C keywords: Don't flame out over volatile

faizan khan (/user/faizk)

AUGUST 05, 2016

Consider the following code,

```
struct _a_struct{
      int x;
      int y
      volatile bool alive=false;
} ASTRUCT;

ASTRUCT a_struct;


//Thread 1
a_struct.x = x;
a_struct.y = y;
a_struct.alive =true;


//thread 2
if (a_struct.alive==true)
{
      draw_struct(a_struct.x, a_struct.y);
}
```

This is a normal scenario of an object being shared between two threads, one thread is updating its value and the other is waiting for the updated values. The above code seems harmless but there is something wrong with it. It will not produce the desired results (https://msdn.microsoft.com/en-us/library/windows/desktop/ee418650(v=vs.85).aspx). We could try to make everything in the structure volatile but this would produce inefficient code. We don't want to lose efficiency; we just want to share data between the two threads. In this article, I will show you what problems can be caused by the above code, and why should we avoid the solution of placing volatile with everything.

This article will have two parts. The first part will try to resolve all the confusion surrounding volatile. We will discuss its semantics: declaration and assignments, its use in a multi-threaded environment and its use in a kernel setting. The second part will lay out use cases where volatile is considered necessary like in a setjmp and longjmp, signal handling and inline assembly.

My motivation for writing this article is to make some sense of the chaos surrounding volatile. I wanted this article to be a complete guide for the use of volatile. But from what I have researched, it's not that we don't know how to use this keyword, it's just that we don't know when to stop using it. The basic use cases, which are as follows, are well known (http://www.barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword).

1. Use volatile on memory mapped IO
2. Use volatile on global data shared between multiple tasks

3. Use volatile on data used in ISR.

A lot has been written on them, and it is because of this reiteration of these basic use cases that we tend to start using volatile everywhere. All of the code I see seems to lie in one of the above categories. For example I was faced with a problem related to global data accessed in an ISR. I placed volatile with everything I thought could cause the problem and the code worked fine. But as we will see later, this solution hid the problem as opposed to solving it. Naturally an issue was reported after a few months regarding loss of data synchronization.

Points of Confusion

The Setting

The second edition (K&R%2520Dennis%2520Ritchie) of K&R introduced volatile as follows.

> "*The purpose of volatile is to force an implementation to suppress optimization that could otherwise occur. For example, for a machine with memory-mapped input/output, a pointer to a device register might be declared as a pointer to volatile, in order to prevent the compiler from removing apparently redundant references through the pointer." —Dennis Ritchie.*

Volatile was introduced to inform the compiler of special memory mapped regions. There is a usenet group (https://groups.google.com/forum/#!msg/comp.std.c/tHvQhiKFtD4/zflgJhbkCXcJ) post that says that before volatile, compilers had to use strange heuristics to identify which regions were memory mapped and which were not. So volatile was a necessary addition to the C language at the time. But beyond this basic use case, volatile is widely misused (http://blog.regehr.org/archives/28).

Back in those days most processors were single core and executed the instructions in-order (https://courses.cs.washington.edu/courses/csep548/06au/lectures/introOOO.pdf). What you wrote and the order you wrote it in got executed without a scratch. Nowadays the code that we write, known as abstract machine by the C standard, is a lot different from the actual implementation that gets executed. In fact according to the C standard the only thing the compiler has to make sure to produce executable code is the end result.

The only thing the compiler cannot do is remove or reorder accesses to volatile qualified objects (C-99 rationale (http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf)). But it can freely remove or reorder non-volatile accesses around it. The compiler can also upgrade some objects to become volatile qualified after certain expressions. Therefore understanding the usage of volatile (http://www.embedded.com/design/programming-languages-and-tools/4415475/Guidelines-for-handling-volatile-variables) in basic use cases like declarations and assignments is also necessary. Some developers still struggle to differentiate between a volatile object and a volatile pointer. In the next section I will start with addressing these points of confusions one by one. They are arranged in order of their increasing confusion regarding the usage of volatile.

Next Page >> (http://www.embedded.com/design/programming-languages-and-tools/4442490/2/C-keywords--Don-t-flame-out-over-volatile)

MOST COMMENTED

Currently no items

RELATED CONTENT

02.24.2012 | DISCUSSION

Combining C's volatile and const keywords (/electronics-blogs/barr-code/4236917/Combining-C-s-volatile-and-const-keywords)

08.05.2016 | DESIGN

C keywords: Don't flame out over volatile (/design/programming-languages-and-tools/4442490/C-keywords--Don-t-flame-out-over-volatile)

07.22.2013 | DISCUSSION

When to use – and not use – the volatile keyword (/electronics-blogs/cole-bin/4418638/When-to-use---and-not-use---the-volatile-keyword)

07.02.2001 | DISCUSSION

Introduction to the volatile keyword (/electronics-blogs/beginner-s-corner/4023801/Introduction-to-the-Volatile-Keyword)

11.16.2005 | DISCUSSION

Destination ESC blog (http://www.embedded.com/electronics-blogs/4219317/Destination-ESC) on Embedded.com and social media accounts Twitter (https://twitter.com/ESC_Conf), Facebook (https://www.facebook.com/EELiveShow), LinkedIn (https://www.linkedin.com/groups?gid=1403157), and Google+ (https://plus.google.com/110966938586722247542/posts).

Embedded.com, EE Times, EBN, EDN, and Planet Analog are owned by AspenCore. The Embedded Systems Conference is owned by UBM.

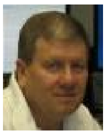
(mailto:?subject=C keywords: Don't flame out over volatile&body=http://www.embedded.com/design/programming-languages-and-tools/4442490/C-keywords--Don-t-flame-out-over-volatile)

< Previous Page 1 of 4 Next > (/design/programming-languages-and-tools/4442490/2/C-keywords--Don-t-flame-out-over-volatile)

Place volatile accurately (/electronics-blogs/programming-pointers/4025609/Place-volatile-accurately)

6 COMMENTS

WRITE A COMMENT



Antedeluvian (/User/Antedeluvian) POSTED: AUG 11, 2016 6:23 PM EDT

Since my C has been self taught, I rarely get into more sophisticated constructs. I even avoid the use of pointers. I recently had to get into volatile declarations to stop the @#$! assembler optimizing out what I want. I often use dummy writes to a location simply as a place holder for a breakpoint. It gets optimized out and there is no breakpoint.

Only today I thought I had found a bug in my debugger where the expected variable update did not occur until several instructions later. Tech support suggested making the variable volatile to allow for the variable to be updated at the right point during debugging, in order to verify correct operation.
I think the compilers are now over-optimizing!

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4442490%2FC-KEYWORDS--DON-T-FLAME-OUT-OVER-VOLATILE)

MichaelM985 (/User/MichaelM985) POSTED: AUG 12, 2016 4:42 AM EDT

@antedeluvian :-
I am like you a late comer to C99 by the way of Cypress Creator 3.X.
As I am at that envious point in life where learning new things is becoming a race against time, I keep my C in the KISS state "Keep It Simple Stupid" and with lots of comments.
I find that I am now asking the compiler to not perform optimisation.
When I wrote code for 2K ROMS for 6502 embedded systems then I had bolt on optimsers and even the dreaded compression compilers which found similar bits of code and reused it with a jump to that point. That made chasing a bug very hard. The point is that then I was able to decide the level of optimisation that I wanted.
So lets have big drop downs on our compiler IDE that lets you easily choose what optimisation or not you need?
Best to all Crusty

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4442490%2FC-KEYWORDS--DON-T-FLAME-OUT-OVER-VOLATILE)

Jorick23 (/User/Jorick23) POSTED: AUG 10, 2016 3:40 PM EDT

On page 3 you write, "For example if an ISR occurs, the value of my_delay is read and modified, and before it is written, another interrupt occurs. The wait() function will read my_delay in the meantime and get the old value, this is called a torn read."

This won't happen. The second interrupt will return to the first, which then finishes writing my_delay. The first interrupt then returns to the wait() function and my_delay is read correctly. The wait() function can't read anything until all interrupts are finished.

A better example would be incrementing a uint16_t in an interrupt on an 8-bit processor, and reading it at task level. Many processors require two 8-bit reads to get the 16-bit value into registers. If the 16 bit value is 0x00FF and the interrupt hits just after the high byte of 0x00 is read, the value increments to 0x0100 and is written. The interrupt returns and the low byte (now 0x00) is read, resulting in 0x0000 in the two registers instead of the correct 0x0100. The read at the task level should be placed in a critical section to avoid this. The write in the interrupt doesn't have to be put in a critical section unless my_delay is used by a higher-level interrupt.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4442490%2FC-KEYWORDS--DON-T-FLAME-OUT-OVER-VOLATILE)

GonzaloSerrano (/User/GonzaloSerrano) **POSTED: AUG 8, 2016 6:36 AM EDT**

I think every embedded engineer must read this column. It helped me alot.

Just one thing: at the end of page 3 tmp = my_delay(); should be tmp = my_delay;

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4442490%2FC-KEYWORDS--DON-T-FLAME-OUT-OVER-VOLATILE)

_mrn (/User/_mrn) **POSTED: AUG 6, 2016 4:10 PM EDT**

Hi,
1. On page 2 you write "When modifying a volatile type with non-volatile type, the compiler will implicitly cast the non-volatile type to a volatile type." I am not sure what you mean, but you give an example of assigning pi to pvi and comment that pi becomes volatile with this assignemnt. I do not think you are right. pi will remain non-volatile, it does not matter it holds the same address as pvi which is a pointer to volatile.
2. On page 3 you write: "Naturally aligned reads are atomic." What is the source of this rule? I don't think this is guaranteed by the C standard.
3. EnterCritical() and ExitCritical() calls are mixed in your example on page 3.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4442490%2FC-KEYWORDS--DON-T-FLAME-OUT-OVER-VOLATILE)

Faizk (/User/Faizk) **POSTED: AUG 6, 2016 6:37 PM EDT**

1. Yes you are correct, pi does not become volatile. But the compiler will cast the non volatile type to volatile type, what I was trying to say is that its better to cast it yourself, this problem should have been discussed in the function parameters topic where this is of actual value...Will fix that

2. This is not a C standard rule. But it is just an observation, and upheld by most of the compilers.

3. Yea thanks for pointing that out will fix it...

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4442490%2FC-KEYWORDS--DON-T-FLAME-OUT-OVER-VOLATILE)