STM32 same while loop code but compiled to different assembly code

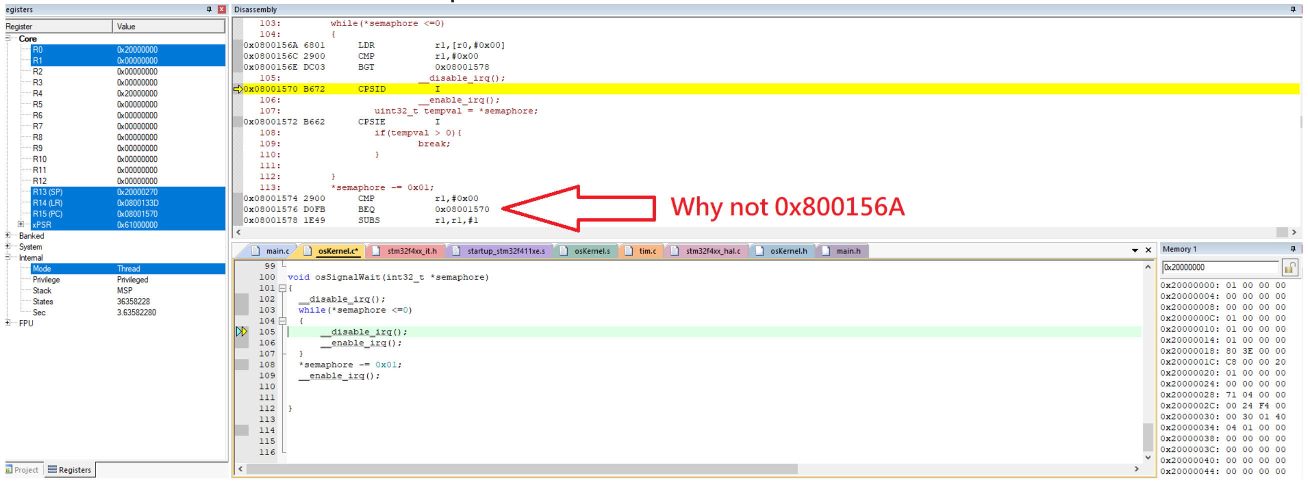
Asked today Active today Viewed 66 times

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.



Register	Value
Core	
R0	0x20000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x20000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000270
R14 (LR)	0x0800133D
R15 (PC)	0x08001570
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	36358228
Sec	3.63582280
FPU	

```
   103:         while(*semaphore <=0)
   104:         {
0x0800156A 6801      LDR          r1,[r0,#0x00]
0x0800156C 2900      CMP          r1,#0x00
0x0800156E DC03      BGT          0x08001578
   105:                   __disable_irq();
0x08001570 B672      CPSID        I
   106:                   __enable_irq();
   107:              uint32_t tempval = *semaphore;
0x08001572 B662      CPSIE        I
   108:              if(tempval > 0){
   109:                     break;
   110:              }
   111:
   112:         }
   113:         *semaphore -= 0x01;
0x08001574 2900      CMP          r1,#0x00
0x08001576 D0FB      BEQ          0x08001570
0x08001578 1E49      SUBS         r1,r1,#1
```

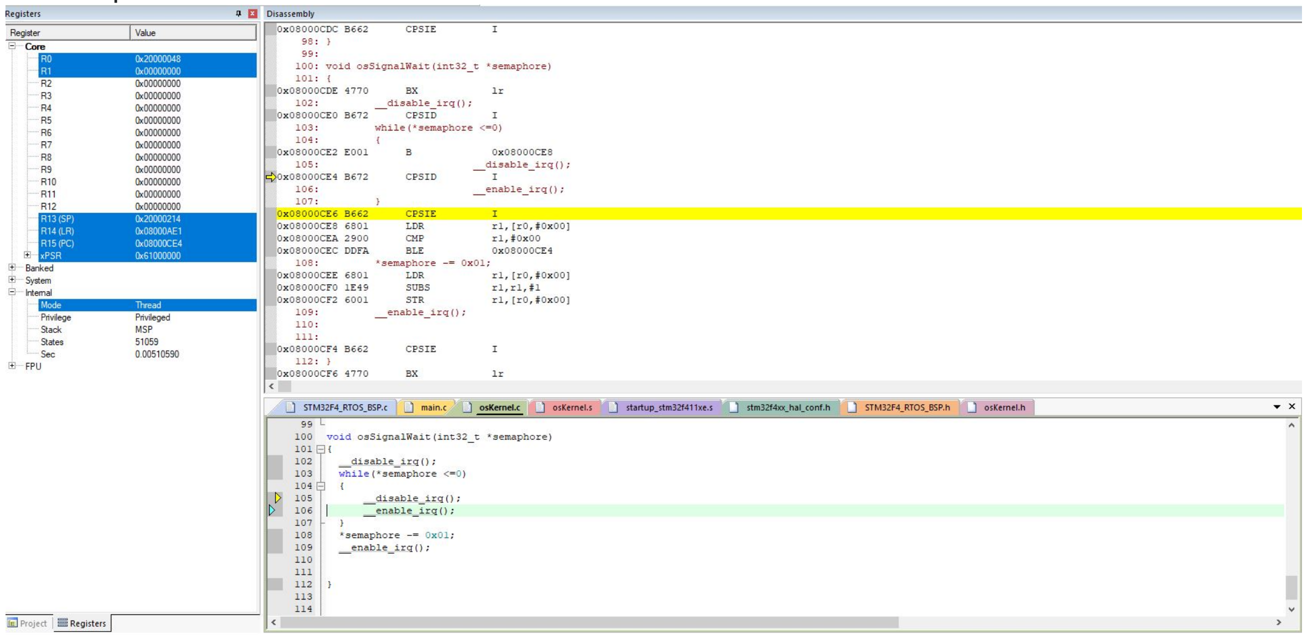
```
void osSignalWait(int32_t *semaphore)
{
  __disable_irq();
  while(*semaphore <=0)
  {
      __disable_irq();
      __enable_irq();
  }
  *semaphore -= 0x01;
  __enable_irq();


}
```

```
0x20000000: 01 00 00 00
0x20000004: 00 00 00 00
0x20000008: 00 00 00 00
0x2000000C: 01 00 00 00
0x20000010: 01 00 00 00
0x20000014: 01 00 00 00
0x20000018: 80 3E 00 00
0x2000001C: C8 00 00 20
0x20000020: 01 00 00 00
0x20000024: 00 00 00 00
0x20000028: 71 04 00 00
0x2000002C: 00 24 F4 00
0x20000030: 00 30 01 40
0x20000034: 04 01 00 00
0x20000038: 00 00 00 00
0x2000003C: 00 00 00 00
0x20000040: 00 00 00 00
0x20000044: 00 00 00 00
```

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.



Register	Value
Core	
R0	0x20000048
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000214
R14 (LR)	0x08000AE1
R15 (PC)	0x08000CE4
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	51059
Sec	0.00510590
FPU	

```
0x08000CDC B662      CPSIE          I
    98: }
    99:
   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: }
0x08000CF6 4770      BX             lr
```

```

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


}


```

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 16 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

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

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

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

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

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

1 No volatile is needed here at all. The builtin asm has the `memory` clobber acting as a compiler memory barrier. – P__J__ 58 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

```
#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 38 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 26 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 19 mins ago

@PeterCordes I do not say what is better. They are not identical and have different uses godbolt.org/z/aXSMB_ – P__J__ 8 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 7 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 2 mins ago