How does Windows decide whether a newly-created window should use LTR or RTL layout?

devblogs.microsoft.com/oldnewthing/20220523-00

May 23, 2022

Raymond Chen

You can specify in a window's extended styles whether it follows left-to-right (LTR) or right-to-left (RTL) layout. The right-to-left layout is used in languages that are written from right to left, of which the most widely used today are probably Arabic and Hebrew. You can request right-to-left layout by setting the `WS_EX_LAYOUTRTL` extended style.

If you don't specify the `WS_EX_LAYOUTRTL` extended style, the system may still apply that style automatically, based on the following rules:

Scenario		Rule
Child window	Parent omits `WS_EX_NOINHERIT-LAYOUT`	Inherit `WS_EX_LAYOUTRTL` from parent.
	Parent has `WS_EX_NOINHERIT-LAYOUT`	Remain LTR.
Top-level window	Owned window	Remain LTR.
	Unowned window	Follow `GetProcessDefault-Layout`.

In the case of a top-level unowned window, the `WS_EX_LAYOUTRTL` extended style is automatically set if the process default layout has the `LAYOUT_RTL` bit set.

As I noted some time ago, if a process never calls `SetProcessDefaultLayout`, then the initial process default layout is inferred by inspecting the FileDescription version property of the primary executable: If it begins with two left-to-right marks (LRMs, represented by Unicode code point U+200E), then the process default layout is set to `LAYOUT_RTL`.

I also noted some time ago that you can ask `GetLocaleInfoEx` for the `LOCALE_IREADING-LAYOUT` to determine whether any particular language is LTR or RTL.

```
// Direction values:
// 0 = left to right (e.g., English)
// 1 = right to left (e.g., Arabic)
// 2 = top to bottom, right to left (e.g., classical Chinese)
// 3 = top to bottom, left to right (e.g., Mongolian)

int GetLanguageReadingLayout(PCWSTR languageName)
{
    int direction = 0;
    THROW_IF_WIN32_BOOL_FALSE(
        GetLocaleInfoEx(languageName,
                        LOCALE_IREADINGLAYOUT | LOCALE_RETURN_NUMBER,
                        reinterpret_cast<LPWSTR>(&direction),
                        sizeof(direction) / sizeof(wchar_t)));
    return direction;
}

int GetSystemDefaultLanguageReadingLayout()
{
    return GetLanguageReadingLayout(LOCALE_NAME_SYSTEM_DEFAULT);
}

int GetUserDefaultLanguageReadingLayout()
{
    return GetLanguageReadingLayout(LOCALE_NAME_USER_DEFAULT);
}
```

You typically are interested in the primary language for the current thread, since that's the one that most influences which language resources your program will use. You can use `GetThreadPreferredUILanguages` to get all the languages that apply to the current thread, and then pass the first one to `GetLanguageReadingLayout`. Calling the `GetThreadPreferredUILanguages` function is a bit frustrating because the list of applicable languages can change asynchronously (if another thread calls `SetProcessPreferredUILanguages` It is double annoying because the wil helper function `AdaptFixedSizeToAllocatedResult` assumes null-terminated strings and doesn't support double-null-terminated strings, so we have to write out the loop manually.

```
namespace wil
{
    template<typename string_type, size_t stackBufferLength = 40>
    HRESULT GetThreadPreferredUILanguages(DWORD flags,
        _Out_ PULONG languageCount, string_type& result)
    {
        wchar_t stackBuffer[stackBufferLength];
        ULONG required = ARRAYSIZE(stackBuffer);
        if (::GetThreadPreferredUILanguages(flags, languageCount,
                                            stackBuffer, &required))
        {
            result = make_unique_string_nothrow<string_type>
                                                (nullptr, required);
            RETURN_IF_NULL_ALLOC(result);
            memcpy(result.get(), stackBuffer,
                   required * sizeof(wchar_t));
            return S_OK;
        }
        DWORD error = ::GetLastError();
        while (error == ERROR_INSUFFICIENT_BUFFER)
        {
            result = make_unique_string_nothrow<string_type>
                                                (nullptr, required);
            RETURN_IF_NULL_ALLOC(result);
            if (::GetThreadPreferredUILanguages(flags,
                         languageCount, result.get(), &required))
            {
                return S_OK;
            }
            error = ::GetLastError();
        }
        RETURN_WIN32(error);
    }

    template <typename string_type = wil::unique_cotaskmem_string,
              size_t stackBufferLength = 40>
    string_type GetThreadPreferredUILanguages(DWORD flags,
        _Out_ PULONG languageCount)
    {
        string_type result;
        THROW_IF_FAILED((wil::GetThreadPreferredUILanguages<
            string_type, stackBufferLength>
            (flags, languageCount, result)));
        return result;
    }
}
```

We can now plug this into our existing function.

```
int GetDefaultThreadLanguageReadingLayout()
{
    ULONG count;
    return GetLanguageReadingLayout(
        wil::GetThreadPreferredUILanguages(MUI_LANGUAGE_NAME |
            MUI_MERGE_UI_FALLBACK, &count).get());
}
```

Sorry it's such a pain.

Bonus chatter: All of this logic assumes that your program has been translated into RTL languages in the first place. If your program is English-only, don't display your English strings in RTL. As I noted in that article, you can leave a breadcrumb in the resources to tell you which direction the resources expect strings to read.

Raymond Chen

Follow