Microsoft Windows Hooks

Kyle Marsh
Microsoft Developer Network Technology Group

Created: March 20, 1992
Revised: July 16, 1992

Abstract

This article explains the Microsoft® Windows™ graphical environment hooks and how to use them. Topics include Windows hook functions, filter functions, and types of hooks:

- WH_CALLWNDPROC
- WH_CBT
- WH_DEBUG
- WH_GETMESSAGE
- WH_HARDWARE
- WH_JOURNALRECORD
- WH_JOURNALPLAYBACK
- WH_KEYBOARD
- WH_MOUSE
- WH_MSGFILTER
- WH_SYSMSGFILTER

Introduction

In the Microsoft® Windows™ graphical environment, a hook is a mechanism by which a function can intercept events (messages, mouse actions, keystrokes) before they reach an application. The function can act on events and, in some cases, modify or discard them. Functions that receive events are called *filter functions* and are classified according to the type of event they intercept. For example, a filter function might want to receive all keyboard or mouse events. For Windows to call a filter function, the filter function must be installed—that is, attached—to a Windows hook (for example, to a keyboard hook). Attaching one or more filter functions to a hook is known as *setting* a hook. If a hook has more than one filter function attached, a chain of filter functions is maintained. The most recently installed (MRI) function is at the beginning of the chain, and the least recently installed (LRI) function is at the end.

When a hook has one or more filter functions attached and an event occurs that triggers the hook, Windows calls the first filter function in the filter function chain. This action is known as *calling* the hook. For example, if a filter function is attached to the CBT hook and an event occurs that triggers the hook (such as a window is about to be created), Windows calls the CBT hook by calling the first function in the filter function chain. For a given hook type, task hooks are called first, followed by system hooks.

To maintain and access filter functions, applications use the **SetWindowsHookEx** and the **UnhookWindowsHookEx** functions. In earlier versions of Windows (before version 3.1), applications use the **SetWindowsHook** and **UnhookWindowsHook** functions.

Hooks are a powerful capability for Windows-based applications, which can use hooks to:

- Process or modify all messages meant for all the dialog boxes, message boxes, scroll bars, or menus for an application.
- Process or modify all messages meant for all the dialog boxes, message boxes, scroll bars, or menus for the system.
- Process or modify all messages, of any type, for the system whenever a **GetMessage** or a **PeekMessage** function is called.
- Process or modify all messages, of any type, whenever a **SendMessage** function is called.
- Record or play back keyboard and mouse events.
- Process, modify, or remove keyboard events.

Applications developed for Windows version 3.1 and later can use hooks to:

- Process, modify, or discard mouse events.
- Process, modify, or discard hardware events (other than mouse or keyboard events).
- Respond to certain system actions, making it possible to develop computer-based training (CBT) for applications.
- Prevent another filter from being called.

Applications have used hooks to:

- Provide F1 help key support to menus, dialog boxes, and message boxes.
- Provide mouse and keystroke record and playback features, often referred to as *macros*. For example, the Windows Recorder accessory program uses hooks to supply record and playback functionality.
- Monitor messages to determine which messages are being sent to a particular window or what actions a message generates. The Microsoft Windows Software Development Kit (SDK) Spy utility program uses hooks to do these tasks.
- Monitor the keyboard and mouse for use in screen-saving programs.
- Simulate mouse and keyboard input. Hooks are the only reliable way to simulate these activities. If you try to simulate these events by sending or posting messages, Windows internals do not update the keyboard or mouse state, which can lead to unexpected behavior. If hooks are used to play back keyboard or mouse events, these events are processed exactly like real keyboard or mouse events. Microsoft Excel uses hooks to implement its SEND.KEYS macro function.
- Provide CBT for applications that run in the Windows environment. In Windows version 3.1, the newly documented WH_CBT hook makes developing CBT applications much easier.

How to Use Hooks

To use hooks you need to know: (1) how to use the Windows hook functions to add and remove filter functions to and from a hook's filter function chain; (2) what action the filter function you are installing will be required to perform; (3) what kind of hooks are available, what they can do, and what information (parameters) they pass to your filter function.

Windows Hook Functions

The **SetWindowsHookEx**, **UnhookWindowsHookEx**, and **CallNextHookEx** functions were added to the API in Windows version 3.1 and later. Previous versions of Windows used **SetWindowsHook**, **UnhookWindowsHook**, and **DefHookProc** to manage the hooks filter function chain. **SetWindowsHook**, **UnhookWindowsHook**, and **DefHookProc** are still available in Windows version 3.1 and can be used to implement applications running on both versions 3.1 and 3.0 and any future 16-bit Windows-based implementations. In 32-bit Windows versions, these functions will be implemented as macros that call the **Ex** versions.

Windows Version 3.1 and Later

SetWindowsHookEx and **UnhookWindowsHookEx** are described below. See "Calling the next function in the filter function chain" for a discussion of **CallNextHookEx**.

SetWindowsHookEx

To add a filter function to a hook in Windows version 3.1 and later, you call the **SetWindowsHookEx** function, which takes four arguments:

- An integer code describing the hook to which to attach the filter function, and the address of the filter function. These codes are defined in WINDOWS.H and are described later.
- The address of the filter function, which must be the procedure-instance address of the filter function; how it is obtained depends on whether the filter function resides in an application or in a dynamic-link library (DLL). If the filter function resides in an application, use **MakeProcInstance**. If the filter function is in a DLL and the DLL is loaded at run time with **LoadLibrary**, use **GetProcAddress**. If the filter function is in a DLL that is linked to the application through an import library, use the procedure address directly. In any case, the filter function must be exported by including it in the **EXPORTS** statement in the module definition file for the application or the DLL.
- The instance handle of the module containing the filter function. This value may not be NULL.
- The task handle for which the hook is to be installed. If the task handle in not NULL, the installed filter function will be called only in the context of the specified task. If the task handle is NULL, the installed filter function has system scope and may be called in the context of any process or task in the system. An application or library can use the **GetCurrentTask** or the **GetWindowTask** function to obtain task handles for use in hooking a particular task.

  Some hooks may be set with system scope only; others may be set only for a specific task, as shown in the following list.

Hook	Scope
WH_CALLWNDPROC	Task or System
WH_CBT	Task or System
WH_DEBUG	Task or System

WH_GETMESSAGE	Task or System
WH_HARDWARE	Task or System
WH_JOURNALRECORD	System Only
WH_JOURNALPLAYBACK	System Only
WH_KEYBOARD	Task or System
WH_MOUSE	Task or System
WH_MSGFILTER	Task or System
WH_SYSMSGFILTER	System Only

For a given hook type, task hooks are called first, followed by system hooks.

It is a good idea to use task hooks instead of system hooks for several reasons. Task hooks:

- Do not incur a systemwide overhead in applications that are not interested in the call.
- Do not require packaging the filter function implementation in a separate DLL.
- Will continue to work even when future versions of Windows prevent applications from installing systemwide hooks for security reasons.

SetWindowsHookEx returns a handle to the installed hook (an HHOOK). This is a 32-bit handle unlike most other 16-bit Windows handles. The application or library must use this handle to identify this hook later when it calls the **CallNextHookEx** and **UnhookWindowsHookEx** functions. **SetWindowsHookEx** will return NULL if it is unable to add the filter function to the hook.

Windows version 3.1 and later versions keep the filter function chain internally (see Figure 1) and do not rely on the filter functions to store the address of the next filter function in the chain correctly (see the discussion of **SetWindowsHook** below). Thus, hooks are much more robust in Windows version 3.1 than they were in Windows version 3.0. In addition, performance is enhanced significantly because the filter function chain is kept internally.

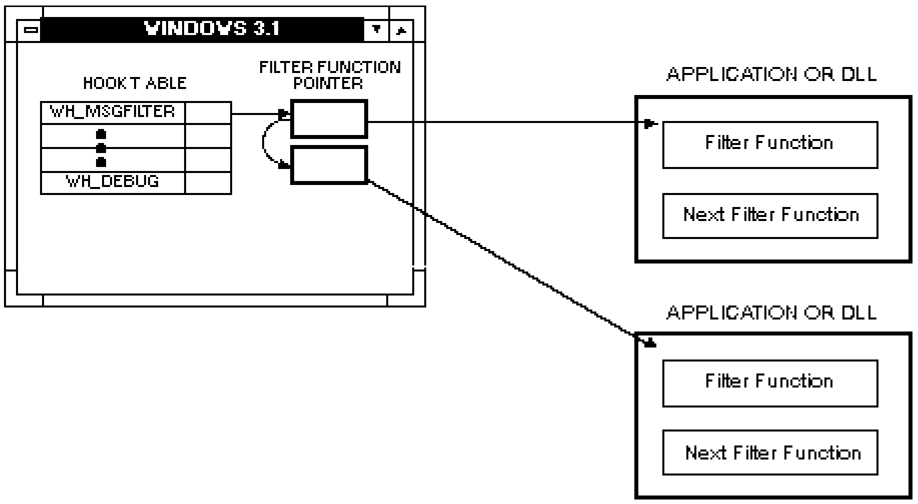

Figure 1.

UnhookWindowsHookEx

To remove a filter function from a hook's chain, call the **UnhookWindowsHookEx** function, which takes the hook handle returned from **SetWindowsHookEx** and returns a value indicating whether the hook is removed.

Windows Versions Before 3.1

SetWindowsHook and **UnhookWindowsHook** are described below. See "Calling the next function in the filter function chain" for a discussion of **DefHookProc**.

SetWindowsHook

To add a filter function to a hook in Windows version 3.0, you call the **SetWindowsHook** function, which takes two arguments: an integer code describing the hook to which to attach the filter function, and the address of the filter function. These arguments are the same as the first two arguments to **SetWindowsHookEx** described previously.

SetWindowsHook returns the address of the filter function that was at the beginning of the filter function chain before the latest filter function was added. If no filter function was on the chain previously, **SetWindowsHook** returns NULL. If other filter functions were attached to the hook, **SetWindowsHook** returns the address of the first function in the filter function chain. The return value is always non-null for the WH_MSGFILTER and WH_KEYBOARD hook types, even when the filter function being installed is the first in the chain.

Store the address that **SetWindowsHook** returns in a static or a global variable in the application or in a DLL that contains the filter function. The address of the next filter function is needed later in calls to **DefHookProc**. Windows version 3.0 uses pointers to the static variables that contain the addresses of the next filter function in the filter function chain and **DefHookProc** (see below) to maintain the filter function chain. If the value returned from **SetWindowsHook** is not properly stored in applications running under Windows version 3.0, Windows is unable to maintain the filter function chain. Windows must know these values to maintain the filter function chain (see Figure 2).

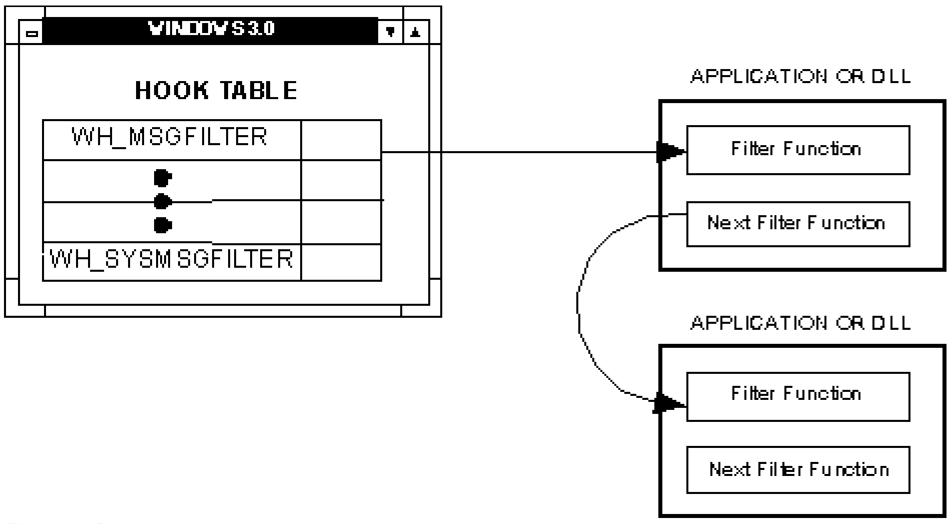

Figure 2.

UnhookWindowsHook

To remove a filter function from a hook's chain, call the **UnhookWindowsHook** function, which takes the same arguments as **SetWindowsHook** and returns a value indicating whether the hook is removed.

Filter Functions

Filter functions are sometimes called *hook functions* or *callback functions*. Hook functions refer to the function being attached to a hook. Because filter functions are called by Windows and not by an application, they are sometimes referred to as callback functions. For consistency, this article uses the term *filter functions.*

Filter functions must use the Pascal calling convention and be declared as far functions. All filter functions must have the following form:

```
DWORD FAR PASCAL FilterFunc( nCode, wParam, lParam )int nCode;
WORD wParam;
DWORD lParam;
```

All filter functions should return a **DWORD**. The Windows version 3.0 SDK documentation indicates that some filter functions return other data types (**VOID**, **INT**), which is not the case. Windows does not always check the filter function's return value (filter functions that the SDK documentation has marked as returning **VOID**) or may only want a particular range of values (marked as returning **INT**). *FilterFunc* is a placeholder for the actual filter function name.

Parameters

Filter functions receive three parameters: *ncode* (the hook code), *wParam*, and *lParam*. The hook code is an integer code that informs the filter function of any additional data it should know. For example, the hook code might indicate what action is causing the hook to be called.

In Windows version 3.0, the hook code also indicates whether the filter function should process the event or call **DefHookProc**. If the hook code is less than zero, the filter function should not process the event. If the hook code is a negative number, the filter function is required to call **DefHookProc**, passing the three parameters it was passed without any modification. Windows version 3.0 uses these negative codes to maintain the filter function chain. Whenever a filter function receives a negative hook code, it must call **DefHookProc**.

In Windows version 3.1, negative hook code values are never passed to a filter function. Passing them to **DefHookProc** (if the installation is using **SetWindowsHook** instead of **SetWindowsHookEx**) causes a **FatalExit** with the code 0x001F and the message "Invalid Hook Code." To run under Windows versions 3.0 and 3.1 (using **SetWindowsHook**), your application must react correctly to negative hook codes.

The second parameter passed to the filter function, *wParam*, is a **WORD**, and the third parameter, *lParam*, is a **DWORD**. These parameters pass information needed by the filter function. Each hook attaches different meanings to *wParam* and *lParam*. For example, filter functions attached to the WH_KEYBOARD hook receive a virtual-key code in *wParam*, and *lParam* contains bit fields that describe the state of the keyboard at the time of the key event. Filter functions attached to the WH_MSGFILTER hook receive a NULL value in *wParam* and a pointer to a message structure in *lParam*. Some hooks attach different meanings for *wParam* and *lParam* depending on the event that causes the hook to be called. For a complete list of what the arguments mean for each hook type, see the SDK *Reference—Volume 1* manual, under **SetWindowsHookEx** or **SetWindowsHook**.

Calling the next function in the filter function chain

When a hook is set, Windows calls the first function in the hook's filter function chain, and the responsibility of Windows ends. The filter function is responsible for ensuring that the next filter function in the chain is called. Windows version 3.1 supplies **CallNextHookEx** to enable a filter function to call the next filter in the filter function chain. **CallNextHookEx** takes four parameters. The first parameter is the value returned from the call to **SetWindowsHookEx**. The next three parameters —*nCode*, *wParam*, and *lParam*—are the parameters that Windows passed to the filter function.

Windows version 3.1 stores the filter function chain internally. When **CallNextHookEx** is called, Windows examines the handle to the hook passed in the first parameter and then calls the next filter function in the chain (see Figure 3).

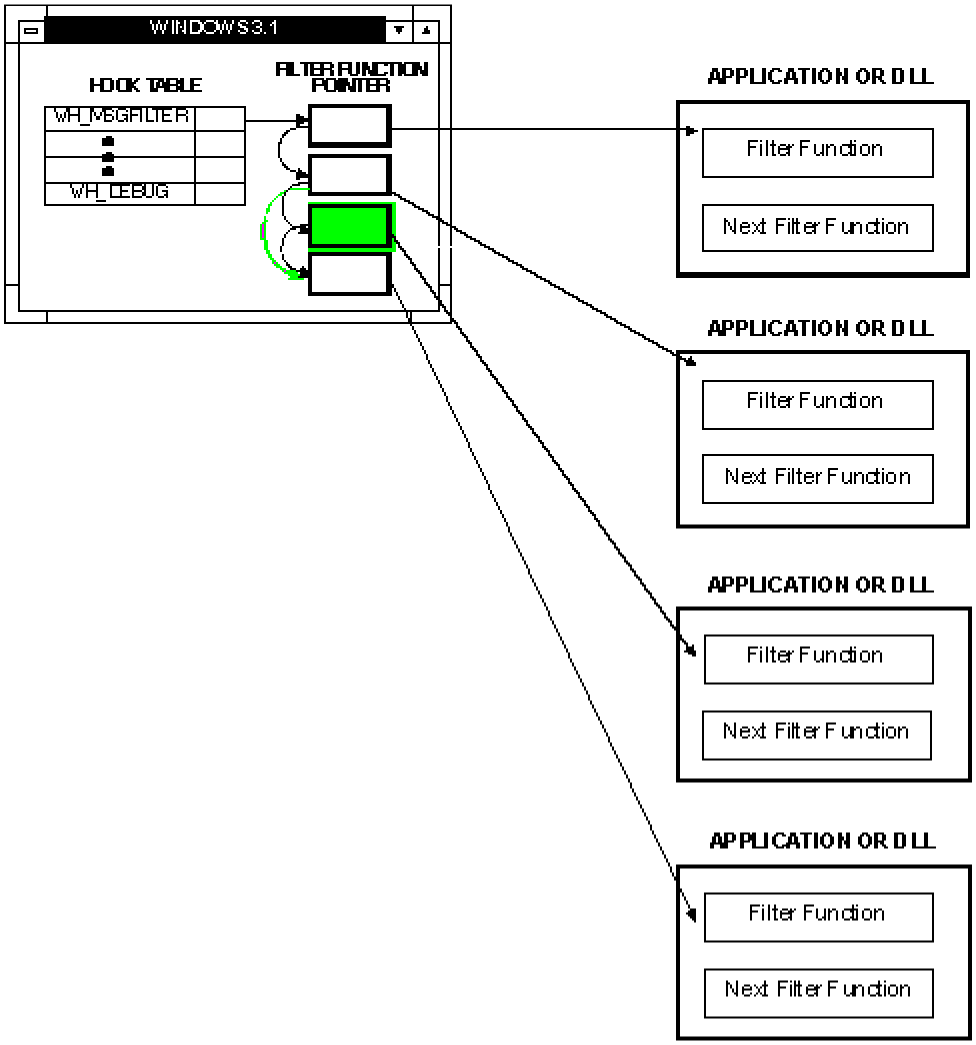

Figure 3.
Windows version 3.0 supplies **DefHookProc**, not **CallNextHookEx**. **DefHookProc** takes four parameters. The first three parameters—*nCode*, *wParam*, and *lParam*—are the parameters that Windows passed to the filter function. The last parameter—*lpfnNextHook*—is a pointer to the static or the global variable that contains the **DWORD** value returned by **SetWindowsHook**. Filter functions must store the value returned by **SetWindowsHook** in a static or a global variable and pass the address of the variable to **DefHookProc**.

Windows version 3.0 uses the address of the next function in the chain passed to **DefHookProc** to call the next filter function in the chain. When a filter function is removed from the middle of a filter function chain, Windows must update the value so that the chain is properly maintained. It does so by calling the filter function being unhooked with a code that tells it to call **DefHookProc** and not to process the event at all. **DefHookProc** gives the address of the variable that contains that function's next filter function to Windows. Windows then calls the first function in the filter function chain, which tells the function to call **DefHookProc** and not to process the event. **DefHookProc** compares the address of the function being removed with the address contained in the last argument passed to **DefHookProc**. If they are the same, **DefHookProc** returns to Windows the address of the pointer to the next filter function in the chain, and Windows sets the value of that pointer to the new next filter function in the chain. If the addresses are not the same, **DefHookProc** calls the next function in the filter function chain (see Figure 4).

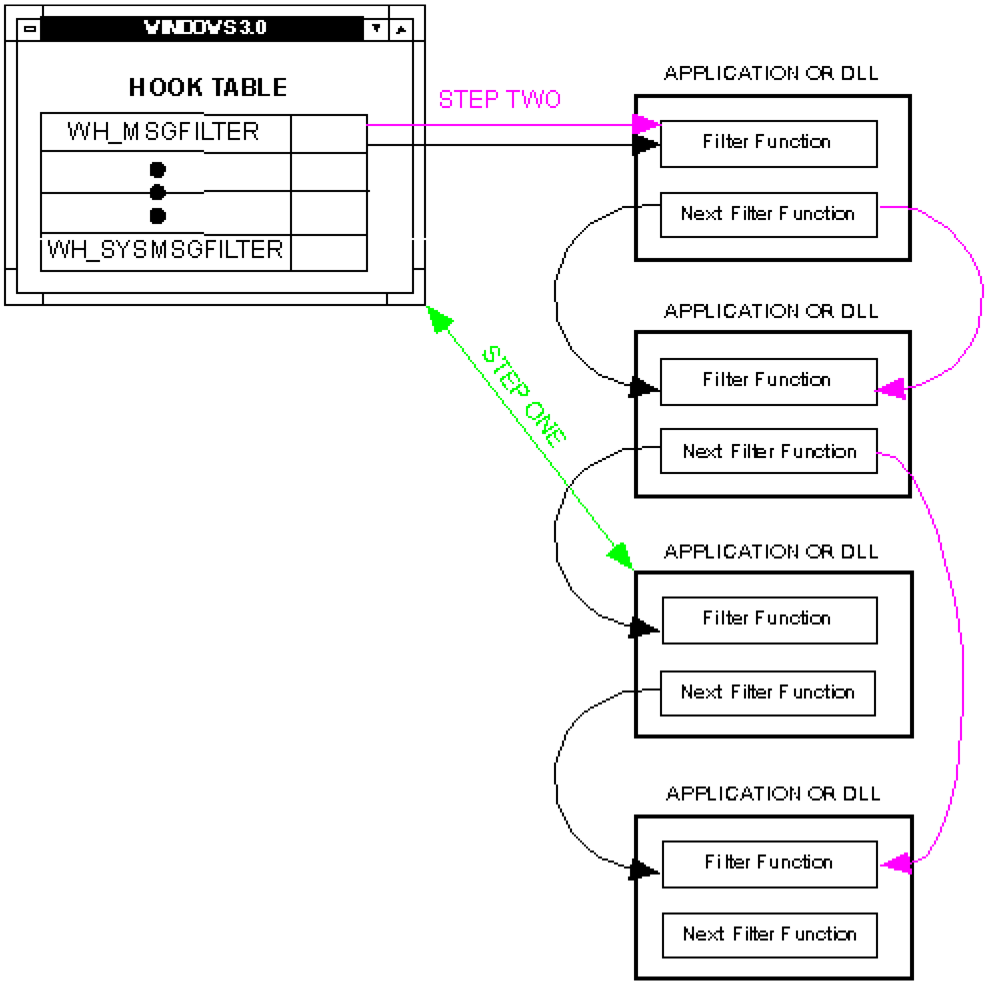

Figure 4.
Because Windows version 3.1 stores the filter function chain internally, when **DefHookProc** is called, it ignores the last parameter and uses the value it has stored internally to call the next function. As a result, the address of the next filter function in the filter function chain, which is stored in the filter function, can be out of date at any moment and should not be relied upon to have any particular meaning when running under Windows version 3.1 and later.

At times a filter function might not want to pass an event to the other hook functions on the same chain. In particular, when a hook allows a filter function to discard an event and it decides to do so, the filter function must not call **CallNextHookEx** (or **DefHookProc**). When a filter function modifies a message, it may or may not want to pass the message to the rest of the filter function chain.

Because filter functions are not installed in any particular order, you cannot be sure where your function is in the filter function chain at any moment except at the moment of installation, when it is the first function in the chain. As a result, you are never absolutely certain that you will get every event that occurs. A filter function installed ahead of your filter function in the chain—a function that was installed after your function timewise—might not pass the event to your filter function.

Filter functions in dynamic-link libraries

Filter functions (except for those attached to application-specific hooks) must reside in a dynamic-link library (DLL) for three reasons: to support Windows version 3.0 real mode, for compatibility with future versions of Windows, and because SS != DS in a DLL.

In real mode, Windows may swap entire inactive applications to expanded memory specification (EMS) memory. Because hooks are a systemwide resource, filter functions must be available at any time. Windows crashes by attempting to call a filter function that has been swapped to EMS memory. The only way to ensure that the filter function is always available is to place the function in a DLL, because DLLs are never swapped to EMS memory. Under Windows standard and enhanced modes, the only modes available in Windows version 3.1, the EMS problem does not exist. Filter functions can, therefore, reside in an application instead of in a DLL. Microsoft strongly recommends, however, that you continue to place filter functions in DLLs to be compatible with future versions of Windows. Although the EMS problem will not exist in future versions of Windows, the separation of address spaces among applications creates a similar problem. Filter functions in a DLL will run correctly in future versions of Windows; those in applications will not.

Another reason for filter functions to reside in DLLs is that when a filter function is called SS will not equal DS. Since SS != DS in all DLLs, programming for this situation will be more natural in a DLL.

Types of Hooks

WH_CALLWNDPROC

Windows calls this hook whenever the Windows **SendMessage** function is called. This call is made at the beginning of **SendMessage** before any task switching occurs. The filter functions receive a hook code from Windows indicating whether the message was sent from the current task and receive a pointer to a structure containing the actual message. Filters can process or modify the message, or both process and modify it. The message, including any modifications, is sent to the Windows function for which it was intended. Filters for this hook must reside in a DLL. Because using this hook is a significant drain on system performance, use it only as a development or debugging tool.

WH_CBT (Windows Version 3.1 Only)

To write a CBT application, the developer must coordinate the CBT application with the application for which it is written. Windows supplies the WH_CBT hook to make this possible. Windows passes a hook code to the filter function, indicating which event has occurred and the appropriate data for the event. Filters for this hook must reside in a DLL. This hook was present but not documented in Windows version 3.0. It is documented in Windows version 3.1 with added functionality that makes developing CBT applications much easier.

A filter function attached to the WH_CBT hook needs to be aware of the following 10 hook codes:

- HCBT_ACTIVATE
- HCBT_CREATEWND
- HCBT_DESTROYWND
- HCBT_MINMAX
- HCBT_MOVESIZE
- HCBT_SYSCOMMAND
- HCBT_CLICKSKIPPED
- HCBT_KEYSKIPPED
- HCBT_SETFOCUS

- HCBT_QS

HCBT_ACTIVATE (Windows Version 3.1 Only)

Windows calls the WH_CBT hook with this hook code when any window is about to be activated. If the filter function returns TRUE, the window is not activated.

The *wParam* parameter contains the handle to the window being activated. The *lParam* parameter contains a far pointer to **CBTACTIVATESTRUCT**, which has the following structure:

```
struct CBTACTIVATESTRUCT
{
   BOOL    fMouse;         // TRUE if activation results from a
                           //  mouse click; otherwise is FALSE
   HWND    hWndActive;     // contains the handle to the
                           //  currently active window
};
```

HCBT_CREATEWND (Windows Version 3.1 Only)

Windows calls the WH_CBT hook with this hook code when a window is about to be created. The WH_CBT hook is called after the WM_GETMINMAXINFO message and before the WM_NCCREATE or WM_CREATE message is sent to the window. Thus, the filter function can return TRUE and not allow the window to be created.

The *wParam* parameter contains the handle to the window being created. The *lParam* parameter contains a pointer to a structure of create parameters (**LPCREATESTRUCT**), which is the same as *lParam* in the WM_CREATE message.

HCBT_DESTROYWND (Windows Version 3.1 Only)

Windows calls the WH_CBT hook with this hook code when Windows is about to destroy any window. The WH_CBT hook is called before the WM_DESTROY message is sent. If the filter function returns TRUE, the window is not destroyed.

The *wParam* parameter contains the handle to the window being destroyed. The *lParam* parameter contains 0L.

HCBT_MINMAX (Windows Version 3.0 and Later)

Windows calls the WH_CBT hook with this hook code when Windows is about to minimize or maximize a window. If the filter function returns TRUE, the action does not occur.

The *wParam* parameter contains the handle to the window being minimized or maximized. The HIWORD of the *lParam* parameter is zero, and the LOWORD of the *lParam* parameter is any one of the SW_* values defined in WINDOWS.H indicating the operation that is taking place.

HCBT_MOVESIZE (Windows Version 3.0 and Later)

Windows calls the WH_CBT hook with this hook code when Windows is about to move or size any window and the user has just finished selecting the new position or size. If the filter function returns TRUE, the action does not occur.

The *wParam* parameter contains the handle to the window being moved or sized. The *lParam* parameter contains an **LPRECT** that points to the drag rectangle.

HCBT_SYSCOMMAND (Windows Version 3.1 Only)

Windows calls the WH_CBT hook with this hook code when Windows processes a system command. The WH_CBT hook is called from within **DefWindowsProc**. If an application does not send the WH_SYSCOMMAND message to **DefWindowsProc**, this hook is not called. If the filter function returns TRUE, the system command is not processed.

The *wParam* parameter contains the system command (SC_TASKLIST, SC_HOTKEY, and so on) that is about to be performed. If *wParam* is SC_HOTKEY, the LOWORD of *lParam* contains the handle to the window for which the hot key applies. If *wParam* contains any value other than SC_HOTKEY, the value of *lParam* is undefined.

The following system commands trigger this hook from within **DefWindowsProc**:

SC_CLOSE	Close the window
SC_HOTKEY	Activate the window associated with the application-specified hot key
SC_HSCROLL	Scroll horizontally
SC_KEYMENU	Retrieve a menu through a keystroke
SC_MAXIMIZE	Maximize the window
SC_MINIMIZE	Minimize the window
SC_MOUSEMENU	Retrieve a menu through a mouse click
SC_MOVE	Move the window
SC_NEXTWINDOW	Move to the next window
SC_PREVWINDOW	Move to the previous window

SC_RESTORE	Save the previous coordinates (checkpoint)
SC_SCREENSAVE	Execute the screen-save application
SC_SIZE	Size the window
SC_TASKLIST	Execute or activate the Windows Task Manager application
SC_VSCROLL	Scroll vertically

HCBT_CLICKSKIPPED (Windows Version 3.1 Only)

Windows calls the WH_CBT hook with this hook code when a mouse event is removed from the system queue and the mouse hook is set. This hook code is not generated unless a filter function is attached to the WH_MOUSE hook. Despite its name, HCBT_CLICKSKIPPED is called not only for skipped mouse events but also for each time a mouse event is removed from the system queue. Its main use is to install a WH_JOURNALPLAYBACK hook in response to a mouse event. (See the WM_QUEUESYNC section below for more information.)

The *wParam* parameter contains the message identifier for the mouse message—for example, the W_LBUTTONDOWN or any W_?BUTTON* messages. The *lParam* parameter contains a far pointer to **MOUSEHOOKSTRUCT**, which has the following structure:

```
struct MOUSEHOOKSTRUCT
  {
    POINT point;          // location of mouse in screen coordinates
    HWND hWnd;            // window that receives this message
    WORD wHitTestCode;    // the result of Hit testing (HT_*)
    DWORD dwExtraInfo;    // extra info associated withthe current
                          //  message
}
```

HCBT_KEYSKIPPED (Windows Version 3.1 Only)

Windows calls the WH_CBT hook with this hook code when a keyboard event is removed from the system queue and the keyboard hook is set. This hook code is not generated unless a filter function is attached to the WH_KEYBOARD hook. Despite its name, HCBT_KEYSKIPPED is called not only for skipped keyboard events but also for each time a keyboard event is removed from the system queue. Its main use is to install a WH_JOURNALPLAYBACK hook in response to a keyboard event. (See the WM_QUEUESYNC section below for more information.)

The *wParam* parameter contains the virtual-key code—the same value as *wParam* of **GetMessage** or **PeekMessage** for WM_KEY* messages. The *lParam* parameter contains the same value as the *lParam* parameter of **GetMessage** or **PeekMessage** for WM_KEY* messages.

WM_QUEUESYNC

While executing, a CBT application often must react to events in the main application. Keyboard or mouse events usually trigger these events. For example, a user clicks an OK button in a dialog box, after which the CBT application wants to play a series of keystrokes to the main application. To determine that the OK button was clicked, the CBT application can use a mouse hook. After determining that the CBT application wants to play some keystrokes to the main application, the CBT application must then wait until the main application completes the processing of the OK button before beginning to play the new keystrokes. The CBT application would not want to apply the keystrokes to the dialog box.

The CBT application uses the WM_QUEUESYNC message to monitor the main application to determine when an action is completed. The CBT application monitors the main application with a mouse or a keyboard hook. In the mouse or the keyboard hook, the CBT application can monitor the main application, looking for events to which the CBT application must respond. By watching the main application with a mouse or a keyboard hook, the CBT application is aware of when an event that needs a response begins. The CBT application must wait until the event is completed before responding to it.

To determine when the action is complete, the CBT application takes these steps. First, the CBT application waits until it receives the WH_CBT hook with an HCBT_CLICKSKIPPED or an HCBT_KEYSKIPPED hook code from Windows. This happens when the event that is causing the action in the main application is removed from the system queue. When the CBT application receives the HCBT_CLICKSKIPPED or the HCBT_KEYSKIPPED hook code, it can install a WH_JOURNALPLAYBACK hook. The CBT application cannot install the WH_JOURNALPLAYBACK hook until it receives either the HCBT_CLICKSKIPPED or the HCBT_KEYSKIPPED hook code. The WH_JOURNALPLAYBACK hook installed by the CBT application plays a WM_QUEUESYNC message to the CBT application. When the CBT application receives the WM_QUEUESYNC message, it can respond to the original event. For example, the CBT application might play some keystrokes to the main application.

HCBT_SETFOCUS (Windows Version 3.1 Only)

Windows calls the WH_CBT hook with this hook code when Windows is about to set the focus to any window. If the filter function returns TRUE, the focus does not change.

The *wParam* parameter contains the handle to the window that receives the focus. The LOWORD of the *lParam* parameter contains the handle to the window that loses the focus. The HIWORD of the *lParam* parameter contains NULL.

HCBT_QS (Windows Version 3.1 Only)

Windows calls the WH_CBT hook with this hook code when a WM_QUEUESYNC message is removed from the system queue while a window is being resized or moved. It does not occur at any other time.

Both the *wParam* parameter and the *lParam* parameter contain zero.

WH_DEBUG (Windows Version 3.1 Only)

Windows calls this hook when Windows is about to call a filter function. This function is meant for debuggers, and other applications should not use it. Filters for this hook must reside in a DLL.

WH_GETMESSAGE

Windows calls this hook when the **GetMessage** or the **PeekMessage** function is about to return a message. The filter functions receive from Windows a pointer to a structure containing the actual message. Filters can process or modify the message or both process and modify it, but cannot discard it. The message, including any modifications, is sent to the application that originally called **GetMessage** or **PeekMessage**. Filters for this hook must reside in a DLL.

WH_HARDWARE (Windows Version 3.1 Only)

In Windows version 3.1, the **hardware_event** function is called by input type device drivers for nonstandard hardware devices to place hardware-related messages in the system queue. Devices such as the "pens" in Windows for Pen Computing use the **hardware_event** function. Windows for Pen Computing needs to gather information from the pointing device in addition to that gathered for the keyboard and the mouse—for example, the speed of the pen, the proximity of the pen, or how hard the user is pressing the pen. Windows for Pen Computing needs these events in chronological order. The device driver can use the **hardware_event** function to add these new events to the system queue. The WH_HARDWARE hook processes these new events.

Windows calls this hook when the **GetMessage** or the **PeekMessage** function is called and an event placed in the system queue by a call to **hardware_event** is about to be processed. Mouse and keyboard events do not trigger this hook. The filter functions receive a pointer to a structure containing the hardware event. Filters can tell Windows to discard the event and must reside in a DLL.

WH_JOURNALRECORD

Windows calls this hook when Windows removes an event from the system queue. Thus, these filters are called for all mouse and keyboard events. This hook is also called when a system modal dialog box appears or is removed. Filters may process the message, which normally is to record or save the event in memory or on disk or both, but cannot modify or discard the message. Filters for this hook must reside in a DLL. A filter function attached to this hook needs to be aware of the following three hook codes:

- HC_ACTION
- HC_SYSMODALON
- HC_SYSMODALOFF

HC_ACTION

Windows calls the WH_JOURNALRECORD hook with this hook code when Windows takes an event from the system queue. It signals the filter function that this is a normal event. The *lParam* parameter to the filter function contains a pointer to an **EVENTMSG** structure. The usual recording procedure is to take all **EVENTMSG** structures passed to the hook and store them in memory or, if events exceed memory storage capacity, write them to disk.

EVENTMSG

The **EVENTMSG** structure is defined in WINDOWS.H and has the following structure:

```
typedef struct tagEVENTMSG
          {
       WORD    message;
       WORD    paramL;
       WORD    paramH;
       DWORD   time;
    } EVENTMSG;
```

The message element of the **EVENTMSG** structure is the message ID for the message, the WM_* value. The *paramL* and *paramH* values depend on whether the event is mouse or keyboard. If it is a mouse event, the values contain the *x*-coordinate and the *y*-coordinate of the event. If it is a keyboard event, *paramL* contains the scan code in the HIBYTE and the virtual-key code in the LOBYTE, and *paramH* contains the repeat count. Bit 15 of the repeat count specifies whether the event is an extended key. The time element of the **EVENTMSG** structure contains the system time (when the event occurred), which it obtained from the return value of **GetTickCount**.

The amount of time between events is determined by comparing the time element of an event with the subsequent events. This time delta is needed when playing back the recorded events.

HC_SYSMODALON

Windows calls the WH_JOURNALRECORD hook with this hook code when Windows brings up a system modal dialog box. All event recording should stop at this point. Windows does not call the WH_JOURNALRECORD hook while a system modal dialog box is up.

HC_SYSMODALOFF

Windows calls the WH_JOURNALRECORD hook with this hook code when Windows removes a system modal dialog box. Event recording, which ceased while the system modal dialog box was in place, can now resume.

WH_JOURNALPLAYBACK

This hook is used to provide mouse and keyboard messages to Windows as if they were inserted in the system queue. Usually this hook is used to play back events recorded with the WH_JOURNALRECORD hook. Whenever a filter function is attached to this hook, Windows calls the first filter function in the function chain to get events. Windows discards any mouse moves while WH_JOURNALPLAYBACK is installed. All other keyboard and mouse input is queued until the WH_JOURNALPLAYBACK hook has no filter functions attached. Filters for this hook must reside in a DLL. A filter function attached to this hook needs to be aware of the following four hook codes:

n HC_GETNEXT

n HC_SKIP

n HC_SYSMODALON

n HC_SYSMODALOFF

HC_GETNEXT

Windows calls the WH_JOURNALPLAYBACK hook with this hook code when Windows accesses the system queue. In most cases, Windows makes this call many times for the same message. The *lParam* parameter to the filter function contains a pointer to an **EVENTMSG** structure (see above). The filter function must put the message, the *paramL* value, and the *paramH* value into the **EVENTMSG** structure. These are usually copied directly from the recorded event made during WH_JOURNALRECORD.

The filter function must tell Windows when to process the message that the filter function is giving Windows. Windows needs two values for its scheduling: (1) the amount of time Windows should wait before processing the message; (2) the time at which the message is to be processed. The usual method of calculating the time to wait before processing is to subtract the **EVENTMSG** time element of the previous message from the **EVENTMSG** time element of the current message. This technique plays back messages at the speed at which they were recorded. If the message is to be processed immediately for much faster playback, the amount of time returned from the function is zero.

The time at which the message should be processed is usually obtained by adding the amount of time Windows should wait before processing the message to the current system time obtained from **GetTickCount**. For immediate playback, use the value returned from **GetTickCount**.

If the system is not otherwise active, Windows uses the values that the filter function has supplied to process the event. If the system is active, Windows examines the system queue. Each time it does, it asks for the same event with an HC_GETNEXT hook code. Each time the filter function receives HC_GETNEXT, it should return the new amount of time to wait, assuming that time elapsed between calls. The time element of the **EVENTMSG** structure and of the message, the *paramH* value, and the *paramL* value will probably not need changing between calls.

HC_SKIP

Windows calls the WH_JOURNALPLAYBACK hook with this hook code when Windows has completed processing a message it received from WH_JOURNALPLAYBACK. This occurs at the time that Windows would have removed the event from the system queue if the event was in the system queue rather than having been generated by a WH_JOURNALPLAYBACK hook. This hook code signals to the filter function that the event that the filter function returned on the prior HC_GETNEXT call has been returned to an application. The filter function should prepare to return the next event on the next HC_GETEVENT call. When the filter function determines that it has no more events to play back, it should unhook itself during this HC_SKIP call.

HC_SYSMODALON

Windows calls the WH_JOURNALPLAYBACK hook with this hook code when Windows brings up a system modal dialog box. The WH_JOURNALPLAYBACK hook is not called while a system modal dialog box is up. The filter function should probably remove itself if it receives this hook code.

HC_SYSMODALOFF

Windows calls the WH_JOURNALPLAYBACK hook with this hook code when Windows removes a system modal dialog box. At this time, the filter function could continue to play back events; however, because a system modal dialog box normally signals a severe error, it is unlikely that playback should continue.

WH_KEYBOARD

Windows calls this hook when the **GetMessage** or the **PeekMessage** function is about to return a

WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP, WM_SYSKEYDOWN, or WM_CHAR message. The filter function receives the virtual-key code and the state of the keyboard at the time of the keyboard hook. Filters can tell Windows to discard the message. Filters for this hook must reside in a DLL. A filter function attached to this hook needs to be aware of the following two hook codes:

- HC_ACTION
- HC_NOREMOVE

HC_ACTION

Windows calls the WH_KEYBOARD hook with this hook code when an event is being removed from the system queue.

HC_NOREMOVE

Windows calls the WH_KEYBOARD hook with this hook code when there is a keyboard event but it is not being removed because an application is calling **PeekMessage** with the PM_NOREMOVE option. If this hook code is passed, the key-state table will not accurately reflect the previous key state. An application needs to be aware of the existence of this condition.

WH_MOUSE (Windows Version 3.1 Only)

Windows calls this hook when a **GetMessage** or a **PeekMessage** function is called and Windows has a mouse message to process. Like the WH_KEYBOARD hook, this filter function receives a hook code, which indicates whether the message is being removed (HC_NOREMOVE), an identifier specifying the mouse message, and the *x*-coordinate and the *y*-coordinate of the mouse. Filters can tell Windows to discard the message. Filters for this hook must reside in a DLL.

WH_MSGFILTER

Windows calls this hook when a dialog box, a message box, a scroll bar, or a menu retrieves a message or when the user presses the ALT+TAB or ALT+ESC keys while the application that set the hook is active. This hook is task specific, so it is always safe for its filter functions to reside in an application or in a DLL. The filter receives a hook code indicating whether the message is for a dialog box or a message box (MSGF_DIALOGBOX), a scroll bar (MSGF_SCROLLBAR), a menu (MSGF_MENU), or the next window action (MSGF_NEXTWINDOW) about to take place. In Windows versions 3.0 and 3.1, message boxes are implemented as dialog boxes, so no specific notification for message boxes occurs. The filter also receives a pointer to a structure containing the message. Filters for this hook may reside in a DLL or in an application. The WH_SYSMSGFILTER hooks are called before the WH_MSGFILTER hooks. If any of the WH_SYSMSGFILTER hook functions return TRUE, the WH_MSGFILTER hooks are not called.

WH_SYSMSGFILTER

This hook is the same as the WH_MSGFILTER hook except that it is a systemwide hook. Windows calls this hook when a dialog box, a message box, a scroll bar, or a menu retrieves a message or when the user presses the ALT+TAB or ALT+ESC keys. The filter receives a hook code indicating whether the message is for a dialog box or a message box (MSGF_DIALOGBOX), a scroll bar (MSGF_SCROLLBAR), a menu (MSGF_MENU), or the next window action (MSGF_NEXTWINDOW) about to take place. In Windows versions 3.0 and 3.1, message boxes are implemented as dialog boxes, so no specific notification for message boxes occurs. The filter also receives a pointer to a structure containing the message. Filters for this hook must reside in a DLL. The WH_SYSMSGFILTER hooks are called before the WH_MSGFILTER hooks. If any of the WH_SYSMSGFILTER hook functions return TRUE, the WH_MSGFILTER hooks are not called.