Hooks Overview - Win32 apps

By Karl-Bridge-Microsoft

Archived: 2026-04-06 01:00:12 UTC

A *hook* is a mechanism by which an application can intercept events, such as messages, mouse actions, and keystrokes. A function that intercepts a particular type of event is known as a *hook procedure*. A hook procedure can act on each event it receives, and then modify or discard the event.

Some uses for hooks include:

- Monitoring messages for debugging purposes
- Providing support for recording and playback of macros
- Providing support for a help key (F1)
- Simulating mouse and keyboard input
- Implementing a computer-based training (CBT) application

Note

Hooks tend to slow down the system because they increase the amount of processing the system must perform for each message. You should install a hook only when necessary, and remove it as soon as possible.

This section discusses the following:

- Hook Chains
- Hook Procedures
- Hook Types
    - WH_CALLWNDPROC and WH_CALLWNDPROCRET
    - WH_CBT
    - WH_DEBUG
    - WH_FOREGROUNDIDLE
    - WH_GETMESSAGE
    - WH_JOURNALPLAYBACK
    - WH_JOURNALRECORD
    - WH_KEYBOARD_LL
    - WH_KEYBOARD
    - WH_MOUSE_LL
    - WH_MOUSE
    - WH_MSGFILTER and WH_SYSMSGFILTER
    - WH_SHELL

The system supports many different types of hooks; each type provides access to a different aspect of its message-handling mechanism. For example, an application can use the WH_MOUSE hook to monitor the message traffic

for mouse messages.

The system maintains a separate hook chain for each type of hook. A *hook chain* is a list of pointers to special, application-defined callback functions called *hook procedures*. When a message occurs that is associated with a particular type of hook, the system passes the message to each hook procedure referenced in the hook chain, one after the other. The action a hook procedure can take depends on the type of hook involved. The hook procedures for some types of hooks can only monitor messages; others can modify messages or stop their progress through the chain, preventing them from reaching the next hook procedure or the destination window.

To take advantage of a particular type of hook, the developer provides a hook procedure and uses the **SetWindowsHookEx** function to install it into the chain associated with the hook. A hook procedure must have the following syntax:

```
 LRESULT CALLBACK HookProc(
  int nCode,
  WPARAM wParam,
  LPARAM lParam
)
{
   // process event
   ...

   return CallNextHookEx(NULL, nCode, wParam, lParam);
}
```

HookProc is a placeholder for an application-defined name.

The *nCode* parameter is a hook code that the hook procedure uses to determine the action to perform. The value of the hook code depends on the type of the hook; each type has its own characteristic set of hook codes. The values of the *wParam* and *lParam* parameters depend on the hook code, but they typically contain information about a message that was sent or posted.

The **SetWindowsHookEx** function always installs a hook procedure at the beginning of a hook chain. When an event occurs that is monitored by a particular type of hook, the system calls the procedure at the beginning of the hook chain associated with the hook. Each hook procedure in the chain determines whether to pass the event to the next procedure. A hook procedure passes an event to the next procedure by calling the **CallNextHookEx** function.

Note that the hook procedures for some types of hooks can only monitor messages. The system passes messages to each hook procedure, regardless of whether a particular procedure calls **CallNextHookEx**.

A *global hook* monitors messages for all threads in the same desktop as the calling thread. A *thread-specific hook* monitors messages for only an individual thread. A global hook procedure can be called in the context of any application in the same desktop as the calling thread, so the procedure must be in a separate DLL module. A thread-specific hook procedure is called only in the context of the associated thread. If an application installs a

hook procedure for one of its own threads, the hook procedure can be in either the same module as the rest of the application's code or in a DLL. If the application installs a hook procedure for a thread of a different application, the procedure must be in a DLL. For information, see Dynamic-Link Libraries.

Note

You should use global hooks only for debugging purposes; otherwise, you should avoid them. Global hooks hurt system performance and cause conflicts with other applications that implement the same type of global hook.

Each type of hook enables an application to monitor a different aspect of the system's message-handling mechanism. The following sections describe the available hooks.

- WH_CALLWNDPROC and WH_CALLWNDPROCRET
- WH_CBT
- WH_DEBUG
- WH_FOREGROUNDIDLE
- WH_GETMESSAGE
- WH_JOURNALPLAYBACK
- WH_JOURNALRECORD
- WH_KEYBOARD_LL
- WH_KEYBOARD
- WH_MOUSE_LL
- WH_MOUSE
- WH_MSGFILTER and WH_SYSMSGFILTER
- WH_SHELL

The **WH_CALLWNDPROC** and **WH_CALLWNDPROCRET** hooks enable you to monitor messages sent to window procedures. The system calls a **WH_CALLWNDPROC** hook procedure before passing the message to the receiving window procedure, and calls the **WH_CALLWNDPROCRET** hook procedure after the window procedure has processed the message.

The **WH_CALLWNDPROCRET** hook passes a pointer to a **CWPRETSTRUCT** structure to the hook procedure. The structure contains the return value from the window procedure that processed the message, as well as the message parameters associated with the message. Subclassing the window does not work for messages set between processes.

For more information, see the *CallWndProc* and *CallWndRetProc* callback functions.

The system calls a **WH_CBT** hook procedure before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the input focus; or before synchronizing with the system message queue. The value the hook procedure returns determines whether the system allows or prevents one of these operations. The **WH_CBT** hook is intended primarily for computer-based training (CBT) applications.

For more information, see the **CBTProc** callback function.

For information, see WinEvents.

The system calls a **WH_DEBUG** hook procedure before calling hook procedures associated with any other hook in the system. You can use this hook to determine whether to allow the system to call hook procedures associated with other types of hooks.

For more information, see the DebugProc callback function.

The **WH_FOREGROUNDIDLE** hook enables you to perform low priority tasks during times when its foreground thread is idle. The system calls a **WH_FOREGROUNDIDLE** hook procedure when the application's foreground thread is about to become idle.

For more information, see the *ForegroundIdleProc* callback function.

The **WH_GETMESSAGE** hook enables an application to monitor messages about to be returned by the **GetMessage** or **PeekMessage** function. You can use the **WH_GETMESSAGE** hook to monitor mouse and keyboard input and other messages posted to the message queue.

For more information, see the *GetMsgProc* callback function.

Warning

Journaling Hooks APIs are unsupported starting in Windows 11 and will be removed in a future release. Because of this, we highly recommend calling the **SendInput** TextInput API instead.

The **WH_JOURNALPLAYBACK** hook enables an application to insert messages into the system message queue. You can use this hook to play back a series of mouse and keyboard events recorded earlier by using WH_JOURNALRECORD. Regular mouse and keyboard input is disabled as long as a **WH_JOURNALPLAYBACK** hook is installed. A **WH_JOURNALPLAYBACK** hook is a global hook—it cannot be used as a thread-specific hook.

The **WH_JOURNALPLAYBACK** hook returns a time-out value. This value tells the system how many milliseconds to wait before processing the current message from the playback hook. This enables the hook to control the timing of the events it plays back.

For more information, see the *JournalPlaybackProc* callback function.

Warning

Journaling Hooks APIs are unsupported starting in Windows 11 and will be removed in a future release. Because of this, we highly recommend calling the **SendInput** TextInput API instead.

The **WH_JOURNALRECORD** hook enables you to monitor and record input events. Typically, you use this hook to record a sequence of mouse and keyboard events to play back later by using WH_JOURNALPLAYBACK. The **WH_JOURNALRECORD** hook is a global hook—it cannot be used as a thread-specific hook.

For more information, see the *JournalRecordProc* callback function.

The **WH_KEYBOARD_LL** hook enables you to monitor keyboard input events about to be posted in a thread input queue.

For more information, see the *LowLevelKeyboardProc* callback function.

The **WH_KEYBOARD** hook enables an application to monitor message traffic for **WM_KEYDOWN** and **WM_KEYUP** messages about to be returned by the **GetMessage** or **PeekMessage** function. You can use the **WH_KEYBOARD** hook to monitor keyboard input posted to a message queue.

For more information, see the *KeyboardProc* callback function.

The **WH_MOUSE_LL** hook enables you to monitor mouse input events about to be posted in a thread input queue.

For more information, see the *LowLevelMouseProc* callback function.

The **WH_MOUSE** hook enables you to monitor mouse messages about to be returned by the **GetMessage** or **PeekMessage** function. You can use the **WH_MOUSE** hook to monitor mouse input posted to a message queue.

For more information, see the *MouseProc* callback function.

The **WH_MSGFILTER** and **WH_SYSMSGFILTER** hooks enable you to monitor messages about to be processed by a menu, scroll bar, message box, or dialog box, and to detect when a different window is about to be activated as a result of the user's pressing the ALT+TAB or ALT+ESC key combination. The **WH_MSGFILTER** hook can only monitor messages passed to a menu, scroll bar, message box, or dialog box created by the application that installed the hook procedure. The **WH_SYSMSGFILTER** hook monitors such messages for all applications.

The **WH_MSGFILTER** and **WH_SYSMSGFILTER** hooks enable you to perform message filtering during modal loops that is equivalent to the filtering done in the main message loop. For example, an application often examines a new message in the main loop between the time it retrieves the message from the queue and the time it dispatches the message, performing special processing as appropriate. However, during a modal loop, the system retrieves and dispatches messages without allowing an application the chance to filter the messages in its main message loop. If an application installs a **WH_MSGFILTER** or **WH_SYSMSGFILTER** hook procedure, the system calls the procedure during the modal loop.

An application can call the **WH_MSGFILTER** hook directly by calling the **CallMsgFilter** function. By using this function, the application can use the same code to filter messages during modal loops as it uses in the main message loop. To do so, encapsulate the filtering operations in a **WH_MSGFILTER** hook procedure and call **CallMsgFilter** between the calls to the **GetMessage** and **DispatchMessage** functions.

```
 while (GetMessage(&msg, (HWND) NULL, 0, 0))
{
    if (!CallMsgFilter(&qmsg, 0))
        DispatchMessage(&qmsg);
}
```

The last argument of **CallMsgFilter** is simply passed to the hook procedure; you can enter any value. The hook procedure, by defining a constant such as **MSGF_MAINLOOP**, can use this value to determine where the procedure was called from.

For more information, see the *MessageProc* and *SysMsgProc* callback functions.

A shell application can use the **WH_SHELL** hook to receive important notifications. The system calls a **WH_SHELL** hook procedure when the shell application is about to be activated and when a top-level window is created or destroyed.

Note that custom shell applications do not receive **WH_SHELL** messages. Therefore, any application that registers itself as the default shell must call the **SystemParametersInfo** function before it (or any other application) can receive **WH_SHELL** messages. This function must be called with **SPI_SETMINIMIZEDMETRICS** and a **MINIMIZEDMETRICS** structure. Set the **iArrange** member of this structure to **ARW_HIDE**.

For more information, see the *ShellProc* callback function.
