Sure, we have RegisterWindowMessage and RegisterClipboardFormat, but where are DeregisterWindowMessage and DeregisterClipboardFormat?

devblogs.microsoft.com/oldnewthing/20150319-00

March 19, 2015

Raymond Chen

The `RegisterWindowMessage` function lets you create your own custom messages that are globally unique. But how do you free the message format when you're done, so that the number can be reused for another message? (Similarly, `RegisterClipboardFormat` and clipboard formats.)

You don't. There is no `DeregisterWindowMessage` function or `DeregisterClipboard-Format` function. Once allocated, a registered window message and registered clipboard format hangs around until you log off.

There is room for around 16,000 registered window messages and registered clipboard formats, and in practice exhaustion of these pools of numbers is not an issue. Even if every program registers 100 custom messages, you can run 160 unique programs before running into a problem. And most people don't even have 160 different programs installed in the first place. (And if you do, you almost certainly don't run all of them!) In practice, the number of registered window messages is well under 1000.

A customer had a problem with exhaustion of registered window messages. "We are using a component that uses the `RegisterWindowMessage` function to register a large number of unique messages which are constantly changing. Since there is no way to unregister them, the registered window message table eventually fills up and things start failing. Should we use `GlobalAddAtom` and `GlobalDeleteAtom` instead of `RegisterWindowMessage`? Or can we use `GlobalDeleteAtom` to delete the message registered by `RegisterWindow-Message`?"

No, you should not use `GlobalAddAtom` to create window messages. The atom that comes back from `GlobalAddAtom` comes from the global atom table, which is different from the registered window message table. The only way to get registered window messages is to call `RegisterWindowMessage`. Say you call `GlobalAddAtom("X")` and you get atom 49443 from the global atom table. Somebody else calls `RegisterWindowMessage("Y")` and they get registered window message number 49443. You then post message 49443 to a window, and it thinks that it is message Y, and bad things happen.

And you definitely should not use `GlobalDeleteAtom` in a misguided attempt to deregister a window message. You're going to end up deleting some unrelated atom, and things will start going downhill.

What you need to do is fix the component so it does not register a lot of window messages with constantly-changing names. Instead, encode the uniqueness in some other way. For example, instead of registering a hundred messages of the form `Contoso user N logged on`, just register a single `Contoso user logged on` message and encode the user number in the `wParam` and `lParam` payloads. Most likely, one or the other parameter is already being used to carry nontrivial payload information, so you can just add the user number to that payload. (And this also means that your program won't have to keep a huge table of users and corresponding window messages.)

Bonus chatter: It is the case that properties added to a window via `SetProp` use global atoms, as indicated by the documentation. This is an implementation detail that got exposed, so now it's contractual. And it was a bad idea, as I discussed earlier.

Sometimes, people try to get clever and manually manage the atoms used for storing properties. They manually add the atom, then access the property by atom, then remove the properties, then delete the atom. This is a high-risk maneuver because there are so many things that can go wrong. For example, you might delete the atom prematurely (unaware that it was still being used by some other window), then the atom gets reused, and now you have a property conflict. Or you may have a bug that calls `GlobalDeleteAtom` for an atom that was not obtained via `GlobalAddAtom`. (Maybe you got it via `GlobalFindAtom` or `EnumProps`.)

I've even seen code that does this:

```
atom = GlobalAddAtom(name);

// Some apps are delete-happy and run around deleting atoms they shouldn't.
// If they happen to delete ours by accident, things go bad really fast.
// Prevent this from happening by bumping the atom refcount a few extra
// times so accidental deletes won't destroy it.
GlobalAddAtom(name);
GlobalAddAtom(name);
```

So we've come full circle. There is a way to delete an unused atom, but people end up deleting them incorrectly, so this code tries to make the atom undeletable. Le Chatelier's Principle strikes again.

Raymond Chen

Follow