# **How to Use C's volatile Keyword**

*The proper use of C's volatile keyword is poorly understood by many programmers. This is not surprising, as most C texts dismiss it in a sentence or two. This article will teach you the proper way to do it.*

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

* Code that works fine--until you enable compiler optimizations
* Code that works fine--until interrupts are enabled
* Flaky hardware drivers
* RTOS tasks that work fine in isolation--until some other task is spawned

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. Unfortunately, most books about the C programming language dismiss volatile in a sentence or two.

[Proper use of volatile is part of the bug-killing [***Embedded C Coding Standard***](https://barrgroup.com/coding-standard)*.*]

C's volatile keyword 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 of C's volatile Keyword**

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 an unsigned 16-bit integer variable to be a volatile integer:

volatile uint16\_t x;

uint16\_t volatile y;

Now, it turns out that pointers to volatile variables are very common, especially with memory-mapped I/O registers. Both of these declarations declare p\_reg to be a pointer to a volatile unsigned 8-bit integer:

volatile uint8\_t \* p\_reg;

uint8\_t volatile \* p\_reg;

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

uint16\_t \* volatile p\_x;

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

uint16\_t volatile \* volatile p\_y;

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), read Dan Sak's column "Top-Level cv-Qualifiers in Function Parameters" (Embedded Systems Programming, February 2000, p. 63).

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

**Proper Use of C's volatile Keyword**

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

1. Memory-mapped peripheral registers

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application

We'll talk about each of these cases in the sections that follow.

**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 that is memory mapped 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:

uint8\_t \* p\_reg = (uint8\_t \*) 0x1234;

// Wait for register to read non-zero

do { ... } while (0 == \*p\_reg)

This code will almost certainly fail as soon as you turn compiler optimization on.  That's because the compiler will generate assembly language (here for an 16-bit x86 processor) that looks something like this:

mov p\_reg, #0x1234

  mov a, @p\_reg

loop:

  ...

bz loop

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

uint8\_t volatile \* p\_reg = (uint8\_t volatile \*) 0x1234;

The assembly language now looks like this:

mov p\_reg, #0x1234

loop:

  ...

mov a, @p\_reg

bz loop

The desired behavior is thus achieved.

Subtler sorts of bugs tend to arise when registers with special properties are accessed without volatile declarations. For instance, a lot of peripherals contain registers that are cleared simply by reading them. Extra (or fewer) reads than you are intending could result in quite unexpected behavior in these cases.

**Interrupt Service Routines**

Interrupt service routines often set variables that are tested in mainline 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:

bool gb\_etx\_found = false;

void main()

{

...

while (!gb\_etx\_found)

{

// Wait

}

...

}

interrupt void rx\_isr(void)

{

...

if (ETX == rx\_char)

{

gb\_etx\_found = true;

}

...

}

[NOTE: We're not advocating use of global variables; this code uses one to keep the example short/clear.]

With compiler optimization turned off, this program might work. However, any half decent optimizer will "break" the program. The problem is that the compiler has no idea that gb\_etx\_found can be changed within the ISR function, which doesn't appear to be ever called.

As far as the compiler is concerned, the expression !gb\_ext\_found will have the same result every time through the loop, and therefore, you must not ever want to 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 gb\_etx\_found to be volatile. After which, this program will work as you intended.

**Multithreaded Applications**

Despite the presence of queues, pipes, and other scheduler-aware communications mechanisms in real-time operating systems, it is still possible that RTOS tasks will exchange information via a shared memory location (i.e., global storage). When you add a preemptive scheduler to your code, your compiler has no idea what a context switch is or when one might occur. Thus, a task asynchronously modifying a shared global is conceptually the same as the ISR scenario discussed above. Thus all shared global objects (variables, memory buffers, hardware registers, etc.) must also be declared volatile to prevent compiler optimization from introducing unexpected behaviors. For example, this code is asking for trouble:

uint8\_t gn\_bluetask\_runs = 0;

void red\_task (void)

{

while (4 < gn\_bluetask\_runs)

{

...

}

// Exit after 4 iterations of blue\_task.

}

void blue\_task (void)

{

  for (;;)

  {

...

gn\_bluetask\_runs++;

...

  }

}

This program will likely fail once the compiler's optimizer is enabled. Declaring gn\_bluetask\_runs with volatile is the proper way to solve the problem.

[NOTE: We're not advocating use of global variables; this code uses a global because it is explaining a relationship between volatile and global variables.]

[WARNING: Global variables shared by tasks and ISRs will also need to be protected against race conditions, e.g. by a mutex.]

**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 when you encounter unexpected program behavior. Modern C/C++ optimizers are so good that I cannot remember the last time I came across an optimization bug. In contrast, I regularly come across failures by programmers to use volatile.

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.