Yes, Virginia, Windows 95 *Does* Do Unicode!

Creating Unicode apps for Windows 95

by Nadine Kano

Windows NT, an operating system built from the ground up, processes characters internally in Unicode.

Windows 95, an operating system that originates from the code base for Windows 3.*x*, processes characters internally in local character sets—the same character sets used by Windows 3.*x*. (For a description of Unicode and other character sets, see "Creating software for a global market: Character sets and Microsoft Windows," *Microsoft Developer Network News*, July 1993.)

Therefore, we can safely say that Windows NT does Unicode, but Windows 95 does not, right? Not exactly.

What does "doing" Unicode mean?

First of all, what does it mean to say that an operating system "does" or "doesn't do" Unicode? Windows 95 itself does not process strings in Unicode. The Win32 API on Windows 95 expects all string parameters to be encoded in the local character set of the system—the default local character set changes depending on the language edition of Windows 95 (except for the five wide character APIs listed below).

For example, Western European editions use the ANSI character set (also called code page 1252), while the Eastern European editions use code page 1250, and the Japanese edition uses Shift-JIS (also called code page 932). So you can say that Windows 95, the operating system, does its internal processing in local character sets only.

This limitation does *not*, however, extend to applications that run on the operating system. Even though Windows 95 doesn't process strings internally using Unicode, applications running on Windows 95 still can!

If you are dealing with multilingual documents, you may prefer to store all your data in Unicode, because it provides unambiguous code point values for a wide range of characters used in a wide range of languages. If your application processes Japanese, Chinese, or Korean characters, you may find it easier to manipulate strings in Unicode, which is a fixed-width character set, unlike alternative character sets, which are a mixture of one- and two-byte encodings. Unicode is also a good means for sharing data with other systems.

Translating needed

If you want to base your application on Unicode and run it on Windows 95, you need to translate string parameters to local character sets before calling API functions. (The only wide character APIs that handle Unicode on Windows 95 are **TextOutW**, **ExtTextOutW**, **GetCharWidthW**, **GetTextExtentW**, and **GetTextExtentPointW**.)

The API for translating strings from Unicode into the system's default local character set is **WideCharToMultiByte**. When you receive string-based information from Windows 95, you need to translate it into Unicode by calling **MultiByteToWideChar** before working with it.

Obviously translating strings to and from Unicode every time you call an API that takes string parameters adds overhead to your program. It's up to you to weigh the pros and cons of this approach to suit your program's requirements.

Local character sets on Windows NT

Perhaps it makes more sense for your application to use local character sets. Unicode's 16 bits per character may be too large to accommodate your space constraints, even with compression. You may have an existing code base that uses local character sets and meets your needs. If your application is based on a local character set, will it still run correctly on Windows NT? Absolutely.

Windows NT supports two flavors of the Win32 API —'W' or wide character entry points, for string parameters encoded in Unicode, and 'A' or ANSI entry points, for string parameters encoded in local character sets. If you call the A entry points, Windows NT will automatically convert the string parameters into Unicode before calling the W version of the same function. For this reason, using local character sets while running on Windows NT does add some overhead to your application.

Creating a single binary

What if you want to create a single binary that will run unchanged on both Windows 95 and Windows NT? Is this possible? If so, which character encoding should you use?

Because both Windows 95 and Windows NT can handle either Unicode or local character sets, you can create a single binary to run on both systems *and* you can choose the character encoding that makes the most sense for your application. The **GetVersionEx** API function will tell you at run time which operating system your application is running on. Based on this information, you can take the appropriate steps.

For example, let's assume that your application is based on Unicode. If you determine that it is running on Windows NT, you can call the wide character API entry points directly. When the same application runs on Windows 95, you will have to convert string parameters to the system's local character set with **WideCharToMultiByte** before calling the ANSI-based API entry points.

Even though W APIs are not implemented on Windows 95, their entry points are stubbed. Therefore, it is safe to write code that calls the W APIs. If your application is based on a local character set, simply call the A versions of the Win32 API functions—both Windows NT and Windows 95 support them. (Don't forget that there are slight differences in the API sets for Windows NT and Windows 95.)

It's important to remember that Unicode support on Win32 is not a black-and-white, all-or-nothing issue. You can create Unicode-based applications for Windows 95 even though the system itself isn't based on Unicode. At the same time, just because Windows NT is fully Unicode-based internally doesn't mean that your application is required to use Unicode when it runs on Windows NT.

Keep in mind that supporting Unicode will not automatically make your application "fully international." Unicode is a character encoding—it can make software internationalization easier in many cases, but it does not help you with culturally accurate sorting; date, time, or currency formats; input methods; user interface translation; text layout; font design and selection; or many other elements that make up an international program.

There are reasons to choose Unicode as your encoding and reasons not to. Win32 gives you the Unicode option, on *both* Windows NT and Windows 95. You need to decide, based on your users' needs and your development goals and constraints, what will work best for your application.

For more information, see chapter 3 in *Developing International Software for Windows 95 and Windows NT* published by Microsoft Press. (That's by me!)

Nadine Kano, a globalization specialist in Microsoft's Developer Relations Group, just completed a 768-page book on software internationalization. Now she spends most of her spare time sleeping and dulling her brain with sleazy tabloid TV shows.