Introduction to the Volatile Keyword

认识关键字 Volatile

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.
很多程序员对于 volatile 的用法都不是很熟悉。这并不奇怪，很多介绍 C 语言的书籍对于他的用法都闪烁其辞。

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

在你们使用 C/C++语言开发嵌入式系统的时候，遇到过以下的情况么？

- 一打开编译器的编译优化选项，代码就不再正常工作了；
- 中断似乎总是程序异常的元凶；
- 硬件驱动工作不稳定；
- 多任务系统中，单个任务工作正常，加入任何其他任务以后，系统就崩溃了。

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.
如果你曾经向别人请教过和以上类似的问题，至少说明，你还没有接触过 **C** 语言关键字 **volatile** 的用法。这种情况，你不是第一个遇到。很多程序员对于 **volatile** 都几乎一无所知。大部分介绍 **C** 语言的文献对于它都闪烁其辞。

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.
Volatile 是一个变量声明限定词。它告诉编译器，它所修饰的变量的值可能会在任何时刻被意外的更新，即便与该变量相关的上下文没有任何对其进行修改的语句。造成这种“意外更新”的原因相当复杂。在我们分析这些原因之前，我们先回顾一下与其相关的语法。

Syntax
语法

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:
要想给一个变量加上 **volatile** 限定，只需要在变量类型声明附之前/后加入一个 **volatile** 关键字就可以了。下面的两个实例是等效的，它们都是将 foo 声明为一个“需要被实时更新”的 int 型变量。

```
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:
同样，声明一个指向 volatile 型变量的指针也是非常类似的。下面的两个声明都是将 foo 定义为一个指向 volatile integer 型变量的指针。

```
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:
一个 Volatile 型的指针指向一个非 volatile 型变量的情况非常少见(我想，我可能使用过一次)，尽管如此，我还是要给出他的语法：

```
int * volatile foo;
```

And just for completeness, if you really must have a volatile pointer to a volatile variable, then:
最后一种形式，针对你真的需要一个 volatile 型的指针指向一个 volatile 型的情形：

```
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).
顺便说一下，如果你想知道关于“我们需要在什么时候在什么地方使用 volatile”和“为什么我们需要 volatile 放在变量类型后面（例如，**int volatile * foo**）”这类问题的详细内容，请参考 Dan Sak`s 的专题，“Top-Level cv-Qualifiers in Function Parameters”。

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.

最后，如果你将 volatile 应用在结构体或者是公用体上，那么该结构体/公用体内的所有内容就都带有 volatile 属性了。如果你并不想这样（牵一发而动全身），你可以仅仅在结构体/公用体中的某一个成员上单独使用该限定。

Use
使用

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

当一个变量的内容可能会被意想不到的更新时，一定要使用 volatile 来声明该变量。通常，只有三种类型的变量会发生这种“意外”：

- 在内存中进行地址映射的设备寄存器；
- 在中断处理程序中可能被修改的全局变量；
- 多线程应用程序中的全局变量；

Peripheral registers
设备寄存器

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 nave and incorrect implementation is as follows:
嵌入式系统的硬件实体中，通常包含一些复杂的外围设备。这些设备中包含的寄存器，其值往往随着程序的流程同步的进行改变。在一个非常简单的例子中，假设我们有一个 8 位的状态寄存器映射在地址 0x1234 上。系统需要我们一直监测状态寄存器的值，直到它的值不为 0 为止。通常错误的实现方法是：

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

// Wait for register to become non-zero.等待寄存器为非 0 值
while (*ptr == 0);
// Do something else.作其他事情
```

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:
一旦你打开了优化选项，这种写法肯定会失败，编译器就会生成类似如下的汇编代码：

```
mov    ptr, #0x1234    mov    a, @ptr 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), 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:
优化的工作原理非常简单：一旦我们我们将一个变量读入寄存器中（参照代码的第二行），如果（从变量相关的上下文看）变量的值总是不变的，那么就没有必要（从内存中）从新读取他。在代码的第三行中，我们使用一个无限循环来结束。为了强迫编译器按照我们的意愿进行编译，我们修改指针的声明为：

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

The assembly language now looks like this:
对应的汇编代码为：

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

The desired behavior is achieved.
我们需要的功能实现了！

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.
对于一些较为特殊的寄存器，（我们上面提到的方法）会导致一些难以想象的错误。事实上，很多设备寄存器在读取一次以后就会被清除。这种情况下，多余的读取操作会导致意想不到的错误。

Interrupt service routines
中断处理程序

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:
中断处理程序经常负责更新一些在主程序中被查询的变量的值。例如，一个串行通讯中断会检测接收到的每一个字节是否为 ETX 信号（以便来确认一个消息帧的结束标志）。如果其中的一个字节为 ETX，中断处理程序就是修改一个全局标志。一个错误的实现方法可能为：

```
int etx_rcvd = FALSE;

void main()
{
   ...
```

```
   while (!ext_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 **!ext_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."
在编译优化选项关闭的时候，代码可能会工作的很好。但是，即便是任何半吊子的优化，也会“破坏”这个代码的意图。问题就在于，编译器并不知道 etx_rcvd 会在中断处理程序中被更新。在编译器可以检测的上下文内，表达式!ext_rcvd 总是为真，所以，你就永远无法从循环中跳出。因此，该循环后面的代码会被当作“不可达到 ”的内容而被编译器的优化选项简单的删除掉。如果你比较幸运，你的编译器也许会给你一个相关的警告；如果你没有那么幸运（或者你没有注意到这些警告），你的代码就会导致严重的错误。通常，就会有人抱怨“该死的优化选项”。

The solution is to declare the variable **etx_rcvd** to be **volatile**. Then all of your problems (well, some of them anyway) will disappear.
解决这个问题的方法很简单：将变量 etx_rcvd 声明为 volatile。然后，所有的（至少是一部分症状）那些错误症状就会消失。

Multi-threaded applications
多线程应用程序

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:
在实时操作系统中，除去队列、管道以及其他调度相关的通讯结构，在两个任务之间采用共享的内存空间（就是全局共享）实现数据的交换仍然是相当常见的方法。当你将一个优先权调度器应用于你的代码时，编译器仍然不知道某一程序段分支选择的实际工作方式以及什么时候某一分支情况会发生。这是因为，另外一个任务修改一个共享的全局变量在概念上通常和前面中断处理程序中提到的情形是一样的。所以，（这种情况下）所有共享的全局变量都要被声明为 volatile。例如：

```
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.
一旦编译器的优化选项被打开，这段代码的执行通常会失败。将 cntr 声明为 volatile 是解决问题的好办法。

Final thoughts

反思

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.
一些编译器允许我们隐含的声明所有的变量为 volatile。最好抵制这种便利的诱惑，因为它很容易让我们“不动脑子”，而且，这也常常会产生一个效率相对较低的代码。

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.
所以，我们又诅咒编译优化或者简单的关掉这一选项来抵制这些诱惑。现在的编译优化已经相当聪明，我不记得在编译优化中找到过什么错误。与之相比，为了解决一些错误，我却常常使用疯狂数量的 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.
如果你恰巧有一段代码需要去修正，先搜索一下有没有 volatile 关键字。如果找不到 volatile，那么这个代码很可能会是一个很好的实例来检测前面提到过的各种错误。

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 NAJones@compuserve.com.
Nigel Jones 在马里兰从事顾问工作。除了为各类嵌入式项目开发充当顾问，他平时的一大爱好就是潜水。你可以通过发送邮件到 NAJones@compuserve.com 与其取得联系。