Raymond’s complete guide to HSTRING semantics

devblogs.microsoft.com/oldnewthing/20160615-00

June 15, 2016

Raymond Chen

The title of today’s article is a blatant ripoff of Eric Lippert’s complete guide to BSTR semantics.

I’m going to start with a lie: An `HSTRING` is a reference-counted Unicode string.

Work with me here.

The string is immutable, and it uses the UTF-16LE encoding, as is traditional in Windows.

Here are the basic operations on `HSTRING`s:

WindowsCreateString creates an `HSTRING` from a UTF-16LE-encoded buffer and a specified length. The buffer does not require a terminating null. If the buffer contains embedded null characters, then the resulting `HSTRING` will have embedded null characters. (In particular, if you pass a null-terminated string and you include the null terminator in the length, then the resulting string has an embedded null character. Note also that the length is in `wchar_t` code units, not in bytes.)

WindowsDuplicateString increments the reference count on an `HSTRING`, and returns a new `HSTRING` which you should use to refer to the string.

WindowsDeleteString decrements the reference count on an `HSTRING`. If the reference count drops to zero, then the string is destroyed. You shouldn’t use the `HSTRING` after passing it to **WindowsDeleteString**.

There are a small number of string manipulation functions like **WindowsSubstring** and **WindowsConcatString** which create new strings from old strings. The set of operations is rather limited, however. If you want to perform fancy operations on `HSTRING`s, you’ll probably need to do them yourself. (Of course, if you’re using a projected language, the `HSTRING` will project as something closer to what your projected language operates on natively, at which point you will most likely have a rich collection of library functions available to do advanced manipulations.)

To access the characters in the `HSTRING`, use the **WindowsGetStringRawBuffer** function, which gives you two things: The return value is a pointer to the first character in the `HSTRING`, and the optional output parameter is the number of code units. The buffer should be treated as read-only because `HSTRING`s are immutable.

The string contents in the buffer are always followed by a null character (which doesn't count toward the string length); as a result, you can treat the string buffer as if it were a null-terminated string and get away with it most of the time.

The time you don't get away with it is if the string contains embedded null characters. In that case, treating it as a null-terminated string will stop prematurely, mistaking the embedded null for the terminal null. You can use the **WindowsStringHasEmbeddedNull** function to detect whether an `HSTRING` contains an embedded null and reject the operation if you don't support embedded nulls.

One of the special rules for `HSTRING` is similar to the corresponding rule for `BSTR`, namely that a null pointer is equivalent to a zero-length string. But `HSTRING` takes it further: Not only is a null pointer equivalent to a zero-length string, but in fact a null pointer *is* the representation of a zero-length string. In other words, if you call **WindowsCreateString** and specify that the string has length zero, then out will come a null pointer. It is legal to assume that a non-null `HSTRING` represents a non-empty string. Conversely, it is legal to test an `HSTRING` against a null pointer to see whether the string is empty.

Okay, so now I cop to the lie: An `HSTRING` is not always a reference-counted string.

There are these things called fast-pass strings. Fast-pass strings are `HSTRING`s that involve no memory allocation. If you have a buffer that you want to turn into an `HSTRING`, and you promise not to modify the buffer for the lifetime of your `HSTRING`, then you can use the **WindowsCreateStringReference** function to create an `HSTRING` *around* your buffer. The resulting `HSTRING` is a legal `HSTRING`, but instead of allocating memory on the heap for a reference-counted object, it uses the `HSTRING_HEADER` structure which you passed to the **WindowsCreateStringReference** function to store the metadata, and it uses the buffer you passed to the function to store the string contents.

It's called a fast-pass string because this special string doesn't require any memory allocation, and no data copying occurs.

When you are finished with a fast-pass string, you just abandon the `HSTRING`. The underlying memory for the fast-pass string was provided by you, so you are still on the hook for freeing that memory as appropriate.

The existence of fast-pass strings explains why the **WindowsDuplicateString** function returns you another `HSTRING`: If the original string is fast-pass, then the **WindowsDuplicateString** function needs to convert it to a true reference-counted heap-allocated object, and then it returns an `HSTRING` to that heap-allocated string. (On the other hand, if the `HSTRING` is already a heap-allocated string with a reference count, then the **WindowsDuplicateString** function merely increments the reference count and returns the same `HSTRING` back.)[1]

The rules for managing `HSTRING`s therefore go like this:

- If you receive an `HSTRING` as a function parameter, you are welcome to use it as-is until your function returns, but don't call **WindowsDeleteString** on that string, because you are not the owner of the string. It was merely lent to you. (This is the same rule that applies to COM reference counts.)[2]
- If you need to keep using the `HSTRING` after the function returns (say, because you're saving it in a member variable), you must use `WindowsDuplicateString` and use the duplicate.
- Each call to **WindowsCreateString** or **WindowsDuplicateString** (or one of the helper functions that creates a string) should be matched to exactly one call to **WindowsDeleteString** which is passed the same handle that **WindowsCreateString** or **WindowsDuplicateString** returned.

You can think of fast-pass strings as lazy-heap-allocated strings: They get copied to the heap only if somebody needs to extend the lifetime of the string beyond the lifetime of the function.

The WRL library has wrapper classes for `HSTRING`s: The `HString` class manages an `HSTRING`, and the `HStringReference` manages a fast-pass `HSTRING`.

[1] In theory, a debugging version of the **WindowsDuplicateString** function could create a full duplicate of the string anyway. That way, when you have an `HSTRING` leak, you can use heap leak tools to find the code that duplicated the string and failed to destroy it. I don't know if this theory actually occurs in practice.

[2] COM violates its own rule with the `CoGetInterfaceAndReleaseStream` function, and that lapse came back to bite us.

Raymond Chen

Follow