Do You Volatile? Should You?

By Kevin P. Dankwardt, Ph.D.

Volatile is an ANSI C type modifier that is frequently needed in C code that is part of signal/interrupt handlers, threaded code, and other kernel code, including device drivers. In general, any data that may be undated asynchronously should be declared to be volatile. Incidentally, this issue is not related to CPU caches except that re-loading of variables into registers may involve cache hits or misses.

Why Use Volatile?

The reason to use volatile is to insure that the compiler generates code to re-load a data item each time it is referenced in your program. Without volatile, the compiler may generate code that merely re-uses the value it already loaded into a register.

Volatile advises the compiler that the data may be modified in a manner that may not be determinable by the compiler. This could be, for example, when a pointer is mapped to a device's hardware registers. The device may independently change the values unbeknownst to the compiler.

With *gcc* the *-O2* option is normally required to see the effect of not using volatile. Without *-O2* or greater optimization, the compiler is likely to re-load registers each time a variable is referenced, anyway. Don't blame the optimizer if a program gets incorrect results because the program does not use volatile where required.

For example, if two threads share a variable, *sum*, and one or both threads modify it, then the other thread may use a stale value in a register instead of going back to memory to get the new value. Instead, each time the thread references sum, it must be re-loaded. The way to insure this occurs in ANSI C is to declare sum to be volatile.

Example 1.

The use of volatile can be required to get correct answers. For example the program wrong will give incorrect results when it is compiled *-O2* and without *volatile*. This slightly obtuse program is designed to stop after 100 ticks of an interval timer that ticks at 100Hz and print the value of the variable *total*. The tick count is incremented in the signal handler. When the count gets to 100, the program should terminate. If the tick count does not get to 100 within 10 seconds then an alarm goes off and the program terminates.

By compiling the program as: gcc -O2 -DVOLATILE=volatile wrong.c -o wrong_v you will see, (unless your program is preempted for quite a while), that the count gets to 100 and the program terminates as designed. With the program compiled as gcc -O2 wrong.c -o wrong_nv you will see, that the count becomes greater than 100 as shown when the handler prints it, but, the while loop does not terminate.

Incidentally, attempts to determine what is happening may thwart your efforts. For example, a function call, such as to printf(), or the use of a breakpoint, in the loop, will likely spill and re-load the registers.

Syntax

The keyword volatile is similar to the *const* keyword. Volatile is used to modify a type. Thus an int, const int, pointer, etc. may be declared to be volatile. In addition, a point may be declared to be a *pointer to volatile*. A pointer to volatile means that the data to which the pointer refers is

volatile as opposed to the pointer itself. Of course, both the pointer and to which it refers, may be declared to be volatile.

To declare a volatile int do:

```
volatile int v;
```

and to declare vp to be a pointer to a volatile int do:

```
volatile int *vp;
```

Since deciphering C declarations can be difficult you may want to consult the C declaration chapter in the Sun manual. This manual references the Decoder Flowchart that can be used to help decipher declarations.

In addition, Linux may have the *cdecl(1)* program that can be used to translate C declarations to English, as for example, in

```
echo 'explain volatile int *v' | cdecl
```

which will answer with

```
declare v as a pointer to volatile int
```

Reading C declarations is made simpler when you realize that they are written boustrophedonically. Of course, even knowing the definition of *boustrophedonically* doesn't really help. The idea is that C declarations are interpreted based on the tricky precedence of operators such as "*", "[]", and "()".

Performance Issues

In some sense, *volatile* is the opposite of *register*. Thus, one can expect to lose performance. This means don't use volatile when it is not needed.

Example 2.

In our performance example we can see the difference that *volatile* may make. If we compile this program with and without *VOLATILE* defined as *volatile* we see an average number of iterations of almost 5,000 for the *volatile* case and almost 20,000 for the *non-volatile* case. Yikes! Remember that we must compile both of them with the *-O2* option. (These iteration counts were made on a 400Mhz AMD-K6.)

Linux Examples

The use of the *volatile* keyword is common in the Linux kernel source. For example, of the 10,607 *.c* and *.h* files in the Fedora Core 1, Linux kernel source directory, 1,694 have the string "volatile" in them somewhere. As an example, the file drivers/net/eepro.c uses *volatile* in three places.

```
385: volatile s32 cmd_status; /* All command and status fields.
*/
392: volatile s32 status;
764: volatile s32 *self_test_results;
```

Generated Code

By examining the code generated by the compiler one can see the difference *volatile* makes. In

this simple example we can see the x86 assembly language when volatile is used and when volatile is not used.

Quiz Yourself

What is *volatile* in each of the following examples? Are they all legal declarations?

```
1) volatile int *s1;
2) int* volatile s2;
3) volatile int* volatile s3;
4) const volatile int * volatile s4;
5) volatile int * (*f)(volatile int *);
```

Check your answers.

Summary

The *volatile* keyword is relatively unknown. There are times when its use is required for correct operation of C/C++ programs. In general, whenever a variable may be altered asynchronously, such as by a signal handler or mapped hardware, the variable must be declared to be volatile.

Since volatile prevents re-using values in registers, volatile comes with a performance penalty that can be substantial.

Also, since declarations involving *volatile* can be difficult to decipher you may want to use cdecl(1).