Understanding Multilingual Software on MS Windows
The answer to the ultimate question of fonts, keyboards and everything

Peter Constable,
SIL IPub/Non-Roman Script Initiative (NRSI)

1. Introduction

A user in Thailand recently asked some intriguing questions about certain problems involving multilingual data. The problems had to do with data going being shared between expatriates and local Thai users. The locals were using regional versions of Microsoft (MS) software: Thai Windows 98 and Thai Office 2000. In contrast, the expatriates were using the corresponding US versions with Keyman keyboards[1] and fonts that date back to Thai Windows 3.1.

The questions reflect a scenario that's not limited to Thailand: *"If I give a local my data, they can view it if they use my font, and they can edit it using a Keyman keyboard. Going the other way, I can see their data if I view the data in Word 2000 and if I use their font, but I can't use my fonts. I also can't view it in some other apps, and I can't edit it."* For this user, the differences between the two situations seemed perplexing.

Explanations were provided as to the reasons for the differences, and how expatriate users could update their US Windows systems so that they could work with data created by their Thai counterparts. Using the very latest versions of MS software, the differences could be eliminated.

All this was interesting, but it was the next question that caught my attention: *"So, can I get this to work for me in Shoebox and Paratext?"*[2] I suddenly realised that the answer, which is "no", was not fully obvious to many users and support personnel.

This question suggested to me that there's a general need here for more education. In many places around the world, similar scenarios have been played out, and users continue to experience frustration working with multilingual or non-Roman data. Users are looking for solutions, but are also looking for solutions that work with the applications they have relied on. There is a need to understand how applications that deal with multilingual data work, to understanding what the limits and potential are of different approaches, and to understand why some things are possible with some applications but not with others.

So, I will describe some significantly different approaches to working with multilingual data that have been available as Windows has developed since Windows 3.1. I will also attempt to explain some of the technical details of the life cycle of a "character" from keyboard to display, but hopefully without getting too

[1] *Tavultesoft Keyboard Manager*, also known as "Keyman", is a utility for creating keyboard input methods. For further information, see http://www.tavultesoft.com/.

[2] *The Linguist's Shoebox* (or "Shoebox") is an application for use in linguistic research that has been developed by SIL International. *Paratext* is an application for use in Bible translation that has been developed by United Bible Societies. Both of these applications have been designed to work on Windows 3.1 or on Windows 95. They follow the *Win3.1* paradigm for handling multilingual data, described in §4.1.

technical. It's a complex issue with lots of permutations, though, so be warned: this is a somewhat long answer to a not-so-easy question.

2. Basics about codepages, fonts and keyboards

I'll start first with a few fundamentals regarding codepages, fonts and keyboards in MS Windows, and a little on Windows programming interfaces. I will assume basic familiarity with fundamental notions such as *characters*, *codepoints*, *keystrokes* and *glyphs*. These notions are discussed in Constable (2000b). Given the technical nature of the issues discussed here, I encourage you to familiarize yourself with these core elements of multilingual text processing before proceeding.

2.1 Codepage basics

A codepage defines a set of characters for a language or for a set of languages, and it also defines a mapping for those characters between an 8-bit encoding and Unicode. So, for example, the Windows codepage for the "Western" (or "Latin 1") character set, codepage 1252 (hereafter, cp1252), specifies a set of Latin characters used for Western European languages, and maps between the Unicode representation of these characters and an 8-bit representation.

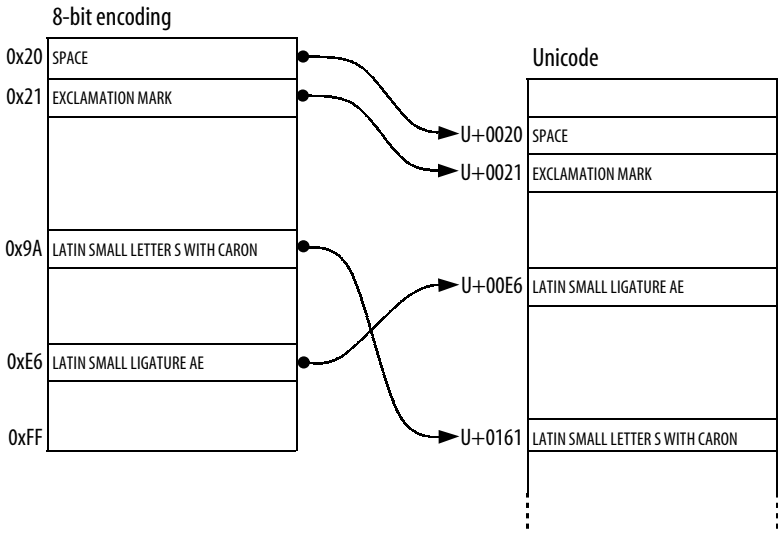

Figure 1: Codepage 1252: a character set, and a mapping from an 8-bit encoding to Unicode

For instance, the Western character set includes "æ" LATIN SMALL LIGATURE AE, and provides the mapping for this character between 0xE6 in 8-bit representation and U+00E6 in Unicode. Similarly, it includes "š" LATIN SMALL LETTER S WITH CARON, and maps this between 0x9A and U+0161. Several other codepages are defined within Windows to cover characters sets for other groups of languages. These include "Arabic", "Baltic", "Cyrillic", "Hebrew", "Greek" and several others (see Figure 2).

It should be noted that, for Far East codepages such as Japanese and Korean, the 8-bit encoding that is used is a *double-byte* encoding, in which certain pairs of bytes are used in combination to encode characters. For example, in the codepage for Traditional Chinese (cp950), the combination <0xB5, 0x44> is used as the 8-bit encoding for the character 湄. These codepages otherwise work as any other, however, defining a character set and providing a mapping to Unicode. Thus, cp950 also provides the mapping from the combination <0xB5, 0x44> to the Unicode value for this character, U+6E44.

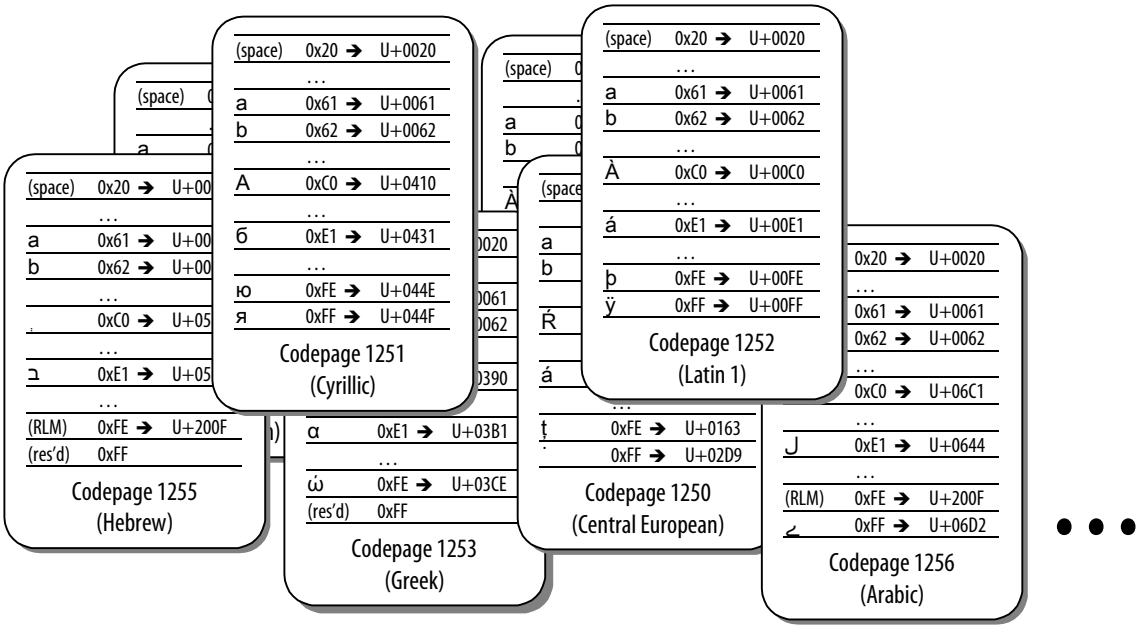

Figure 2: Windows has codepages for several character sets

For more information about codepages, see §2 of Constable (2000c).

2.2 Keyboard basics

Next, some facts about keyboards. Keyboards on Windows 3.x, Windows 95, Windows 98 and Windows Me (hereafter, Win3.1/9x/Me) generate only 8-bit values. This is true even when using an app like Word 2000 that stores text in terms of Unicode. In contrast, keyboards on Windows NT 4 and Windows 2000 (hereafter, WinNT/2K) generate only 16-bit Unicode character codes.

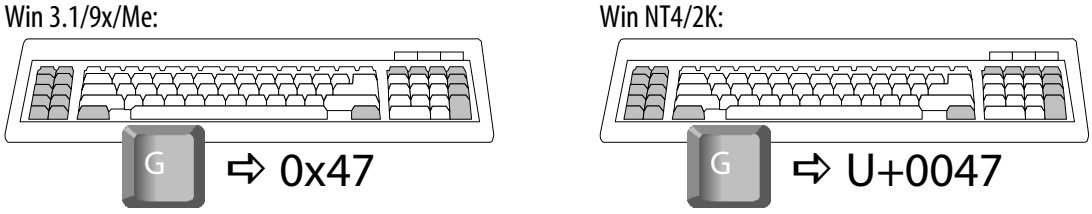

Figure 3: Windows keyboards: 8-bit on Win 3.1/9x/Me, Unicode on Win NT/2K

A "keyboard" configuration in Windows combines a particular language and a particular layout; each language has a *language identifier* (a "LANGID") and a default layout associated with it. Each LANGID has a character set (and, therefore, a codepage) associated with it.[3] Thus, software that needs to can know how to convert 8-bit values in this language into Unicode, or Unicode values into 8-bit values. The way this works on Win3.1/9x/Me and on WinNT/2K is illustrated for English in Figure 4 and for Greek in Figure 5.

[3] For newer languages supported on Win2K, such as Hindi, the character set is the entire Unicode character set; there is no 8-bit codepage for these languages.

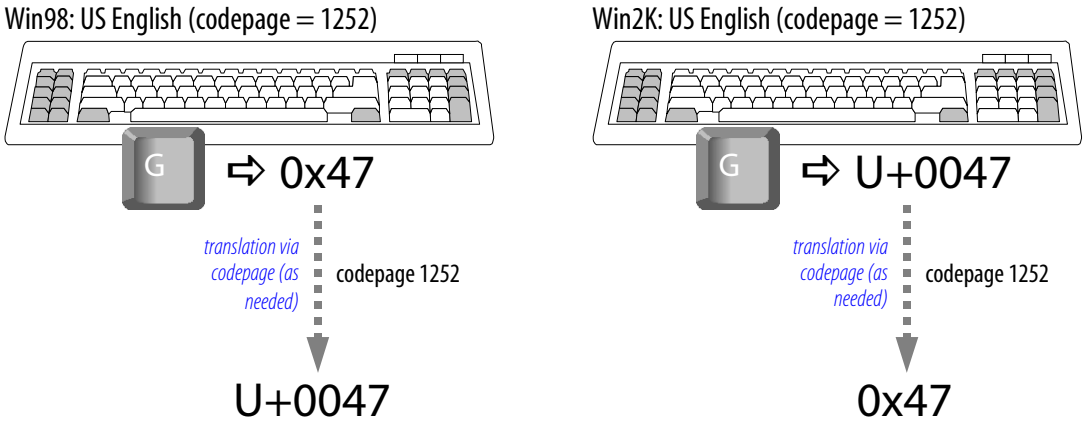

Figure 4: US English keyboard on Win98 and Win2K

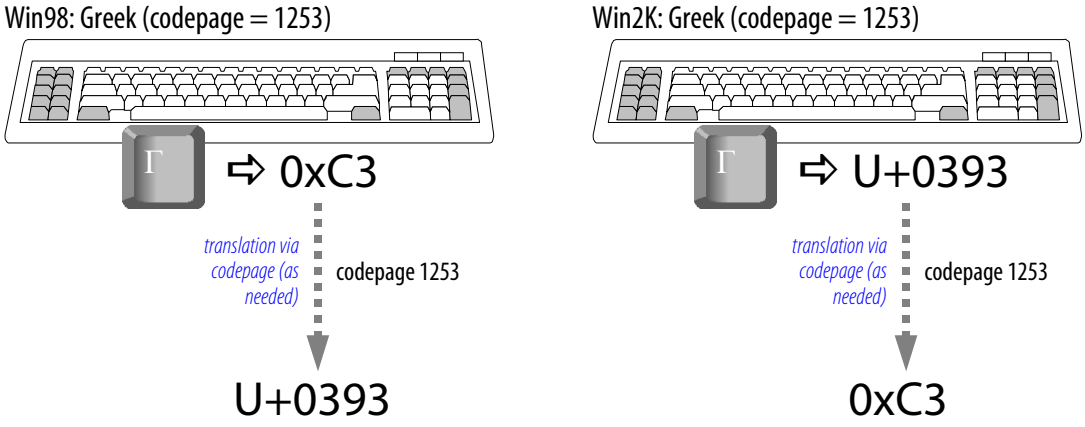

Figure 5: Greek keyboards on Win98 and Win2K

2.3 TrueType font basics

There are some important points about TrueType fonts that we need to be aware of. First of all, TrueType fonts are not limited to a maximum of 256 glyphs. In fact, TrueType fonts can have over 65,000 glyphs. Most fonts have only a small number—a few hundred, perhaps. Many users already have at least one font with tens of thousands of glyphs installed on their system, however. For instance, the Arial Unicode MS font has 51,180 glyphs.

Since fonts can have a large number of glyphs, it isn't difficult to create a font that supports different scripts; for example, to have a font that supports Roman, Arabic and Hebrew. This leads to a second important point about TrueType fonts: since Win95, there have been ways to include information in a TrueType font to indicate what codepages the font supports. Thus, a font that has glyphs for Roman (for Western European languages), Hebrew and Arabic could contain information to indicate that it will work with cp1252 ("Western"), cp1255 ("Hebrew") and cp1256 ("Arabic").

The third important point is that TrueType fonts on the Windows platforms are usually encoded in terms of Unicode character values.[4] That is, Unicode values are always used to access glyphs inside the font. This has been true for all varieties of Windows since TrueType fonts were introduced with Win3.1.

The way this happens is illustrated in Figure 6. A font contains a set of tables. One of these (the 'glyf' table) contains all of the glyph outlines. There is a separate table (the 'cmap' table) that is used to identify the glyph that is associated with each character. On MS Windows, this table references characters in terms of their Unicode values. When an application asks Windows to display some text, Windows obtains the Unicode values for the characters in the string, and uses those Unicode values to find the appropriate glyph for each character.

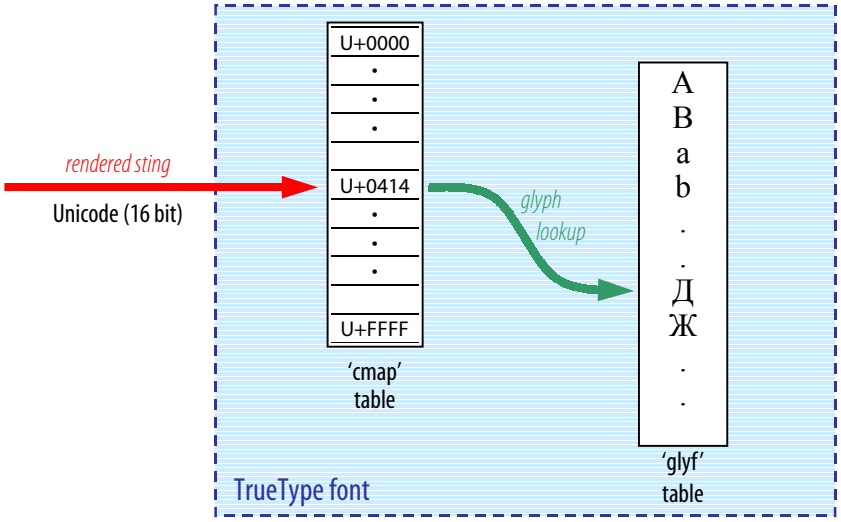

Figure 6: TrueType fonts on Windows: glyphs accessed via Unicode values

It is essential to understand that only Unicode values are used inside the font, even in the case of custom fonts that use non-standard encodings. For example, suppose you created a custom font and thought that the glyph for (say) stroke-L "Ł" was encoded in the font as d131 (0x83). In fact, inside the font this glyph is accessed using some Unicode value. Even though an application may store only 8-bit characters, by the time the codepoint gets to the font, something has had to change it to Unicode. Typically, custom fonts were designed to run on US versions of Windows, and what was actually happening was that these fonts were using character values from the Western character set but applying glyphs for other custom characters. Here's what happens with a normal TrueType font:

[4] This is not a requirement of TrueType fonts on Windows, but it is true of the vast majority. 8-bit-encoded TrueType fonts are the rare exception on Windows.

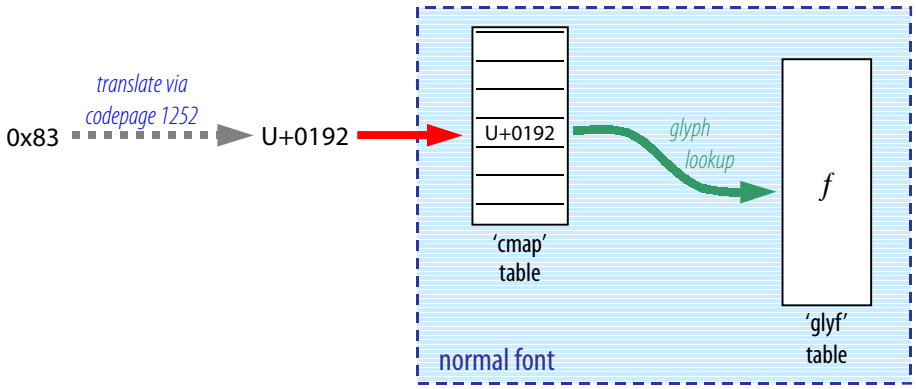

Figure 7: Rending a Western character using a normal font

In contrast, here's what happens with a custom font that assigns a non-standard character set to codepage 1252:

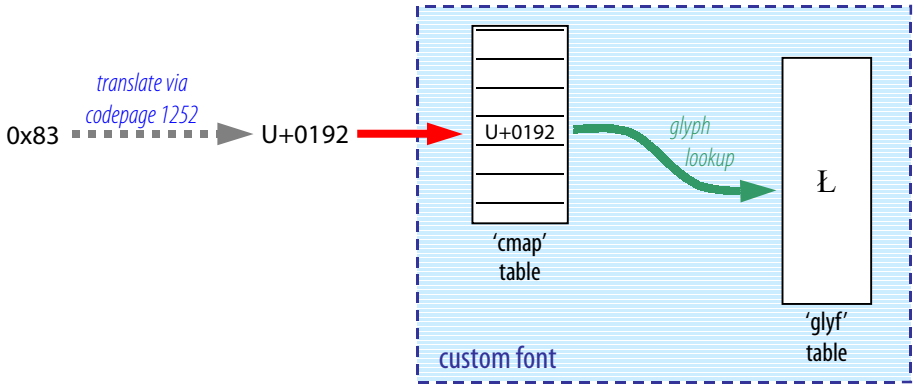

Figure 8: Rending a non-standard character using a custom font and codepage 1252

The custom font simply replaces the appropriate glyph for U+0192 with a different glyph. Custom fonts are hacking both Unicode, and the codepage for the Western character set (cp1252). The key point to remember is that, in every case, the font is using Unicode values for characters.

At this point, you may be wondering about some things; for instance, *If a TrueType font only knows Unicode, and my Win98 keyboard outputs only 8-bit characters, where does the conversion from 8-bit to Unicode take place? Do custom-encoded fonts have any effect on how that happens?* I'll explain these things beginning in §3. First, though, we need to talk briefly about Windows programming interfaces and root beer (or, at least, A&W).

2.4 "A&W" variations of Win32 interfaces

One way to think of Windows is as a collection of programming subroutines and functions that an application developer can call upon to perform various low level tasks, such as opening a file or displaying text on the screen. These subroutines and functions are known as *application programming interfaces* (APIs). There are some important differences between Win9x/Me and WinNT/2K with regard to Unicode support. These differences appear in the interfaces that an application developer would use.

The Windows API functions require various parameters, many of them being integer values. Integer parameters are used for a number of important things, like memory locations or screen co-ordinates. Up to Win3.x, most of these integer parameters were 16-bits wide. Therefore, the versions of the Windows APIs used in these early versions of Windows became known as *Win16*.

For Windows NT and Win95, the Windows APIs were totally revamped. As part of this, the 16-bit integer parameters used in Win16 were changed to 32-bit. Accordingly, this version of the Windows APIs became known as *Win32*.

As the Win32 system was being developed, Microsoft had decided that Unicode was the way of the future for text. At the same time, they needed to be able to support legacy systems and data that used 8-bit text. To deal with this, they created Win32 with alternate 8-bit and Unicode text-handling capabilities. In particular, any application interface into Win32 that related to string handling was provided in two varieties: an "A" ("ANSI", i.e. 8-bit) version, and a "W" ("wide", i.e. 16-bit, Unicode) version.[5]

There was a significant difference between the way Win32 was implemented on Win95 and on NT, however: on Win95, only a small number of the "wide"-version interfaces are supported; for most interfaces that relate to string handling, only the "ANSI" versions are available. This is true of Win98 and Me as well.

We will look later at some particular interfaces and the impact of the two variations of those interfaces on how multilingual software works. For now, I will just mention one group of interfaces: the interfaces used for drawing text on a screen or printer (`TextOut` and `ExtTextOut`) are among the few that are available in both "wide" and "ANSI" versions on Win9x/Me. This makes it possible for an application on Win9x/Me to store text that is encoded in Unicode and display it without requiring any codepage translation before the rendering process. For instance, this allows Word 97 to display Yi or Ethiopic characters, for which there is no codepage, even when running on Win9x/Me.

3. The life-cycle of a "character": from keystroke to display

In this section, we want to get a general overview of how characters are processed in Windows, starting with a keystroke, and ending with a rendered glyph.

People often equate characters, codepoints, keystrokes and glyphs. These are all different, but there are relationships between them, and certain processes by which we get from a keystroke to something appearing on the screen. As mentioned in §1, I'm assuming familiarity with some basic notions: that codepoints are the encoded representation of characters, that keystrokes are used to generate them, that glyphs are used to present them, and that the relationships between keystrokes and codepoints and between codepoints and glyphs are not necessarily one-to-one. (This is all discussed in detail in Constable 2000b.) For simplicity, I'll use language that assumes one-to-one relationships between keystrokes, codepoints and glyphs, even though we know that's not always true.

As we consider how characters are processed in Windows, there are some differences between Win3.1/9x/Me and WinNT/2K. I'll consider Win3.1/9x/Me first, then go on to explain the differences that pertain to WinNT/2K.

---

[5] The term *ANSI* has been used in a variety of ways in the context of Windows. In some cases, it has been used as an alternate name for cp1252. In other cases, it has been used to refer to all 8-bit encodings associated with the Windows codepages. The term actually is an acronym for "American National Standards Institute". The reason why this name was originally introduced into the Windows context was that cp1252 was adapted from a draft ANSI encoding standard. That standard (now an international standard: ISO 8859-1) turned out to be different from cp1252, however, and so the term is really not appropriate for use in relation to Windows. The disparate uses also creates confusion. Because ANSI in the sense of *any 8-bit text* was the basis for the "A" in the "A" vs. "W" distinction in Win32, though, we need to maintain that use of the term here.

3.1 Character processing on Win3.1/9x/Me

As mentioned in §2.2, keyboards on Win3.1/9x/Me generate only 8-bit codes. It was mentioned in §2.3, however, that a Unicode codepoint is needed to access the corresponding glyph in the font. Somehow, at some point, an 8-bit code needs to get converted into a 16-bit code, as illustrated in Figure 9.

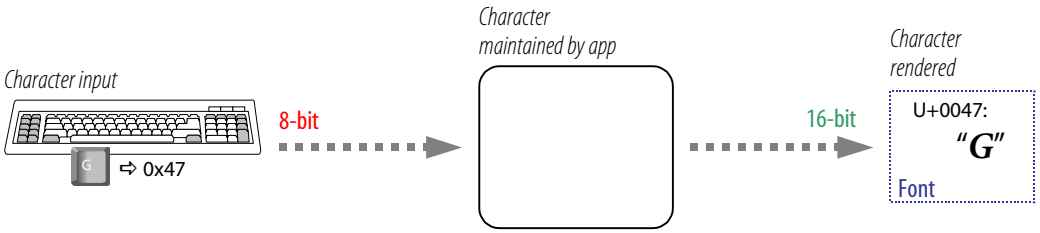

Figure 9: Characters on Win3.1/9x/Me: begin as 8-bit, end as 16-bit

In principle the conversion from an 8-bit codepoint to a 16-bit Unicode value can happen at one of three points. First, the conversion could be done by the operating system before the character reaches the app:

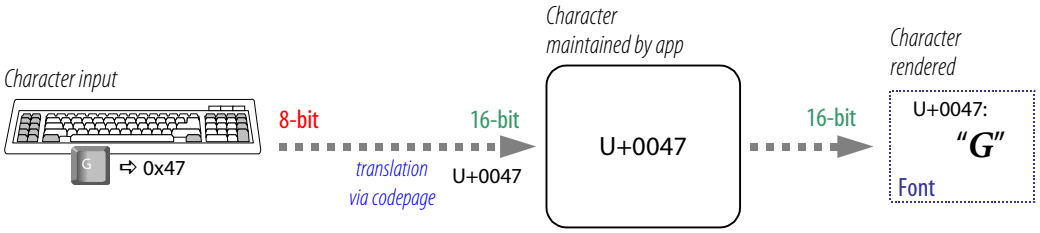

Figure 10: Win3.1/9x/Me: conversion to 16-bit between keyboard and app

In practice, this has never been done in Win3.1/9x/Me for a very simple reason: these versions of Windows have never supported a mechanism to pass a character from a keyboard to an application when that character is encoded in Unicode. Thus, the statement about 8-bit keyboard input on these platforms that was made in §2.2 is not as strictly worded as it could be: not only is it true that the keyboard handlers provided by Win3.1/9x/Me output only 8-bit codepoints, but also Win3.1/9x/Me have provided means for apps to receive only 8-bit codepoints from a keyboard.[6] Further details of this will be provided in §3.2.

Another point in the life of a character at which the conversion from 8-bit to 16-bit could also happen is after the character is output by the app in the rendering process:

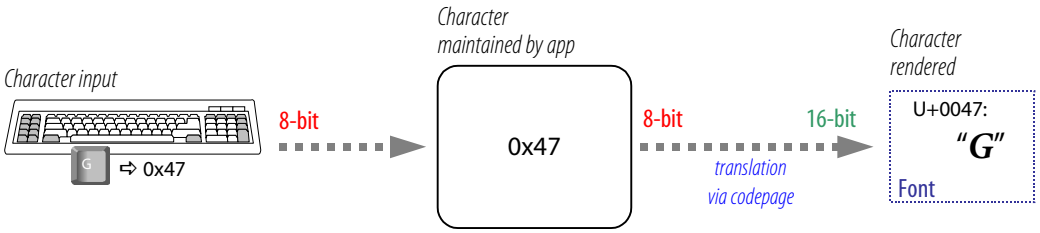

Figure 11: Win3.1/9x/Me: conversion to 16-bit handled by OS after characters have been output by app for rendering

[6] An application could avoid the issue by handling all aspects of keyboard input itself, starting from the keyboard hardware device driver. From an application developer's perspective, this would be a very costly approach, and not worth it.

In this situation, the app receives input characters encoded as 8-bit codepoints, and does all it's processing and storage in terms of that 8-bit encoding. It also uses that encoding when it asks Windows to display the characters.[7] At that point, Windows automatically converts the 8-bit codepoints to Unicode in order to access the glyphs in the font. This is, in fact, the way most software has worked in the past. We will examine how this is done in more detail in §4.

The third point in the life of a character at which the conversion from 8-bit to 16-bit could also happen lies between the previous two: while the character is being maintained by the app:

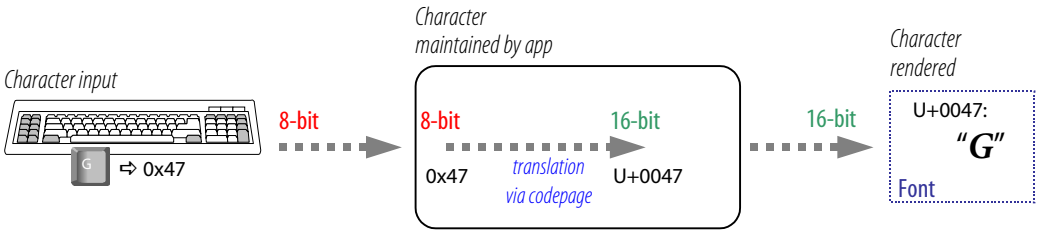

Figure 12: Win3.1/9x/Me: conversion to 16-bit handled while data is maintained by the app

In this situation, the application receives input characters encoded as 8-bit codepoints. Now, the app could continue to do most of its processing and also store the document using the 8-bit encoding, and then convert the characters into Unicode just prior to rendering.[8] There would not be any strong reason for doing this, however: the app doesn't gain any benefit from the Unicode representation since it only sees the Unicode values just immediately before outputting the character(s) to a display. Furthermore, the same effect could have been accomplished if the app had output the 8-bit encoding of the data and allowed the operating system to do the conversion, as described in the previous scenario.[9]

On the other hand, the app could convert characters as soon as it receives them from the keyboard, and then do all of its processing and storage using Unicode. This is, in fact, what Unicode-based applications such as Word 2000 do when running on Win9x/Me (this will be discussed further below).

So, of the various points at which conversion from 8-bit to Unicode could potentially occur, two are actually used: for many applications, conversion to Unicode is handled by the operating system when the app asks Windows to display the text. Some apps handle converting characters to Unicode themselves, doing this as soon as a character has been received from the keyboard. These two situations correspond to two important, basic categories of apps: those that store 8-bit text, and those that store Unicode text. These different categories of applications will be discussed further in §3.3.

3.2 Character processing on WinNT/2K

As mentioned in §2.2, keyboards on WinNT/2K generate only 16-bit codes. Since this is also what is required to access glyphs in a TrueType font, we might think that everything is simple: characters on WinNT/2K are only ever represented using 16-bit codepoints. Not surprisingly, things are never that easy. Bear in mind it has to be possible for applications that run on Win9x, and that work with 8-bit text, to be able to run on WinNT/2K as well. Before I can explain how character processing occurs on WinNT/2K, it will be necessary to explain some fairly technical background issues.

---

[7] Recall from §2.4 that the Windows API functions for drawing text can have "ANSI" and "wide" variants. An application gets Windows to display 8-bit-encoded text by using the "ANSI" variant of a text-drawing function.

[8] An application gets Windows to display Unicode-encoded text by using the "wide" variant of a text-drawing function.

[9] The *Script Definition File* rendering system is able to work this way, however. See note 35.

Recall from §2.4 that some Windows APIs come in "ANSI" and "wide" versions. One of these is a function that is used as an application is loading to make Windows aware of its presence. This function is called `RegisterClassEx`.[10] For this function, the choice between the "A" and "W" variants has significant implications for almost everything else that happens while the app is running. This is particularly true for keyboard input.

The standard mechanism in Windows by which keyboards send characters to an application is a special system message known as `WM_CHAR`. In general, this message is capable of sending character codes that are either 8-bit (ANSI) or 16-bit (Unicode). The choice between whether Windows passes 8- or 16-bit codes in the `WM_CHAR` message is determined by which version of the `RegisterClassEx` function the app used during initialisation.

On WinNT/2K, both versions of the `RegisterClassEx` function are available. That means that an app can initialise itself either as an ANSI app or as a wide app, and Windows will pass `WM_CHAR` messages that match. Keyboards will always generate a 16-bit codepoint, but if the receiving application used the ANSI version of `RegisterClassEx`, then Windows will automatically convert the 16-bit value into an 8-bit codepoint using the appropriate codepage for the given keyboard.[11] The wide version of `RegisterClassEx` is also available, though, and an app that uses the wide version will receive `WM_CHAR` messages that contain 16-bit Unicode characters. That has made it possible to create keyboards on Win2K for languages like Hindi and Tamil that use characters for which there is no codepage support.

On Win9x/Me, however, the wide version of the `RegisterClassEx` function is not supported. This means that, on those platforms, the `WM_CHAR` message can pass characters only as 8-bit codepoints.[12] Since application developers usually would like their application to run on both Win9x/Me and on WinNT/2K, they need to accommodate the limitations of Win9x/Me. As a result, most current applications use the "ANSI" version of `RegisterClassEx`, and therefore can only receive characters that are input from the keyboard in terms of 8-bit codepoints, even when the application is running on WinNT/2K. This will be discussed further in §4.

Given this background, we can now explore how characters are handled on WinNT/2K. This can happen in several different ways. The simple case applies to applications that register themselves as "wide" by using the "wide" version of `RegisterClassEx`. In this situation, everything is done in terms of 16-bit values:

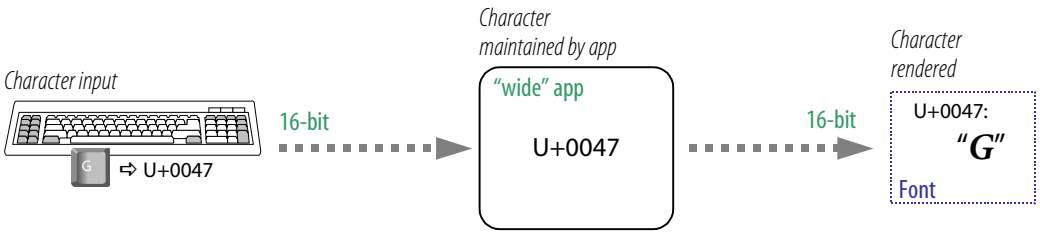

Figure 13: WinNT/2K: 16-bit everywhere for "wide" apps

If, however, an application registers itself by using the "ANSI" version of `RegisterClassEx`, then the 16-bit codes generated by a keyboard get converted into 8-bit, and so we have mixed 8- and 16-bit processing as with Win3.1/9x/Me. In fact, we have exactly the same possibilities that we saw in §3.1, but with the

[10] Win3.x used an earlier, Win16 function, `RegisterClass`. Win16 API functions did not have "ANSI" and "wide" versions. Effectively, all Win16 interfaces were "ANSI" versions.

[11] Recall from §2.2 that keyboards on WinNT/2K, as on other versions of Windows, have a LANGID and also codepage associated with them.

[12] This is the reason why 16-bit character input has not been possible on Win3.1/9x/Me, as described in the previous section. It is also makes codepages essential on Win9x/Me.

additional complication that the codepoints generated by a keyboard first get converted from 16-bit to 8-bit. That we should encounter these same ways of processing characters shouldn't be surprising when we're considering applications that were designed to be compatible with Win9x/Me.[13]

So, for ANSI-registered applications running on WinNT/2K, one option is that the keyboard generates 16-bit codepoints which automatically get converted to 8-bit by Windows before being passed to the application, and then the 8-bit codepoints get automatically converted back to 16-bit by Windows after the application has output the text for rendering:

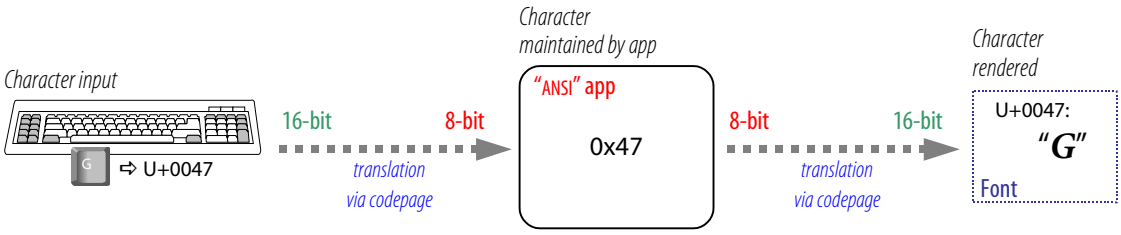

Figure 14: WinNT/2K: automatic conversion by the OS to 8-bit and back to 16-bit

The other option for ANSI-registered applications running on WinNT/2K is for the application to receive an 8-bit codepoint and immediately convert it to 16-bit, and then to store and process text using 16-bit codes:

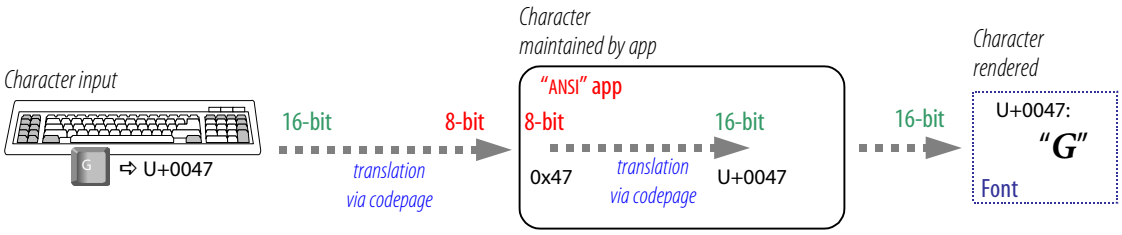

Figure 15: WinNT/2K: automatic conversion by OS to 8-bit, conversion to 16-bit handled by app

So, we have seen that characters can live one of two types of lives on Win3.1/9x/Me, and one of three types of lives on WinNT/2K.

3.3 8-bit versus Unicode, and "ANSI" versus "wide"

We saw at the end of §3.1 that there in an important 2-way categorization of applications: those that store 8-bit text, and those that store text as Unicode. We also saw in the previous section that applications must register themselves using either "ANSI" or wide version of `RegisterClassEx`,[14] and that this choice has an impact on whether 8-bit or 16-bit characters are received from the keyboard. It would be natural to expect that the 8-bit versus Unicode distinction correlates with the "ANSI" versus "wide" distinction. There are clear similarities, but we have seen that the two distinctions are independent: it is possible for an ANSI-

[13] The first option discussed in the previous section for Win3.1/9x/Me, involving conversion from 8-bit to 16-bit between the keyboard and the application, never occurs, either on Win3.1/9x/Me or on WinNT/2K. We saw that it doesn't occur for Win3.1/9x/Me since those versions of Windows do not provide any mechanism for an application to receive an input character as a 16-bit codepoint (and we saw in this section that this was due to the fact that the "wide" version of `RegisterClassEx` is not available on Win3.1/9x/Me). It doesn't occur on WinNT/2K for the simple reason that, if an app was capable of receiving input characters as 16-bit codepoints, then they would not have been converted into 8-bit in the first place: they would come directly from the keyboard as 16-bit codepoints.

[14] Or, on Win3.x, using the "ANSI"-only function `RegisterClass`.

registered application on Win9x/Me to store text as Unicode (see Figure 12). It would also be possible for a "wide" application to store its text as 8-bit, converting everything it receives into 8-bit, but there is not much reason to do so, and thus such an application is unlikely.

The following table summarises which combinations for these two distinctions can occur on various versions of Windows.[15] It also indicates which of the preceding figures showing the various processing models correspond in each case.

	app stores 8-bit text	**app stores Unicode**
app registered as "ANSI"	Win3.1/9x/Me (Figure 11), WinNT/2K (Figure 14)	Win9x/Me (Figure 12), WinNT/2K (Figure 15)
app registered as "wide"	WinNT/2K (possible, but not used)	WinNT/2K (Figure 13)

Table 1: Support for 8-bit vs. Unicode, and "ANSI" vs. "wide" on various versions of Windows

4. Windows multilingual software paradigms

In this section, we will look at the various paradigms that have been available to applications for dealing with multilingual data since Windows 3.1. Many of the important issues have been covered in §3. There are still some additional details that need to be mentioned, however.

In §3, we saw two different processing models used on Win3.1/9x/Me, and three used on WinNT/2K. Taken together, we saw in §3.3 that applications can work in one of three ways, when considered in relation to two independent parameters: storing text as 8-bit or as Unicode, and registering as either "ANSI" or "wide". Because of some additional details that have not yet been mentioned, there are actually a total of five paradigms for working with multilingual text that Windows applications can follow. We will examine each of these in this section.

4.1 The *Win3.1* paradigm

Windows 3.1 was designed using an approach we might call a *localization* approach, as opposed to a *multilingual* approach. What this meant was that an app was first designed for English and other Western European languages (the Western character set, i.e. cp1252). It would later be adapted for a different regional market; that is, adapted to support English plus some other language/character set, such as the Hebrew character set, which uses cp1255.

The key point here is that Win3.1 could handle only a single Windows codepage. Windows itself came in localised versions, each supporting one particular codepage to match the language or languages of the target market for that version. On a given system, text was assumed always to be defined in terms of that one codepage.

In theory, an app could have stored text in terms of Unicode if it wanted to, but it would have been difficult to do it, and there was no real benefit. Unless an app implemented its own proprietary keyboards and rendering system, which would have gone completely contrary to the whole Windows ethos, it could handle input and output only for the limited range of characters in a single Windows codepage. Note also that the app would still have to be aware of the system codepage so that it can interact with the operating system, and in order to handle behaviours that might be hard-wired into Windows. For example, in a Far East version, certain "upper-ASCII" bytes would be interpreted by Windows as lead bytes in a two-byte

[15] Note that Win3.1 is not included under ANSI-registered apps that store Unicode. This is discussed further in §4.1.

sequence; or in the Thai version, a codepoint might get remapped into a sequence of different codepoints. Applications designed for a different localised version of Windows might not behave appropriately. In summary, Win3.1 and Unicode simply were not made for each other, and apps that support Unicode have never been made to work on Win3.1.

So, Win3.1-style applications store text in terms of 8-bits, and interact with Windows using only one codepage.[16] The codepage used is determined by the particular installation of Windows. Because only one codepage is used, the number of characters supported is limited—fewer than 224 characters on non-Far East systems.[17] In terms of how characters are processed, the model shown in Figure 11 above is used. Overall, Win3.x was not really designed to be particularly multilingual.

There is one other detail that needs to be considered in terms of the *Win3.1* paradigm: Far East versions of Windows 3.x/9x/Me use *multi-byte* encodings, in which a character may be represented by a single byte, or by a pair of bytes. (Some byte values can even be used either as a single-byte codepoint, or as the second byte in a double-byte codepoint.) For a *Win3.1*-style application to work on a Far East version of Windows and to support a multi-byte codepage, it must be specially designed to do so. In general, *Win3.1* applications have been designed specifically for particular localised versions of Windows. For example, Shoebox and Paratext were designed with only US versions of Windows in mind.

4.2 The *Win95* paradigm: multilingual, 8-bit apps

By the time Windows 95 came out, Microsoft had realised that they needed to allow for greater flexibility in terms of multilingual support. The key change in Win9x that made this possible was the ability for an application to make use of multiple codepages and keyboards.[18]

When you install Win9x, many different codepages are installed on the system. Note, however, that one of these is defined as the default. That default codepage is used for the Windows user interface. It is also the one that is used if no specific codepage is specified when a mapping between 8-bit and Unicode is required.

To get multiple keyboards installed on a Win9x system, some special but easy steps are required:

1. Go into the Windows setup (use the Add/Remove Programs control panel) and install "multilingual support". (This option can also be enabled during installation.)
2. Once multilingual support is enabled, if you go into the keyboards control panel, you will see a tab labelled "Language"; this is where you can add keyboards for additional languages.
3. Once you add more than one keyboard, you will see an icon in the system tray, probably showing a two-letter language identifier (e.g. "En" for English). Clicking on this icon will bring up a menu from which you can switch to a different keyboard.

So, we can have multiple keyboards, and these may use multiple codepages. Remember, though, that somewhere along the way, the 8-bit codepoint needs to be converted to Unicode before a glyph can be retrieved. In Figure 11, we saw that Windows automatically does a codepage conversion in the process of rendering text. That model made no mention of selecting alternate codepages, however. To understand what an app needs to do in order for alternate codepages to be used, I need to explain how applications handle font selection.

---

[16] Win9x/Me and WinNT/2K provide multiple codepages. On any given installation, however, there is one that is designated as a default, or "system", codepage. This is the codepage that would be used for a Win3.1-style application when running on one of these versions of Windows.)

[17] Many have used custom fonts in order to overcome this character-set limitation. Such implementations are considered further in §6.

[18] Note: this is not talking about Keyman keyboards, but rather keyboards that are distributed with the operating system. The impact of Keyman will be considered in §6.

For an application to present a list of fonts to the user, it must first ask Windows for that list of fonts. It can do this in different ways. It can retrieve a list of real font names; that is, a name corresponding to each font file. For example, *Arial*; *Arial, Italic*; *Arial, Bold*; etc.. For the user, this is not the most useful way to list fonts, however. Instead, the app can ask for a list of *logical fonts* that correspond to font families, such as *Arial* and *Tahoma*. In other words, variations such as italic and bold are suppressed.[19]

There is another way in which an application can enumerate logical fonts. To understand this, recall from §2.3 that a a single TrueType font or font family can easily have glyphs that cover several different Windows codepages, and that it can contain information regarding which codepages it will support. Thus, an application can also enumerate fonts as logical fonts that correspond to a font family in combination with a particular character set.[20] So, for example, Arial has glyphs that cover the Arabic, Baltic, Central European, Cyrillic, Greek, Hebrew, Turkish and Western codepages. If fonts are enumerated using this type of logical font, the list of fonts will contain names such as *Arial (Arabic)*, *Arial (Baltic)*, etc. This type of enumeration is used, for example, in WordPad on Win95 and Win98. Enumerating logical fonts in terms of character set-specific variants is important for multilingual 8-bit apps, as I'll explain next.

When a keyboard is selected from the system tray, the app is notified and given the LANGID and the character set identifier (charset ID) for that keyboard. The app needs to take note of that information; in particular, it needs to mark the text to indicate which charset is used. At this point, it only makes sense to use a font that supports that charset. If the app has enumerated logical fonts in terms of charset-specific variants (as explained above), then it will be easy to present to the user a restricted list of font families that support that charset. The app could even automatically activate one of these for text that is subsequently entered. This behaviour is found in WordPad on Win95 and Win98.This could work the other way as well: a user can select a particular logical font that combines a font family with a charset (e.g. "Times New Roman (Hebrew)"), and the app could activate a keyboard that matches.

So, a keyboard and a charset-specific logical font are selected, each of them corresponding to a particular codepage. As 8-bit characters are entered into the app, the app keeps track of the charset-specific logical font that is applied to the text. When the app asks Windows to draw the text, it tells Windows to use that logical font. Based on this, Windows knows what codepage should be applied to that text. It will then take the 8-bit text and translate it into Unicode using that codepage, and the resulting Unicode characters are used to access glyphs in the font.

And so, we start with 8-bit codes from the keyboard, which get stored in the computer as 8-bit codes, but which get translated into Unicode between the app and the font. This is also what happened with Win3.1, and follows the basic character processing model shown in Figure 11. There are important differences, however. Win3.1 allowed only a single Windows codepage, the system codepage, to be used to map to Unicode, and therefore limited characters to that one character set. But, in a multilingual, 8-bit *Win95* app, many codepages can be used, and the codepage can be matched by activating specific keyboards and logical fonts. These differences are reflected in Figure 16.

What is also different is that an application needs to do some extra work to gain these benefits. It needs to decide if the codepage associated with a keyboard is one that it supports and decide whether to allow the keyboard layout to be switched, and it needs to keep track of which fonts can be used with which keyboards. It also needs to enumerate logical fonts in a particular way that lists character set-specific variants, and it needs to track which logical font, or at least which charset, is used for each run of text.

---

[19] These two ways of enumerating fonts can be seen in the Fonts control panel by toggling the "Hide variations" option in the View menu.

[20] In addition to numerical identifiers for codepages, such as 1252 for the "Western" character set, Windows also makes use of a distinct set of numerical character set identifiers, known as *charset IDs*. So, for example, the "Western" character set has a charset ID of 0, while the "Hebrew" character set has a charset ID of 177. Given either a codepage ID or a charset ID, Windows provides a means to find the other value.

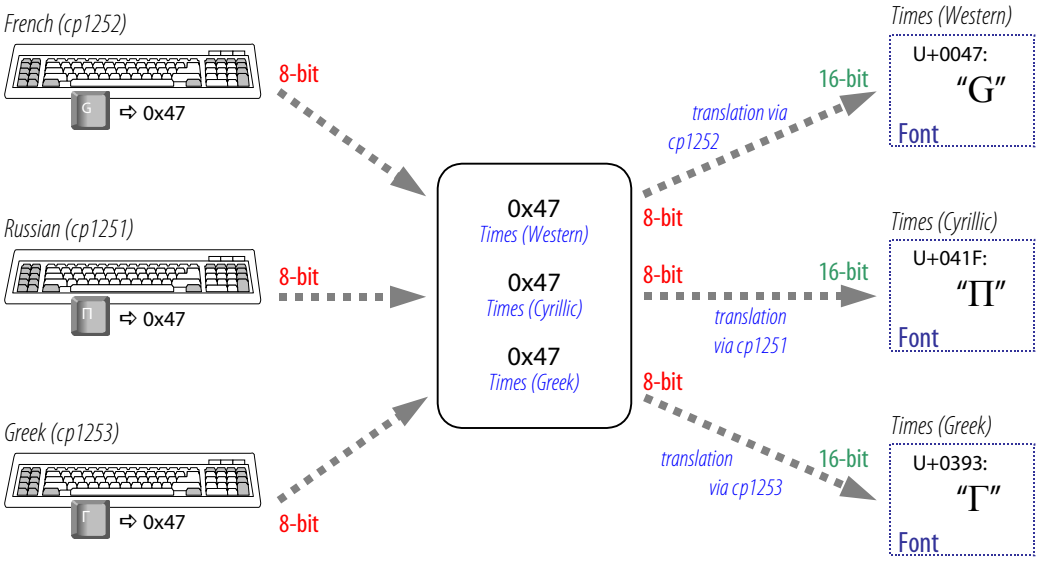

Figure 16: Win95 multilingual apps: multiple keyboards, logical fonts, and codepages[21]

This paradigm has the potential to support a very large variety of characters. It is constrained to the inventory of characters supported by the codepages installed on a system, though in principle MS could continue to invent new codepages to cover more and more languages. Codepages are cumbersome to deal with, however. Having a single, universal character set makes much more sense, which is the motivation for Unicode. Thus, MS have chosen not to provide any codepages beyond those that are available for Win9x:

Codepage ID	Character set
1250	Central European
1251	Cyrillic
1252	Latin 1 (also known as "Western" or "ANSI")
1253	Greek
1254	Turkish
1255	Hebrew
1256	Arabic
1257	Baltic
1258	Vietnamese
874	Thai
932	Japanese
936	Simplified Chinese
949	Korean (Wansung encoding)
950	Traditional Chinese

Table 2: Windows codepages[22]

[21] Note that, in Figure 16, the three logical fonts shown on the right correspond to a single font file.

Even so, applications designed for Win95 had the potential for working with an interesting variety of languages that used very different scripts. In actual practice, Win95 was not limited by codepages as much as it was by limitations in input methods and rendering. These issues are discussed further in §5.

As noted in §4.1, support for multi-byte codepages used for Far Eastern languages (cp932, cp936, cp949, cp950) requires special programming. This represents an additional limitation in applications that follow the *Win95* paradigm: they will not necessarily be able to support multi-byte codepages. For example, MS Word 95 was developed using this paradigm, but the US version of that application does not support multi-byte codepages.

Before continuing to look at other paradigms, let's consider what happens with a *Win3.1*-style app when it runs on Win9x/Me. Such apps don't do anything in the way of keeping track of what keyboard is active, what character set applies to each run of text, or what charset-specific logical fonts were available. It simply accepts and stores 8-bit text, and asks Windows to draw that text. While drawing the text, Windows automatically converts the text to Unicode using the Windows system codepage. All of that still works on Win9x/Me. Those versions of Windows provide extra multilingual functionality, but apps don't need to take advantage of it. *Win3.1*-style apps simply don't.

So, for example, when running Shoebox 4 on a US version of Win98 (for which the system codepage is cp1252), it will allow me to select a Thai keyboard (which is associated with cp874), but instead of showing Thai characters, it will simply interpret the 8-bit codepoints that come from the keyboard in terms of cp1252 (i.e. I see various accented Roman characters from the upper half of cp1252).

These first two paradigms fall into the single "8-bit/ANSI" cell of Table 1 (the upper left cell). The distinction between them has to do with whether the app uses one codepage or multiple codepages. This suggests a third, independent dimension of variation, in addition to the 8-bit versus Unicode and "ANSI" versus "wide" distinctions we have discussed. In practice, the one- versus many-codepage distinction is relevant only for the "8-bit/ANSI" combination: it doesn't make sense for an "ANSI" app to store Unicode if it is only working with the characters in a single 8-bit codepage (there is no advantage over storing 8-bit text), and a "wide/Unicode" app is not dependent upon codepages at all.

4.3 The *"ANSI"/Unicode* paradigm

The third paradigm is defined by applications that register themselves with Windows as "ANSI" (using the "ANSI" variant of `RegisterClassEx`—see §2.4 and §3.2) and that store text in terms of Unicode rather than a legacy, 8-bit encoding. (This corresponds to the upper right cell of Table 1.) The processing models used are those shown in Figure 12, when running on Win9x/Me, and Figure 15, when running on WinNT/2K. The difference between these two ways of processing lies entirely in the operating system; there are no differences within the application itself.

The additional mechanisms that such an app uses—keyboards, charsets, etc.—are very similar to those described above for the *Win95* paradigm. There are some key differences, however.

As seen in figures 12 and 15, an app that follows this paradigm will always receive 8-bit codepoints from the keyboard. When the keyboard is activated, the app is given the LANGID and charset. It can take either piece of information and ask Windows what the corresponding codepage is. Using that codepage, the app immediately converts each 8-bit codepoint as they are received into 16-bit values, which is how the text is stored. Since the app now has the Unicode value for the character, a call into the font to get the appropriate glyph can be made directly without the need for any further conversion.

---

[22] The Korean word 원성 "wanseong" means 'precomposing'. The Korean codepage, cp949, encodes Korean syllables as precomposed forms. It covers the most common combinations of jamo, but not all possible combinations. At one time, MS had defined a different codepage for Korean, cp1361, that used "Johab" (조합, 'combining') encoding, which represents each syllable in terms of the component jamo. This was capable of representing all possible combinations. The Johab codepage is no longer supported, however.

A Unicode app will have different font handling than an 8-bit, Win95-style app. Since there is no need to convert to Unicode when accessing the font, it is not necessary to enumerate logical fonts that distinguish among charset-specific variants. This has a benefit of being able to present a shorter list of fonts to the user, and the user is not burdened with understanding the charset distinctions. On the other hand, it also means that a change in font will not, in general, give the app enough information to know whether the keyboard needs to be changed. So, for example, a user may want to enter Hebrew text using the Arial font family, but simply selecting Arial isn't enough for the app to know that the user wants to enter Hebrew text—Arial can also support text in a number of other languages. Thus, the user must do something else to indicate a change, and this would typically be done by activating a different keyboard using the keyboard menu in the system tray.

The behaviour described here can be seen in Word 2000 when running on Win9x/Me.[23]

In principle, applications that follow this paradigm are capable of storing any Unicode character. They are still limited by codepages for keyboard, input, however. This issue is discussed further in §5.

4.4 The *"Wide"/Unicode* paradigm

The fourth paradigm is defined by applications that register using the "wide" variant of `RegisterClassEx`. In actual practice, all such apps store text as Unicode, even though this is not strictly necessary.[24] Thus, the paradigm corresponds to the lower right cell in Table 1. The character processing model that is used is shown in Figure 13.

This paradigm uses the simplest processing model, which doesn't involve any conversion between 8- and 16-bit encodings, and doesn't depend in any way upon codepages. When a keyboard is activated, the application may want to note the LANGID, but it does not need to be concerned with the charset or codepage—the charset is always Unicode. The application can also enumerate fonts as font families, without needing to distinguish between charset-specific logical fonts.

The behaviour described here can be seen in Word 2000 when running on WinNT/2K.[25]

In terms of the range of characters supported, applications that follow this paradigm are limited only by input methods and rendering issues (see §5). They are otherwise capable of supporting any Unicode character.

The only drawback to this paradigm is that applications that use it can only run on WinNT/2K. For many developers, this would limit the market for the product too much.[26] One way for a developer to deal with this would be to create *two* versions of an application: a "wide"/Unicode version for use on WinNT/2K, and an "ANSI"/Unicode version for use on Win9x/Me. These can even be provided together in a single distribution package using an installer that detects which version of Windows the software is being installed on. A better solution, though, might be to adopt the *combined "wide" / "ANSI" plus Unicode* paradigm, which is discussed next.

---

[23] Word 2000 does not follow this exact paradigm, however. Rather, it uses the *combined "wide" / "ANSI" plus Unicode* paradigm discussed in §4.5.

[24] See the discussion regarding the possibility of "wide" 8-bit apps in §3.3

[25] As mentioned in note 23, though, Word 2000 actually uses the *combined "wide" / "ANSI" plus Unicode* paradigm discussed in §4.5.

[26] That may not be true for applications that are intended for specific "vertical" markets. It does apply to most or all of the kinds of applications that would be of interest to linguists, translators and other language workers, however.

4.5 The *combined "wide" / "ANSI" plus Unicode* paradigm

This last paradigm recognises the benefits of the *"wide"/Unicode* paradigm but also the limitations of Win9x/Me, and aims to offer the best solution available for each platform within a single application. To do this, before an application calls the `RegisterClassEx` function while it is initialising (i.e. when the user runs the program), it first determines what version of Windows it is running on. If it is running on Win9x/Me, then it will register as "ANSI". But if it is running on WinNT/2K, then it will register as "wide". This will allow it to take advantage of the benefits of "wide"/Unicode support on WinNT/2K, but also run on Win9x/Me, offering the best capabilities that are available on those systems.

This has significant implications for how the application is designed: it has to be able to work in two somewhat different modes. In "wide" mode, there is no need to pay attention to charsets or codepages, and no need to convert between 8- and 16-bit encodings. When running in "ANSI" mode, however, it must pay attention to the charset and codepage for a given keyboard, and it must convert characters as they are received. In other words, an application of this sort must be able to use either the character processing model shown in Figure 12 or that shown in Figure 13, depending upon which version of Windows is present. This involves additional work for the developer. It provides the best overall capabilities, however.

Given that this paradigm actually combines the previous two paradigms, using one or the other according to the system on which an application is running, this paradigm doesn't fit into any of the individual cells shown in Table 1. Rather, it encompasses both of the cells in the right-hand column.

This is the paradigm that has been used, for example, with MS Office 2000.

The range of characters that can be supported by applications that follow this paradigm is affected by the platform on which it is running. Codepage limitations apply when running on Win9x/Me, but not when running on WinNT/2K. Input methods and rendering may also be factors in either case. See §5 for further details.

5. Input method, rendering and codepage limitations

The five paradigms I have described vary considerably in the range of writing systems that they can support. Because the *Win3.1* paradigm uses a single codepage, it is capable of supporting only certain character sets, and it is not designed to allow very different writing systems to be combined in a single document (except that English and other writing systems that require only the ASCII character set are always available). On the other hand, a *"wide"/Unicode* app (or a *combined "wide" / "ANSI" plus Unicode* app when running in "wide" mode) is capable of working with any Unicode characters. There are still limitations related to input methods and rendering systems, however. Also, we have seen that "ANSI" applications are constrained by codepages. We will consider these various limitations further here.

Some implementations have attempted to overcome some of these limitations by using custom encodings, custom-encoded fonts and special input or rendering systems, such as Keyman and *SDF*.[27] These are discussed in §6.

5.1 Input method and rendering limitations

We have seen that the *Win3.1* paradigm supports the characters of only a single codepage, but that the other paradigms can support a much larger inventory of characters. There are issues related to input methods and rendering that apply to all of these, however. I'll discuss those here.

Obviously, a user can work only with languages for which appropriate keyboards and fonts are installed on their system. Note that the keyboards and fonts don't have to go together. For instance, a system can have a

[27] The *Script Definition File* system was developed by SIL International to provide complex-script rendering capabilities within the *Win3.1* paradigm. See §6.

Japanese font without having to have a Japanese keyboard. In fact, Office 97 included Japanese fonts and codepages that could be installed on a US version of Win95, so that users could open and view Japanese documents created using Japanese versions of software. It did not provide keyboards, however, so there was no way for someone using US Win95 and Office 97 to edit such documents.

In addition to having fonts and keyboards installed, some languages also requires special support for either input or for complex rendering. For Far East languages, input is done using input method editors[28], and these require special support in 8-bit applications in order to deal with multi-byte encoding issues. Several languages, such as Thai, Arabic and Hebrew, require special rendering support. For Thai, it is possible for this to be provided entirely by the operating system. For Arabic and Hebrew, on the other hand, applications must be specially designed to handle right-to-left paragraph layout.

Special, script specific needs have been handled in different ways as Windows has evolved. As has been mentioned, Win3.1 was adapted into localised versions designed for specific regional markets. Each of the localised versions added additional code to handle the special input, encoding and rendering requirements of the writing systems for that one region. It was also assumed that applications would be localised for regional markets, just as Windows itself was. Thus, a developer might create a Korean version of their application, in which they added support for a Korean input method editor and for multi-byte encodings, and perhaps a separate Thai version, with additional code to deal with Thai line-breaking behaviours.

The situation for Win95 was slightly improved: the core of Win95 came in only three versions: one for Arabic and Hebrew, to provide right-to-left support; one for Far East languages, to provide mechanisms for input method editors and multi-byte encodings; and another version for other markets.[29] Again, it was expected that applications would be adapted for the different regions.

Around the time that Win95 was being developed, MS began to develop new rendering technologies to address the needs of complex scripts. They continued with the approach that applications needed to be adapted for particular markets. This can be seen from the TrueType Open 1.0 Specification:

> As much as possible, the tables of TrueType Open define only the information that is specific to the font layout. The tables do not try to encode information that remains constant within the conventions of a particular language or the typography of a particular script. Such information that would be replicated across all fonts in a given language belongs in the text-processing application for that language, not in the fonts. [Microsoft (1995), p. 3.]

The implication of this statement means, for example, that if an application is to support Arabic script, then the application needs to understand the rendering behaviours of that script. All that the TrueType Open support will provide is whatever is specific to a particular font.

This situation has changed considerably, however. In recent years, MS has committed to a globalised approach to software in which a single version of software is designed to work for all regional markets. Thus, Win2K, Office 2000 and Internet Explorer 5.x have each been designed to support the writing systems of all of the regional markets that MS has targeted.[30] A key element in facilitating this has been to incorporate support for complex-script rendering directly into Win2K, or in the case of Internet Explorer and Office to make it an installable addition to Win9x/Me. As a result, it is possible on US versions of Win9x/Me to edit and view text in various languages that use complex scripts, such as Thai, Arabic or Hebrew.

---

[28] Input method editors are special types of input methods designed to handle very large character sets, such as are needed for Chinese, Japanese and Korean. For more information, see Constable (2000b) or Kano (1995).

[29] Official announcements indicated that there were only three code bases. In Thai Win95, additional code was added to the US version in order to support Thai rendering and line breaking.

[30] To be completely accurate, the code for Thai support in Office 2000 was not ready in time. Thus, there is still a separate version of Office 2000 to support Thai.

For instance, if you install Internet Explorer 5.5 on one of these systems, the installation options allow you to enable support for Thai, Hebrew, Arabic, and Far East languages. Enabling support for one of these will cause Internet Explorer to install the necessary fonts, keyboards, codepages (if not present) and rendering support.

Much of this global support is provided at the system level. Thus, if you install Internet Explorer 5.5 onto US Win98 and select Thai language support in the setup options, this will make it possible to work with Thai text in WordPad. This is possible because Thai input can be handled by a keyboard that generates codepoints in an 8-bit, single byte encoding (WordPad doesn't need to handle input method editors in order to deal with Thai), and because the system extensions for rendering are automatically available to the standard text-drawing API functions.

There are some minimal requirements to take advantage of such system-level support, though. First of all, rendering these various scripts is possible only if the Unicode values of characters are available. This means that an application must use one of the Unicode-based paradigms, or must use the appropriate codepages. For *Win3.1*-style apps, support is limited to the system codepage only. If the app is able to work with that codepage (not necessarily true in the case of multi-byte codepages), it may be able to display the text. (There may also be other rendering issues, though, as described below.) But it will not be able to work with more than the system codepage, no matter what support amount of support for other scripts may also be available on that system. Applications that follow the other paradigms are much better positioned in this regard, since they either support multiple codepages (though not necessarily multi-byte codepages), or else they support Unicode directly.

There may be other requirements for rendering, however, specifically in the case of right-to-left scripts like Arabic and Hebrew. If an application is not designed to support right-to-left paragraph layout, it will not be able to correctly render these scripts, regardless of which paradigm it follows. Likewise, special support must be designed into an application if it is to be able to handle vertical layout of East Asian scripts.

There are also some minimal requirements in terms of input: for any application to support input of Far East languages, it must be specially written to work with input method editors.

Which of the five paradigms an application uses has a major impact on the multilingual capabilities of that application. We have seen here, though, that there are other important factors: right-to-left paragraph layout, vertical paragraph layout, and the ability to work with input method editors. Each of these requires additional work on the part of an application developer. Many applications simply do not include the additional code that is required for these capabilities.

There is an additional rendering limitation in what MS offers that is beyond the control of an application developer: MS has thus far provided fonts and rendering support for only so many scripts and writing systems around the world. They have been making significant advances in this regard, and are offering support for an increasing number of languages that use non-Roman scripts. For example, recent work has included Hindi, Divehi, and Assyrian. For linguists working with minority languages, the ability of MS-supplied to render minority-language writing systems may continue to be a concern for some time, but significant progress has been made. Windows is capable of working with many more languages today than just a few years ago. As mentioned, though, everything the MS contributes is of no use unless applications are written to take advantage of it.

5.2 Limitations of "ANSI" apps and codepages

Each of the paradigms other than the *Win3.1* paradigm are able to support a large variety of characters. We have seen, however, that applications that register themselves as "ANSI" are constrained by a significant limiting factor: codepages. Whether an app stores text using Unicode or 8-bit representations, characters are received from the keyboard as 8-bit codes. Therefore, there must be a conversion via a codepage at some point before the character reaches the font. There are only a limited number of Windows codepages

defined, however, as seen in Table 2. There are many scripts for which there are no codepages; for example, Devanagari, Bengali, Burmese, Khmer, Ethiopic, and many others.

In terms of rendering, the codepage limitation creates a minor distinction between apps that store text as Unicode (whether they are "wide" or "ANSI") and apps that follow the 8-bit *Win95* paradigm. Because an 8-bit *Win95*-style app uses an "ANSI" variant of a text-display API function to display text, a codepage is needed in the rendering process (see figures 11 and 14). As a result, such applications can only display characters that are supported by a Windows codepage installed on the given system. In contrast, applications that store text as Unicode will use a "wide" API function to display text, meaning that no codepage is necessary in the rendering process (see figures 12, 13 and 15). So, for example, Word 97 can display *any* Unicode character that gets entered into the text (assuming appropriate fonts and complex-script rendering support), even if there is no corresponding codepage, and regardless of whether it is running on Win95 or Win2K.

Of course, there needs to be some way to get Unicode characters into the application. For an ANSI-registered app, keyboard input of Unicode characters requires a codepage (see figures 12 and 15). This means that there is no way to enter characters into such an app if the characters are not supported by some keyboard. This is, therefore, a much bigger limitation than exists for rendering. So, for example, Word 2000 can display Ethiopic characters, but no mechanism exists for keying Ethiopic characters into Word 2000 when running on Win9x/Me.[31]

Because Win9x/Me only allows ANSI-registered apps, the keyboard/codepage limitation applies to *any* application when running on these platforms.

5.3 Breaking the codepage barrier for keyboarding in ANSI-registered apps

We have seen that apps that initialise themselves as "ANSI" (and therefore all apps running on Win9x/Me) will receive WM_CHAR messages containing only 8-bit codepoints, and that this limits these apps to working only with characters that are in some Windows codepage. If it were not for the codepage barrier on the WM_CHAR messages, an ANSI-registered app that stores text as Unicode would be able to support any Unicode characters.

If there were another message that could be used in place of WM_CHAR that always contained Unicode characters, this limitation could be removed. Recently, Microsoft did just that: they added a new system message into Win32, WM_UNICHAR. This message always passes Unicode-encoded characters, regardless of the version of Windows or whether the app has registered itself as "ANSI" or as "wide". It requires applications and keyboards that are specifically designed to support this new message, but it doesn't require any changes in the operating system itself. Thus, it can be used on existing Win9x systems.

The net effect of this message is to make it possible for an "ANSI" application to process characters as though it were a "wide" application. This gives us a new processing model:

[31] Since Word 2000 follows the *combined "wide" / "ANSI" plus Unicode* paradigm, it registers as wide when running on WinNT/2K, and so it would be possible to key Ethiopic characters into Word when running on those systems.

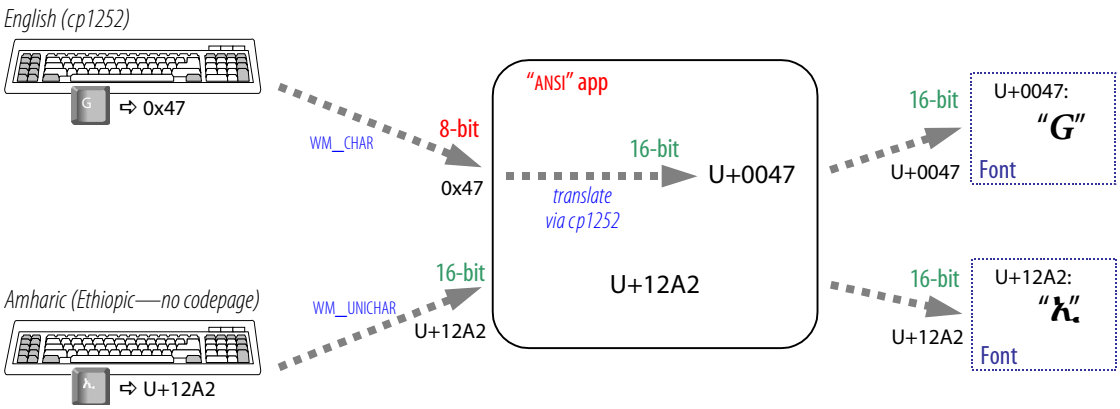

Figure 17: "ANSI"/Unicode app: 8-bit keyboard input, or Unicode keyboard input using `WM_UNICHAR`

Use of the `WM_UNICHAR` message makes sense mainly for the *"ANSI"/Unicode* and *combined "wide" / "ANSI" plus Unicode* paradigms. Support for this message would be a useful addition for applications that follow either of these paradigms, significantly increasing the range of Unicode characters that users can work with in these applications. It particularly benefits users working on Win9x/Me, but can also be of benefit for "ANSI"-only apps when running on WinNT/2K.

The SIL FieldWorks applications will support this new system message, as will Keyman 5.[32] We assume that support is likely to start appearing in new versions of at least some Microsoft applications.

6. Multilingual apps and custom-encoded fonts

We have dealt with a lot of information regarding how Windows is designed to work with multilingual applications (or *not really* work with multilingual applications, in the case of Win3.1 and earlier). An interesting question that arises is how these mechanisms for handling multilingual text are affected by the use of custom-encoded fonts.

Custom-encoded font implementations have been used primarily to overcome one of two limitations:

- *Win3.1*-style apps only support the system codepage
- complex rendering support for a given script is not available on the versions of Windows being used, or are not available to the application, which is most typical of *Win3.1*-style apps[33]

There are other possible reasons why one might want to create a custom implementation, but in every case with which I am familiar, one of these two reasons has been the motivating factor.

These custom implementations are, therefore, mainly intended for applications that follow the *Win3.1* paradigm. In most cases, they are used together with special keyboards that have been created using a program such as Keyman. In many cases where complex script behaviours are involved, the necessary transformations are handled within an intelligent keyboard. In some cases, implementations have been done using a rendering system such as SDF to handle contextual glyph selection.[34] In both cases, all that is

[32] Version 5 of Keyman is the first version of that product to support Unicode-encoded characters as well as 8-bit characters. At the time of writing, Keyman 5 is still in the beta-review process.

[33] The reason that this is most typical of *Win3.1*-style apps is because they don't use Unicode, and they also don't track codepage or charset IDs so that the Unicode values of characters can be determined.

[34] The *Script Definition File* system was introduced briefly in note 27. Basically, it allows codepoints to be mapped into other codepoints using context-sensitive rules.

happening is that codepoints are being mapped to other codepoints within the same codepage. Otherwise, the character processing model that the application is utilising is unchanged.[35]

Potentially, a custom-encoded font can still work in applications that follow one of the other paradigms. It is important to understand, though, that a custom-encoded font is tied to a particular codepage: the codepage that they redefine, which is the system codepage for the systems on which the implementation was expected to be used. In most cases, custom fonts were built to work on top of codepage 1252. In order to work as intended, the codepage which the font is redefining must be available to an application. Because the life cycle of a character often involves translations between 8-bit codes and Unicode values, the fact that custom-encoded fonts are dependent upon a particular codepage presents many opportunities for problems.

For example, suppose you created a plain text file in Notepad on US Win98 using a custom-encoded font that was designed for use on a US version of Windows (for which the system codepage is cp1252). Suppose that you then give the file to someone using a Russian version of Win98 (on which the system codepage is 1251). They would not be able to view the file, even if they have the custom font. The reason for this is that Notepad on Win9x/Me follows the *Win3.1* paradigm: it encodes 8-bit text but utilises only the Windows system codepage. Because the font and file were created in the context of cp1252, the recipient will likely see lots of empty boxes when displaying the file on a system that uses cp1251 as its default. In the contexts in which the font was expected to be used, the transformations from codepoint to glyph would be as in Figure 18:

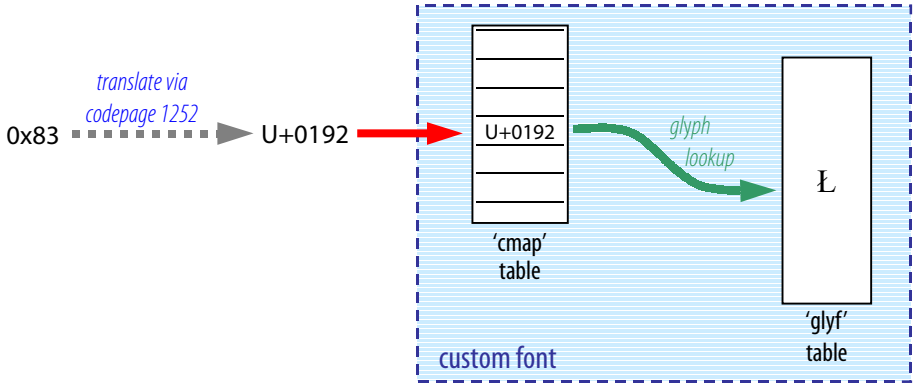

Figure 18: Rending a non-standard character using a custom font and codepage 1252[36]

[35] The SDF system is actually a little more capable in that it is not limited on output to using the same codepage. Rather, it can provide a mapping directly to Unicode. When used in this way, therefore, it is acting like a custom codepage that supports context-sensitive mappings and that is used in the rendering process.

An application that utilizes this feature would need to use a variant of the processing model shown in Figure 12. Rather than receiving an 8-bit codepoint from the keyboard, converting it via a codepage, and then storing the Unicode value, the application would store the 8-bit value, and then convert it to Unicode using the SDF system just prior to rendering. The only application to date that supports this capability of SDF is *ScriptPad*, which has also been developed by SIL International.

It is important to note that the SDF system differs from more advanced rendering technologies, such as *OpenType* and the *SIL Graphite* system, in several important respects. In particular, OpenType and Graphite operate directly on glyphs, transforming one string of glyphs into another. In contrast, the SDF system transforms character codes into other character codes. Considered another way, the output from a font's cmap table constitutes the input to OpenType and Graphite. But the output of SDF is used as input into the cmap table in a font. Solutions based on SDF generally use custom encodings, or custom-encoded fonts, or both.

[36] It should be noted that the Unicode value U+0192 is "*ƒ*", not "Ł". This is a "custom-encoded" font, remember.

On a Russian system, however, the following transformation would occur:

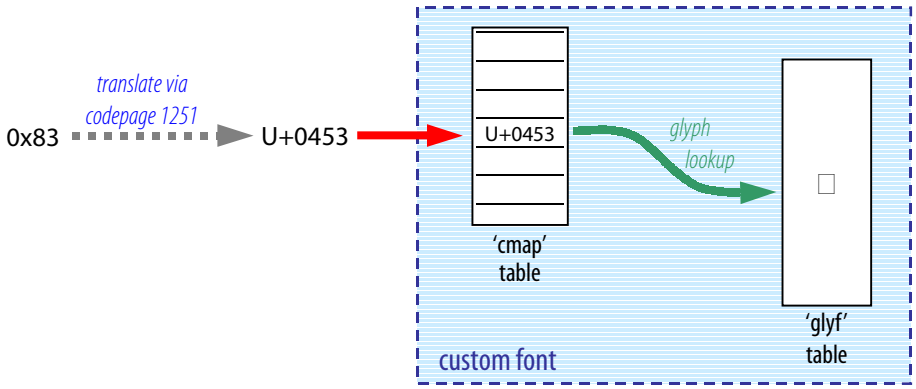

Figure 19: Rending a non-standard character using a custom font and codepage 1251[37]

Stating this problem more generally, custom-encoded "hacked cp1252" fonts will not work on localised versions of Win3.1 that use a different system codepage. They will also not work in localised versions of Win9x/Me or WinNT/2K in applications that rely on the default Windows codepage if that codepage is something other than cp1252.

Because keyboards are associated with codepages, mismatches can cause problems for custom fonts. For example, if you work in a multilingual app such as WordPad (on Win98) and activate a keyboard that uses a codepage other than the one for which the custom font was designed, you will end up seeing boxes. Keyman 3.2 and earlier were somewhat neutral regarding codepages, but that means that a multilingual app will convert input from Keyman 3.2 according to the codepage associated with whatever Windows keyboard is currently activated. Again, that can result in seeing empty boxes.

In Keyman 4, it became possible to specify a specific LANGID with a keyboard. What wasn't obvious to all is that this would cause a specific codepage to be associated with that keyboard. That would mean, for example, that if a keyboard intended for use with the SIL Ezra font package (a custom-encoded font based on cp1252) was created for use with Keyman 4 and was assigned the LANGID for Hebrew, then that keyboard would not work with any app that made use of the multilingual support mechanisms in Windows. Again, the user would be seeing empty boxes because 8-bit-to-Unicode conversion would be done using the Hebrew codepage (cp1255) rather than cp1252.

The codepage intended for a custom-encoded font must, therefore, be available to the application and applied to any text that is formatted with that font. It must not be enforced too strongly, though, otherwise the user will experience unintended behaviours. This can affect things such as upper- or lower-case mappings, and line breaking. Thus, an application may break lines in what the user perceives to be the middle of words. Or an application's "intelligent" features for things like sentence-initial capitalization or smart quotation marks may result in undesired changes.

In order to work around some of these behavioural problems, many custom fonts have been implemented as *symbol-encoded* fonts. This effectively gives them special "symbol" codepage, which is available as a separate mapping on any version of Windows. In fact, symbol encoding uses part of the Unicode private-use area for encoding. Specifically, the glyphs in a symbol-encoded font are accessed in the font's cmap table using Unicode values in the range U+F020–U+F0FF.

[37] In fact, what happens in this situation is that the cmap table does not contain any entry for U+0453. In this situation, Windows selects a special default glyph, which is a box in most TrueType fonts.

Symbol-encoded fonts that are used for custom character sets are susceptible to a number of problems, specifically when used in applications that store text as Unicode. These problems can result in the loss of data. These issues have been discussed in depth in Constable (2000c) and Hallissy (1998).

Custom-encoded fonts were a sensible solution for people who had to create multilingual documents in the days of Win3.1, when there was no other way to do so. In the environment of newer technologies, however, they are increasingly becoming an albatross that gets in our way. For some users, there may still be need to use custom-encoded fonts for a while yet. But wherever possible, we need to be moving away from custom encodings and toward industry standards, which at last are beginning to provide the kinds of solutions we have been needing.

7. Old wineskins and new wine: why Windows' multilingual support works for WordPad but not for Shoebox

At last, we return to the original question that came from that person in Thailand: *"I installed support for Thai on my US Win98 machine and can now edit Thai in programs like WordPad and Word; can I get this to work in programs like Shoebox and Paratext?"* The answer to this is, "No." By now, hopefully you have some idea why. I'll explain, just to be sure.

The multilingual support components—fonts, keyboards, rendering engines—that are provided by Microsoft are base on Unicode or the standard Windows codepages. They are useful only for applications based on the *Win95* paradigm or on one of the three Unicode-based paradigms. In contrast, our legacy linguistic applications that deal with text—Shoebox, Paratext, LinguaLinks, BART, Fiesta and others—were designed around the *Win3.1* paradigm. They assume a single codepage, and have relied on the use of custom-encoded fonts to deal with alternate character sets. They don't make use of any of the mechanisms for multilingual support available in Windows: Unicode, LANGIDs, codepages, charsets, logical fonts, etc. Multilingual support in Windows is starting to flourish, but the capabilities being provided by Windows depend upon applications utilising the multilingual support mechanisms that are made available. Our legacy applications miss out on all the fun because they don't know the first thing about any of this stuff.

What's the answer? We need some new wineskins—new applications that will build on new technologies. The SIL FieldWorks applications are intended to do just that.

Fortunately, we are at a point at which codepages are really starting to become obsolete. Managing multiple codepages made life more difficult for software developers, and limited users for whom existing codepages were inadequate. Unicode removes the need for codepages, but supporting pieces in the infrastructure, such as fonts and keyboards, also need to work with Unicode. Keyboarding has been a problem, particularly for Win9x/Me, but developments such as `WM_UNICHAR` and Keyman 5 are beginning to remove those barriers. There are still limitations to the number of scripts and writing system for which MS provides fonts and rendering support, but they continue to expand their coverage. The other piece to the puzzle is applications: they need to follow a Unicode-based paradigm, and they need to provide special support for input method editors and non-Roman paragraph layout options.

The necessary pieces are starting to fall into place. For users working with multilingual text, there is finally light at the end of the tunnel, and it is getting brighter.

8. Bibliography

Constable, Peter. 2000a. Font issues in MS Windows and Office. *NRSI Update, 13.*

Constable, Peter. 2000b. Understanding characters, keystrokes, codepoints and glyphs: Encoding and working with multilingual text. Available in *CTC Resource Collection 2000* CD-ROM, by SIL International. Dallas, SIL International.

Constable, Peter. 2000c. Unicode issues in Microsoft Word 97 and Word 2000. Available in *CTC Resource Collection 2000* CD-ROM, by SIL International. Dallas, SIL International.

Hallissy, Bob. 1998. The BoxChar Mysteries presents… The €uro case. Available in *Resource Collection 98 CD*, by International Publishing Services (1998). Dallas: SIL International. Also available in *CTC Resource Collection 2000*, by SIL International (2000). Dallas: SIL International.

Kano, Nadine. 1995. *Developing international software for Windows® 95 and Windows NT™*. Redmond, WA: Microsoft Press. Also available online at http://msdn.microsoft.com/library/books/devintl/S24AE.HTM.

Microsoft Corporation. 1995. *TrueType Open font specification. Version 1.0.* Redmond, WA: Microsoft Corporation. Available online at http://www.microsoft.com/typography/tt/tt.htm.