CHAPTER THREE

THE SECRET LIFE OF GETWINDOWTEXT

THE GETWINDOWTEXT function is more complicated than you think. The documentation tries to explain its complexity with small words, which is great if you don't understand long words, but it also means that the full story becomes obscured.

Here's an attempt to give the full story.

How windows manage their text

THERE ARE TWO ways window classes can manage their text. They can do it manually or they can let the system do it. The default is to let the system do it.

If a window class lets the system manage its text, the system will do the following:

- Default handling of the `WM_NCCREATE` message takes the `lpWindowName` parameter passed to `CreateWindow/Ex` and saves the string in a "special place."
- Default handling of the `WM_GETTEXT` message retrieves the string from that special place.

- Default handling of the WM_SETTEXT message copies the string to that special place.

On the other hand, if a window class manages its window text manually, the system does not do any special handling, and it is the window class's responsibility to respond to the WM_GETTEXT/WM_SETTEXT messages and return/save the strings explicitly.

Frame windows typically let the system manage their window text. Custom controls typically manage their window text manually.

Enter GetWindowText

The GetWindowText function has a problem: Window text needs to be readily available without hanging. FindWindow needs to get window text to find a window. Task-switching applications need to get window text so that they can display the window title in the switcher window. It should not be possible for a hung application to clog up other applications. This is particularly true of the task-switcher scenario.

This argues *against* sending WM_GETTEXT messages, because the target window of the WM_GETTEXT might be hung. Instead, GetWindowText should use the "special place" because that cannot be affected by hung applications.

On the other hand, GetWindowText is used to retrieve text from controls on a dialog, and those controls frequently employ custom text management. This argues *for* sending WM_GETTEXT messages, because that is the only way to retrieve custom-managed text.

GetWindowText strikes a compromise:

- If you are trying to get the window text from a window in your own process, GetWindowText will send the WM_GETTEXT message.
- If you are trying to get the window from a window in another process, GetWindowText will use the string from the special place and not send a message.

According to the first rule, if you are trying to get text from a window in your own process, and the window is hung, `GetWindowText` will also hang. But because the window belongs to your process, it's your own fault, and you deserve to lose. Sending the `WM_GETTEXT` message ensures that text from windows that do custom text management (typically, custom controls) are properly retrieved.

According to the second rule, if you are trying to get text from a window in another process, `GetWindowText` will not send a message; it just retrieves the string from the special place. Because the most common reason for getting text from a window in another process is to get the title of the frame, and because frame windows typically do not do custom window text manipulation, this usually gets the right string.

The documentation simplifies this as "`GetWindowText` cannot retrieve text from a window from another application."

What if I don't like these rules?

IF THE SECOND rule bothers you because you need to get text from a custom control in another process, you can send the `WM_GETTEXT` message manually. Because you are not using `GetWindowText`, you are not subject to its rules.

Note, however, that if the target window is hung, your application will also hang because `SendMessage` will not return until the target window responds.

Note also that because `WM_GETTEXT` is in the system message range (0 to `WM_USER-1`), you do not need to take any special action to get your buffer transferred into the target process and to get the result transferred back to the calling process (a procedure known as *marshalling*). In fact, any special steps you take to this end are in error. The window manager does the marshalling for you.

Can you give an example where this makes a difference?

CONSIDER THIS CONTROL:

```
SampleWndProc(...)
{
    case WM_GETTEXT:
        lstrcpyn((LPTSTR)lParam, TEXT("Booga!"), (int)wParam);
        return lstrlen((LPTSTR)lParam);
    case WM_GETTEXTLENGTH: return 7; // lstrlen("Booga!") + null
    ...
}
```

And application A, which does this:

```
hwnd = CreateWindow("Sample", "Frappy", ...);
```

Now consider process B, which gets the handle to the window created by application A (by whatever means):

```
TCHAR szBuf[80];
GetWindowText(hwnd, szBuf, 80);
```

This will return `szBuf = "Frappy"` because it is getting the window text from the special place. However

```
SendMessage(hwnd, WM_GETTEXT, 80, (LPARAM)szBuf);
```

will return `szBuf = "Booga!"`

Why are the rules for GetWindowText so weird?

SET THE WAYBACK machine to 1983. Your typical PC had an 8086 processor running at a whopping 4.7MHz, two 360K 5¼-inch floppy drives (or if you

were really loaded, one floppy drive and a 10MB hard drive), and 256KB of memory

This was the world of Windows 1.0.

Windows 1.0 was a cooperatively multitasked system. No preemptive multitasking here. When your program got control, it had control for as long as it wanted it. Only when you called a function such as `PeekMessage` or `GetMessage` did you release control to other applications.

This was important because in the absence of a hardware memory manager, you really had to make sure that your memory didn't get ripped out from under you.

One important consequence of cooperative multitasking is that if your program is running, not only do you know that no other program is running, but you also know that *every window is responding to messages*. Why? Because if they are hung, they won't release control to you!

This means that it was *always* safe to send a message. You never had to worry about the possibility of sending a message to a hung window, because you knew that no windows were hung.

In this simpler world, `GetWindowText` was a straightforward function:

```
int WINAPI
GetWindowText(HWND hwnd, LPSTR pchBuf, int cch)
{
  // ah for the simpler days
  return SendMessage(hwnd, WM_GETTEXT, (WPARAM)cch, (LPARAM)pchBuf);
}
```

This worked for all windows, all the time. No special handling of windows in a different process.

It was the transition to Win32 and preemptive multitasking that forced the change in the rules, because for the first time, there was the possibility that (gasp) the window you were trying to communicate with was not responding to messages.

Now you have the backward-compatibility problem. As noted previously, many parts of the system and many programs rely on the capability to retrieve window text without hanging. So how do you make it possible

to retrieve window text without hanging, while still enabling controls such as the edit control to do their own window text management?

The Win32 rules on `GetWindowText` are the result of this attempt to reconcile conflicting goals.