# [xp、win7下如何取消屏保、取消电源管理](http://www.cnblogs.com/cswuyg/archive/2011/12/11/2284023.html)

2011.12.11整理

烛秋

## 一、网络上流传的方法

网络上流传的方法很多，也很复杂。之前在实现这个功能的时候在网络上寻找了很多，如下：

<http://blog.csdn.net/rainharder/article/details/6101317>

<http://topic.csdn.net/t/20021009/15/1081843.html#r_7006146>

<http://topic.csdn.net/u/20100414/12/A5265E43-6288-43A0-B337-C5DA0CF1D7DD.html#r_64661983>

<http://topic.csdn.net/u/20100203/12/6B7E7AF7-D5DA-4BA5-A6EF-8A38866B4C1B.html#r_63377844>

<http://topic.csdn.net/u/20080603/17/5EBACC7D-5F96-4A29-AD7C-55FD8F47CD45.html>

<http://topic.csdn.net/u/20100203/12/6b7e7af7-d5da-4ba5-a6ef-8a38866b4c1b.html>

一些代码试了之后发现很多并没有实现功能。譬如下面的代码：

[复制代码](javascript:void(0);)

static UINT dss\_GetList[] = { SPI\_GETLOWPOWERTIMEOUT,

SPI\_GETPOWEROFFTIMEOUT, SPI\_GETSCREENSAVETIMEOUT };

static UINT dss\_SetList[] = { SPI\_SETLOWPOWERTIMEOUT,

SPI\_SETPOWEROFFTIMEOUT, SPI\_SETSCREENSAVETIMEOUT };

m\_pValue = new int[dss\_ListCount];

for (int x = 0; x < dss\_ListCount; x++)

{

SystemParametersInfo(dss\_GetList[x], 0, &m\_pValue[x], 0);

SystemParametersInfo(dss\_SetList[x], 0, NULL, 0);

}

和

for (int x = 0; x<dss\_ListCount; x++)

{

SystemParametersInfo(dss\_SetList[x], m\_pValue[x], NULL, 0);

}

[复制代码](javascript:void(0);)

**上边的代码可以在很多地方看到，但在xp\win7下都没有效果。这些无效的代码到处黏贴，误人不浅。**

## 二、通过测试确定可行的代码

### 去掉屏保:

bool bAct;

::SystemParametersInfo(SPI\_GETSCREENSAVEACTIVE, 0, &bAct, 0);

if (bAct)

{

::SystemParametersInfo(SPI\_SETSCREENSAVEACTIVE, false, 0, 0);

}

### 恢复屏保：

if ( bAct )

{

::SystemParametersInfo( SPI\_SETSCREENSAVEACTIVE, true, 0, 0 );

}

### 取消电源管理，避免睡眠、待机：

::SetThreadExecutionState( ES\_CONTINUOUS | ES\_SYSTEM\_REQUIRED | ES\_DISPLAY\_REQUIRED );

### 恢复电源管理：

::SetThreadExecutionState( ES\_CONTINUOUS );

## 三、附上解决这一问题时的记录

~~2011.10.24~~

~~2011.10.25~~

### 一、解决屏保问题的解决方式有两种：

  1、可以通过SystemParametersInfo(SPI\_SETSCREENSAVEACTIVE,FALSE,NULL,0);实现。

  2、通过在处理WM\_SYSCOMMAND消息，当wParam参数为：SC\_SCREENSAVE时，处理该消息，返回true。要注意的是，这种方式的局限性：只有当前的Activate窗口，才能收到这个进入屏保的消息！

### 二、解决的电源管理的解决方式有三种：

  1、 通过SystemParamerinfo设置，参数为 SPI\_SETLOWPOWERTIMEOUT和SPI\_SETPOWEROFFTIMEOUT。但有时候不灵，如在xp、win7下都不行。这个方法是网络上的，实地测试之后，发现不靠谱！！

  2、不关闭显示器的解决方式：通过处理WM\_SYSCOMMAND消息，当wParam参数为：SC\_MONITORPOWER时。这个方式的局限性跟上边的一样，可能无法收到该消息。

  3、使用定时器调用SetThreadExecutionState函数，可以使得系统不关闭显示器、不睡眠。

### 三、具体实现：

1、干掉屏保：SystemParametersInfo( SPI\_SETSCREENSAVEACTIVE, false, 0, 0);

   获取屏保设置：SystemParametersInfo( SPI\_GETSCREENSAVEACTIVE, 0, &m\_bSCreenSave, 0 );

   恢复屏保设置：SystemParametersInfo( SPI\_SETSCREENSAVEACTIVE, m\_bSCreenSave, 0, 0);

2、干掉电源管理：

   设置定时器：SetTimer(1, 5000);

   定时调用：SetThreadExecutionState(ES\_SYSTEM\_REQUIRED | ES\_DISPLAY\_REQUIRED);

2011.10.26

### 四、具体实现修正

不需要定时调用SetThreadExecutionState，只需要参数加上ES\_CONTINUOUS即可。

SetThreadExecutionState( ES\_CONTINUOUS | ES\_SYSTEM\_REQUIRED | ES\_DISPLAY\_REQUIRED );

用完之后：SetThreadExecutionState( ES\_CONTINUOUS );

ES\_CONTINUOUS参数的作用是“冻结”空闲计数器，直到下一次调用

参考资料：http://msdn.microsoft.com/en-us/library/ms995870.aspx

Calling the SetThreadExecutionState function without the ES\_CONTINUOUS flag will simply reset the idle timers and the system will go to sleep if the timers run out again. To keep the system or the display awake it may be necessary to use the ES\_CONTINUOUS flag along with either the ES\_SYSTEM\_REQUIRED or the ES\_DISPLAY\_REQUIRED flag. This will freeze the respective timers. If the ES\_CONTINUOUS flag is used in conjunction with ES\_DISPLAY\_REQUIRED or ES\_SYSTEM\_REQUIRED, applications should then release the idle timers by resending just the ES\_CONTINUOUS flag when there is no longer a need to prevent them from timing out.

## 四、相关问题

1、屏幕保护在注册表中的位置：HKEY\_CURRENT\_USER\Control Panel\Desktop

2、电源管理相关：HKEY\_CURRENT\_USER\Control Panel\PowerCf

3、发现注册表里修改了屏保时间，并没有生效。

4、睡眠跟待机不同，睡眠需要重新按开机键。

5、笔记本的电源管理。联想的笔记本有电源管理设置用于关闭屏幕背光，几分钟无操作之后，笔记本屏幕就变黑，但这个时候系统没有待机、没有睡眠、没有屏保。怎么让它不关闭背光呢？查找了很久，感觉这是无法解决的问题，它不属于电源管理，可能是硬件级别上的实现。

# SetThreadExecutionState Function---MSDN

Enables applications to inform the system that it is in use, thereby preventing the system from entering the sleeping power state or turning off the display while the application is running.

EXECUTION\_STATE WINAPI SetThreadExecutionState(

\_\_in EXECUTION\_STATE *esFlags*

);

**Parameters**

*esFlags*

The thread's execution requirements. This parameter can be one or more of the following values.

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| ES\_CONTINUOUS 0x80000000 | Informs the system that the state being set should remain in effect until the next call that uses ES\_CONTINUOUS and one of the other state flags is cleared. |
| ES\_DISPLAY\_REQUIRED 0x00000002 | Forces the display to be on by resetting the display idle timer. |
| ES\_SYSTEM\_REQUIRED 0x00000001 | Forces the system to be in the working state by resetting the system idle timer. |

**Return Value**

If the function succeeds, the return value is the previous thread execution state.

If the function fails, the return value is NULL.

**Remarks**

Activities that are automatically detected include local keyboard or mouse input, server activity, and changing window focus. Activities that are not automatically detected include disk or CPU activity and video display.

Calling **SetThreadExecutionState** without ES\_CONTINUOUS simply resets the idle timer; to keep the display or system in the working state, the thread must call **SetThreadExecutionState** periodically.

To run properly on a power-managed computer, applications such as fax servers, answering machines, backup agents, and network management applications must use ES\_SYSTEM\_REQUIRED | ES\_CONTINUOUS when they process events. Multimedia applications, such as video players and presentation applications, must use ES\_DISPLAY\_REQUIRED when they display video for long periods of time without user input. Applications such as word processors, spreadsheets, browsers, and games do not need to call **SetThreadExecutionState**.

The **SetThreadExecutionState** function cannot be used to prevent the user from putting the computer in standby mode. Applications should respect that the user expects a certain behavior when they close the lid on their laptop or press the power button.

This function does not stop the screen saver from executing either.

**Requirements**

|  |  |
| --- | --- |
| **Client** | Requires Windows Vista, Windows XP, or Windows 2000 Professional. |
| **Server** | Requires Windows Server 2008, Windows Server 2003, or Windows 2000 Server. |
| **Header** | Declared in Winbase.h; include Windows.h. |
| **Library** | Use Kernel32.lib. |
| **DLL** | Requires Kernel32.dll. |

**See Also**

[Power Management Functions](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/power/base/power_management_functions.htm)  
[**SetSystemPowerState**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/power/base/setsystempowerstate.htm)  
[**WM\_POWERBROADCAST**](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/power/base/wm_powerbroadcast.htm)

# [SetConsoleCtrlHandler 处理控制台消息](http://andylin02.iteye.com/blog/661431)

一、如何处理所有的控制台消息。

　　第一步，首先要安装一个事件钩子，也就是说要建立一个回调函数。调用Win32 API，原型如下：

BOOL SetConsoleCtrlHandler（

PHANDLER\_ROUTINE HandlerRoutine, // 回调函数

BOOL Add // 表示添加还是删除

）；

　　参数HandlerRoutine是一个指向函数的指针，原型如下:

BOOL WINAPI HandlerRoutine（

DWORD dwCtrlType // 控制事件类型

）；

　　所有的HandlerRoutine函数只有一个参数dwCtrlType，他表示控制台发出了什么消息。参数有下列值:

CTRL\_C\_EVENT - 当用户按下了CTRL+C,或者由GenerateConsoleCtrlEvent API发出.

CTRL\_BREAK\_EVENT - 用户按下CTRL+BREAK, 或者由GenerateConsoleCtrlEvent API发出.

CTRL\_CLOSE\_EVENT - 当试图关闭控制台程序，系统发送关闭消息。

CTRL\_LOGOFF\_EVENT - 用户退出时，但是不能决定是哪个用户.

CTRL\_SHUTDOWN\_EVENT - 当系统被关闭时.

　当收到事件的时候，HandlerRoutine可以选择处理，或者简单的忽略。如果回调函数选择忽略，函数返回FALSE，系统将处理下一个钩子程序。如果处理消息，程序在处理完消息后应该返回TRUE。

CTRL\_CLOSE\_EVENT, CTRL\_LOGOFF\_EVENT和CTRL\_SHUTDOWN\_EVENT通常被用来处理一些程序的清理工作，然后调用ExitProcess API。另外，这三个事件有超时机制，CTRL\_CLOSE\_EVENT是5秒，另外两个是20秒。如果程序超时候，系统将会弹出结束进程的对话框。如果用户选择了结束进程，任何清理工作都不会做，所以应该在超时时间内完成工作。下面是一个回调函数的例子：

BOOL WINAPI ConsoleHandler（DWORD CEvent）

{

char mesg[128]；

switch（CEvent）

{

case CTRL\_C\_EVENT:

MessageBox（NULL,

"CTRL+C received!","CEvent",MB\_OK）；

break；

case CTRL\_BREAK\_EVENT:

MessageBox（NULL,

"CTRL+BREAK received!","CEvent",MB\_OK）；

break；

case CTRL\_CLOSE\_EVENT:

MessageBox（NULL,

"Program being closed!","CEvent",MB\_OK）；

break；

case CTRL\_LOGOFF\_EVENT:

MessageBox（NULL,

"User is logging off!","CEvent",MB\_OK）；

break；

case CTRL\_SHUTDOWN\_EVENT:

MessageBox（NULL,

"User is logging off!","CEvent",MB\_OK）；

break；

}

return TRUE；

}

　　好，现在已经有了回调函数，再来看看怎么安装钩子：

if （SetConsoleCtrlHandler（

（PHANDLER\_ROUTINE）ConsoleHandler,TRUE）==FALSE）

{

// unable to install handler...

// display message to the user

printf（"Unable to install handler!\n"）；

return -1；

}

　　第一个参数是函数指针，就是上面的那个函数。第二个参数是标志，如果为TRUE那么就安装钩子，如果为FALSE那么删除钩子。

　　好了，在安装了钩子后，我们就能收到控制台消息了，在程序退出前，要删除钩子。很简单吧。

# VC定时器的用法:SetTimer和Ontimer

Settimer是设置一个计时器并开始执行计时器Ontimer中的代码，Ontimer是计时器所执行的代码。KillTimer用于停止计时器。 Settimer是设置，Ontimer是响应Settimer消息的。

SetTimer函数的用法

1 ）用WM\_TIMER来设置定时器

先请看SetTimer这个API函数的原型

UINT\_PTR SetTimer(            
    HWND hWnd,//和定时器相关联的窗口   
    UINT\_PTR nIDEvent,//一个非0的数字标志这个定时器   
    UINT uElapse,//指定时间间隔，以毫秒为单位   
    TIMERPROC lpTimerFunc//一般指定为null   
);   
当在MFC的工程中如下：   
UINT\_PTR SetTimer(            
    UINT\_PTR nIDEvent,//一个非0的数字标志这个定时器   
    UINT uElapse,//指定时间间隔，以毫秒为单位   
    TIMERPROC lpTimerFunc//一般指定为null   
);

用法如下：

SetTimer(1,1000,NULL);

1:计时器的名称；

1000：时间间隔，单位是毫秒；

NULL:使用onTime函数。

在每次时间到1s时会触发WM\_TIMER消息，其消息响应函数如下：   
void CXXX::OnTimer(UINT\_PTR nIDEvent)   
{   
// TODO: 在此添加消息处理程序代码和/或调用默认值   
        // 此处加入如下代码就好了：   
        switch()   
        {   
        case 1:   
              //做该做的事情   
                //当不需要的时候在此处调用KillTimer(1);   
              break;   
        default:   
              break;   
        }   
CDialog::OnTimer(nIDEvent);   
}

当不需要计时器的时候调用KillTimer(nIDEvent);

例如：KillTimer(1);

如果我要加入两个或者两个以上的 timer怎么办？

继续用SetTimer函数吧，上次的timer的ID是1，这次可以是2，3，4。。。。

SetTimer(2,1000,NULL);

SetTimer(3,500,NULL);

嗯，WINDOWS会协调他们的。当然onTimer函数体也要发生变化，要在函数体内添加每一个timer的处理代码：

onTimer(nIDEvent)

{

switch(nIDEvent)

{

case 1:........;

break;

case 2:.......;

break;

case 3:......;

break;

}

}

# [time\_t 与字符串之间的相互转换](http://blog.csdn.net/nanhaizhixin/article/details/8349668)

本文章已收录于：

方法1， SYSTEMTIME 类型

//指定time\_t类型的时间，格式化为YYYYMMDDHH24MISS型的字符串

void FormatTime(SYSTEMTIME &tm1)

{

TCHAR \* szTime = new TCHAR[128];

GetLocalTime(&tm1);

\_stprintf( szTime, \_T("%4.4d-%2.2d-%2.2d %2.2d:%2.2d:%2.2d"),

tm1.wYear, tm1.wMonth+1, tm1.wDay,

tm1.wHour, tm1.wMinute,tm1.wSecond);

delete szTime;

szTime = NULL;

}

//指定YYYYMMDDHH24MISS型的时间，格式化为time\_t型的时间

SYSTEMTIME FormatTime2(TCHAR \* szTime)

{

SYSTEMTIME tm1;

\_stscanf( szTime, \_T("%4d-%2d-%2d %2d:%2d:%2d"),

&tm1.wYear,

&tm1.wMonth,

&tm1.wDay,

&tm1.wHour,

&tm1.wMinute,

&tm1.wSecond );

return time1;

}

方法2， time\_t类型

//指定time\_t类型的时间，格式化为YYYYMMDDHH24MISS型的字符串

void FormatTime(time\_t time1, char \*szTime)

{

struct tm tm1;

#ifdef WIN32

tm1 = \*localtime(&time1);

#else

localtime\_r(&time1, &tm1 );

#endif

sprintf( szTime, "%4.4d%2.2d%2.2d%2.2d%2.2d%2.2d",

tm1.tm\_year+1900, tm1.tm\_mon+1, tm1.tm\_mday,

tm1.tm\_hour, tm1.tm\_min,tm1.tm\_sec);

}

//指定YYYYMMDDHH24MISS型的时间，格式化为time\_t型的时间

time\_t FormatTime2(char \* szTime)

{

struct tm tm1;

time\_t time1;

sscanf(szTime, "%4d%2d%2d%2d%2d%2d",

&tm1.tm\_year,

&tm1.tm\_mon,

&tm1.tm\_mday,

&tm1.tm\_hour,

&tm1.tm\_min,

&tm1.tm\_sec);

tm1.tm\_year -= 1900;

tm1.tm\_mon --;

tm1.tm\_isdst=-1;

time1 = mktime(&tm1);

return time1;

}

# [struct stat结构体简介](http://www.cnblogs.com/CSU-PL/archive/2013/06/06/3120757.html)

在使用这个结构体和方法时，需要引入：

<sys/types.h>

<sys/stat.h>

struct stat这个结构体是用来描述一个linux系统文件系统中的文件属性的结构。

可以有两种方法来获取一个文件的属性：

1、通过路径：

int stat(const char \*path, struct stat \*struct\_stat);

int lstat(const char \*path,struct stat \*struct\_stat);

两个函数的第一个参数都是文件的路径，第二个参数是struct stat的指针。返回值为0，表示成功执行。

执行失败是，error被自动设置为下面的值：

EBADF： 文件描述词无效

EFAULT： 地址空间不可访问

ELOOP： 遍历路径时遇到太多的符号连接

ENAMETOOLONG：文件路径名太长

ENOENT：路径名的部分组件不存在，或路径名是空字串

ENOMEM：内存不足

ENOTDIR：路径名的部分组件不是目录

这两个方法区别在于stat没有处理字符链接(软链接）的能力，如果一个文件是符号链接，stat会直接返回它所指向的文件的属性；而lstat返回的就是这个符号链接的内容。这里需要说明一下的是软链接和硬链接的含义。我们知道目录在linux中也是一个文件，文件的内容就是这这个目录下面所有文件与inode的对应关系。那么所谓的硬链接就是在某一个目录下面将一个文件名与一个inode关联起来，其实就是添加一条记录！而软链接也叫符号链接更加简单了，这个文件的内容就是一个字符串，这个字符串就是它所链接的文件的绝对或者相对地址。

2、通过文件描述符

int fstat(int fdp, struct stat \*struct\_stat);　　//通过文件描述符获取文件对应的属性。fdp为文件描述符

下面是这个结构的结构

struct stat {

        mode\_t     st\_mode;       //文件对应的模式，文件，目录等

        ino\_t      st\_ino;       //inode节点号

        dev\_t      st\_dev;        //设备号码

        dev\_t      st\_rdev;       //特殊设备号码

        nlink\_t    st\_nlink;      //文件的连接数

        uid\_t      st\_uid;        //文件所有者

        gid\_t      st\_gid;        //文件所有者对应的组

        off\_t      st\_size;       //普通文件，对应的文件字节数

        time\_t     st\_atime;      //文件最后被访问的时间

        time\_t     st\_mtime;      //文件内容最后被修改的时间

        time\_t     st\_ctime;      //文件状态改变时间

        blksize\_t st\_blksize;    //文件内容对应的块大小

        blkcnt\_t   st\_blocks;     //伟建内容对应的块数量

      };

stat结构体中的st\_mode 则定义了下列数种情况：  
    S\_IFMT   0170000    文件类型的位遮罩  
    S\_IFSOCK 0140000    scoket  
    S\_IFLNK 0120000     符号连接  
    S\_IFREG 0100000     一般文件  
    S\_IFBLK 0060000     区块装置  
    S\_IFDIR 0040000     目录  
    S\_IFCHR 0020000     字符装置  
    S\_IFIFO 0010000     先进先出  
  
    S\_ISUID 04000     文件的(set user-id on execution)位  
    S\_ISGID 02000     文件的(set group-id on execution)位  
    S\_ISVTX 01000     文件的sticky位  
  
    S\_IRUSR(S\_IREAD) 00400     文件所有者具可读取权限  
    S\_IWUSR(S\_IWRITE)00200     文件所有者具可写入权限  
    S\_IXUSR(S\_IEXEC) 00100     文件所有者具可执行权限  
  
    S\_IRGRP 00040             用户组具可读取权限  
    S\_IWGRP 00020             用户组具可写入权限  
    S\_IXGRP 00010             用户组具可执行权限  
  
    S\_IROTH 00004             其他用户具可读取权限  
    S\_IWOTH 00002             其他用户具可写入权限  
    S\_IXOTH 00001             其他用户具可执行权限  
  
    上述的文件类型在POSIX中定义了检查这些类型的宏定义：  
    S\_ISLNK (st\_mode)    判断是否为符号连接  
    S\_ISREG (st\_mode)    是否为一般文件  
    S\_ISDIR (st\_mode)    是否为目录  
    S\_ISCHR (st\_mode)    是否为字符装置文件  
    S\_ISBLK (s3e)        是否为先进先出  
    S\_ISSOCK (st\_mode)   是否为socket  
若一目录具有sticky位(S\_ISVTX)，则表示在此目录下的文件只能被该文件所有者、此目录所有者或root来删除或改名，在linux中，最典型的就是这个/tmp目录啦。

# [VC++的AssertValid和Dump函数的应用详解](http://blog.csdn.net/pizi0475/article/details/5467018)

**CObject::AssertValid**  成员函数提供对对象内部状态的运行时检查。尽管从CObject派生类时不需要重写 AssertValid，但可以通过重写使您的类更安全可靠。AssertValid应在对象的所有成员变量上执行断言，以验证它们包含有效值。例如，它应检查指针成员变量不为   NULL。  下面的示例显示如何声明   AssertValid   函数：    
  class   CPerson   :   public   CObject    
  {    
  protected:    
        CString   m\_strName;    
        float       m\_salary;    
  public:    
  #ifdef   \_DEBUG    
        virtual   void   AssertValid()   const;       //   Override    
  #endif    
        //   ...    
  };    
  当重写   AssertValid   时，在执行您自己的检查之前请调用   AssertValid   的基类版本。然后使用   ASSERT   宏检查您的派生类特有的成员，如下所示：    
     
  #ifdef   \_DEBUG    
  void   CPerson::AssertValid()   const    
  {    
        //   call   inherited   AssertValid   first    
        CObject::AssertValid();    
     
        //   check   CPerson   members...    
        ASSERT(   !m\_strName.IsEmpty());   //   Must   have   a   name    
        ASSERT(   m\_salary   >   0   );   //   Must   have   an   income    
  }    
  #endif    
  如果任何成员变量存储对象，则可以使用   ASSERT\_VALID   宏测试它们的内部有效性（如果它们的类重写了   AssertValid）。    
     
  例如，考虑   CMyData   类，该类在其成员变量之一中存储了一个   CObList。CObList   变量   m\_DataList   存储了一个   CPerson   对象的集合。CMyData   的简化声明如下所示：    
     
  class   CMyData   :   public   CObject    
  {    
        //   Constructor   and   other   members   ...    
        protected:    
              CObList\*   m\_pDataList;    
        //   Other   declarations   ...    
        public:    
  #ifdef   \_DEBUG    
              virtual   void   AssertValid(   )   const;   //   Override    
  #endif    
        //   Etc.   ...    
  };    
  CMyData   中重写的   AssertValid   如下所示：    
     
  #ifdef   \_DEBUG    
  void   CMyData::AssertValid(   )   const    
  {    
        //   Call   inherited   AssertValid    
        CObject::AssertValid(   );    
        //   Check   validity   of   CMyData   members    
        ASSERT\_VALID(   m\_pDataList   );    
        //   ...    
  }    
  #endif    
  CMyData   使用   AssertValid   机制测试其数据成员中存储的对象的有效性。CMyData   中重写的   AssertValid   为它自己的   m\_pDataList   成员变量调用   ASSERT\_VALID   宏。    
     
  因为   CObList   类也重写   AssertValid，所以有效性测试不在该级别停止。该重写对列表的内部状态执行附加有效性测试。因此，对   CMyData   对象的有效性测试将导致对存储的   CObList   列表对象内部状态的附加有效性测试。    
     
  再多进行一些操作，还可以添加对存储在列表中的   CPerson   对象的有效性测试。可以从   CObList   派生   CPersonList   类，并重写   AssertValid。在重写中可调用   CObject::AssertValid，然后循环访问列表，在列表中存储的每个   CPerson   对象上调用   AssertValid。本主题开始所示的   CPerson   类已重写了   AssertValid。    
     
  当为调试生成时，这是一种功能极强的机制。当接着为发布生成时，该机制自动关闭。    
     
  AssertValid   的限制    
  给定类的   AssertValid   函数的用户应注意该函数的限制。触发的断言指示对象一定有误，并且执行将暂停。但是，缺少断言只指示未找到任何问题，并不保证对象是好的。

**Dump**

当从   CObject   派生类时，在使用   DumpAllObjectsSince   将对象转储到“输出”窗口时，可以重写   Dump   成员函数以提供附加信息。    
     
  Dump   函数将对象的成员变量的文本化表示形式写入转储上下文   (CDumpContext)。转储上下文类似于   I/O   流。可以使用插入运算符   (<<)   向   CDumpContext   发送数据。    
     
  重写   Dump   函数时，应先调用   Dump   的基类版本以转储基类对象的内容。然后为派生类的每个成员变量输出文本化说明和值。    
     
  Dump   函数的声明如下所示：    
     
  class   CPerson   :   public   CObject    
  {    
  public:    
  #ifdef   \_DEBUG    
        virtual   void   Dump(   CDumpContext&   dc   )   const;    
  #endif    
     
        CString   m\_firstName;    
        CString   m\_lastName;    
        //   And   so   on...    
  };    
  由于对象转储只在调试程序时有意义，所以   Dump   函数的声明用   #ifdef   \_DEBUG   /   #endif   块括起来。    
     
  在下面的示例中，Dump   函数先为其基类调用   Dump   函数。然后，它将每个成员变量的简短说明与该成员的值一起写入诊断流。    
     
  #ifdef   \_DEBUG    
  void   CPerson::Dump(   CDumpContext&   dc   )   const    
  {    
        //   Call   the   base   class   function   first.    
        CObject::Dump(   dc   );    
     
        //   Now   do   the   stuff   for   our   specific   class.    
        dc   <<   "last   name:   "   <<   m\_lastName   <<   "/n"    
              <<   "first   name:   "   <<   m\_firstName   <<   "/n";    
  }    
  #endif    
  必须提供   CDumpContext   参数以指定转储输出的目的地。MFC   的“Debug”版本提供名为   afxDump   的预定义   CDumpContext   对象，它将输出发送到调试器。    
     
  CPerson\*   pMyPerson   =   new   CPerson;    
  //   Set   some   fields   of   the   CPerson   object.    
  //...    
  //   Now   dump   the   contents.    
  #ifdef   \_DEBUG    
  pMyPerson->Dump(   afxDump   );    
  #endif    
  在   MFC   程序中，可以使用   DumpAllObjectsSince   转储有关堆中尚未释放的所有对象的说明。DumpAllObjectsSince   转储自上个   CMemoryState::Checkpoint   以来分配的所有对象。如果未发生   Checkpoint   调用，则   DumpAllObjectsSince   将转储当前在内存中的所有对象和非对象。    
     
  注意       必须先启用诊断跟踪，然后才能使用   MFC   对象转储。    
  注意       程序退出时   MFC   将自动转储所有泄漏的对象，因此不必创建代码在该点转储对象。    
  以下代码通过比较两个内存状态来测试内存泄漏，并在检测到泄漏时转储所有对象：    
     
  if(   diffMemState.Difference(   oldMemState,   newMemState   )   )    
  {    
        TRACE(   "Memory   leaked!/n"   );    
        diffMemState.DumpAllObjectsSince();    
  }    
  转储的内容如下所示：    
     
  Dumping   objects   ->    
     
  {5}   strcore.cpp(80)   :   non-object   block   at   $00A7521A,   9   bytes   long    
  {4}   strcore.cpp(80)   :   non-object   block   at   $00A751F8,   5   bytes   long    
  {3}   strcore.cpp(80)   :   non-object   block   at   $00A751D6,   6   bytes   long    
  {2}   a   CPerson   at   $51A4     
  Last   Name:   Smith    
  First   Name:   Alan    
  Phone   #:   581-0215     
  {1}   strcore.cpp(80)   :   non-object   block   at   $00A7516E,   25   bytes   long    
  大多数行开始处的大括号中的数字指定对象的分配顺序。最近分配的对象具有最高编号，并显示在转储的顶部。

# [DeviceIOControl详解-各个击破](http://blog.csdn.net/gis_wudi/article/details/6639739)

标签： [microsoft](http://www.csdn.net/tag/microsoft)[attributes](http://www.csdn.net/tag/attributes)[function](http://www.csdn.net/tag/function)[access](http://www.csdn.net/tag/access)[file](http://www.csdn.net/tag/file)[api](http://www.csdn.net/tag/api)

2011-07-28 10:55 23961人阅读 [评论](http://blog.csdn.net/gis_wudi/article/details/6639739#comments)(6) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/gis_wudi/article/details/6639739#report)

分类：

VC相关（9）

[作者同类文章](http://blog.csdn.net/GIS_wudi/article/category/849098)*X*

Win32 Api（2）

[作者同类文章](http://blog.csdn.net/GIS_wudi/article/category/854950)*X*

版权声明：本文为博主原创文章，未经博主允许不得转载。

DeviceIoControl这个api我们用的不多，但是很重要，有时会帮助我们实现一些特别的需求, 如获取硬件设备信息、与硬件设备通信（读写数据）等，对照msdn，下面我们详细解释一下这个api的用法（有什么错误再所难免，各位不吝指教啊）。  
DeviceIoControl是用来控制我们指定设备的输入输出操作，使设备按照我们发的指令去工作。

DeviceIoControl是kernel32中的函数，需要包含的头文件是winbase.h，先看函数原型

**BOOL DeviceIoControl(**

**HANDLE** *hDevice***,**

**DWORD** *dwIoControlCode***,**

**LPVOID** *lpInBuffer***,**

**DWORD** *nInBufferSize***,**

**LPVOID** *lpOutBuffer***,**

**DWORD** *nOutBufferSize***,**

**LPDWORD** *lpBytesReturned***,**

**LPOVERLAPPED** *lpOverlapped*

**);**

哈哈，参数不少，而且还都很抽象，没关系，我们一个一个击破它

来到第一个参数，hDevice,当然是要操作的设备的句柄了，这个句柄需要通过CreateFile的返回值中获取，对于createfile这里只做一个粗略的解释：

* **HANDLE CreateFile(**
* **LPCTSTR** *lpFileName***,**
* **DWORD** *dwDesiredAccess***,**
* **DWORD** *dwShareMode***,**
* **LPSECURITY\_ATTRIBUTES** *lpSecurityAttributes***,**
* **DWORD** *dwCreationDisposition***,**
* **DWORD** *dwFlagsAndAttributes***,**
* **HANDLE** *hTemplateFile*
* **);**

参数：要打开的文件名，访问权限，共享模式，安全属性，文件存在与不存在时的文件创建模式，文件属性设定（隐藏、只读、压缩、指定为系统文件等），文件副本句柄。要说明的是第一个参数lpFileName,是设备的名称或者是和设备关连的驱动的名称，一般用\\.\DeviceName的形式，比如要打开逻辑驱动盘A就用\\.\a,也可以用\\.\PhysicalDevice0,\\.\PhsycalDebive1来指定物理驱动器，\\.\PhysicalDevice0表示本机的物理驱动器0(一般是主硬盘),从而来获取硬盘的序列号、模块名、扇区数、磁头数等相关信息

搞定hDevice!

dwIoControlCode: 当然就是控制设备的指令了，指令怎么来是个问题，微软已经定义好了很多种操作，在winioctl.h文件中，但最终都是通过CTL\_CODE宏来实现的，其实这就是一种通信协议。CTL\_CODE的具体用法在最后来介绍。

lpInBuffer: 设备操控请求数据的缓冲区基址，如果dwIoControlCode 指定了一个操作，该操作不需要输入数据，那么这个参数设为NULL

nInBufferSize：lplnBuffer的size

lpOutBuffer:存放输出数据的buffer，同样，如果dwIoControlCode 指定了一个操作，该操作不需要处理输出数据，那么这个参数设为NULL

nOutBufferSize：haha，别说你不知道什么什么意思，pass

lpBytesReturned：实际输出数据的bytes

lpOverlapped：Ignored; set to NULL.（Are you understand?）

下面来到第二个参数的详解，CTL\_CODE的定义与应用：

CTL\_CODE原型：

**#define CTL\_CODE(***DeviceType*, *Function*, *Method*, *Access***) (**

**((***DeviceType***) << 16) | ((***Access***) << 14) | ((***Function***) << 2) | (***Method***)**

**)**

可以看到，这个宏四个参数，自然是一个32位分成了4部分，高16位存储设备类型（ 这里不列举了，看msdn哦），14~15位访问权限，2～13位操作功能，最后一个就是确定缓冲区（别忘记上面DeviceIOControl中缓冲区的定义哦）是如何与I/O和文件系统数据缓冲区进行数据传递的方式（具体取值查看msdn）我们最常用的就是METHOD\_BUFFERED

Function codes 0-2047 are reserved for Microsoft; codes 2048-4095 are reserved for OEMs and IHVs. （我们能用的是2048～4095）

看以下一段：

这个宏经常用来定义IOCTL（I/O控制）和FSCTL（文件系统控制）功能控制代码，所有的IOCTLs**必须**通过这种方式定义，以确保这些指令能被Microsoft，以及其他的硬件厂商通信接口所识别

The following illustration shows the format of the resulting IOCTL.

援引微软定义的一个指令：锁卷

#define FSCTL\_LOCK\_VOLUME               CTL\_CODE(FILE\_DEVICE\_FILE\_SYSTEM,  6, METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

Game over!

# [DeviceIoControl的使用说明](http://blog.csdn.net/lujunql/article/details/2532152)

标签： [null](http://www.csdn.net/tag/null)[buffer](http://www.csdn.net/tag/buffer)[parameters](http://www.csdn.net/tag/parameters)[output](http://www.csdn.net/tag/output)[application](http://www.csdn.net/tag/application)[byte](http://www.csdn.net/tag/byte)

2008-06-10 16:29 15436人阅读 [评论](http://blog.csdn.net/lujunql/article/details/2532152#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/lujunql/article/details/2532152#report)

分类：

技术（141）

[作者同类文章](http://blog.csdn.net/lujunql/article/category/398326)*X*

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/lujunql/article/details/2532152)

1. [Parameters参数](http://blog.csdn.net/lujunql/article/details/2532152#t0)
2. [Return Values返回值](http://blog.csdn.net/lujunql/article/details/2532152#t1)

应用程序和驱动程序的通信过程是：应用程序使用CreateFile函数打开设备，然后用DeviceIoControl与驱动程序进行通信，包括读和写两种操作。还可以用ReadFile读数据用WriteFile写数据。操作完毕时用CloseHandle关闭设备。我们比较常用的就是用DeviceIoControl对设备进行读写操作。先看看DeviceIoControl是怎么定义的：

BOOL DeviceIoControl(

HANDLE hDevice,

DWORD dwIoControlCode,

LPVOID lpInBuffer,

DWORD nInBufferSize,

LPVOID lpOutBuffer,

DWORD nOutBufferSize,

LPDWORD lpBytesReturned,

LPOVERLAPPED lpOverlapped

);

#### Parameters（参数）

hDevice （CreateFile返回的设备句柄）

[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

dwIoControlCode （应用程序调用驱动程序的控制命令，就是IOCTL\_XXX IOCTLs ）

[in] IOCTL for the operation. This value identifies the specific operation to perform and the type of device on which to perform the operation. There are no specific values defined for the dwIoControlCode parameter. However, you can define custom IOCTL\_XXX IOCTLs with the [CTL\_CODE](http://msdn.microsoft.com/en-us/library/ms904001.aspx) macro. You can then advertise these IOCTLs and an application can use these IOCTLs with **DeviceIoControl** to perform the driver-specific functions.

lpInBuffer （应用程序传递给驱动程序的数据缓冲区地址）

[in] Long pointer to a buffer that contains the data required to perform the operation. Set to NULL if the dwIoControlCode parameter specifies an operation that does not require input data.

nInBufferSize （应用程序传递给驱动程序的数据缓冲区大小，字节数）

[in] Size, in bytes, of the buffer pointed to by lpInBuffer.

lpOutBuffer （驱动程序返回给应用程序的数据缓冲区地址）

[out] Long pointer to a buffer that receives the output data for the operation. Set to NULL if the dwIoControlCode parameter specifies an operation that does not produce output data.

nOutBufferSize （驱动程序返回给应用程序的数据缓冲区大小，字节数）

[out] Size, in bytes, of the buffer pointed to by lpOutBuffer.

lpBytesReturned （驱动程序实际返回给应用程序的数据字节数地址）

[out] Long pointer to a variable that receives the size, in bytes, of the data stored in lpOutBuffer. The **DeviceIoControl** function may unnecessarily use this parameter. For example, if an operation does not produce data for lpOutBuffer and lpOutBuffer is NULL, the value of lpBytesReturned is meaningless.

lpOverlapped （重叠操作结构）

[in] Ignored; set to NULL.

#### Return Values（返回值）

Nonzero indicates success. Zero indicates failure. To obtain extended error information, call the [GetLastError](http://msdn.microsoft.com/en-us/library/ms885627.aspx) function. （非0成功，0失败）

具体使用我们看看列子：

1，向设备传递数据，我们定义一个函数来实现

bool CDeviceOperDlg::SendKeyData(HANDLE handle, BYTE \*bData, int iSize)  
{  
 ULONG nOutput;  
 BYTE bTemp[512];

 //将数据放置到发送数组  
 memset(bTemp,0,sizeof(bTemp));  
 memcpy(bTemp,&bData[0],iSize);  
 //向设备发送  
 if (!DeviceIoControl(handle,           
       ATST2004\_IOCTL\_WRITE,     //根据具体的设备有相关的定义  
       bTemp,                                        //向设备传递的数据地址  
       iSize,                                            //数据大小，字节数  
       NULL,                                          //没有返回的数据，置为NULL  
       0,                                                  //没有返回的数据，置为0

       &nOutput,  
       NULL)  
    )  
 {  
  return false;  
 }

 return true;  
}

2，从设备读取数据

bool CDeviceOperDlg::ReviceKeyData(HANDLE handle, BYTE \*bData, int iSize)  
{

 ULONG nOutput;  
 BYTE bTemp[512];  
 //数组清零  
 memset(bTemp,0,sizeof(bTemp));  
 //向设备发送  
 if (!DeviceIoControl(handle,  
       ATST2004\_IOCTL\_READ,           //根据具体的设备有相关的定义  
       NULL,                                              //没有向设备传递的数据，置为NULL  
       0,                                                      //没有向设备传递的数据，置为NULL  
       bTemp,                                           //读取设备的数据返回地址  
       iSize,                                               //读取数据的字节数  
       &nOutput,  
       NULL)  
    )  
 {  
  return false;  
 }  
 //放置到公用数组  
 memcpy(&bData[0],&bTemp[0],iSize);  
 return true;  
}

# [关于setvbuf()函数的详解](http://www.cnblogs.com/lichenlong/p/5170059.html)

为什么要使用setvbuf函数

如果你的内存足够大，可以把文件IO的BUF设置大一些，这样每次你用fopen/fread/fwrite/fscanf/fprintf语句的时候，都会在内存里操作，减少内存到磁盘IO读写的操作次数，提高系统效率。  
  
如果你的程序的功能涉及到类似数据库、视频、音频、图像处理等大量需要爆发式磁盘到内存的IO情况下，可以考虑用setvbuf进行[优化内存](https://www.baidu.com/s?wd=%E4%BC%98%E5%8C%96%E5%86%85%E5%AD%98&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dbPH-9nyNbujDvn10YmWms0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnH6vn1n4PjTk)IO，其他情况下可以不考虑，LINUX/WINDOWS会自动处理这些问题。

功 能: 把缓冲区与流相关  
　　用 法: int setvbuf([FILE](https://www.baidu.com/s?wd=FILE&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWPWTLmW0vuyn1PvNWmWIh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWTLnWc1rj6z) \*stream, char \*buf, int type, unsigned size);  
　　参数：stream ：指向流的指针 ；  
　　buf ： 期望缓冲区的地址；  
　　type ： 期望缓冲区的类型：  
　　\_IOFBF(满缓冲）：当缓冲区为空时，从流读入数据。或者当缓冲区满时，向流写入数 据。  
　　\_IOLBF(行缓冲）：每次从流中读入一行数据或向流中写入一行数据。  
　　\_IONBF(无缓冲）：直接从流中读入数据或直接向流中写入数据，而没有缓冲区。  
　　size ： 缓冲区内字节的数量。   
<http://baike.baidu.com/view/1983906.htm>   
if (setvbuf(input, bufr, \_IOFBF, 512) != 0)   
是设置 input这个文件流使用 bufr 所指的512个字节作为 input文件的buffer, 当你操作input文件时，数据都会暂存在 bufr 里面,每次读input时，系统会一次性读512字节到bufr里暂存。  
 if (setvbuf(output, [NULL](https://www.baidu.com/s?wd=NULL&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWPWTLmW0vuyn1PvNWmWIh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWTLnWc1rj6z), \_IOLBF, 132) != 0)   
是设置output文件的buffer，这个buffer会由系统自行调用malloc来申请，buffer是按行模式工作的，每行最大132个字节，也就是当你写output的时候，每写完一行(遇见\n），就会把整行真正的写到磁盘文件上，在遇到[换行符](https://www.baidu.com/s?wd=%E6%8D%A2%E8%A1%8C%E7%AC%A6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dWPWTLmW0vuyn1PvNWmWIh0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnWTLnWc1rj6z)前，都咱存在系统自动申请的buffer中，而不会写到真正的磁盘文件上。