浅谈 HOOK 技术在 VC 编程中的应用　　选择自 lithe 的 Blog
关键字　浅谈 HOOK 技术在 VC 编程中的应用
出处

摘要：本文针对 HOOK 技术在 VC 编程中的应用进行讨论，并着重对应用比较广泛的全局 HOOK 做了阐述。

引言

Windows 操作系统是建立在事件驱动机制之上的，系统各部分之间的沟通也都是通过消息的相互传递而实现的。但在通常情况下，应用程序只能处理来自进程内部的消息或是从其他进程发过来的消息，如果需要对在进程外传递的消息进行拦截处理就必须采取一种被称为 HOOK（钩子）的技术。钩子是 Windows 操作系统中非常重要的一种系统接口，用它可以轻松截获并处理在其他应用程序之间传递的消息，并由此可以完成一些普通应用程序难以实现的特殊功能。基于钩子在消息拦截处理中的强大功能，本文即以 VC++ 6.0 为编程背景对钩子的基本概念及其实现过程展开讨论。为方便理解，在文章最后还给出了一个简单的有关鼠标钩子的应用示例。

钩子的基本原理

钩子的本质是一段用以处理系统消息的程序，通过系统调用，将其挂入到系统。钩子的种类有很多，每一种钩子负责截获并处理相应的消息。钩子机制允许应用程序截获并处理发往指定窗口的消息或特定事件，其监视的窗口即可以是本进程内的也可以是由其他进程所创建的。在特定的消息发出，并在到达目的窗口之前，钩子程序先行截获此消息并得到对其的控制权。此时在钩子函数中就可以对截获的消息进行各种修改处理，甚至强行终止该消息的继续传递。

任何一个钩子都由系统来维护一个指针列表（钩子链表），其指针指向钩子的各个处理函数。最近安装的钩子放在链的开始，最早安装的钩子则放在最后，当钩子监视的消息出现时，操作系统调用链表开始处的第一个钩子处理函数进行处理，也就是说最后加入的钩子优先获得控制权。在这里提到的钩子处理函数必须是一个回调函数（callback function），而且不能定义为类成员函数，必须定义为普通的 C 函数。在使用钩子时可以根据其监视范围的不同将其分为全局钩子和线程钩子两大类，其中线程钩子只能监视某个线程，而全局钩子则可对在当前系统下运行的所有线程进行监视。显然，线程钩子可以看作是全局钩子的一个子集，全局钩子虽然功能强大但同时实现起来也比较烦琐：其钩子函数的实现必须封装在动态链接库中才可以使用。

钩子的安装与卸载

由于全局钩子具有相当的广泛性而且在功能上完全覆盖了线程钩子，因此下面就主要对应用较多的全局钩子的安装与使用进行讨论。前面已经提过，操作系统是通过调用钩子链表开始处的第一个钩子处理函数而进行消息拦截处理的。因此，为了设置钩子，只需将回调函数放置于链首即可，操作系统会使其首先被调用。在具体实现时由函数 SetWindowsHookEx() 负责将回调函数放置于钩子链表的开始位置。SetWindowsHookEx()函数原型声明如下：

```
HHOOK SetWindowsHookEx(int idHook;
HOOKPROC lpfn;
```

火龙果•整理
uml.org.cn

```
HINSTANCE hMod;
DWORD dwThreadId);
```

其中：参数 **idHook** 指定了钩子的类型，总共有如下 **13** 种：

WH_CALLWNDPROC 系统将消息发送到指定窗口之前的"钩子"
WH_CALLWNDPROCRET 消息已经在窗口中处理的"钩子"
WH_CBT 基于计算机培训的"钩子"
WH_DEBUG 差错"钩子"
WH_FOREGROUNDIDLE 前台空闲窗口"钩子"
WH_GETMESSAGE 接收消息投递的"钩子"
WH_JOURNALPLAYBACK 回放以前通过 **WH_JOURNALRECORD**"钩子"记录的输入消息
WH_JOURNALRECORD 输入消息记录"钩子"
WH_KEYBOARD 键盘消息"钩子"
WH_MOUSE 鼠标消息"钩子"
WH_MSGFILTER 对话框、消息框、菜单或滚动条输入消息"钩子"
WH_SHELL 外壳"钩子"
WH_SYSMSGFILTER 系统消息"钩子"

参数 **lpfn** 为指向钩子处理函数的指针，即回调函数的首地址；参数 **hMod** 则标识了钩子处理函数所处模块的句柄；第四个参数 **dwThreadId** 指定被监视的线程，如果明确指定了某个线程的 **ID** 就只监视该线程，此时的钩子即为线程钩子；如果该参数被设置为 **0**，则表示此钩子为监视系统所有线程的全局钩子。此函数在执行完后将返回一个钩子句柄。

虽然对于线程钩子并不要求其象全局钩子一样必须放置于动态链接库中，但是推荐其也在动态链接库中实现。因为这样的处理不仅可使钩子可为系统内的多个进程访问，也可以在系统中被直接调用，而且对于一个只供单进程访问的钩子，还可以将其钩子处理过程放在安装钩子的同一个线程内，此时 **SetWindowsHookEx()**函数的第三个参数也就是该线程的实例句柄。

在 **SetWindowsHookEx()**函数完成对钩子的安装后，如果被监视的事件发生，系统马上会调用位于相应钩子链表开始处的钩子处理函数进行处理，每一个钩子处理函数在进行相应的处理时都要考虑是否需要把事件传递给下一个钩子处理函数。如果要传递，就通过函数 **CallNestHookEx()**来解决。尽管如此，在实际使用时还是强烈推荐无论是否需要事件传递而都在过程的最后调用一次 **CallNextHookEx( )**函数，否则将会引起一些无法预知的系统行为或是系统锁定。该函数将返回位于钩子链表中的下一个钩子处理过程的地址，至于具体的返回值类型则要视所设置的钩子类型而定。该函数的原型声明如下：

```
LRESULT CallNextHookEx(HHOOK hhk;int nCode;WPARAM wParam;LPARAM lParam);
```

其中，参数 **hhk** 为由 **SetWindowsHookEx()**函数返回的当前钩子句柄；参数 **nCode** 为传给钩子过程的事件代码；参数 **wParam** 和 **lParam** 则为传给钩子处理函数的参数值，其具体含义同设置的钩子类型有关。

最后，由于安装钩子对系统的性能有一定的影响，所以在钩子使用完毕后应及时将其卸载以释放其所占资源。释放钩子的函数为 **UnhookWindowsHookEx()**，该函数比较简单只有一个参数用于指定此前由 **SetWindowsHookEx()**函数所返回的钩子句柄，原型声明如下：

火龙果•整理
uml.org.cn

```
BOOL UnhookWindowsHookEx(HHOOK hhk);
```

鼠标钩子的简单示例

最后，为更清楚展示 HOOK 技术在 VC 编程中的应用，给出一个有关鼠标钩子使用的简单示例。在钩子设置时采用的是全局钩子。下面就对鼠标钩子的安装、使用以及卸载等过程的实现进行讲述：

由于本例程需要使用全局钩子，因此首先构造全局钩子的载体--动态链接库。考虑到 Win32 DLL 与 Win16 DLL 存在的差别，在 Win32 环境下要在多个进程间共享数据，就必须采取一些措施将待共享的数据提取到一个独立的数据段，并通过 def 文件将其属性设置为读写共享：

```
#pragma data_seg("TestData")
HWND glhPrevTarWnd=NULL; // 窗口句柄
HWND glhHook=NULL; // 鼠标钩子句柄
HINSTANCE glhInstance=NULL; // DLL 实例句柄
#pragma data_seg()
……
SECTIONS // def 文件中将数据段 TestData 设置为读写共享
TestData READ WRITE SHARED
```

在安装全局鼠标钩子时使用函数 SetWindowsHookEx()，并设定鼠标钩子的处理函数为 MouseProc()，安装函数返回的钩子句柄保存于变量 glhHook 中：

```
void StartHook(HWND hWnd)
{
……
glhHook=(HWND)SetWindowsHookEx(WH_MOUSE,MouseProc,glhInstance,0);
}
```

鼠标钩子安装好后，在移动、点击鼠标时将会发出鼠标消息，这些消息均经过消息处理函数 MouseProc()的拦截处理。在此，每当捕获到系统各线程发出的任何鼠标消息后首先获取当前鼠标所在位置下的窗口句柄，并进一步通过 GetWindowText()函数获取到窗口标题。在处理函数完成后，通过 CallNextHookEx()函数将事件传递到钩子列表中的下一个钩子处理函数：

```
LRESULT WINAPI MouseProc(int nCode,WPARAM wParam,LPARAM lParam)
{
LPMOUSEHOOKSTRUCT pMouseHook=(MOUSEHOOKSTRUCT FAR *) lParam;
if(nCode>=0)
{
HWND glhTargetWnd=pMouseHook->hwnd;
//取目标窗口句柄
```

火龙果•整理
uml.org.cn

```
HWND ParentWnd=glhTargetWnd;
while(ParentWnd !=NULL)
{
glhTargetWnd=ParentWnd;
//取应用程序主窗口句柄
ParentWnd=GetParent(glhTargetWnd);
}
if(glhTargetWnd!=glhPrevTarWnd)
{
char szCaption[100];
//取目标窗口标题
GetWindowText(glhTargetWnd,szCaption,100);
……
}
}
//继续传递消息
return CallNextHookEx((HHOOK)glhHook,nCode,wParam,lParam);
}
```

最后，调用 **UnhookWindowsHookEx()**函数完成对钩子的卸载：

```
void StopHook()
{
……
UnhookWindowsHookEx((HHOOK)glhHook);
}
```

现在完成的是鼠标钩子的动态链接库，经过编译后需要经应用程序的调用才能实现对当前系统下各线程间鼠标消息的拦截处理。这部分同普通动态链接库的使用没有任何区别，在将其加载到进程后，首先调用动态链接库的 **StartHook()**函数安装好钩子，此时即可对系统下的鼠标消息实施拦截处理，在动态链接库被卸载即终止鼠标钩子时通过动态链接库中的 **StopHook()**函数卸载鼠标钩子。

经上述编程，在安装好鼠标钩子后，鼠标在移动到系统任意窗口上时，马上就会通过对鼠标消息的拦截处理而获取到当前窗口的标题。实验证明此鼠标钩子的安装、使用和卸载过程是正确的。

小结

钩子，尤其是系统钩子具有相当强大的功能，通过这种技术可以对几乎所有的 **Windows** 系统消息和事件进行拦截处理。这种技术广泛应用于各种自动监控系统对进程外消息的监控处理。本文只对钩子的一些基本原理和一般的使用方法做了简要的探讨，感兴趣的读者完全可以在本文所述代码基础之上用类似的方法实现对诸如键盘钩子、外壳钩子等其他类型钩子的安装与使用。本文所述代码在 **Windows 98** 下由 **Microsoft Visual C++ 6.0** 编译通过。

火龙果•整理
uml.org.cn

2005-3-6
HOOK 专题 [转]

HOOK 专题

-

目录
基本概念
运行机制
钩子类型
作者

-

基本概念
钩子(Hook)，是 Windows 消息处理机制的一个平台,应用程序可以在上面设置子程以监视指定窗口的某种消息，而且所监视的窗口可以是其他进程所创建的。当消息到达后，在目标窗口处理函数之前处理它。钩子机制允许应用程序截获处理 window 消息或特定事件。
钩子实际上是一个处理消息的程序段，通过系统调用，把它挂入系统。每当特定的消息发出，在没有到达目的窗口前，钩子程序就先捕获该消息，亦即钩子函数先得到控
制权。这时钩子函数即可以加工处理（改变）该消息，也可以不作处理而继续传递该消息，还可以强制结束消息的传递。

-

运行机制
1、钩子链表和钩子子程：
每一个 Hook 都有一个与之相关联的指针列表，称之为钩子链表，由系统来维护。这个列表的指针指向指定的，应用程序定义的，被 Hook 子程调用的回调函数，也就是该钩子的各个处理子程。当与指定的 Hook 类型关联的消息发生时，系统就把这个消息传递到 Hook 子程。一些 Hook 子程可以只监视消息，或者修改消息，或者停止消息的前进，避免这些消息传递到下一个 Hook 子程或者目的窗口。最近安装的钩子放在链的开始，而最早安装的钩子放在最后，也就是后加入的先获得控制权。
Windows 并不要求钩子子程的卸载顺序一定得和安装顺序相反。每当有一个钩子被卸载，Windows 便释放其占用的内存，并更新整个 Hook 链表。如果程序安装了钩子，但是在尚未卸载钩子之前就结束了，那么系统会自动为它做卸载钩子的操作。
钩子子程是一个应用程序定义的回调函数(CALLBACK Function),不能定义成某个类的成员函数，只能定义为普通的 C 函数。用以监视系统或某一特定类型的事件，这些事件可以是与某一特定线程关联的，也可以是系统中所有线程的事件。

火龙果•整理
uml.org.cn

钩子子程必须按照以下的语法：

```
LRESULT CALLBACK HookProc
(
  int nCode,
      WPARAM wParam,
      LPARAM lParam
);
```

HookProc 是应用程序定义的名字。

nCode 参数是 Hook 代码，Hook 子程使用这个参数来确定任务。这个参数的值依赖于 Hook 类型，每一种 Hook 都有自己的 Hook 代码特征字符集。

wParam 和 lParam 参数的值依赖于 Hook 代码，但是它们的典型值是包含了关于发送或者接收消息的信息。

2、钩子的安装与释放：

使用 API 函数 SetWindowsHookEx()把一个应用程序定义的钩子子程安装到钩子链表中。SetWindowsHookEx 函数总是在 Hook 链的开头安装 Hook 子程。当指定类型的 Hook 监视的事件发生时，系统就调用与这个 Hook 关联的 Hook 链的开头的 Hook 子程。每一个 Hook 链中的 Hook 子程都决定是否把这个事件传递到下一个 Hook 子程。Hook 子程传递事件到下一个 Hook 子程需要调用 CallNextHookEx 函数。

```
HHOOK SetWindowsHookEx(
          int idHook,       // 钩子的类型，即它处理的消息类型
          HOOKPROC lpfn,    // 钩子子程的地址指针。如果 dwThreadId 参数为 0
   // 或是一个由别的进程创建的线程的标识，
   // lpfn 必须指向 DLL 中的钩子子程。
   // 除此以外，lpfn 可以指向当前进程的一段钩子子程代码。
   // 钩子函数的入口地址，当钩子钩到任何消息后便调用这个函数。
          HINSTANCE hMod,  // 应用程序实例的句柄。标识包含 lpfn 所指的子程的
DLL。
   // 如果 dwThreadId 标识当前进程创建的一个线程，
   // 而且子程代码位于当前进程，hMod 必须为 NULL。
   // 可以很简单的设定其为本应用程序的实例句柄。
          DWORD dwThreadId // 与安装的钩子子程相关联的线程的标识符。
   // 如果为 0，钩子子程与所有的线程关联，即为全局钩子。
                    ;
```

函数成功则返回钩子子程的句柄，失败返回 NULL。

以上所说的钩子子程与线程相关联是指在一钩子链表中发给该线程的消息同时发送给钩子子程，且被钩子子程先处理。

在钩子子程中调用得到控制权的钩子函数在完成对消息的处理后，如果想要该消息继续传递，那么它必须调用另外一个 SDK 中的 API 函数 CallNextHookEx 来传递它，以执行钩子链表所指的下一个钩子子程。这个函数成功时返回钩子链中下一个钩子过程的返回值，返回值的类型依赖于钩子的类型。这个函数的原型如下：

```
LRESULT CallNextHookEx
(
```

火龙果•整理
uml.org.cn

```
HHOOK hhk;
int nCode;
WPARAM wParam;
LPARAM lParam;
 ;
```

hhk 为当前钩子的句柄，由 SetWindowsHookEx()函数返回。
NCode 为传给钩子过程的事件代码。
wParam 和 lParam 分别是传给钩子子程的 wParam 值，其具体含义与钩子类型有关。
钩子函数也可以通过直接返回 TRUE 来丢弃该消息，并阻止该消息的传递。否则的话，其他安装了钩子的应用程序将不会接收到钩子的通知而且还有可能产生不正确的结果。
钩子在使用完之后需要用 UnHookWindowsHookEx()卸载，否则会造成麻烦。释放钩子比较简单，UnHookWindowsHookEx()只有一个参数。函数原型如下：

```
UnHookWindowsHookEx
(
HHOOK hhk;
);
```

函数成功返回 TRUE，否则返回 FALSE。
3、一些运行机制：
在 Win16 环境中，DLL 的全局数据对每个载入它的进程来说都是相同的；而在 Win32 环境中，情况却发生了变化，DLL 函数中的代码所创建的任何对象（包括变量）都归调用它的线程或进程所有。当进程在载入 DLL 时，操作系统自动把 DLL 地址映射到该进程的私有空间，也就是进程的虚拟地址空间，而且也复制该 DLL 的全局数据的一份拷贝到该进程空间。也就是说每个进程所拥有的相同的 DLL 的全局数据，它们的名称相同，但其值却并不一定是相同的，而且是互不干涉的。
因此，在 Win32 环境下要想在多个进程中共享数据，就必须进行必要的设置。在访问同一个 Dll 的各进程之间共享存储器是通过存储器映射文件技术实现的。也可以把这些需要共享的数据分离出来，放置在一个独立的数据段里，并把该段的属性设置为共享。必须给这些变量赋初值，否则编译器会把没有赋初始值的变量放在一个叫未被初始化的数据段中。
#pragma data_seg 预处理指令用于设置共享数据段。例如：

```
#pragma data_seg("SharedDataName"
HHOOK hHook=NULL;
#pragma data_seg()
```

在#pragma data_seg("SharedDataName"和#pragma data_seg()之间的所有变量将被访问该 Dll 的所有进程看到和共享。再加上一条指令 #pragma comment(linker,"/section:.SharedDataName,rws",那么这个数据节中的数据可以在所有 DLL 的实例之间共享。所有对这些数据的操作都针对同一个实例的，而不是在每个进程的地址空间中都有一份。
当进程隐式或显式调用一个动态库里的函数时，系统都要把这个动态库映射到这个进程的虚拟地址空间里(以下简称"地址空间"。这使得 DLL 成为进程的一部分，以这个进程的身份执行，使用这个进程的堆栈。
4、系统钩子与线程钩子：

火龙果•整理
uml.org.cn

SetWindowsHookEx()函数的最后一个参数决定了此钩子是系统钩子还是线程钩子。
线程勾子用于监视指定线程的事件消息。线程勾子一般在当前线程或者当前线程派生的线程内。
系统勾子监视系统中的所有线程的事件消息。因为系统勾子会影响系统中所有的应用程序，所以勾子函数必须放在独立的动态链接库(DLL) 中。系统自动将包含"钩子回调函数"的 DLL 映射到受钩子函数影响的所有进程的地址空间中，即将这个 DLL 注入了那些进程。
几点说明：
（1）如果对于同一事件（如鼠标消息）既安装了线程勾子又安装了系统勾子，那么系统会自动先调用线程勾子，然后调用系统勾子。
（2）对同一事件消息可安装多个勾子处理过程，这些勾子处理过程形成了勾子链。当前勾子处理结束后应把勾子信息传递给下一个勾子函数。
（3）勾子特别是系统勾子会消耗消息处理时间，降低系统性能。只有在必要的时候才安装勾子，在使用完毕后要及时卸载。

-

钩子类型
每一种类型的 Hook 可以使应用程序能够监视不同类型的系统消息处理机制。下面描述所有可以利用的 Hook 类型。
1、WH_CALLWNDPROC 和 WH_CALLWNDPROCRET Hooks
WH_CALLWNDPROC 和 WH_CALLWNDPROCRET Hooks 使你可以监视发送到窗口过程的消息。系统在消息发送到接收窗口过程之前调用 WH_CALLWNDPROC Hook 子程，并且在窗口过程处理完消息之后调用 WH_CALLWNDPROCRET Hook 子程。
WH_CALLWNDPROCRET Hook 传递指针到 CWPRETSTRUCT 结构，再传递到 Hook 子程。
CWPRETSTRUCT 结构包含了来自处理消息的窗口过程的返回值，同样也包括了与这个消息关联的消息参数。
2、WH_CBT Hook
在以下事件之前，系统都会调用 WH_CBT Hook 子程，这些事件包括：
1. 激活，建立，销毁，最小化，最大化，移动，改变尺寸等窗口事件；
2. 完成系统指令；
3. 来自系统消息队列中的移动鼠标，键盘事件；
4. 设置输入焦点事件；
5. 同步系统消息队列事件。
Hook 子程的返回值确定系统是否允许或者防止这些操作中的一个。
3、WH_DEBUG Hook
在系统调用系统中与其他 Hook 关联的 Hook 子程之前，系统会调用 WH_DEBUG Hook 子程。你可以使用这个 Hook 来决定是否允许系统调用与其他 Hook 关联的 Hook 子程。
4、WH_FOREGROUNDIDLE Hook
当应用程序的前台线程处于空闲状态时，可以使用 WH_FOREGROUNDIDLE Hook 执行低优先级的任务。当应用程序的前台线程大概要变成空闲状态时，系统就会调用 WH_FOREGROUNDIDLE Hook 子程。
5、WH_GETMESSAGE Hook
应用程序使用 WH_GETMESSAGE Hook 来监视从 GetMessage or PeekMessage 函数返回的消息。

火龙果•整理
uml.org.cn

你可以使用 WH_GETMESSAGE Hook 去监视鼠标和键盘输入，以及其他发送到消息队列中的消息。

6、WH_JOURNALPLAYBACK Hook

WH_JOURNALPLAYBACK Hook 使应用程序可以插入消息到系统消息队列。可以使用这个 Hook 回放通过使用 WH_JOURNALRECORD Hook 记录下来的连续的鼠标和键盘事件。只要 WH_JOURNALPLAYBACK Hook 已经安装，正常的鼠标和键盘事件就是无效的。

WH_JOURNALPLAYBACK Hook 是全局 Hook，它不能象线程特定 Hook 一样使用。

WH_JOURNALPLAYBACK Hook 返回超时值，这个值告诉系统在处理来自回放 Hook 当前消息之前需要等待多长时间（毫秒）。这就使 Hook 可以控制实时事件的回放。

WH_JOURNALPLAYBACK 是 system-wide local hooks，它們不會被注射到任何行程位址空間。

7、WH_JOURNALRECORD Hook

WH_JOURNALRECORD Hook 用来监视和记录输入事件。典型的，可以使用这个 Hook 记录连续的鼠标和键盘事件，然后通过使用 WH_JOURNALPLAYBACK Hook 来回放。

WH_JOURNALRECORD Hook 是全局 Hook，它不能象线程特定 Hook 一样使用。

WH_JOURNALRECORD 是 system-wide local hooks，它們不會被注射到任何行程位址空間。

8、WH_KEYBOARD Hook

在应用程序中，WH_KEYBOARD Hook 用来监视 WM_KEYDOWN and WM_KEYUP 消息，这些消息通过 GetMessage or PeekMessage function 返回。可以使用这个 Hook 来监视输入到消息队列中的键盘消息。

9、WH_KEYBOARD_LL Hook

WH_KEYBOARD_LL Hook 监视输入到线程消息队列中的键盘消息。

10、WH_MOUSE Hook

WH_MOUSE Hook 监视从 GetMessage 或者 PeekMessage 函数返回的鼠标消息。使用这个 Hook 监视输入到消息队列中的鼠标消息。

11、WH_MOUSE_LL Hook

WH_MOUSE_LL Hook 监视输入到线程消息队列中的鼠标消息。

12、WH_MSGFILTER 和 WH_SYSMSGFILTER Hooks

WH_MSGFILTER 和 WH_SYSMSGFILTER Hooks 使我们可以监视菜单，滚动条，消息框，对话框消息并且发现用户使用 ALT+TAB or ALT+ESC 组合键切换窗口。WH_MSGFILTER Hook 只能监视传递到菜单，滚动条，消息框的消息，以及传递到通过安装了 Hook 子程的应用程序建立的对话框的消息。WH_SYSMSGFILTER Hook 监视所有应用程序消息。

WH_MSGFILTER 和 WH_SYSMSGFILTER Hooks 使我们可以在模式循环期间过滤消息，这等价于在主消息循环中过滤消息。

通过调用 CallMsgFilter function 可以直接的调用 WH_MSGFILTER Hook。通过使用这个函数，应用程序能够在模式循环期间使用相同的代码去过滤消息，如同在主消息循环里一样。

13、WH_SHELL Hook

外壳应用程序可以使用 WH_SHELL Hook 去接收重要的通知。当外壳应用程序是激活的并且当顶层窗口建立或者销毁时，系统调用 WH_SHELL Hook 子程。

WH_SHELL 共有 5 钟情况：

1. 只要有个 top-level、unowned 窗口被产生、起作用、或是被摧毁；
2. 当 Taskbar 需要重画某个按钮；
3. 当系统需要显示关于 Taskbar 的一个程序的最小化形式；
4. 当目前的键盘布局状态改变；
5. 当使用者按 Ctrl+Esc 去执行 Task Manager（或相同级别的程序）。

火龙果•整理
uml.org.cn

按照惯例，外壳应用程序都不接收 **WH_SHELL** 消息。所以，在应用程序能够接收 **WH_SHELL** 消息之前，应用程序必须调用 **SystemParametersInfo function** 注册它自己。

火龙果•整理
uml.org.cn