

Volatile Demystified

In C programming, the use of the key word 'volatile' is intended to prevent the compiler from applying any optimisations on objects that can be changed in ways that the compiler cannot determine. In this article, the author demystifies the use of the volatile key word.



Volatile is a qualifier in C, which is applied to a variable when it is declared. It is used extensively while writing programs for embedded systems, especially when dealing with hardware. The aim of this article is to give you an idea of how to use a Volatile qualifier. So, what are its instructions to the compiler? It tells the compiler that the value of the variable may change at any time during the execution of the code without the knowledge of the compiler. If proper precautions are not taken, the desired output may not be achieved. A variable should be declared volatile whenever its value may change unexpectedly.

Volatile variables are, therefore, variables that can be changed at any time by other external programs or by the same program.

The syntax for declaring the variable as 'volatile' is:

```
volatile dataType variable;
```

Let us understand the 'volatile' key word thoroughly through the following illustrations.



Note : Remember that all codes are compiled in the gcc compiler version: gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

Illustration 1: Polling mechanism

Let us consider the following small and simple code snippet to study the behaviour of the 'volatile' key word in C:

```

1 int main()
2 {
3     int i = 0;
4     int flag = 0;
5
6     //Some code
7
8     while(flag != 1)
9     {
10         //Keep polling until flag becomes 1
11         ;
12     }
13
14     //rest of the code
15     return 0;
16 }

```

Here, the intention of the programmer is to keep polling inside the *while* loop until the flag value is SET to the value 1, which might be done by a hardware/peripheral device. However, during the compilation phase, the compiler will find

```
Elegance
[satya]
ls -l while_loop_without_volatile.s
-rw-rw-r-- 1 satya satya 482 Oct 24 12:27 while_loop_without_volatile.s
[satya]
```

Figure 1: Size of assembly code generated by the compiler without qualifying the variable as volatile

```
Elegance
[satya]
ls -l while_loop_with_volatile.s
-rw-rw-r-- 1 satya satya 501 Oct 24 14:20 while_loop_with_volatile.s
[satya]
```

Figure 2: Size of assembly code generated by the compiler with the variable qualified as volatile

that this piece of code is not achieving any valuable results; hence, the code will be optimised by removing this.

If one observes the code that follows below, the condition inside the *while* loop is replaced by the compiler to *while* (TRUE). This is primarily done in compilers in the embedded systems environment, where generating optimal machine code is very important. As a programmer, if you are not aware of this, it could result in unexpected behaviour at runtime.

```
1 int main()
2 {
3     int i = 0;
4     int flag = 0;
5
6     //Some code
7
8     while(TRUE)
9     {
10         //Infinite loop
11         ;
12     }
13
14     //rest of the code
15     return 0;
16 }
```

Now, the question is, “How can one confirm that the compiler is really optimising the code?” Let us check the size of the assembly code (call this *while_without_volatile.s*) generated by the compiler using the steps given below; the size is shown in Figure 1.

How to generate assembly code from the C source code in the gcc compiler

Step 1: Compile the C source code with the *-save-temps* option as shown below:

```
gcc -o sample sample.c -save-temps
```

Step 2: Check out the *sample.s* file generated by issuing the *ls* command.

When we compile code with the *-save-temps* option of

gcc, it generates three output files:

- Pre-processed code (with the *.i* extension)
- Assembly code (with the *.s* extension)
- Object code (with the *.o* option)

Now, if you observe Figure 1, the size is found to be 482 bytes in the fifth column. Next, qualify the flag variable to ‘volatile’ for the code shown in Illustration 1, and generate the assembly code (call this *while_with_volatile.s*) before checking the size by issuing the *ls* command. The size obtained in my system is shown in Figure 2.

Now, if you observe Figure 2, the size is found to be 501 bytes in the fifth column. So, when we compare the sizes of both the codes, with and without the ‘volatile’ key word, it is obvious that the compiler is not optimising the ‘flag’ variable when it is qualified as ‘volatile’.

Let us experiment further to explore where exactly the compiler is optimising the code. To find this out, apply the *vimdiff* command to the assembly codes generated with and without the keyword ‘volatile’—the difference is shown in Figure 3.

How to apply the vimdiff command

First, type:

```
vimdiff while_without_volatile.s while_with_volatile.s
```

In the disassembly of the non-volatile version (*while_loop_without_volatile.s*) of the *while* loop shown in Figure 3, the statements in lines 14 and 15 load the value of the flag into memory locations [-8(%ebp) & -4(%ebp)] outside the loop labelled *.L2*. This is because, since the flag variable is not declared volatile, the compiler assumes that its value cannot be modified outside the program. Having already read the value of the flag into memory locations [-8(%ebp) & -4(%ebp)], the compiler omits reloading the value of the flag variable when optimisation is enabled, because its value cannot change. The result is ultimately the control getting into

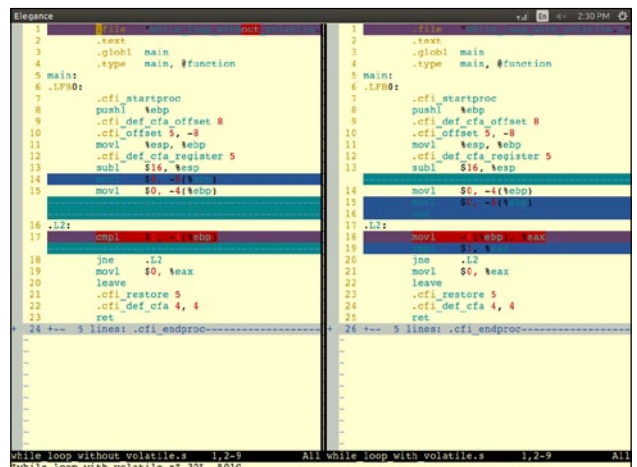


Figure 3: Difference between the assembly codes generated with and without the ‘volatile’ key word

```

satya@Elegance: ~/Blog/ForExample
└─$ ls -l for_without_volatile.s
-rw-rw-r-- 1 satya satya 490 Oct 24 18:26 for_without_volatile.s
└─$

satya@Elegance: ~/Blog/ForExample
└─$ ls -l for_with_volatile.s
-rw-rw-r-- 1 satya satya 542 Oct 24 18:26 for_with_volatile.s
└─$

```

Figure 4: Size of assembly codes generated by the compiler with and without volatile qualifier

the infinite loop labelled *.L2*.

In contrast, in the disassembly of the volatile version (*while_loop_with_volatile.s*) of the *while* loop shown in Figure 3, the compiler assumes that the value of the flag variable can change outside the program and performs no optimisations. Consequently, the value of the flag is loaded into the register *%eax* every time from the memory *[-8(%ebp)]* inside the loop labelled *.L2*. As a result, the value of the flag is checked every time, and further decisions are taken depending upon the value of the flag variable.

To avoid optimisation problems caused by changes to the program state external to the implementation, it is always safer to declare the variable as 'volatile'. This helps to avoid unexpected results. From Figure 3, we can conclude that the 'volatile' key word prevents optimisation of the code by the compiler.

Illustration 2: Delay generations using loops

Let us consider another example, where 'for' loops are used commonly in the Embedded C code to generate small delays as shown in the following code:

```

1 int main()
2 {
3     int i;
4
5     //Some code goes here
6
7     //Loop for delay generation
8     for( i = 0; i < 100; i++)
9     {
10         ;
11     }
12
13     //Again the remaining code goes here
14
15     return 0;
16 }

```

In fact, a compiler might optimise the code shown above into nothing. A local variable 'i' is the counter for a loop that does nothing but increment value 'i' until it's equal to 100. Thus, the optimiser can replace the loop with a single assignment that just sets 'i' to its final value. When that happens, the delay code doesn't achieve what the programmer had intended. So, it is

always better to declare the local variable 'i' as 'volatile' even though the code might be less efficient, since we will get the desired results, as shown in the code below:

```

1 int main()
2 {
3     volatile int i;
4
5     //Some code goes here
6
7     //Loop for delay generation
8     for( i = 0; i < 100; i++)
9     {
10         ;
11     }
12
13     //Again the remaining code goes here
14
15     return 0;
16 }

```

Let us generate the assembly codes for the examples given in the last two code snippets above, using the commands given in 'How to generate assembly code from the C source code in the gcc compiler', and then get the size of the assembly codes using the *ls* command as given in Figure 4.

Comparing the sizes in Figure 4, one can conclude that the compiler is applying the optimisation techniques without the volatile qualifier. In other words, the compiler is not allowed to reorder the instructions on volatile variables with respect to other memory operations. The disassembly code, both with and without the 'volatile' key word, for Illustration 2 is shown in Figure 5.

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

Let us consider one more example to show how the global variable will be affected by the compiler optimisation in a multi-threaded application. The example code snippet is shown below:

```

1 #define FALSE 0
2 #define TRUE 1
3
4 volatile unsigned int global_item_count;
5
6 //Other functions
7 void thread_one(void)
8 {
9     global_item_count = FALSE;
10    while(global_item_count == FALSE)
11    {
12        sleep(1);
13    }

```

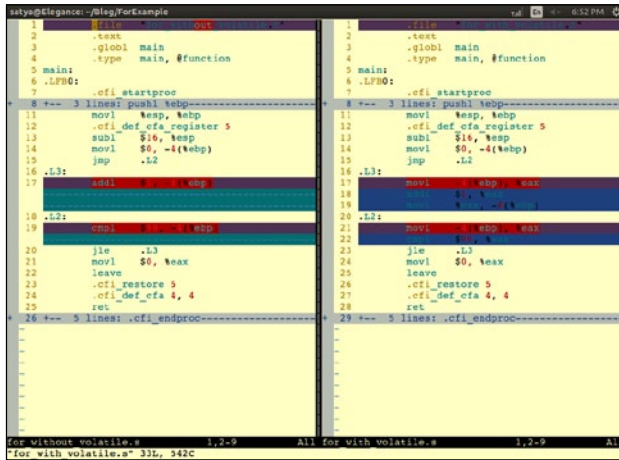


Figure 5: Differences between the assembly codes generated with and without the 'volatile' keyword

```

14     //Some code goes here
15 }
16
17 void thread_two(void)
18 {
19     //some code goes here
20
21     global_item_count++;
22     sleep(5);
23
24     //some code goes here
25
26 }
    
```

In the above demo program, the compiler doesn't have any knowledge of the context switching between two threads. If the compiler optimisations are turned 'ON', then the compiler will assume that the *global_item_count* variable is always 'ZERO' and no other part of the thread is attempting to modify it. So, the compiler may replace the *while* loop in the code above, as shown in the code below:

```

.
.
.
while(TRUE)
{
    sleep(1);
}
.
.
.
    
```

... which is nothing but the infinite loop; so in order to avoid such optimisations by the compiler, it is safe to declare the variable *global_item_count* as 'volatile'. Similarly, one can realise the effect of the producer-consumer problem accessing the global variable without declaring it as 'volatile'.

Illustration 4: Interrupt service routines

Let us consider another example given in the code snippet below, where 'volatile' plays a very important role in the ISR (Interrupt Service Routines):

```

1 int flag = 0;
2 void rx_isr(void)
3 {
4     flag = 1;
5 }
6 int main()
7 {
8     // ...
9     while(!flag)
10    {
11        //Some code goes here
12    }
13    // ...
14 }
    
```


In the above example, if the flag is not declared as 'volatile', the compiler may optimise the code (assuming always that the flag is ZERO) and replace the *while(!flag)* to *while(TRUE)*, which is nothing but the infinite loop. But the flag value might change when the interrupt occurs.



Note : Whether to declare the variable as 'volatile' or not is cross-compiler dependent. Anyhow it is a good practice to declare the variable as 'volatile' to achieve the portability of the code.

A variable should be declared volatile whenever its value can change asynchronously. In real time, three types of variables can change:

- Memory-mapped peripheral registers (e.g., polling and waiting)
- Global variables modified by an Interrupt Service Routine
- Global variables accessed by multiple tasks within a multi-threaded application

The main use of the 'volatile' key word is to prevent the compiler from optimising the code in terms of time complexity, by generating a code that uses CPU registers as faster ways to represent variables. Declaring the variable as 'volatile' forces compiled code to access the exact memory location in RAM on every access to the variable to get its latest value, thereby avoiding any runtime surprises for the programmer. 

By: Satyanarayana Sampangi

The author is a member of the embedded software team at Emertxe Information Technology (P) Ltd (<http://www.emertxe.com>). His areas of interest are embedded C programming combined with data structures and microcontrollers. He can be reached at satya@emertxe.com