[**QueryPerformanceFrequency用法**](http://www.cppblog.com/deane/articles/113151.html)

**精确获取时间**

QueryPerformanceFrequency() - 基本介绍

类型：Win32API

原型：BOOL QueryPerformanceFrequency(LARGE\_INTEGER \*lpFrequency);

作用：返回硬件支持的高精度计数器的频率。

返回值：非0，硬件支持高精度计数器；0，硬件不支持，读取失败。

QueryPerformanceFrequency() - 技术特点

供WIN9X使用的高精度定时器：QueryPerformanceFrequency()和QueryPerformanceCounter()，要求计算机从硬件上支持高精度定时器。需包含windows.h头文件。

函数的原形是：

BOOL QueryPerformanceFrequency(LARGE\_INTEGER \*lpFrequency);

BOOL QueryPerformanceCounter (LARGE\_INTEGER \*lpCount);

数据类型LARGEINTEGER既可以是一个作为8字节长的整数，也可以是作为两个4字节长的整数的联合结构，其具体用法根据编译器是否支持64位而定。该类型的定义如下：

typeef union \_ LARGE\_INTEGER

{

struct

{

DWORD LowPart;

LONG HighPart;

};

LONGLONG QuadPart;

} LARGE\_INTEGER;

在定时前应该先调用QueryPerformanceFrequency()函数获得机器内部计时器的时钟频率。接着在需要严格计时的事件发生前和发生之后分别调用QueryPerformanceCounter()，利用两次获得的计数之差和时钟频率，就可以计算出事件经历的精确时间。

测试Sleep的精确时间：

#include <stdio.h>

#include <windows.h>

void main()

{

     LARGE\_INTEGER nFreq;

     LARGE\_INTEGER nBeginTime;

     LARGE\_INTEGER nEndTime;

     double time;

     QueryPerformanceFrequency(&nFreq);

     QueryPerformanceCounter(&nBeginTime);

     Sleep(1000);

     QueryPerformanceCounter(&nEndTime);

     time=(double)(nEndTime.QuadPart-nBeginTime.QuadPart)/(double)nFreq.QuadPart;

     printf("%f\n",time);

}

结果为

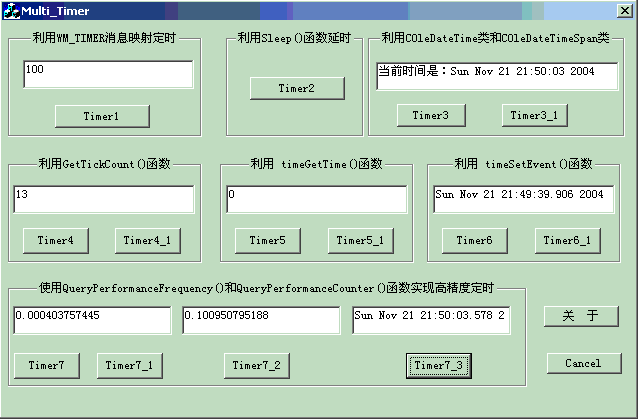
0.999982

1.000088

1.000200

等，所以Sleep的精度还是比较低的。

**VC中基于 Windows 的精确定时**

[示例工程下载](http://www.vckbase.com/code/downcode.asp?id=2537)  
  
　　在工业生产控制系统中，有许多需要定时完成的操作，如定时显示当前时间，定时刷新屏幕上的进度条，上位 机定时向下位机发送命令和传送数据等。特别是在对控制性能要求较高的实时控制系统和数据采集系统中，就更需要精确定时操作。  
　　众所周知，Windows 是基于消息机制的系统，任何事件的执行都是通过发送和接收消息来完成的。 这样就带来了一些问题，如一旦计算机的CPU被某个进程占用，或系统资源紧张时，发送到消息队列 中的消息就暂时被挂起，得不到实时处理。因此，不能简单地通过Windows消息引发一个对定时要求 严格的事件。另外，由于在Windows中已经封装了计算机底层硬件的访问，所以，要想通过直接利用 访问硬件来完成精确定时，也比较困难。所以在实际应用时，应针对具体定时精度的要求，采取相适 应的定时方法。  
　　VC中提供了很多关于时间操作的函数，利用它们控制程序能够精确地完成定时和计时操作。本文详细介绍了 VC中基于Windows的精确定时的七种方式，如下图所示：  
  
 方式一：VC中的**WM\_TIMER**消息映射能进行简单的时间控制。首先调用函数**SetTimer()**设置定时间隔，如SetTimer(0,200,NULL)即为设置200ms的时间间隔。然后在应用程序中增加定时响应函数 **OnTimer()**，并在该函数中添加响应的处理语句，用来完成到达定时时间的操作。这种定时方法非常简单，可以实现一定的定时功能，但其定时功能如同Sleep()函数的延时功能一样，精度非常低，最小计时精度仅为30ms，CPU占用低，且**定时器消息在多任务操作系统中的优先级很低，不能得到及时响 应**，往往不能满足实时控制环境下的应用。只可以用来实现诸如位图的动态显示等对定时精度要求不高的情况。如示例工程中的Timer1。   
　　方式二：VC中使用**sleep()**函数实现延时，它的**单位是ms**，如延时2秒，用sleep(2000)。精度非常低，最小计时精度仅为30ms，用sleep函数的不利处在于**延时期间不能处理其他的消息**，如果时间太长，就好象死机一样，CPU占用率非常高，只能用于要求不高的延时程序中。如示例工程中的Timer2。  
　　方式三：利用**COleDateTime**类和**COleDateTimeSpan**类结合WINDOWS的消息处理过程来实现秒级延时。如示例工程中的Timer3和Timer3\_1。以下是实现2秒的延时代码：

COleDateTime start\_time = COleDateTime::GetCurrentTime();

COleDateTimeSpan end\_time=

COleDateTime::GetCurrentTime()- start\_time;

while(end\_time.GetTotalSeconds() < 2) //实现延时2秒

{

MSG msg;

GetMessage(&msg,NULL,0,0);

TranslateMessage(&msg);

DispatchMessage(&msg);

//以上四行是实现在延时或定时期间能处理其他的消息，

//虽然这样可以降低CPU的占有率，

//但降低了延时或定时精度，实际应用中可以去掉。

end\_time=COleDateTime::GetCurrentTime() - start\_time;

} //这样在延时的时候我们也能够处理其他的消息。

　　方式四：在精度要求较高的情况下，VC中可以利用GetTickCount()函数，该函数的返回值是  DWORD型，表示**以ms为单位的计算机启动后经历的时间间隔**。精度比WM\_TIMER消息映射高，在较短的定时中其计时误差为15ms，在较长的定时中其计时误差较低，如果定时时间太长，就好象死机一样，CPU占用率非常高，只能用于要求不高的延时程序中。如示例工程中的Timer4和Timer4\_1。下列代码可以实现50ms的精确定时：

DWORD dwStart = GetTickCount();

DWORD dwEnd = dwStart;

do

{

dwEnd = GetTickCount()-dwStart;

}while(dwEnd <50);

为使GetTickCount()函数在延时或定时期间能处理其他的消息，可以把代码改为：

DWORD dwStart = GetTickCount();

DWORD dwEnd = dwStart;

do

{

MSG msg;

GetMessage(&msg,NULL,0,0);

TranslateMessage(&msg);

DispatchMessage(&msg);

dwEnd = GetTickCount()-dwStart;

}while(dwEnd <50);

虽然这样可以降低CPU的占有率，并在延时或定时期间也能处理其他的消息，但降低了延时或定时精度。  
　　方式五：与GetTickCount()函数类似的多媒体定时器函数DWORD **timeGetTime**(void)，该函数定时精度为ms级，**返回从Windows启动开始经过的毫秒数**。微软公司在其多媒体Windows中提供了精确定时器的底 层API持，利用多媒体定时器可以很精确地读出系统的当前时间，并且能在非常精确的时间间隔内完成一个事件、函数或过程的调用。不同之处在于调用DWORD timeGetTime(void) 函数之前必须将 **Winmm.lib**  和 **Mmsystem.h** 添加到工程中，否则在编译时提示DWORD timeGetTime(void)函数未定义。由于使用该 函数是通过查询的方式进行定时控制的，所以，应该建立定时循环来进行定时事件的控制。如示例工程中的Timer5和Timer5\_1。  
　　方式六：使用多媒体定时器**timeSetEvent()**函数，该函数定时精度为ms级。利用该函数可以实现周期性的函数调用。如示例工程中的Timer6和Timer6\_1。函数的原型如下：

MMRESULT timeSetEvent（ UINT uDelay,

UINT uResolution,

LPTIMECALLBACK lpTimeProc,

WORD dwUser,

UINT fuEvent ）

　　该函数设置一个**定时回调事件**，此事件可以是一个一次性事件或周期性事件。事件一旦被激活，便调用指定的回调函数， 成功后返回事件的标识符代码，否则返回NULL。函数的参数说明如下：

uDelay：以毫秒指定事件的周期。

Uresolution：以毫秒指定延时的精度，数值越小定时器事件分辨率越高。缺省值为1ms。

**LpTimeProc：指向一个回调函数**。

DwUser：存放用户提供的回调数据。

FuEvent：指定定时器事件类型：

(1)TIME\_ONESHOT：uDelay毫秒后**只产生一次事件**

(2)TIME\_PERIODIC ：每隔uDelay毫秒**周期性地产生事件**。

　　具体应用时，可以通过调用timeSetEvent()函数，**将需要周期性执行的任务定义在LpTimeProc回调函数中**(如：定时采样、控制等)，从而完成所需处理的事件。需要注意的是，任务处理的时间不能大于周期间隔时间。另外，在定时器使用完毕后， 应及时调用**timeKillEvent()**将之释放。   
　　方式七：对于精确度要求更高的定时操作，则应该使用**QueryPerformanceFrequency**()和 **QueryPerformanceCounter**()函数。这两个函数是VC提供的仅供Windows 95及其后续版本使用的精确时间函数，并要求计算机从硬件上支持精确定时器。如示例工程中的Timer7、Timer7\_1、Timer7\_2、Timer7\_3。  
QueryPerformanceFrequency()函数和QueryPerformanceCounter()函数的原型如下：

BOOL QueryPerformanceFrequency(LARGE\_INTEGER\* lpFrequency);

BOOL QueryPerformanceCounter(LARGE\_INTEGER\* lpCount);

　　数据类型ARGE\_INTEGER既可以是一个**8字节**长的整型数，也可以是**两个4字节**长的整型数的联合结构， 其具体用法根据编译器是否支持64位而定。该类型的定义如下：

typedef **union** \_LARGE\_INTEGER

{

struct

{

DWORD LowPart ;// 4字节整型数

LONG HighPart;// 4字节整型数

};

**LONGLONG QuadPart** ;// 8字节整型数

}LARGE\_INTEGER ;

　　在进行定时之前，先调用QueryPerformanceFrequency()函数获得机器内部**定时器的时钟频率**， 然后在需要严格定时的事件发生之前和发生之后分别调用QueryPerformanceCounter()函数，利用两次获得的计数之差及时钟频率，计算出事件经 历的精确时间。下列代码实现1ms的精确定时：

LARGE\_INTEGER litmp;

LONGLONG QPart1,QPart2;

double dfMinus, dfFreq, dfTim;

QueryPerformanceFrequency(&litmp);

dfFreq = (double)litmp.QuadPart;// 获得计数器的时钟频率

QueryPerformanceCounter(&litmp);

QPart1 = litmp.QuadPart;// 获得初始值

do

{

QueryPerformanceCounter(&litmp);

QPart2 = litmp.QuadPart;//获得中止值

dfMinus = (double)(QPart2-QPart1);

dfTim = dfMinus / dfFreq;// 获得对应的时间值，**单位为秒**

}while(dfTim<0.001);

其定时误差不超过1微秒，精度与CPU等机器配置有关。

【示例】

#include <windows.h>  
  
double PCFreq = 0.0;  
\_\_int64 CounterStart = 0;  
  
void StartCounter()  
{  
    LARGE\_INTEGER li;  
    if(!QueryPerformanceFrequency(&li))  
        cout << "QueryPerformanceFrequency failed!\n";  
  
    PCFreq = double(li.QuadPart)/1000.0;  
  
    QueryPerformanceCounter(&li);  
    CounterStart = li.QuadPart;  
}  
double GetCounter()  
{  
    LARGE\_INTEGER li;  
    QueryPerformanceCounter(&li);  
    return double(li.QuadPart-CounterStart)/PCFreq;  
}  
  
int main()  
{  
    StartCounter();  
    Sleep(1000);  
    cout << GetCounter() <<"\n";  
    return 0;  
}

This program should output a number close to 1000 (windows sleep isn't that accurate, but it should be like 999).

The StartCounter() function records the number of ticks the performance counter has in the CounterStart variable. The GetCounter() function returns the number of milliseconds since StartCounter() was last called as a double, so if GetCounter() returns 0.001 then it has been about 1 microsecond since StartCounter() was called.

If you want to have the timer use seconds instead then change

PCFreq = double(li.QuadPart)/1000.0;

to

PCFreq = double(li.QuadPart);

or if you want microseconds then use

PCFreq = double(li.QuadPart)/1000000.0;

But really it's about convenience since it returns a double.