Killing a window timer prevents the WM_TIMER message from being generated for that timer, but it doesn't retroactively remove ones that were already generated

devblogs.microsoft.com/oldnewthing/20141205-00

December 5, 2014

Raymond Chen

Calling `KillTimer` to cancel a window timer prevents `WM_TIMER` messages from being generated for that timer, even if one is overdue. In other words, give this sequence of operations:

```
SetTimer(hwnd, IDT_MYTIMER, 1000, NULL);
Sleep(2000);
KillTimer(hwnd, IDT_MYTIMER);
```

no `WM_TIMER` message is ever generated. Even though a timer became due during the `Sleep`, no timer message was generated during the sleep because timer messages are generated on demand, and nobody demanded one. Killing the timer then removes the ability to demand a timer message, and the result is that no message ever appears.

In general, this means that once you kill a timer, you will not receive any `WM_TIMER` messages for that timer.

Unless you demanded one while the timer was active and didn't process it.

Let's try a variation:

```
SetTimer(hwnd, IDT_MYTIMER, 1000, NULL);
Sleep(2000);
if (PeekMessage(&msg, NULL, WM_TIMER, WM_TIMER, 0)) {
 DispatchMessage(&msg);
}
KillTimer(hwnd, IDT_MYTIMER);
```

In this case, the `PeekMessage` function looks for a `WM_TIMER` message in the queue, and if none is found, it asks for one to be generated on the fly if a timer is due. It so happens that one is due (`IDT_MYTIMER`), so the `PeekMessage` causes a `WM_TIMER` to be generated and placed in the queue. But it doesn't remain in this state for long, because the message is removed from the queue by the `PeekMessage` function.

Okay, now let's make things weird:

```
SetTimer(hwnd, IDT_MYTIMER, 1000, NULL);
Sleep(2000);
if (PeekMessage(&msg, NULL, WM_TIMER, WM_TIMER, PM_NOREMOVE)) {
 // oh hey there is an overdue timer, how about that
}
KillTimer(hwnd, IDT_MYTIMER);
```

This time, we passed the `PM_NOREMOVE` flag. The window manager goes through the same process as before, first looking for a `WM_TIMER` message in the queue, and then failing to find one, generates one on the fly since the `IDT_MYTIMER` timer is overdue. But the `PM_NOREMOVE` flag makes things weird because it says, "Thanks for generating that message for me. But don't remove it from the queue. Leave it there. I'll deal with it later."

You might do this if you want to stop processing if a timer elapses, but you don't want to handle the timer immediately because you are in some sensitive state at the point you realize that you need to stop processing. Instead, you want to return back out to the main message loop and let it deal with the timer.

```
BOOL DoWorkUntilTheNextTimer()
{
 BOOL fFinished = FALSE;
 MSG msg;
 PrepareToDoWork();
 while (!PeekMessage(&msg, NULL, WM_TIMER, WM_TIMER, PM_NOREMOVE)) {
  if (AnyWorkLeft()) DoSomeWork();
  else { fFinished = TRUE; break; }
 }
 CleanUpAfterDoingWork();
 return fFinished;
}
```

And then you might call it like this:

```
void DoWorkForUpToOneSecond()
{
 SetTimer(hwnd, IDT_MYTIMER, 1000, NULL);
 DoWorkUntilTheNextTimer();
 KillTimer(hwnd, IDT_MYTIMER);
}
```

The `KillTimer` will prevent any new timer messages from being generated for `IDT_MYTIMER`, but it does not go back in time and retroactively un-generate the timer message that was generated when `DoWorkUntilTheNextTimer` asked to see if there were any timer messages.

You are now in the strange situation where a subsequent call to `PeekMessage` or `GetMessage` will retrieve a timer message for a timer that is no longer active!

This is captured in the MSDN documentation with the simple sentence, “The **KillTimer** function does not remove `WM_TIMER` messages already posted to the message queue.”

Raymond Chen

Follow