Modality, part 3: The WM_QUIT message

devblogs.microsoft.com/oldnewthing/20050222-00

February 22, 2005

Raymond Chen

After our two quick introductions to modality, we're now going to dig in a little deeper.

The trick with modality is that when you call a modal function, the responsibility of message dispatch is handled by that function rather than by your main program. Consequently, if you have customized your main program's message pump, those customizations are lost once you lose control to a modal loop.

The other important thing about modality is that a `WM_QUIT` message always breaks the modal loop. Remember this in your own modal loops! If ever you call the `PeekMessage` function or The [typo fixed 10:30am] `GetMessage` function and get a `WM_QUIT` message, you must not only exit your modal loop, but you must also re-generate the `WM_QUIT` message (via the `PostQuitMessage` message) so the next outer layer will see the `WM_QUIT` message and do its cleanup as well. If you fail to propagate the message, the next outer layer will not know that it needs to quit, and the program will seem to "get stuck" in its shutdown code, forcing the user to terminate the process the hard way.

In a later series, we'll see how this convention surrounding the `WM_QUIT` message is useful. But for now, here's the basic idea of how your modal loops should re-post the quit message to the next outer layer.

```
BOOL WaitForSomething(void)
{
  MSG msg;
  BOOL fResult = TRUE; // assume it worked
  while (!SomethingFinished()) {
    if (GetMessage(&msg, NULL, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    } else {
      // We received a WM_QUIT message; bail out!
      CancelSomething();
      // Re-post the message that we retrieved
      PostQuitMessage(msg.wParam);
      fResult = FALSE; // quit before something finished
      break;
    }
  }
  return fResult;
}
```

Suppose your program starts some operation and then calls `WaitForSomething()`. While waiting for something to finish, some other part of your program decides that it's time to exit. (Perhaps the user clicked on a "Quit" button.) That other part of the program will call `PostQuitMessage(wParam)` to indicate that the message loop should terminate.

The posted quit message will first be retrieved by the `GetMessage` in the `WaitForSomething` function. The `GetMessage` function returns `FALSE` if the retrieved message is a `WM_QUIT` message. In that case, the "else" branch of the conditional is taken, which cancels the "Something" operation in progress, then posts the quit message back into the message queue for the next outer message loop to handle.

When `WaitForSomething` returns, control presumably will fall back out into the program's main message pump. The main message pump will then retrieve the `WM_QUIT` message and do its exit processing before finally exiting the program.

And if there were additional layers of modality between `WaitForSomething` and the program's main message pump, each of those layers would retrieve the `WM_QUIT` message, do their cleanup, and then re-post the `WM_QUIT` message (again, via `PostQuitMessage`) before exiting the loop.

In this manner, the `WM_QUIT` message gets handed from modal loop to modal loop, until it reaches the outermost loop, which terminates the program.

"But wait," I hear you say. "Why do I have to do all this fancy `WM_QUIT` footwork? I could just have a private little global variable named something like `g_fQuitting`. When I want the program to quit, I just set this variable, and all of my modal loops check this variable and exit prematurely if it is set. Something like this:

```
BOOL MyWaitForSomething(void) // code in italics is wrong
{
  MSG msg;
  while (!SomethingFinished()) {
    if (g_fQuitting) {
    CancelSomething();
      return FALSE;
    }
    if (GetMessage(&msg, NULL, 0, 0)) {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return TRUE;
}
```

And so I can solve the problem of the nested quit without needing to do all this `PostQuitMessage` rigamarole."

And you'd be right, if you controlled every single modal loop in your program.

But you don't.

For example, when you call the `DialogBox` function, the dialog box code runs its own private modal loop to do the dialog box UI until you get around to calling the `EndDialog` function. And whenever the user clicks on any of your menus, Windows runs its own private modal loop to do the menu UI. Indeed, even the resizing of your application's window is handled by a Windows modal loop.

Windows, of course, has no knowledge of your little `g_fQuitting` variable, so it has no idea that you want to quit. It is the `WM_QUIT` message that serves this purpose of co-ordinating the intention to quit among separate parts of the system.

Notice that this convention regarding the `WM_QUIT` message cuts both ways. You can use this convention to cause modal loops to exit (we'll see more of this later), but it also obliges you to respect this convention so that **other** components (including the window manager itself) can get your modal loops to exit.

Raymond Chen

Follow