Just because you're a control doesn't mean that you're necessarily inside a dialog box

devblogs.microsoft.com/oldnewthing/20070820-00

August 20, 2007

Raymond Chen

Prerequisites: Moderate to advanced understanding of the window and dialog managers. When you're implementing a control, you need to be aware that you aren't necessarily being hosted inside a dialog box. One commenter suggested handling `WM_KEYDOWN` and closing the dialog box as a way to prevent multi-line edit controls from eating the Enter key. But the edit control can't do that because people create edit controls outside of dialog boxes. How do you "close the dialog box" when there isn't one? This leads to a related topic brought up by another comment:

> Doesn't `ES_WANTRETURN` do exactly this? The MSDN states the following (emphasis mine): "ES_WANTRETURN: Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiple-line edit control in a dialog box. Without this style, **pressing the ENTER key has the same effect as pressing the dialog box's default pushbutton**. This style has no effect on a single-line edit control."

I remarked that `ES_WANTRETURN` is a messy subject. Now I'm going to show you the mess. It's sort of like visiting your friend's house when they're not expecting you and wandering into their bedroom where they haven't tidied up and there's clothes all over the floor. The authors of the edit control back in 1981 didn't follow the above guidance. Probably[1] because back in the days when the edit control was first written, the window manager was still in a state of flux and its design hadn't settled down. You can't blame the edit control for not following guidance that didn't exist. The edit control implements `ES_WANTRETURN` as you might expect: It include `DLGC_WANTALLKEYS` in its response to `WM_GETDLCODE`, which causes all keys, including Enter, to go to the edit control. What's more interesting is how the edit control implemented the absence of `ES_WANTRETURN`: It still includes `DLGC_WANTALLKEYS`, but when it receives the Enter key, it first attempts to detect whether it's inside a dialog box, and if so, it tries to mimic what the dialog box would have done: It asks its parent dialog box for the default ID, sets focus to the corresponding control, and simulates input via `PostMessage` to make that control act as if the user had pressed Enter. Since only button controls can be the default ID, the edit control "knows" that the recipient of the simulated input is the button control. The author of the edit control then went in and

modified the button control so that it didn't rely on virtualized input state when handling the `WM_KEYDOWN` message. This is ugly no matter how you slice it, and it violates so many principles of control design it isn't funny. For one thing, the way it detects whether the control it hosted inside a dialog is fragile and can be tricked into guessing wrong. Next, its mimcry of the `IsDialogMessage` function is incorrect. When it wants to invoke the default button, it does so by simulating input, which we already know is wrong. And before it does so, it sets focus to the control, which is also wrong; the `IsDialogMessage` function generates a `WM_COMMAND` message *without changing focus*. And finally, it totally misses the boat if the edit control is inside a nested dialog. As I noted, all these mistakes are obvious in retrospect, but when the control was first written, these mistakes might not[1] even have been mistakes. (For example, nested dialogs didn't appear on the scene until Windows 95.) Why haven't these mistakes been fixed? Well, how can you prove that there aren't any programs that rely on the mistakes? One thing you quickly learn in application compatibility is that a bug once shipped gains the status of a feature, because you can be pretty sure that some program somewhere relies on it. (I've seen a plugin that relies on a memory leak in Explorer, for example.) This goes doubly true for core controls like the edit control. Any change to the edit control must be taken with a great deal of trepidation, because your change affects pretty much every single Windows program on the entire planet. With that high a degree of risk, the prudent choice is often to let sleeping dogs lie. **Nitpicker's Corner**

[1]Note weasel words. This is my educated guess as to what happened based on personal observation and thought. It is not a statement of the official position of Microsoft Corporation, and this guess may ultimately prove incorrect.

Raymond Chen

Follow