ELECTRICALENGINEERING

Volatile keyword with ADC, USB interrupt readings

Asked 8 months ago Active 8 months ago Viewed 140 times

1

I was just reading an answer about global variables, it also stated that I should use volatile affix whenever a global is used in an interrupt.

- In my program I am using a 30 byte ADC buffer without interrupt, it just gets regularly updated with DMA transfers.
- There is also a 4 byte USB buffer which gets updated whenever I receive an USB interrupt.
- There is a variable that gets changed every time main while loop starts over, it reads one of the adc values and updates itself with a proper math function, and then gets used in another function frequently to generate something else.
- There is a counter which I increment in every while loop start over, it is used widely in my code ecosystem. Has nothing to do with interrupts or external sources, just gets incremented whenever code finishes a full cycle.

In which of these four situations I should use volatile and would it lead to any memory based side effects in my system?

I wasn't using volatile fix since now and not sure if I ever faced a problem related to this.

I am using cortex M-4, pure C code.

c embedded non-volatile-memory

edited Apr 9 at 18:58
Electric_90
1,988 6 20

asked Apr 9 at 15:49
Can Uysal
79 5

1 The key thing to understand here is that **volatile is not enough** - you also need to understand when non-atomic access may occur and when that may lead to invalid values, ie, those that are neither a self consistent old value nor a self consistent new value, but an invalid mix of the two, for example counting 99, 100 but seeing 199. – Chris Stratton Apr 9 at 16:30

2

> In my program I am using a 30 byte ADC buffer without interrupt, it just gets regularly updated with DMA transfers.

Any DMA buffer needs to be volatile qualified to let the compiler know it cannot make any assumptions about the contents there.

> There is also a 4 byte USB buffer which gets updated whenever I receive an USB interrupt.

If it is shared between an interrupt and the main program, you need to volatile qualify it to prevent optimization hiccups. You *must* also use some means of protection against race conditions.

> There is a variable that gets changed every time main while loop starts over, it reads one of the adc values and updates itself with a proper math function, and then gets used in another function frequently to generate something else.

It does not need to be volatile. The memory-mapped ADC register itself needs to be volatile-qualified since it can change at any time.

> There is a counter which I increment in every while loop start over, it is used widely in my code ecosystem. Has nothing to do with interrupts or external sources, just gets incremented whenever code finishes a full cycle.

No volatile needed.

More info here.

answered Apr 10 at 6:39

Lundin
5,719 13 35

Thanks for the answer Lundin, I have few other questions if you don't mind. In my program I am using spi half/full complete callbacks and only change state type definiton such as spi_state_half_complete or complete. Should I make that enumeration be volatile as well? In my main execution I'm waiting for it to change state after I have finished all the stuff to do. Other question is, related to the race conditions, I'm not using a lock mechanism when I'm directly reading from the RNG register itself, register is volatile. What can go wrong while doing so due to race conditions? – Can Uysal Apr 10 at 22:36

There is also an other thing which is I completely disabled ADC interrupts due to over-occupying my program memory, and I'm just reading the register whenever I need it without thinking if it is getting transferred by DMA at that instance or not. What is bad about this, can it lead to a bug? – Can Uysal Apr 10 at 22:37

@CanUysal Depends what you mean with callback. It's quite simple: is the variable shared between an ISR and the background program? If so, it must be volatile. If not, it need not be volatile. – Lundin Apr 11 at 11:26

@CanUysal Regarding race conditions, who reads from the register where? You need to protect variables shared by the ISR and the background program from race conditions. It doesn't matter if the shared data is plain variables or MCU registers. Volatile does nothing to protect against race conditions. As for what can go wrong, see the linked answer and dig through the comments below it. I gave an example from real life where I encountered such a bug during register access. electronics.stackexchange.com/questions/409545/… – Lundin Apr 11 at 11:28

@CanUysal Regarding your ADC, is it DMA:d or not? If DMA then volatile, always. Though many ADC have a continuous conversion feature to overwrite last sampled value internally, in which case you don't need DMA but can just grab the value when needed. DMA is for hard real time when you need to catch every single ADC sample. – Lundin Apr 11 at 11:39

4

You need the `volatile` keyword because a good optimizing compiler will try to reduce memory accesses. If the variable in question can change outside of what the code is doing to it, or if the pertinent memory location is used as an output, then that's not behavior you want.

The `volatile` keyword tells the compiler "keep the physical memory up to date, and don't assume that the physical memory stays the same". There's some detail about exactly where it reads and writes, which you can find in a longer article about this, but that's the essence.

So, this tells you:

- If it gets updated by hardware, or if it affects hardware, it needs to be volatile.
- If it gets updated or used by an interrupt and used or updated elsewhere, it needs to be volatile. You can think of an interrupt as "looking like" hardware to the compiler, because it's not within the thread of control that the compiler knows about.
- If it's getting used to communicate between threads (as in an RTOS), then it needs to be volatile (and yes, there are RTOS mechanisms for this -- and sometimes they're the right mechanisms. When they aren't, and you need to just write to or read from a global -- use volatile).
- If it's set and read in the *same thread of control*, it does not need to be volatile, even if it's global.

edited Apr 10 at 14:56

answered Apr 9 at 16:13

TimWescott

16.6k 1 16 31

"If it gets updated or used by an interrupt" Rather, if it is shared between an ISR and the main program. Local variables only used by the ISR need not be volatile. – Lundin Apr 10 at 6:50

1 @Lundin thanks -- I've clarified that point. – TimWescott Apr 10 at 14:57

Thanks for the answer Tim, I wish I could have selected two solutions. – Can Uysal Apr 10 at 22:39

2

The rule is that you need `volatile` whenever memory contents are modified outside of regular program flow:

- hardware registers change contents without the program doing anything
- DMA buffers are modified external to the program
- interrupts change program flow

The compiler's data flow analysis follows normal execution paths and decides when to store data in memory or keep it in registers, based on access frequency and availability of registers.

The obvious example is

```
int i; // global

…
    while(i == 0); // wait for something to happen
```

Without optimization, you get

```
1:
    ldr r0, =i   ; get address
    ldr r0, [r0] ; get value
    tst r0       ; update flags
    beq 1b
```

The compiler knows that the address of `i` never changes, so it can move the first `ldr` out of the loop, and no code path stores anything to this address, so it can also move the second `ldr`. The remainder is an empty endless loop, so the code becomes

```
    ldr r0, =i
    ldr r0, [r0]
    tst r0
    bne 2f
1:
    b 1b
2:
```

This is obviously wrong, but normally the compiler runs out of registers rather quickly, so if the loop isn't empty and doesn't refer to `i` anymore, it is quite likely that `i` will be reloaded even if it isn't marked `volatile`.

As @Evan suggests, `volatile` isn't used for multithreaded code, even though the compiler cannot see the effects of the other threads. The important bit here: the only way you can be sure to have a consistent view is to hold a lock, and the locking functions are then treated as a barrier by the compiler, forcing a reload.

This is also how the Linux kernel gets away with non- `volatile` DMA buffers: this is correct because the driver is called only from paths that have locked some data structure (and the compiler can see that call path, DMA buffers are generally not accessed from interrupt code!), and this causes the compiler to emit loads and stores.

answered Apr 9 at 16:26

Simon Richter
7,437 1 12 31

What I noticed is that when I add few volatiles in my code, cpu usage jumps about %5 or so, this optimization must be the reason right? – Can Uysal Apr 10 at 22:16

Quite possible, or the program behaving differently and doing more work now because it would throw data away before. – Simon Richter Apr 11 at 8:52

1

The volatile qualifier prevents the optimizer from eliminating or reordering access to a memory address -- every evaluation or assignment will result in its own memory load or store.

a) The buffer should be declared volatile as it is updated by the DMA mechanism which the compiler doesn't know about.

b) The buffer should be declared volatile because it is updated in the interrupt context and the main thread of control.

c) As long as this counter is only manipulated from the main thread of control it does not need to be volatile. The compiler knows that function calls can modify memory.

d) Again, you don't need volatile here.

e) You didn't ask but there is at least one other major reason to use volatile, and that is for memory mapped IO or machine registers. Here, the memory reads or writes have important side effects so need to happen exactly as specified. If you have a hardware RNG that is memory mapped such that each read returns a new random number, that should be volatile otherwise the compiler may coalesce multiple reads. These hardware specific cases are usually handled by the platform SDK assuming you are using it. If you manually declare hardware regions you will need to take care of the volatile qualifier yourself.

f) One place *not* to use volatile: multi-threading. Superficially this appears to be the same as the interrupt context, and indeed there are some similarities. However, volatile only ensures that

memory reads and writes are preserved, it does not enforce that anything in particular is atomic. On a single core CPU interrupts are atomic -- the IRS will finish before the main thread resumes so you don't have to worry about the main thread seeing a partially updated state. The reverse is still a problem: the ISR can generally interrupt the main thread at any time, so the ISR shouldn't make assumptions about the state. If you do, you have to guard changes to that with interrupt disable functions.

For multi-threading you generally want to use concurrency primitives like mutex locks and thread safe queues. These include the appropriate memory barriers to make sure that other threads see the updates. Those concurency primitives may internally use volatile values, but you generally won't need or want to explicitly use volatile qualifications for multi-threading (unless one of the other conditions applies).

For much more detail about why not to use volatile in high-level multi-threaded application see this question and answer:

https://stackoverflow.com/questions/4557979/when-to-use-volatile-with-multi-threading

edited Apr 9 at 17:12

answered Apr 9 at 16:08

Evan

2,175 5 16

Unfortunately, your arguments about ISRs vs. threads seriously garble the issue of atomicity to the degree that what you are saying is neither accurate nor useful guidance. Nor do mutexes remove the need for volatile! – Chris Stratton Apr 9 at 16:29

in e, what do you mean by memory mapped IO? I do have a variable that reads directly from the RNG hardware registers without looking if its done or not, does it count as mmIO? Should I make it volatile as well? Other thing is should GPIO input variables be volatile too? I'm confused. – Can Uysal Apr 9 at 16:29

Typically the definitions for memory mapped I/O registers in your MCU specific files already declare them volatile, often through some other I/O-flavored intermediate macro. – Chris Stratton Apr 9 at 16:34

2 "On a single core CPU interrupts are atomic -- the IRS will finish before the main thread resumes so you don't have to worry about the main thread seeing a partially updated state" This is plain wrong! Even on single-thread ISR, any variable which is not `_Atomic` qualified (C11) may be updated in several instructions. An interrupt may fire between two such instructions. And if programming in C, it doesn't matter the slightest if it is a 32 bit variable and the CPU is 32 bit - unless it is `_Atomic` then you have no guarantee that the variable will get updated in a single instruction, period. – Lundin Apr 10 at 6:45

1 @Evan Regarding compiler optimizations " All C compilers even for embedded systems assume explicit callbacks can change memory". Well, on a bare metal system you don't have thread callbacks but only ISRs. Historically almost every embedded system compiler gets this wrong, particularly with optimizations enabled. I've been debugging that particular bug numerous times. This is why we declare such variables volatile regardless of compiler - better safe than sorry. As for higher level callbacks like thread/process context switches, they may very well be implemented through ISRs. – Lundin Apr 11 at 11:22