

Introduction to the volatile keyword

[Embedded Staff](#)

July 2, 2001

The use of volatile is poorly understood by many programmers. This is not surprising, as most C texts dismiss it in a sentence or two.

Have you experienced any of the following in your C/C++ embedded code?

- Code that works fine-until you turn optimization on
- Code that works fine-as long as interrupts are disabled
- Flaky hardware drivers
- Tasks that work fine in isolation-yet crash when another task is enabled

If you answered yes to any of the above, it's likely that you didn't use the C keyword **volatile**. You aren't alone. The use of **volatile** is poorly understood by many programmers. This is not surprising, as most C texts dismiss it in a sentence or two.

volatile is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time-without any action being taken by the code the compiler finds nearby. The implications of this are quite serious. However, before we examine them, let's take a look at the syntax.

Syntax

To declare a variable volatile, include the keyword **volatile** before or after the data type in the variable definition. For instance both of these declarations will declare foo to be a volatile integer:

```
volatile int foo;  
int volatile foo;
```

Now, it turns out that pointers to volatile variables are very common. Both of these declarations declare foo to be a pointer to a volatile integer:

```
volatile int * foo;  
int volatile * foo;
```

Volatile pointers to non-volatile variables are very rare (I think I've used them once), but I'd better go ahead and give you the syntax:

```
int * volatile foo;
```

And just for completeness, if you really must have a volatile pointer to a volatile variable, then:

```
int volatile * volatile foo;
```

Incidentally, for a great explanation of why you have a choice of where to place volatile and why you should place it after the data type (for example, **int volatile * foo**), consult Dan Sak's column, "Top-Level cv-Qualifiers in Function Parameters" (February 2000, p. 63).

Finally, if you apply **volatile** to a struct or union, the entire contents of the struct/union are volatile. If you don't want this behavior, you can apply the volatile qualifier to the individual members of the struct/union.

Use

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

- Memory-mapped peripheral registers

- Global variables modified by an interrupt service routine
- Global variables within a multi-threaded application

Peripheral registers

Embedded systems contain real hardware, usually with sophisticated peripherals. These peripherals contain registers whose values may change asynchronously to the program flow. As a very simple example, consider an 8-bit status register at address 0x1234. It is required that you poll the status register until it becomes non-zero. The naive and incorrect implementation is as follows:

```
UINT1 * ptr = (UINT1 *) 0x1234;
```

```
// Wait for register to become non-zero.
```

```
while (*ptr == 0);
```

```
// Do something else.
```

This will almost certainly fail as soon as you turn the optimizer on, since the compiler will generate assembly language that looks something like this:

```
mov ptr,#0x1234  mov a,@ptrloop  bz  loop
```

The rationale of the optimizer is quite simple: having already read the variable's value into the accumulator (on the second line), there is no need to reread it, since the value will always be the same. Thus, in the third line, we end up with an infinite loop. To force the compiler to do what we want, we modify the declaration to:

```
UINT1 volatile * ptr =  
(UINT1 volatile *) 0x1234;
```

The assembly language now looks like this:

```
mov ptr, #0x1234
loop mov a, @ptr
bz loop
```

The desired behavior is achieved.

Subtler problems tend to arise with registers that have special properties. For instance, a lot of peripherals contain registers that are cleared simply by reading them. Extra (or fewer) reads than you are intending can cause quite unexpected results in these cases.

Interrupt service routines



Interrupt service routines often set variables that are tested in main line code.

For example, a serial port interrupt may test each received character to see if it is an ETX character (presumably signifying the end of a message). If the character is an ETX, the ISR might set a global flag. An incorrect implementation of this might be:

```
int etx_rcvd = FALSE;
```

```
void main()
{
    ...
    while (!etx_rcvd)
    {
        // Wait
    }
    ...
}
```

```
interrupt void rx_isr(void)
```

```
{  
    ...  
    if (ETX == rx_char)  
    {  
        etx_rcvd = TRUE;  
    }  
    ...  
}
```

With optimization turned off, this code might work. However, any half decent optimizer will “break” the code. The problem is that the compiler has no idea that **etx_rcvd** can be changed within an ISR. As far as the compiler is concerned, the expression **!etx_rcvd** is always true, and, therefore, you can never exit the while loop. Consequently, all the code after the while loop may simply be removed by the optimizer. If you are lucky, your compiler will warn you about this. If you are unlucky (or you haven't yet learned to take compiler warnings seriously), your code will fail miserably. Naturally, the blame will be placed on a “lousy optimizer.”

The solution is to declare the variable **etx_rcvd** to be **volatile** . Then all of your problems (well, some of them anyway) will disappear.

Multi-threaded applications

Despite the presence of queues, pipes, and other scheduler-aware communications mechanisms in real-time operating systems, it is still fairly common for two tasks to exchange information via a shared memory location (that is, a global). When you add a pre-emptive scheduler to your code, your compiler still has no idea what a context switch is or when one might occur. Thus, another task modifying a shared global is conceptually identical to the problem of interrupt service routines discussed previously. So all shared global

variables should be declared **volatile**. For example:

```
int cntr;

void task1(void)
{
    cntr = 0;
    while (cntr == 0)
    {
        sleep(1);
    }
    ...
}

void task2(void)
{
    ...
    cntr++;
    sleep(10);
    ...
}
```

This code will likely fail once the compiler's optimizer is enabled. Declaring `cntr` to be volatile is the proper way to solve the problem.

Final thoughts

Some compilers allow you to implicitly declare all variables as volatile. Resist this temptation, since it is essentially a substitute for thought. It also leads to potentially less efficient code.

Also, resist the temptation to blame the optimizer or turn it off. Modern

optimizers are so good that I cannot remember the last time I came across an optimization bug. In contrast, I come across failures to use **volatile** with depressing frequency.

If you are given a piece of flaky code to "fix," perform a grep for **volatile** . If grep comes up empty, the examples given here are probably good places to start looking for problems.

Nigel Jones is a consultant living in Maryland. When not underwater, he can be found slaving away on a diverse range of embedded projects. He can be reached at .

[Return to July 2001 Table of Contents](#)