# How were window hooks implemented in 16-bit Windows?

devblogs.microsoft.com/oldnewthing/20060809-18

August 9, 2006

Raymond Chen

The mechanism for keeping track of window hooks was very different in 16-bit Windows. The functions involved were `SetWindowsHook`, `UnhookWindowsHook` and `DefHookProc`. The first two functions still exist today, but the third one has been replaced with a macro:

```
// 16-bit prototype
DWORD WINAPI DefHookProc(int nCode, WPARAM wParam,
                         LPARAM lParam, HHOOK FAR *phk);
// 32-bit macro
#define DefHookProc(nCode, wParam, lParam, phhk)\
        CallNextHookEx(*phhk, nCode, wParam, lParam)
```

Disclaimer: All code below is "reconstructed from memory". The spirit of the code is intact, but the precise details may be off.

To install a windows hook in 16-bit Windows, you started by calling `SetWindowsHook`:

```
HHOOK g_hhkPrev;
g_hhkPrev = SetWindowsHook(WH_WHATEVER, MyHookProc);
```

The return value from `SetWindowsHook` must be saved in a global variable, which we gave the somewhat provocative name `g_hhkPrev`. The hook procedure itself went something like this:

```
// In Win16, hook procedures returned a DWORD, not an LRESULT.
DWORD CALLBACK MyHookProc(int nCode, WPARAM wParam, LPARAM lParam)
{
  if (nCode >= 0) { ... }
  return DefHookProc(nCode, wParam, lParam, &g_hhkPrev);
}
```

And then when you were finished, you removed the hook by calling `UnhookWindowsHook`:

```
UnhookWindowsHook(WH_WHATEVER, MyhookProc);
g_hhkPrev = NULL;
```

Internally, the chain of hook functions was managed as a linked list, but instead of using some internal data structure to keep track of the hooks, the linked list was managed **inside the HHOOK variables themselves**.

The internal implementation of `SetWindowsHook` was simply this:

```
// This array is initialized with a bunch
// of "do nothing" hook procedures.
HOOKPROC g_rgHook[NUMHOOKS];
HHOOK WINAPI SetWindowsHook(int nType, HOOKPROC pfnHookProc)
{
 HHOOK hhkPrev = (HHOOK)g_rgHook[nType];
 g_rgHook[nType] = pfnHookProc;
 return hhkPrev;
}
```

Installing a hook merely set your hook procedure as the head of the hook chain, and it returned the previous head. Invoking a hook was a simple matter of calling the hook at the head of the chain:

```
DWORD CallHook(int nType, int nCode, WPARAM wParam, LPARAM lParam)
{
 return g_rgHook[nType](nCode, wParam, lParam);
}
```

Each hook procedure did its work and then sent the call down the hook chain by calling `DefHookProc`, passing the `HHOOK` **by address**.

```
DWORD WINAPI DefHookProc(int nCode, WPARAM wParam,
                         LPARAM lParam, HHOOK FAR *phk)
{
 HOOKPROC pfnNext = (HOOKPROC)*phk;
 if (nCode >=0) {
  return pfnNext(nCode, wParam, lParam);
 }
 ... more to come ...
}
```

As you can see, it's all blindingly simple: Invoking a hook calls the first hook procedure, which then calls `DefHookProc`, which knows that a `HHOOK` is just a `HOOKPROC`, and it forwards the call down the chain by merely calling the next hook procedure directly.

The real magic happens when somebody wants to unhook. Recall that the rule for hook procedures is that a negative hook code should be passed straight to `DefHookProc` (or in modern times, `CallNextHookEx`). This convention allows the hook system to use negative codes to manage its own internal bookkeeping. In this case, we're using `-1` as the "unhook this hook procedure" code.

```
BOOL WINAPI UnhookWindowsHook(int nType, HOOKPROC pfnHookProc)
{
 return DefHookProc(-1, 0, (LPARAM)pfnHookProc,
                   (HHOOK FAR*)&g_rgHook[nType]);
}
```

And then the real magic begins:

```
DWORD WINAPI DefHookProc(int nCode, WPARAM wParam,
                        LPARAM lParam, HHOOK FAR *phk)
{
 HOOKPROC pfnNext = (HOOKPROC)*phk;
 if (nCode >=0) {
  return pfnNext(nCode, wParam, lParam);
 }
 switch (nCode) {
 case -1: // trying to unhook a node
  if (pfnNext == (HOOKPROC)lParam) { // found it
   *phk = (HHOOK)pfnNext(-2, 0, 0);
   return TRUE;
  }
  // else keep looking
  return pfnNext(nCode, wParam, lParam);
 case -2: // report the next hook procedure
   return (DWORD)*phk;
 }
 return 0;
}
```

And there you have it, the entire window hook system in two dozen lines of code. You have to give 16-bit Windows credit for being small.

Let’s walk through hook installation, dispatch, and removal to see how this all works. Suppose there is one `WH_KEYBOARD` hook in the system. Our variables are therefore set up like this:

```
// In USER
g_rgHook[WH_KEYBOARD] = Hook1;
// In HOOK1.DLL
HHOOK g_hhkPrev1 = DoNothingHookProc;
DWORD CALLBACK Hook1(int nCode, WPARAM wParam, LPARAM lParam)
{
 if (nCode >= 0) { ... work ... }
 return DefHookProc(nCode, wParam, lParam, &g_hhkPrev1);
}
```

Now suppose you want to install a new hook, `Hook2`.

```
// In HOOK2.DLL
HHOOK g_hhkPrev2;
g_hhkPrev = SetWindowsHook(WH_KEYBOADR, Hook2);
```

The `SetWindowsHook` function just puts your function in as the new “head” hook function and returns the old one.

```
// In USER
g_rgHook[WH_KEYBOARD] = Hook2;
// In HOOK2.DLL
HHOOK g_hhkPrev2 = Hook1;
DWORD CALLBACK Hook2(int nCode, WPARAM wParam, LPARAM lParam)
{
 if (nCode >= 0) { ... work ... }
 return DefHookProc(nCode, wParam, lParam, &g_hhkPrev2);
}
// In HOOK1.DLL
HHOOK g_hhkPrev1 = DoNothingHookProc;
DWORD CALLBACK Hook1(int nCode, WPARAM wParam, LPARAM lParam)
{
 if (nCode >= 0) { ... work ... }
 return DefHookProc(nCode, wParam, lParam, &g_hhkPrev1);
}
```

Now suppose the window manager decides it’s time to fire the `WH_KEYBOARD` hook. It starts with `CallHook` which calls `g_rgHook[WH_KEYBOARD]` that takes us to `Hook2`. That hook function does its work, then calls `DefHookProc(..., &g_hhkPrev2)`, which dispatches the hook to `g_hhkPrev2 == Hook1`. Similarly, the hook travels through `Hook1`, then `DefHookProc(..., &g_hhkPrev1)`, where it finally reaches the `DoNothingHookProc` which does nothing and ends the hook chain.

Now suppose that `HOOK1.DLL` decides to uninstall its hook. It therefore calls `UnhookWindowsHook(WH_KEYBOARD, Hook1)`. This starts off the hook chain with the internal hook code `-1` and `&g_rgHook[WH_KEYBOARD]` as the first hook pointer. This activates the `case -1` in `DefHookProc` code path, which dereferences its `phk` parameter and obtains `g_rgHook[WH_KEYBOARD] == Hook2`. Since this is not equal to `Hook1`, the call forwards down the chain to `Hook2`.

Like a good hook function, `Hook2` reacts to the negative hook code by handing the call directly to `DefHookProc(-1, ..., &g_hhkPrev2)`. This time, `*phk == g_hhkPrev2 == Hook1`, so the test succeeds and we dispatch the hook down the chain with a new internal code of `-2`, which means, “Tell me what the next hook procedure is”.

This dispatch calls `Hook1` which (since the notification code is negative) immediately passes the call to `DefHookProc(-2, ..., &g_hhkPrev1)`. This now triggers the `case -2` code path, which just returns `*phk == g_hhkPrev1 == DoNothingHookProc`. This value is returned to the `DefHookProc(-1, ...)` which stores the result into `*phk == g_hhkPrev2`; the result is that you have `g_hhkPrev2 = DoNothingHookProc`. Finally, `DefHookProc` returns `TRUE` to indicate that the hook was successfully uninstalled. This value is then returned out from all the nested function calls to the original caller of `UnhookWindowsHook`.

Observe that at the end of this unhook exercise, we get the desired result:

```
// In USER
g_rgHook[WH_KEYBOARD] = Hook2; // unchanged
// In HOOK2.DLL
g_hhkPrev2 = DoNothingHookProc; // updated!
DWORD CALLBACK Hook2(int nCode, WPARAM wParam, LPARAM lParam)
{
 if (nCode >= 0) { ... work ... }
 return DefHookProc(nCode, wParam, lParam, &g_hhkPrev2);
}
```

And `Hook1` is out of the hook chain, as we desired.

This really isn't all that complicated. All we did was delete a node from a linked list. It's just that this particular linked list cannot be traversed by just dereferencing pointers. Instead, we have to issue a function call and ask the recursive function to perform the work on the "next" node for us. That's what the negative `nCode` values are for.

Every time I work through this exercise, I am impressed by how compactly 16-bit Windows was written. In just two dozen lines of code, we managed a linked list of function calls, including a dispatching system as well as arbitrary deletion from the middle of the linked list, and all without any memory allocation.

(And because I know people are going to try to change the topic: Remember, I'm talking about 16-bit Windows, not 32-bit window hooks.)

Next time, we'll look at one way people abused this simple system.

Raymond Chen

**Follow**