Combining C's volatile and const keywords

Tags: Industry

Does it ever make sense to declare a variable in C or C++ as both volatile (in other words, "ever-changing") and const ("read-only")? If so, why? And how should you combine `volatile` and `const` properly?

One of the most consistently popular articles on the Netrino website is about C's volatile keyword. The volatile keyword, like const, is a type qualifier. These keywords can be used by themselves, as they often are, or together in variable declarations.

I've written about volatile and const individually before. If you haven't previously used the volatile keyword, I recommend you read "How to Use C's volatile Keyword" on Netrino.com before going on. As that article makes plain:

> *"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."*

How to use C's volatile keyword

By declaring a variable volatile, you're effectively asking the compiler to be as inefficient as possible when it comes to reading or writing that variable. Specifically, the compiler should generate object code to perform each and

every read from a volatile variable as well as each and every write to a volatile variable–even if you write it twice in a row or read it and ignore the result. Not a single read or write can be skipped. In other words, no compiler optimizations are allowed with respect to volatile variables.

The use of volatile variables may also create additional sequence points within the functions that access them. In a nutshell, the order of accesses of volatile variables A and B in the object code must be the same as the order of those accesses in the source code. The compiler is not allowed to reorder volatile variable accesses for any reason. (Consider what might go wrong if the referenced memory locations were hardware registers.)

Here are two examples of declarations of volatile variables:

```
int volatile g_shared_with_isr;
```

```
uint8_t volatile * p_led_reg = (uint8_t *) 0x80000;
```

The first example declares a global flag that can be shared between an ISR and some other part of the code (such as a background processing loop in `main()` or a task on top of an RTOS) without fear that the compiler will optimize (in other words, "delete") the code you write to check for asynchronous changes to the flag's value. It's important to use volatile to declare all variables that are shared by asynchronous software entities, which is important in any kind of multithreaded programming. (Remember, though, that access to global variables shared by tasks or with an ISR must always also be protected via a mutex or interrupt disable, respectively.)

The second example declares a pointer to a hardware register at a known physical memory address (80000h)–in this case to manipulate the state of one or more LEDs. Because the pointer to the hardware register is declared

volatile, the compiler must always perform each individual write. Even if you write C code to turn an LED on followed immediately by code to turn the same LED off, you can trust that the hardware really will receive both instructions. Because of the sequence point restrictions, you are also guaranteed that the LED will be off after both lines of the C code have been executed. The volatile keyword should always be used with creating pointers to memory-mapped I/O such as this.

[See my Coding Standard Rule #4: Use **volatile** whenever possible for best practice recommendations on the use of volatile by itself.]

How to use C's const keyword

The const keyword is can be used to modify parameters, as well as in variable declarations. Here we are only interested in the use of const as a type qualifier in variable declarations, as in:

```
uint16_t const max_temp_in_c = 1000;
```

This declaration creates a 16-bit unsigned integer value of 1,000 with a scoped name of `max_temp_in_c`. In C, this variable will exist in memory at run time, but will typically be located, by the linker, in a non-volatile memory area such as ROM or flash. Any reference to the const variable will result in a read from that location. (In C++, a const integer may no longer exist as an addressable location in run-time memory.)

Any attempt the code makes to write to a const variable directly (i.e., by its name) will result in a compile-time error. To the extent that the const variable is located in ROM or flash, an indirect write (i.e., via a pointer to its address) will also be thwarted–though at run time, obviously.

Another use of const is to mark a hardware register as read-only. For example:

```
uint8_t const * p_latch_reg = 0x10000000;
```

Declaring the pointer this way, any attempt to write to that physical memory address via the pointer (e.g., `*p_latch_reg = 0xFF;`) should result in a compile-time error.

[See my Coding Standard Rule #2: Use **const** whenever possible for best practice recommendations on the use of const by itself.]

How to use const and volatile together
Though the essence of the volatile ("ever-changing") and const ("read-only") keywords may seem at first glance opposed, they are in fact orthogonal. Thus sometimes it makes sense to use both keywords in the declaration of a single variable or pointer. Typical use scenarios for both involve either pointers to memory-mapped hardware registers or variables located in shared memory areas.

(#1) Constant addresses of hardware registers
The following declaration uses both const and volatile in the frequently useful scenario of declaring a constant pointer to a volatile hardware register.

```
uint8_t volatile * const p_led_reg = (uint8_t *) 0x80000;
```

The proper way to read a complex declaration like this is from the name of the variable back to the left, as in:

"`p_led_reg` IS A constant pointer TO A volatile 8-bit unsigned integer."

Reading it that way, we can see that the keyword const modifies only the pointer (i.e., the fixed address 80000h), which should obviously not change at run time. Whereas the keyword volatile modifies only the type of integer. This

style of declaration is actually quite useful and is a much safer version of the volatile-only declaration of `p_led_reg` that appears earlier in this article.

In this example, adding const means that the common typo of a missed pointer dereference (i.e., '`*` ') can be caught at compile time. That is, the mistaken code

```
p_led_reg = LED1_ON;
```

won't overwrite the address with the non-80000h value of `LED1_ON` . The compiler error leads us to correct this to:

```
*p_led_reg = LED1_ON;
```

which is almost certainly what we meant to write in the first place.

(#2) Read-only shared-memory buffer

Another use for a combination of const and volatile is where you have two or more processors communicating via a shared memory area and you're coding the side of this communications that will only be reading from a shared memory buffer. In this case you could declare variables such as:

```
int const volatile comm_flag;uint8_t const volatile comm_buffer[
```

Of course, you'd usually want to instruct the linker to place these global variables at the correct addresses in the shared memory area or to declare the above as pointers to specific physical memory addresses. In the case of pointers, the use of const and volatile may become even more complex, as in the next category.

(#3) Read-only hardware register

Sometimes you will run across a read-only hardware register. In addition to enforcing compile-time checking so that the software doesn't try to overwrite the memory location, you also need to be sure that each and every requested read actually occurs. By declaring your variable IS A (constant) pointer TO A constant and volatile memory location you request all of the appropriate protections, as in:

```
uint8_t const volatile * const p_latch_reg = (uint8_t *) 0x10000
```

As you can see, the declarations of variables that involve both the volatile and const decorators can quickly become complicated to read. However, the technique of combining C's volatile and const keywords can be useful and even important in the above scenarios.

The proper use of volatile and const and the combination of the two keywords is definitely something you should learn along you path to becoming a master embedded software engineer.

Michael Barr *is a former lecturer at University of Maryland and Johns Hopkins University and the author of three books and more than sixty-five articles and papers about embedded systems design. In addition to his work designing embedded systems ranging from medical devices to consumer electronics, Mr. Barr has been admitted as a testifying expert witness in numerous U.S. and Canadian court cases covering issues of patent infringement and validity, software quality, theft of copyrighted source code, and satellite TV piracy. Mr. Barr holds BSEE and MSEE degrees. Contact him via e-mail at mbarr@netrino.com or read his blog at http://embeddedgurus.com/barr-code.*

Additional reading on volatile:

“Introduction to the Volatile Keyword” Nigel Jones (ESP magazine, 2001)

“Place volatile accurately” Dan Saks, 11/16/2005

More by Michael Barr