code is broken: **the `while()` loop body leaves interrupts *enabled* before doing `-=`**. I think you want `enable(); disable();` as the loop body to give interrupt handlers a chance to run while you're spin-waiting. Or better, spin with interrupts enabled until you see `*sem <= 0`, then try disable, check, and decrement. – Peter Cordes 1 hour ago

@Elijan9: I think enable/disable is just there so interrupt handlers can run while spinning, not to actually do anything between them. – Peter Cordes 1 hour ago

1 @Elijan9 - The loop body currently disables interrupts and then enables them, which as Peter Cordes mentions, is backwards. The purpose of those in the loop body (when done in the correct order) is to ensure that when the semaphore becomes available, the loop body exits with interrupts disabled. In the meantime, repeatedly enabling them ensures that interrupts can be serviced during the spin-wait. – phonetagger 1 hour ago

Thank you guys so much. I appreciate your kindness based on my simple and a little bit silly question. (I've done so much research for almost 1 week but cannot find the correct keyword for the answer. I hope the question is not duplicate) – Dung-Yi 1 hour ago

1 @Dung-Yi - The spin-wait technique is a very crude way of handling semaphores. In a good preemptive RTOS, you would be able to wait on the semaphore without spinning. Spinning wastes CPU resources (instruction cycles). The OS (or RTOS) should have a way of waiting/pending on the semaphore by calling some OS API function, and when it returns, either you have the semaphore or the wait timed out (if you gave it a timeout). – phonetagger 1 hour ago

@phonetagger Do you mean using the Cooperative spin-lock? Insert a yield function in the middle of disable(); enable(); I use the following code to do the yield #define ICSR (*((volatile uint32_t *)0xE000ED04)) void osThreadYield(void) { SysTick->VAL = 0; ICSR = 0x04000000; // trigger SysTick } – Dung-Yi 1 hour ago

1 No volatile is needed here at all. The builtin asm has the `memory` clobber acting as a compiler memory barrier. – P__J__ 1 hour ago

@P__J__ - Though the assembly that results from compiling the OP's posted source may read the value that `semaphore` points to on each loop, it would be legal, if `semaphore` is **not** defined as `volatile`, for the compiler to perform an optimization by emitting instructions that read it only once, before the loop, and keep re-using the value without reading it again on each loop. By defining it as `volatile int32_t *semaphore` instead of `int32_t *semaphore`, you prevent the compiler from making that optimization. A memory barrier by itself doesn't solve that problem. – phonetagger 1 hour ago

1 @Dung-Yi I do not mean using a "cooperative spin-lock". I don't know what sort of RTOS you are using; home-brew perhaps? (Did your professor write it?) Any good commercial RTOS would have a "wait on semaphore" API function that puts your thread to sleep until the semaphore is made available by another thread or interrupt. While your thread is sleeping, it consumes no CPU instruction cycles at all. Using a yield or delay function is better than plain spinning, but it still takes intermittent CPU cycles; it's not as good as an API function that's specifically made for waiting on a semaphore. – phonetagger 59 mins ago

@phonetagger - it is not memory barrier instruction only compiler memory clobber. It is something different – P__J__ 59 mins ago

@P__J__ - Where is this `memory` clobber thing you speak of? In which asm listing? Part of which instruction? – phonetagger 53 mins ago

the CMSIS functions: `__attribute__( ( always_inline ) ) __STATIC_INLINE void __enable_irq(void) { __ASM volatile ("cpsie i" : : : "memory"); }` and disable as well. – P__J__ 50 mins ago

@phonetagger: P__J__ is talking about a GNU C inline asm statement with an empty template string but a clobber list that declares to the compiler that arbitrary vars in memory might be modified. `asm("" ::: "memory")`. The optimizer treats it like a black-box function call it can't inline, so only non-escaped local vars can stay in regs. This is a barrier against compile-time reordering and load hoisting, but compiles to zero instructions. preshing.com/20120625/memory-ordering-at-compile-time. I don't see any advantage to doing that over using `volatile` or `_Atomic int`, though – Peter Cordes 44 mins ago

@P__J__ - OK, perhaps it ***should*** work as you say, but clearly the asm in the OP's first listing shows it doesn't work as you say. According to stackoverflow.com/a/47103378/1245420 , either `volatile` or `"memory"` should work, but in this case clearly `"memory"` doesn't. Using `volatile` is the idiomatic (portable) C and C++ way of telling the compiler to avoid optimizing accesses to/from a variable, and I suspect it will work properly for the OP. – phonetagger 40 mins ago

@phonetagger: It works with GCC 8.2: godbolt.org/z/fbpe2d. Perhaps Keil is buggy? If memory access gets reordered wrt. enable / `__disable_irq()` that's a potential problem, if any code ever depends on that for non-volatile accesses. Also note that GCC avoids a 2nd load of the semaphore with the barriers but not volatile case. It still has the value in a register from the loop condition when compiling `*semaphore -= 1;` so it only needs a `sub` and `str`. To get that with `volatile` or `_Atomic`, you'd need to load into a tmp var as part of the loop condition. – Peter Cordes 30 mins ago

0

This very well known keil over optimisation bug. Reported many times. Having memory clobber it should read the memory every time.

Here is an example how clobbers work

```c
#include <stdint.h>

unsigned x;
volatile unsigned y;


int foo()
{
    while(x < 1000);
}

int bar()
{
    while(x < 1000) asm("":::"memory");
}
```

```
foo:
        ldr     r3, .L5
        ldr     r3, [r3]
        cmp     r3, #1000
        bxcs    lr
.L3:
        b       .L3
.L5:
        .word   x
bar:
        ldr     r1, .L11
        ldr     r2, .L11+4
        ldr     r3, [r1]
        cmp     r3, r2
        bxhi    lr
.L9:
        ldr     r3, [r1]
        cmp     r3, r2
        bls     .L9
        bx      lr
.L11:
        .word   x
        .word   999
```

answered 52 mins ago

P__J__
20.4k ● 3 ● 12 ● 36

That's not an over-optimization bug, it's data-race UB in the C source that you can work around with a memory barrier or volatile. Or actually *avoid* UB with `_Atomic` (with `mo_relaxed` so it can compile to the same asm). Why do you say a barrier is better than `volatile`, or `_Atomic int` (with mo_relaxed)? Since this is for a semaphore object that's *only* ever used for synchronization so you never want to let the compiler keep it in a register across multiple reads. `volatile` ensures we don't have invented loads from one read. – Peter Cordes 42 mins ago

Thank you for point out the issue. I did find this bug reported in stack overflow before but never realize that my problem is caused by this!! Thank you. – Dung-Yi 35 mins ago

@PeterCordes I do not say what is better. They are not identical and have different uses godbolt.org/z/aXSMB_ – P__J__ 24 mins ago

@Dung-Yi: Just to be clear; it's not a *compiler* bug, it's a bug in your code. The compiler is working as intended and optimizing variables into registers under the assumption that no other thread can modify them, unless you tell it otherwise. It's allowed to assume that for non-`_Atomic` and non-`volatile`

variables because otherwise that would be data-race UB. If compilers didn't do this, any code that used global variables or even pointers would run slowly, storing after every change and reloading before every use. – Peter Cordes 23 mins ago

@P__J__: I think I was getting mixed up between the comment thread on the other answer vs. this. But why post it when there's already an answer suggesting `volatile`? The interesting point is that the `"memory"` barrier that should be part of the definition of `__disable_irq()` should have already blocked reordering, like it does on GCC. godbolt.org/z/fbpe2d. (But yes I'm aware that memory barriers don't affect non-escaped local vars, like a function arg.) – Peter Cordes 18 mins ago

@PeterCordes Thank you for correction. – Dung-Yi just now Edit