# [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;

}