volatile的重要性对于搞嵌入式的程序员来说是不言而喻的，对于volatile的了解程度常常被不少公司在招聘嵌入式编程人员面试的时候作为衡量一个应聘者是否合格的参考标准之一，为什么volatile如此的重要呢？这是因为嵌入式的编程人员要经常同中断、底层硬件等打交道，而这些都用到volatile，所以说嵌入式程序员必须要掌握好volatile的使用。
其实就象读者所熟悉的const一样，volatile是一个类型修饰符。在开始讲解volatile之前我们先来讲解下接下来要用到的一个函数，知道如何使用该函数的读者可以跳过该函数的讲解部分。
原型：int gettimeofday (struct timeval * tv , struct timezone * tz);
头文件：#include <sys/time.h>
功能：获取当前时间
返回值：如果成功返回0，失败返回 - 1，错误代码存于errno中。
gettimeofday()会把目前的时间用tv所指的结构返回，当地时区的信息则放到tz所指的结构中。

[cpp] view plaincopy

```
1. timeval结构定义为：
2. struct timeval{
3.     long tv_sec;
4.     long tv_usec;
5. };
6. timezone 结构定义为：
7. struct timezone{
8.     int tz_minuteswest;
9.     int tz_dsttime;
10. };
```

先来说说timeval结构体，其中的tv_sec存放的是秒，而tv_usec存放的是微秒。其中的timezone成员变量我们很少使用，在此简单的说说它在gettimeofday()函数中的作用是把当地时区的信息则放到tz所指的结构中，在其中tz_minuteswest变量里存放的是和Greenwich时间差了多少分钟，tz_dsttime日光节约时间的状态。我们在此主要的是关注前一个成员变量timeval，后一个我们在此不使用，所以使用gettimeofday()函数的时候我们把有一个参数设定为NULL，下面先来看看一段简单的代码。

[cpp] view plaincopy

```
1. #include <stdio.h>
2. #include <sys/time.h>
3.
4. int main(int argc, char * argv[])
```

```
5. {
6.     struct timeval start,end;
7.     gettimeofday( &start, NULL );  /*测试起始时间*/
8.     double timeuse;
9.     int j;
10.     for(j=0;j<1000000;j++)
11.         ;
12.     gettimeofday( &end, NULL );    /*测试终止时间*/
13.
    timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_sec - start
.tv_sec ;
14.     timeuse /= 1000000;
15. printf("运行时间为：%f\n",timeuse);
16.
17.     return 0;
18.
19. }
```

运行结果为：

[cpp] view plaincopy

```
1. root@ubuntu:/home# ./p
2. 运行时间为：0.002736
```

现在来简单的分析下代码，通过end.tv_sec - start.tv_sec 我们得到了终止时间跟起始时间以秒为单位的时间间隔，然后使用end.tv_sec - start.tv_sec 得到终止时间跟起始时间以微妙为单位的时间间隔。因为时间单位的原因，所以我们在此对于(end.tv_sec - start.tv_sec) 得到的结果乘以1000000转换为微秒进行计算，之后再使用timeuse /= 1000000;将其转换为秒。现在了解了如何通过gettimeofday()函数来测试start到end代码之间的运行时间，那么我们现在接下来看看volatile修饰符。

通常在代码中我们为了防止一个变量在意想不到的情况下被改变，我们会将变量定义为volatile，这从而就使得编译器就不会自作主张的去“动”这个变量的值了。准确点说就是每次在用到这个变量时必须每次都重新从内存中直接读取这个变量的值，而不是使用保存在寄存器里的备份。

在举例之前我们先大概的说下Debug和Release 模式下编译方式的区别，Debug 通常称为调试版本，它包含调试信息，并且不作任何优化，便于程序员调试程序。Release 称为发布版本，它往往是进行了各种优化，使得程序在代码大小和运行速度上都是最优的，以便用户很好地使用。大致的知道了Debug和Release的区别之后，我们下面来看看一段代码。

[cpp] view plaincopy

```
1. #include <stdio.h>
2.
3. void main()
4. {
5. int a=12;
6. printf("a的值为:%d\n",a);
7. __asm {mov dword ptr [ebp-4], 0h}
8. int b = a;
9. printf("b的值为:%d\n",b);
10. }
```

先分析下上面的代码，我们使用了一句__asm {mov dword ptr [ebp-4], 0h}来修改变量a在内存中的值，如果有对这句代码功能不清楚的读者可以参考我之前的一篇《C语言的那些小秘密之堆栈》，在此就不做过多的讲解了。前面已经讲解了Debug和Release 编译方式的区别，那么我们现在来对比看下结果。注：使用vc6编译运行，如无特殊说明，均在linux环境下编译运行。读者自己在编译的时候别忘了选择编译运行的模式。

使用Debug模式的结果为：

[cpp] view plaincopy

```
1. a的值为:12
2. b的值为:0
3. Press any key to continue
```

使用Release模式的结果为：

[cpp] view plaincopy

```
1. a的值为:12
2. b的值为:12
3. Press any key to continue
```

看看上面的运行结果我们发现在Release模式进行了优化之后b的值为了12，但是使用Debug模式的时候b的值为0。为什么会出现这样的情况呢？我们先不说答案，再来看看下面一段代码。注：使用vc6编译运行

[cpp] view plaincopy

```
1. #include <stdio.h>
2.
3. void main()
4. {
5. int volatile a=12;
6. printf("a的值为:%d\n",a);
```

```
7. __asm {mov dword ptr [ebp-4], 0h}
8. int b = a;
9. printf("b的值为:%d\n",b);
10. }
```

使用Debug模式的结果为：

[cpp] view plaincopy

```
1. a的值为:12
2. b的值为:0
3. Press any key to continue
```

使用Release模式的结果为：

[cpp] view plaincopy

```
1. a的值为:12
2. b的值为:0
3. Press any key to continue
```

我们发现这种情况下不管使用Debug模式还是Release模式都是一样的结果。现在我们就来分析下，在此之前我们先说了Debug和Release 模式下编译方式的区别。

先分析上一段代码，由于在Debug模式下我们并没有对代码进行优化，所以对于在代码中每次使用a值得时候都是从它的内存地址直接读取的，所以在我们使用了__asm {mov dword ptr [ebp-4], 0h}语句改变了a的值之后，接下来使用a值的时候从内存中直接读取，所以得到的是更新后的a值；但是当我们在Release模式下运行的时候，发现b的值为a之前的值，而不是我们更新后的a值，这是由于编译器在优化的过程中做了优化处理。编译器发现在对a赋值之后没有再次改变a的值，所以编译器把a的值备份在了一个寄存器中，在之后的操作中我们再次使用a值的时候就直接操作这个寄存器，而不去读取a的内存地址，因为读取寄存器的速度要快于直接读取内存的速度。这就使得了读到的a值为之前的12。而不是更新后的0。

第二段代码中我们使用了一个volatile修饰符，这种情况下不管在什么模式下都得到的是更新后的a的值，因为volatile修饰符的作用就是告诉编译器不要对它所修饰的变量进行任何的优化，每次取值都要直接从内存地址得到。从这儿我们可以看出，对于我们代码中的那些易变量，我们最好使用volatile修饰，以此来得到每次对其进行更新后的值。为了加深下大家的印象我们再来看看下面一段代码。

[cpp] view plaincopy

```
1. #include <stdio.h>
2. #include <sys/time.h>
```

```
3.
4. int main(int argc, char * argv[])
5. {
6.     struct timeval start,end;
7.     gettimeofday( &start, NULL );  /*测试起始时间*/
8.     double timeuse;
9.     int j;
10.     for(j=0;j<10000000;j++)
11.         ;
12.     gettimeofday( &end, NULL );    /*测试终止时间*/
13.     timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
start.tv_usec;
14.     timeuse /= 1000000;
15. printf("运行时间为：%f\n",timeuse);
16.
17.     return 0;
18.
19. }
```

与之前我们测试时间的代码一样，我们只是增大了for()循环的次数。
先来看看我们不使用优化的结果：

[cpp] view plaincopy

```
1. root@ubuntu:/home# gcc time.c -o p
2. root@ubuntu:/home# ./p
3. 运行时间为：0.028260
```

使用了优化的运行结果：

[cpp] view plaincopy

```
1. root@ubuntu:/home# gcc -o p time.c -O2
2. root@ubuntu:/home# ./p
3. 运行时间为：0.000001
```

从结果显然可以看出差距如此之大，但是如果我们在上面的代码中修改一下int j为int volatile j之后再来看看如下代码：

[cpp] view plaincopy

```
1. #include <stdio.h>
2. #include <sys/time.h>
3.
```

```
4. int main(int argc, char * argv[])
5. {
6.     struct timeval start,end;
7.     gettimeofday( &start, NULL );  /*测试起始时间*/
8.     double timeuse;
9.     int volatile j;
10.     for(j=0;j<10000000;j++)
11.         ;
12.     gettimeofday( &end, NULL );   /*测试终止时间*/
13.     timeuse = 1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
start.tv_usec;
14.     timeuse /= 1000000;
15. printf("运行时间为：%f\n",timeuse);
16.
17.     return 0;
18.
19. }
```

先来看看我们不使用优化的运行结果为：

[cpp] view plaincopy

```
1. root@ubuntu:/home# gcc time.c -o p
2. root@ubuntu:/home# ./p
3. 运行时间为：0.027647
```

使用了优化的运行结果为：

[cpp] view plaincopy

```
1. root@ubuntu:/home# gcc -o p time.c -O2
2. root@ubuntu:/home# ./p
3. 运行时间为：0.027390
```

我们发现此时此刻不管是否使用优化语句运行，时间几乎没有变化，只是有微小的差异，这微小的差异是由于计算机本身所导致的。所以我们通过对于上面一个没有使用volatile和下面一个使用了volatile的对比结果可知，使用了volatile的变量在使用优化语句是for()循环并没有得到优化，因为for()循环执行的是一个空操作，那么通常情况下使用了优化语句使得这个for()循环被优化掉，根本就不执行。就好比编译器在编译的过程中将i的值设置为大于或者等于10000000的一个数，使得for()循环语句不会执行。但是由于我们使用了volatile，使得编译器就不会自作主张的去动我们的i值，所以循环体得到了执行。举这个例子的原因是要让读者牢记，如果我们定义了volatile变量，那么它就不会被编译器所优化。

当然volatile还有那些值得注意的地方呢？由于访问寄存器的速度要快过直接访问内存的速度，所以编译器一般都会作减少对于内存的访问，但是如果将变量加上volatile修饰，则编译器保证对此变量的读写操作都不会被优化。这样说可能有些抽象了，再看看下面的代码，在此就简要的写出几步了。

```
main()
{
	int i=o;
	while(i==0)
	{
		……
	}
}
```

分析以上代码，如果我们没有在while循环体结构里面改变i的值，编译器在编译的过程中就会将i的值备份到一个寄存器中，每次执行判断语句时就从该寄存器取值，那么这将是一个死循环，但是如果我们做如下的修改：

```
main()
{
	int volatile i=o;
	while(i==0)
	{
		……
	}
}
```

我们在i的前面加上了一个volatile，假设while()循环体里面执行的是跟上一个完全一样的操作，但是这个时候就不能说是一个死循环了，因为编译器不会再对我们的i值进行"备份"操作了，每次执行判断的时候都会直接从i的内存地址中读取，一旦其值发生变化就退出循环体。

最后给出一点就是在实际使用中volatile的使用的场合大致有以下几点：

1、中断服务程序中修改的供其它程序检测的变量需要加volatile；

2、多任务环境下各任务间共享的标志应该加volatile；

3、存储器映射的硬件寄存器通常也要加volatile说明，因为每次对它的读写都可能有不同意义。

对于volatile的讲解我们到此就结束了。由于本人水平有限，博客中的不妥或错误之处在所难免，殷切希望读者批评指正。同时也欢迎读者共同探讨相关的内容，如果乐意交流的话请留下你宝贵的意见。