

Unicode Issues in Microsoft Word 97 and Word 2000

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

Abstract

With the introduction of Unicode support in the Microsoft Office 97 applications, the Word application gained some interesting new capabilities. At the same time, some users began to encounter unexpected behaviour with their fonts and their data, such as text displaying as boxes. This has been especially true for people using custom-built fonts that use non-standard encodings. In this paper, we explore the Unicode-related capability in Word, and look at several of the problems users may encounter. In discussing these problems, we explain the causes, and suggest some fixes. We also consider the direction in which the software industry has been heading with regard to Unicode and argue that users should adopt a Unicode strategy to avoid recurrences of the kind of problems discussed here.

1. Introduction

Over the past several years, Microsoft (MS) has gradually been incorporating support for Unicode into their software products. Beginning with Windows NT 3.1 and Windows 95, there has been at least some measure of support for Unicode in these two lines of Windows operating systems. Building on this, MS began to add support for Unicode to some of their office productivity applications beginning with the release of Office 97. Similar capabilities began to be brought to some of the Macintosh versions of these applications with the release of Office 98 for the Macintosh. With Office 2000, all of the core applications in the Office suite had been revised to use the Unicode standard.

Word 97 and later versions store all text encoded in 16-bit Unicode. There are times when Word has to translate 16-bit characters to and from 8-bit characters, however. This is needed in order to run on a primarily 8-bit-character system (Windows 95/98/Me), and in order to provide backwards compatibility with various 8-bit-character file formats, such as plain-text formats and the formats used by earlier versions of Word. MS engineers had to forge a hybrid of 8-bit and 16-bit character support. The solutions they adopted were designed primarily with the average business user in mind. For users working with minority-language orthographies using customized fonts and encodings, however, these solutions have presented various challenges—for those working with Roman-script orthographies as well as non-Roman.

* This paper is a revised version of an earlier paper (Constable 1998) that focused on Word 97. The information has been updated to include information related to Word 2000, and to reflect a better understanding of some of the issues. Some of the issues that relate to Word 2000 have also been discussed in Constable (2000a).

Some of the facts discussed in this paper were first discovered by Bob Hallissy or Martin Hosken, and I gained much from their insights. These paper also benefited from helpful comments by Bob Hallissy and Peter Martin. Any shortcomings that remain are, of course, my own.

In this document, we discuss several Unicode-related areas of difficulty that users may face using Word 97 and later versions:

- Text sometimes appears as boxes when the font is changed.
- When a file is saved in Word 6.0/95 or other formats, text appears as question marks.
- When text is formatted with certain fonts, lines wrap at any character rather than only at word-breaking characters.
- With certain custom fonts that had worked in Windows 95, some characters no longer worked after upgrading to Windows 98.
- Text is formatted with a certain font in Word 2000, but the text seems to be displayed with a different font.
- Word 2000 does not allow text to be formatted with certain fonts.
- When a file is saved as text, certain characters are converted to sequences; for example, “©” is converted to “(c)”.
- Certain characters do not work with Word 2000 when Arabic or Hebrew support is enabled.
- When running on Thai Windows 95/98, character code d211 does not work properly.
- Since Word 97 and later are storing text encoded in Unicode, there may be times when a user would like to know exactly what Unicode value is being stored, and the user may also like to be able to enter particular Unicode values or to search for a Unicode value.

Some of these issues result from mixing custom encodings with software designed to use the Unicode encoding standard. Other issues are known bugs. Also, some issues do not pertain specifically to Word but relate to factors in Windows itself. All are issues that have been encountered in using Word 97 and Word 2000, however, and so are discussed together here.

In sections 6–14, these issues are discussed, each in a separate section. Beside describing the problems and explaining why they occur, we also consider ways in which the problems may be dealt with, where solutions to work around the problems exist.

Before discussing the first of these issues, we briefly present some technical background in sections 2–4. These sections give background information on character sets and codepages and on symbol versus non-symbol fonts, and also provides an overview of Unicode support in Word 97 and later versions.

For some of the issues discussed, we have written VBA macros to provide fixes. These macros are available in the file `UnicodeWordMacros.dot` which is available from the SIL Non-Roman Script Initiative (NRSI; see the contact information in §16). For some of these issues, a different fix is available in the form of a modified codepage 1252 file created by Martin Hosken, which may also be obtained from the NRSI. Details regarding the installation and use of the Word template and the codepage file will be provided with those files. Unfortunately, not all of the macros work in Word 98 for the Macintosh, and Martin’s Windows codepage file just wouldn’t be at home in a Mac system folder.

After describing these various problems and the available fixes, we consider the current situation faced by users, and how we might address such problems in the future.

Most of the comments regarding Word apply to Word 97 and later versions. We will refer to these collectively as just “Word”; where a comment applies only to a specific version, that version will be named explicitly (e.g. “Word 2000”). Occasional reference will be made to Word 98 for the Macintosh. These issues have not been researched as thoroughly in that version, however. No testing has been done with Word 2001 for the Macintosh. References to “Word 10” or “Office 10” are referring to the next version for the Windows platforms following Office 2000. (Office 10 was in early beta as of the time of writing.)

2. Character sets and codepages

Let's begin with basics: Text data is always stored in a computer as a sequence of numbers where the numbers are given a standard interpretation which determines what character each number represents. For example, ASCII is a universally-known encoding standard in which characters are stored as 7-bit numbers and the number 66, for example, is interpreted to represent the character "B", as illustrated in Figure 1.

Stored ASCII data:

...	66	105	108	108	32	115	108	111	119	108	121	32	116	117	114	110	101	100	...
-----	----	-----	-----	-----	----	-----	-----	-----	-----	-----	-----	----	-----	-----	-----	-----	-----	-----	-----

How the data is interpreted:

... B i l l s l o w l y t u r n e d ...

Figure 1: Interpretation of encoded text

All text encoding works by these same principles; all that changes are the details of the syntax (e.g. 16-bit values rather than 8-bit, or pairs of numbers rather than single numbers) and the semantics (what character is associated with a given number or combination of numbers).

When data is stored as 8-bit values, it is possible to distinguish 256 (2^8) different values. For many languages, this is more than adequate to meet the orthographic needs of the language. This is not enough to handle all orthographies of every language, however. (It isn't even adequate for the orthographies of certain individual languages, such as Amharic or Chinese, let alone every orthography of every language.) In fact, 8-bits is enough to handle only the orthographies of a relatively small number of languages.

In most Microsoft software, from the time of the first version of DOS until recently, it has been assumed that text data is usually stored as 8-bit data encoded using one of a certain set of 8-bit encoding standards. (This is not the case with Windows NT, however.) Since early in the history of DOS, MS had to support users who spoke a wide variety of languages. In order to overcome the limitations of an 8-bit encoding, MS provided a way in which the user could specify that a different encoding, and thus a different set of characters, be used. The mechanism they introduced was that of *codepages*.

A codepage provided a particular set of encoding definitions that would determine the encoding syntax and semantics, and the system would provide mechanisms to switch between different codepages. In DOS, for example, you could use the MODE command to control what codepage would be used by the display or printer device (assuming the device was capable of supporting the codepage), and then every character of text data would be interpreted by that device in terms of the selected codepage. Thus, a text character stored as the number 119 might be interpreted as the character "w" when one codepage was selected, or it might be interpreted as the character "ו" if a different codepage was selected, as illustrated in Figure 2.

Stored data:

...	66	105	108	108	32	115	108	111	119	108	121	32	116	117	114	110	101	100	...
-----	----	-----	-----	-----	----	-----	-----	-----	-----	-----	-----	----	-----	-----	-----	-----	-----	-----	-----

How the data is interpreted according to one codepage:

... B i l l s l o w l y t u r n e d ...

How the data might be interpreted according to another codepage:

... ב ד ה ה ו ה ש ל ה א מ ת ר ב ע ה ...

Figure 2: Data interpreted according to selected codepage

Clearly, it would be essential to know what codepage should be used to correctly interpret a given text, but that detail was left to be taken care of by the user or the application developer. The main point was that codepages provided a mechanism by which the operating system could allow the user to work with a greater variety of orthographies.¹

Word keeps track of the language and codepage associated with a run of text, and also the language and codepage associated with whatever keyboard layout (or other input method) is currently active. Whenever Word needs to have text converted from one encoding to another, it asks Windows to do this using the appropriate codepage.

With the advent of Unicode support in Windows NT and partial Unicode support in Windows 95,² codepages took on an important, new characteristic: not only does a codepage define a particular set of characters, but it now also provides a mapping between the 8-bit character codes in that encoding and the corresponding 16-bit Unicode values, as shown in Figure 3.

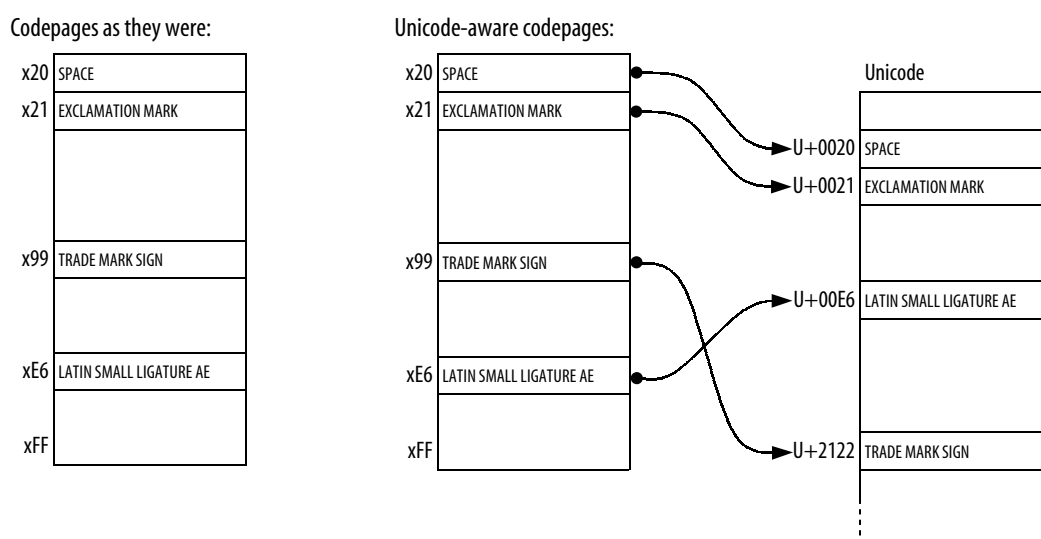


Figure 3: Codepages then and now

The codepage provides mapping tables to go in either direction: from 8-bit to Unicode, or from Unicode to 8-bit. For example, as shown in Figure 4, codepage 1253 defines the interpretation of the 8-bit number 0xC4 (d196) as GREEK CAPITAL LETTER DELTA “Δ”. Codepage 1253 also provides the information that 0xC4 corresponds to the 16-bit number U+0394, which is the Unicode value that represents GREEK CAPITAL LETTER DELTA. Likewise, if given the Unicode value U+0394, codepage 1253 can provide the translation back to the 8-bit value, 0xC4.³

¹ In fact, Windows also uses a second mechanism, *charset*, which is nearly identical to codepage. For the purposes of this discussion, there is no important distinction between them. If you look at the contents of an RTF file, however, you will find that a charset is associated with each font rather than a codepage.

² In fact, codepage 8-bit-to-Unicode mapping was introduced with Windows 3.1. Unicode information was only used internally for displaying TrueType fonts. Applications did not have access to any Unicode information until the introduction of the Win32 and Win32s APIs.

³ The mapping from Unicode to 8-bit isn't just the reverse of the 8-bit to Unicode mapping. It will map as many Unicode characters into the codepage as might make converted text legible. Thus, for example, it might map U+0104, “Ą”, to x41, “A”, if the codepage doesn't support U+0104.

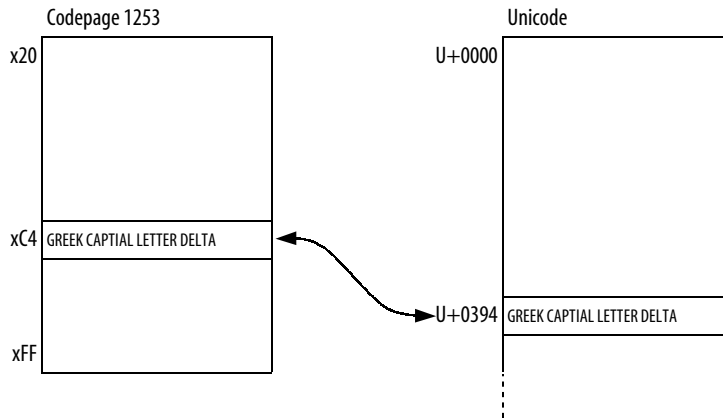


Figure 4: 8-bit-to-Unicode and Unicode-to-8-bit mappings

If a different codepage were used, however, this information may not be available. For example, suppose we asked Windows to translate the Unicode value U+0394 using codepage 1252 (“US English”). As shown in Figure 5, the Unicode value U+0394 is not within the domain of codepage 1252 since GREEK CAPITAL LETTER DELTA is not part of the set of characters defined by that codepage. Since Windows can’t get an 8-bit value from codepage 1252 that corresponds to U+0394, it will return an error value of 0x3F, “?”.

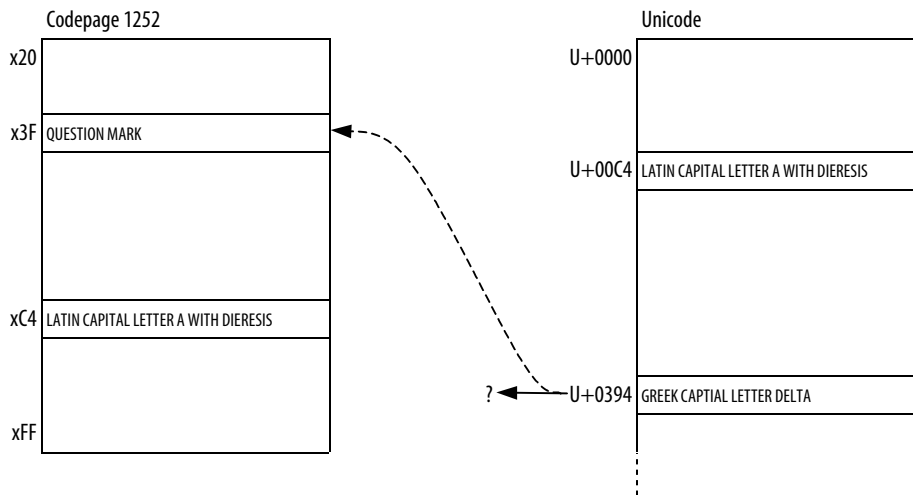


Figure 5: Unicode-to-8-bit mapping for characters not in codepage

The function of codepages in translating between 8-bit values and Unicode values is important to understand since they are regularly used within Windows for this purpose. In particular, they are constantly used by Word for performing translations as it manages text data.

For further information on codepages and character sets in Windows, see Constable (2000c), Hosken (1997) or Kano (1995). For further information on the basic issues of character encoding and the representation or orthographic systems within computers, see Constable (2000b).

3. Symbol fonts and non-symbol fonts

When a font contains glyphs for orthographic characters, it can be desirable for an application to have some awareness of the properties of those characters so that it can provide better functionality. An example of such functionality is the ability of Word to cycle selected text between lower case, upper case and mixed case by keying `SHIFT-F3`.

Providing such functionality for orthographic characters is feasible in principle if we assume that the complete set of orthographic characters is limited and defined. In contrast, symbols and dingbats are an open set since they potentially can include anything that can be represented graphically. There is no way to list a comprehensive set of symbols in order to identify their properties. Moreover, even if such a listing were possible, there would not likely be any interesting properties that would be useful for any processing purposes.

As a result, fonts in Windows are flagged to indicate whether or not they contain only symbols.⁴ In Windows, a font must be flagged as being symbol or non-symbol,⁵ and the way in which a font is flagged will affect how software behaves when that font is used. For example, in Word, if the selected text is formatted with a symbol font such as Wingdings, keying `SHIFT-F3` to change case will have no effect. When a symbol font is used, Word will make no assumptions whatsoever about properties of any characters formatted with that font.⁶ For text that is formatted with a non-symbol font, however, Word will behave on the basis of character-property information specified in the Unicode standard (or, in versions before Word 97, provided by whatever codepage is in use).

When users create fonts to support minority-language orthographies, often the set of characters that are required do not correspond to any existing codepage definition. If such a font is flagged as a non-symbol font and is used in conjunction with a codepage that doesn't match it, applications like Word may exhibit undesirable behaviour.

For example, Word supports a “smart quotes” feature by which it automatically converts straight quotation marks to left or right curly quotation marks as needed. Let's suppose you were writing a phonology paper in Word on US Windows and you had textual examples in IPA. Suppose further that you were using an IPA font that was flagged as being non-symbol, and that this font had some IPA symbol, e.g. an over-striking dieresis, that was encoded using the 8-bit value 34 (0x22, which is the straight quotation mark in codepage 1252). Then Word would convert the over-striking dieresis to whatever character was encoded by the 8-bit value 148 (0x94, which is the right curly quotation mark in codepage 1252). The reason for this is that Word would see that the font was a non-symbol font and would request information from codepage 1252 about the characters. Codepage 1252 can only assume that the character encoded by the number 34 is the straight quotation mark, and Word would therefore automatically make the conversion from 34 to 148. Neither Word nor the codepage are aware or even care that the IPA font has an over-striking dieresis rather than a straight quotation mark. The only way for the user to avoid this change from happening would be to turn off the smart quotes feature, which may not be convenient.

To avoid such undesirable behaviour, fonts for minority-language orthographies or for IPA have often been created as symbol fonts. This trick has worked in the past, but beginning with Word 97, this introduces some new problems, as will be described below.

To determine whether a font is flagged as a symbol font or not, you need a tool that can report internal information regarding a font. A good example is the *Microsoft font properties extension*, which is available as a free download from <http://www.microsoft.com/typography/property/property.htm>.

⁴ Technically, the distinction is found in parameters within the `cmap` and `name` tables in the TrueType font. Fonts that contain non-symbol characters are often referred to as “UGL” (Unicode glyph list) fonts.

⁵ Far East versions of Windows may permit other options.

⁶ This is slightly different from earlier versions of Word. Further details are provided below.

4. Unicode support in Word

Beginning with Word 97, text is stored as 16-bit Unicode values. Unicode support in Word 2000 is very similar to that in Word 97. There are some differences, though, mostly refinements in areas pertaining to backwards compatibility with earlier versions of Word and with legacy 8-bit encoding standards. The most important differences with regard to Unicode support are described below.⁷

That Word is storing text as 16-bit values can be illustrated by viewing a Word document using a hex editor, as shown in Figure 6.

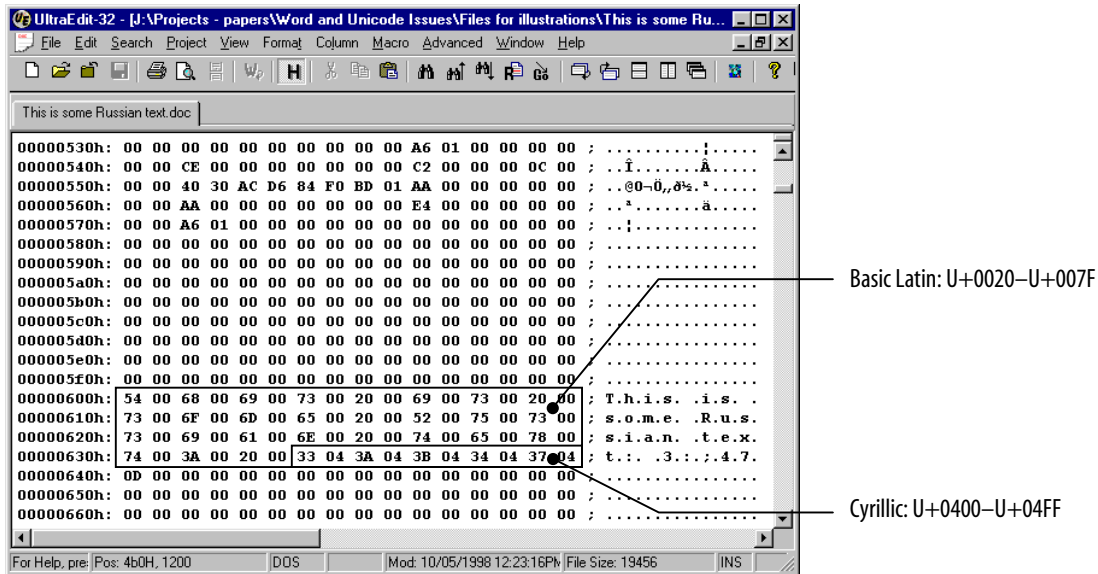


Figure 6: Text in Word 97 document stored in Unicode

Note that each character is two bytes long, e.g. “T” is stored as 0x54 0x00 (U+0054—the high order byte follows the low order byte). Note also the five characters of Russian text (the 10 bytes that are selected): U+0433 U+043A U+043B U+0434 U+0437. These are Unicode values, and the text would appear in Word (using Times New Roman) as “гклдз”.

When text is entered, Word converts 8-bit values to Unicode values according to whatever codepage is active. (If you have the Windows multilingual extensions installed, this can be changed by choosing a different keyboard and language using the applet that appears in the tray of the taskbar.) This has important consequences even if you are working with US English. For example, suppose you key ALT-0151 to get the em dash “—”. Now, you may be expecting that Word has stored that character using the number 151 since that is how you entered it, and you know that “the em dash is at position 151 in the font”.⁸

⁷ The most significant development made in Word 2000 in relation to multilingual text support is in the area of globalization. Up to and including Office 97, Microsoft used to create different executables for various regions of the world: one for Western European languages; one for Far Eastern languages, with support for double-byte encodings and input method editors; another for Arabic and Hebrew, with right-to-left support; and separate versions for Thai and for Vietnamese. As of Office 2000, all of these groups of languages are supported in the same executable except for Thai and Vietnamese (the code for these was not ready in time; the goal of merging support for all of their markets in a single version will be achieved with Office 10). The incorporation of Unicode support in Word 97 was the first step in globalization that made the advances in Word 2000 possible.

⁸ This common perception is, in fact, not true. Windows TrueType fonts have always used Unicode values internally to reference characters, not 8-bit character codes. Thus, the em dash is always at U+2014 in the font.

However, Word takes the number you entered, 151, and asks Windows to translate that to a Unicode value using the currently active codepage (1252, say). Windows returns the Unicode value for the em dash, U+2014, and that is what Word stores.

We should point out some of the locations in Unicode of characters in codepage 1252, the default codepage for a typical installation of US Windows. When using a non-symbol font and entering text encoded according to codepage 1252, characters in the range 0x20–0x7E (d32–d126) and 0xA0–0xFF (d160–d255) will be converted to “equivalent” values in Unicode; e.g. 0x41 maps to U+0041, 0xD9 maps to U+00D9, etc. For characters in the range 0x80–0x9F (d128–d159), however, this is not the case. Characters in the range U+0080–U+009F are defined in Unicode as control characters, and the characters found in the range 0x80–0x9F in codepage 1252 are found in various ranges of Unicode. For example, the em dash is 0x97 in codepage 1252, and U+2014 in Unicode (as described above); but 0x98 in codepage 1252, “~”, is U+02DC in Unicode. Each of the characters of codepage 1252 in the range 0x80–0x9F are shown in Table 1 together with the corresponding Unicode value.

Character code	Character	Unicode value	Character code	Character	Unicode value
x80	€	U+20AC	x90	(reserved)	U+0090
x81	(reserved)	U+0081	x91	'	U+2018
x82	,	U+201A	x92	'	U+2019
x83	f	U+0192	x93	“	U+201C
x84	„	U+201E	x94	”	U+201D
x85	...	U+2026	x95	•	U+2022
x86	†	U+2020	x96	–	U+2013
x87	‡	U+2021	x97	—	U+2014
x88	^	U+02C6	x98	~	U+02DC
x89	‰	U+2030	x99	™	U+2122
x8A	Š	U+0160	x9A	š	U+0161
X8B	‹	U+2039	X9B	›	U+203A
X8C	Œ	U+0152	X9C	œ	U+0153
X8D	(reserved)	U+008D	X9D	(reserved)	U+009D
X8E	Ž	U+017D	X9E	ž	U+017E
X8F	(reserved)	U+008F	X9F	ÿ	U+0178

Table 1: Codepage 1252 characters 0x80–0x9F and their Unicode equivalents^{9,10}

When a symbol font is selected, text that is entered is translated to Unicode by a different translation than that provided by the current codepage. Since a symbol font could potentially contain any character or graphic symbol, there is no way to know if the character being entered is defined in Unicode or not. As a

⁹ The characters shown reflect the version of codepage 1252 found in Windows 98. Just prior to the release of Windows 98, Microsoft has added three new characters to codepage 1252: the euro currency sign, “€”, at 0x80 (U+20AC); latin capital letter z with caron, “Ž”, at 0x8E (U+017D); and latin small letter z with caron, “ž”, at 0x9E (U+017E). Previously, the code values 0x80, 0x8E and 0x9E were “reserved” in codepage 1252. The impact of this change on existing fonts and data is discussed briefly in §9. For further details, see Hallissy (1998).

¹⁰ Unicode characters in the range U+0080–U+009F are control codes and not printing characters. According to Kano (1995), those character codes shown here as reserved map to U+FFFE. Our testing indicates, however, that they are, in fact, mapped to their equivalents in the range U+0080–U+009F.

result, Word will translate the 8-bit value entered to a special range in Unicode known as the “Private Use Area”. Specifically, Word will translate the 8-bit value by adding 0xF000,¹¹ as illustrated in Figure 7.

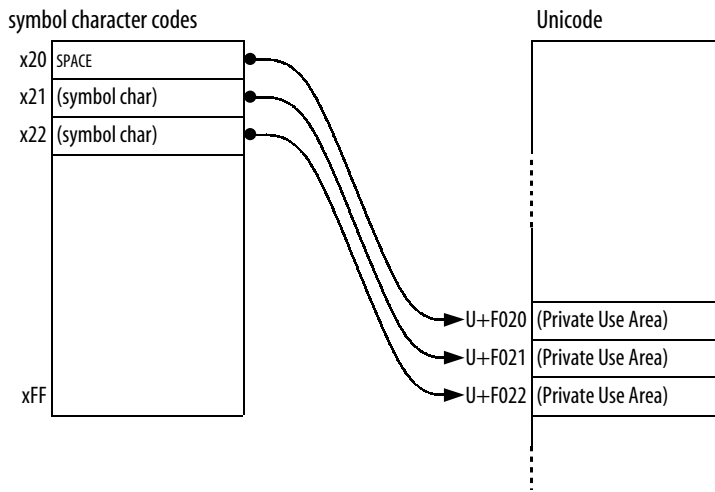


Figure 7: Text formatted with symbol font in a Word 97 document

That characters formatted with a symbol font are being stored this way can be illustrated by viewing a Word document using a hex editor, as shown in Figure 8.

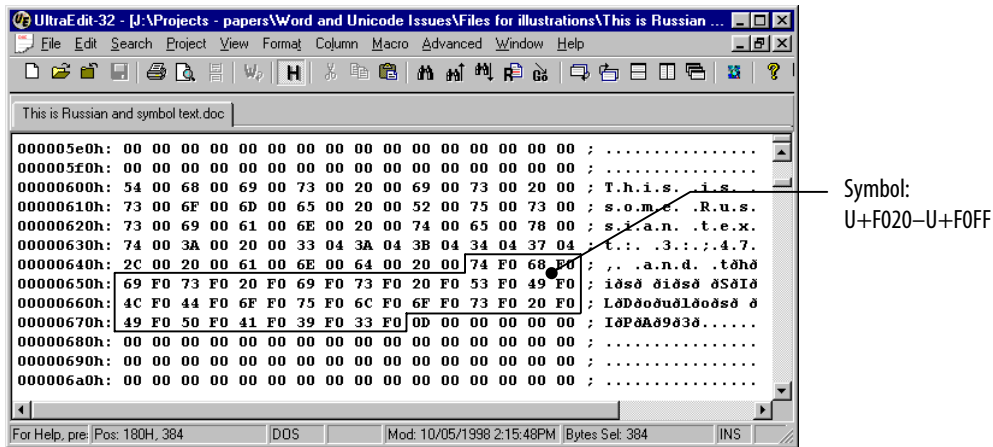


Figure 8: Characters formatted with symbol font map to Unicode Private Use Area

¹¹ What’s really happening here appears to be rather messier than this. We don’t know this for certain, but it appears that Word actually stores a symbol character as an 8-bit value and calculates a Unicode value to associate with that character by adding xF000, but that Word doesn’t use that calculated Unicode value to display the character. (It would use that Unicode value for searches, however.) To display the character, it appears that Word passes the 8-bit value to the TextOutA () Win32 API, and that TextOutA () is finding the character in the font using the translation charcode – 0x20 + usFirstCharIndex (usFirstCharIndex is found in the OS/2 table in the font). What this means is that you could have a symbol font in which the characters are at (say) U+E020–U+E0FF. If you selected this font and typed “A”, Word would store 0x41, it would display the character at U+E041 in the font, but Word would tell you that the character is actually U+F061, and you would need to search for U+F061 to find it.

A similar translation occurs when existing text that was entered with a non-symbol font selected is reformatted using a symbol font. For example, suppose the character “a” is entered while the Times New Roman font is selected. The 8-bit value for “a” in codepage 1252 is 0x61, and this gets translated to the Unicode value U+0061. If that character is then selected and the font is changed to Wingdings, Word will replace the value U+0061 with the value U+F061. Thus, any text that is formatted with a symbol font will contain characters in the range U+F020–U+F0FF, regardless of how it was originally entered.¹²

Word is able to save to or read from various file formats, including several that predate Word 97. Examples include MS-DOS text, Word 2.x, Word 6/Word 95, and RTF.¹³ Since all of these file formats store text as 8-bit values, Word must convert between 8-bit encodings and 16-bit Unicode whenever it reads from or saves to one of these formats. These conversions are handled by the codepages installed in the system, and the conversion may or may not succeed without loss of information. For example, if a Word document contains a Chinese character, e.g. “脣” (Unicode value U+8123), and the file is exported as MS-DOS text on a non-Far East version of Windows, the Chinese character would be converted to a question mark “?” since the relevant codepage (the “OEM” codepage for that system) does not include that character. The same will be true in some situations if the file is exported as, say, a Word 2.x document. (This would not happen on a non-Far East version of Windows if system files for Far East support are added later. This is explained below.)

Word 2000 provides better support for exporting to 8-bit plain text files than does Word 97. In addition to the “MS-DOS Text” and “Text Only” export options, Word 2000 offers an “Encoded Text” option that presents a separate dialog to allow the user to select the encoding to be used for the exported file. Thus, given the document described above containing Chinese characters, the user could specify that the plain text file should be encoded using an encoding standard that supports Chinese, as illustrated in Figure 9.

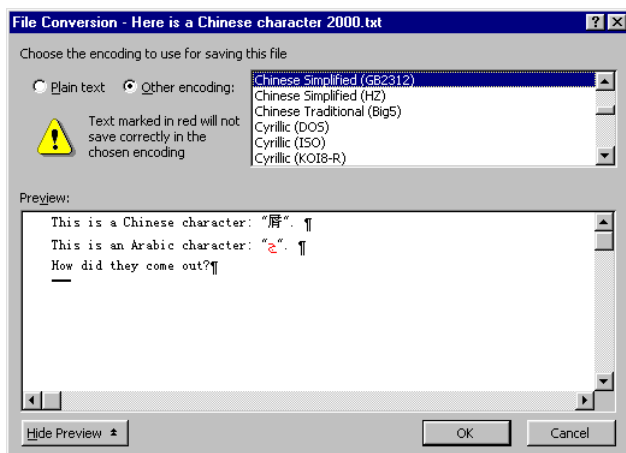


Figure 9: Word 2000’s “Encoded Text” file export option: user can select the encoding for the exported text file

Of course, this assumes that there is a codepage that supports Chinese characters installed on the users system, which is not always the case. (It is not the case by default on US Windows 95, for example.) Also, only one codepage at a time can be used. Thus, if a file mixes characters that are not all supported by any

¹² If the previously existing text were from a different codepage than 1252, for example if it were Cyrillic text, Word will convert back to an 8-bit value using the Cyrillic codepage before doing the translation into the range U+F020–U+F0FF. Interestingly, if the existing text is not defined in any codepage, if it were Ethiopic or Yi, for example, then Word would not change the character code. Thus, it is possible to have text that is formatted with a symbol font but that is not in the range U+F020–U+F0FF.

¹³ The RTF format was updated for Word 97.

one codepage, then not all will survive the export process. This is illustrated in Figure 9, for example, by the Arabic character shown in red in the preview window.

If this same document containing Chinese characters is exported to RTF format, the RTF file stores the Unicode value (albeit in the decimal representation of a signed-integer value), and also Word's translation of that character back to 8-bit, "?" (on non-Far East systems without any codepage support for Chinese characters). Thus, the RTF file stores U+8123 as "\u-32477\3f". The RTF file also correctly maintains the font information. As a result, when the RTF file is opened again, the character "脣" will appear again as it originally did.

Office 97 and Office 2000 include fonts and system files that provide some Far East support. These will allow you to view documents with Chinese, Japanese or Korean text in Word. (Office 97 did not provide any means to type in these scripts, though Office 2000 does). On a system that has these files installed, if a file contains the Chinese character U+8123, "脣", and the file is exported to Word 2.x, then when the Word 2.x document is opened, that Chinese character will appear as "Ã". Note that this is a two-byte sequence. Word has used the Far East support files (specifically, the codepage files) to convert the Unicode value U+8123 to the double-byte sequence that would be used on a Far East version of Windows, 0xC3 0x8B.¹⁴ Word is able to do this conversion when saving to Word 2.x format, but along the way it loses information regarding the character set/codepage.¹⁵

Word 97 and Word 2000 will behave differently when saving to Word 6.0/95 format. Word 97 gives the same behaviour when saving to Word 6.0/95 format as when saving to Word 2.x format: when you re-open the document in Word 97 (or Word 2000), the Chinese character will have been converted to "Ã". The codepage information has been lost. In contrast, Word 2000 manages to retain the codepage information when saving to Word 6.0/95 format. Thus, when the file is re-opened, the Chinese character U+8123, "脣" is retained.

If the same file were saved to RTF format with the Far East support files installed, Word does the same translation giving the double-byte sequence, 0xC3 0x8B, and the RTF file also stores the Unicode value. Thus, the RTF file stores "\uc2\u-32477\c3\8b". When the file is saved to MS-DOS text on a US version of Windows, however, the character is converted to "?", even with the Far East support files installed. This is because the MS-DOS text option uses the default "OEM" codepage for the system, which doesn't include Chinese characters on US Windows.

In theory, the general principle here for how Word behaves in saving to file formats for earlier versions of Word that use 8-bit text is that, if there is a codepage available on the system for the characters in the file, then Word will convert the text using that codepage. There are ranges of Unicode for which no Windows codepage has ever been defined, for instance Devanagari or Yi. In these cases the characters will always be converted as "?" since there is no alternative. For characters in ranges that do have codepages, though, in principle they should get converted using the appropriate codepage.

In practice, Word 2000 does a better job at this than Word 97, though there are still some situations in which Word 2000 may not get things completely right. For instance, I did a test in Word 2000 using some Thai characters and saving to Word 6.0/95 format. I expected it to work, since Word 2000 did a good job with other scripts, like Chinese and Arabic, and because I know that the appropriate codepage for Thai is installed on my system. It did not work, however: the Thai characters were converted to question marks. (In fairness, it should be noted that the common version of Word 2000 was designed to support both Arabic and Chinese, but not Thai. See note 7.) There are also certain edge cases in which Word 2000 may

¹⁴ This is the mapping defined by codepage 936, which is used for Simplified Chinese.

¹⁵ Word 2.x was developed for Windows 3.x, which only allowed applications to work with a single codepage. Thus, Word 2.x had no reason to store this information.

have character encoding or font problems in conversion to Word 6.0/95, but these are unlikely to affect many users.¹⁶

Word also supports import and export to Unicode text format. As with MS-DOS format, all layout and character formatting information is removed, but the Unicode character data is retained. Figure 10 shows the contents of a Unicode text file that was created by exporting a Word document as Unicode text. The Word document contained English and Russian text, and text formatted with a symbol font.

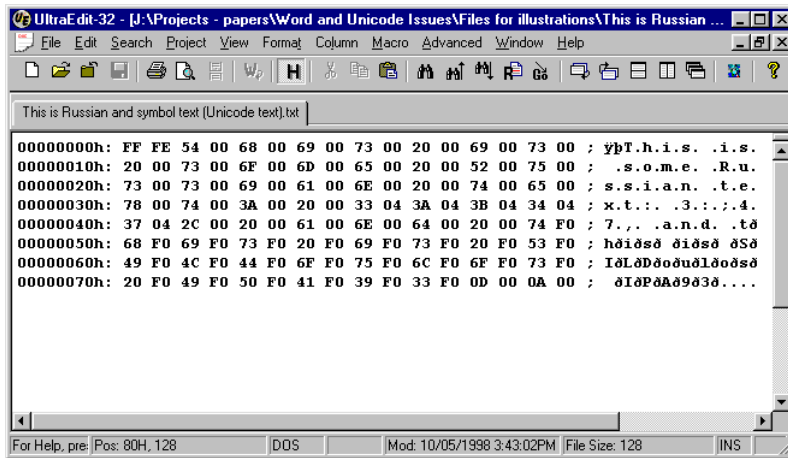


Figure 10: Word 97 document exported to Unicode text

Note that the first two bytes of the file were not in the original text. This Unicode character is the BYTE ORDER MARK, U+FEFF, which indicates the byte order of the data (i.e. whether the high-order byte precedes or follows the low-order byte) and should be the first character of any Unicode text file.¹⁷ In the case of this file, the high order byte follows the low order byte.

For more information on Unicode support in Word 97 and on exporting text from Word 97, see Constable (1997).

Having given some technical background as well as an overview of Unicode support in Word, we can now move on to look at some specific issues that arise in Word as a result of Unicode support.

5. Viewing stored Unicode values, entering specific Unicode values and searching for Unicode values

We will address these issues first since some of them are useful for the sections that follow.

Suppose you'd like to find out what Unicode value Word is actually storing for a given character. How would you do it? Word 97 doesn't provide any way to do this. Word 2000 does provide a way, though: if you select a character or a run of text and then open the Insert/Symbol dialog box, you will see the Unicode value of the first character in the selection reported on the status bar, as shown in Figure 11.

¹⁶ For example, I found that if a Word 2000 file contains a Chinese character next to an Arabic character, and the file is saved to Word 6.0/95 format, something will get lost. So, for instance, in one test the string “霄ح” came back as “اُح”, with the codepage and font family for the Chinese character having been lost. In another test, the same string came back as “اُح”, with codepage and font family information for *both* characters having been lost.

¹⁷ This is true for files that use the UTF-16 encoding form, and in situations where the byte order is not determined by some external protocol. The byte order mark is not required in situations in which a particular byte order is required by a higher protocol, or if the UTF-8 encoding form is used.

codepages, the Insert/Symbol dialog will provide a way to access characters in that font from each of those codepages. Figure 13 shows what the dialog looks like when Times New Roman is selected.

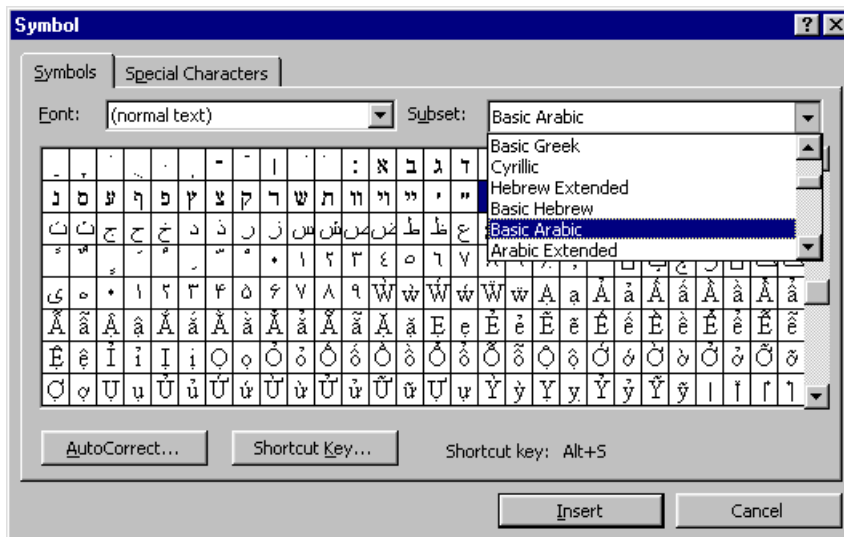


Figure 13: Inserting Unicode characters using Insert/Symbol dialog

Note that the Times New Roman font that comes with Window 95 contains a lot more than just Roman characters (though not all versions contain Hebrew and Arabic characters).

This method may not always work, however. If you have a font that supports a codepage which is not installed in your system, or if the font has characters defined in ranges of Unicode for which there is no standard codepage defined, then Word's Insert/Symbol dialog will not allow you to access those characters. To make up for this, we have provided the `EnterUnicodeCharacters` macro in `UnicodeWordMacros.dot`. This macro presents a dialog box with a textbox control in which you can specify a Unicode value in hexadecimal, as shown in Figure 14:

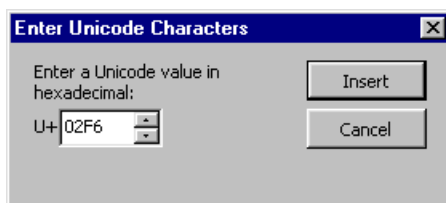


Figure 14: `EnterUnicodeCharacters` macro dialog

To use the macro, simply run it. Only hexadecimal characters can be entered in the textbox, but UP and DOWN ARROW keys and PAGE UP and PAGE DOWN keys can be used to increment and decrement the value. UP and DOWN ARROW keys change the value in increments of 0x0001. If you hold down the SHIFT key, then UP and DOWN ARROW keys change the value in increments of 0x0010. The PAGE UP and PAGE DOWN keys change the value in increments of 0x0100, but if you hold down the SHIFT key then they change the value in increments of 0x1000.

Once the Unicode value has been specified, press `ENTER` or click the Insert button to insert the character into the text. When you are finished inserting characters, press `ESC` or click the Cancel/Close button to close the dialog.^{19,20}

The `EnterUnicodeCharacters` macro provides a convenient way to enter a limited number of characters. What if a user wants to be able to type larger amounts of text that is in a range of Unicode for which no keyboard handling facility is available on their system? Fortunately, an increasing number of keyboards for various languages are being shipped from MS with products like Office and Internet Explorer, and these will work with recent versions of Office. Third-party products that makes it easy to define keyboards and which can generate Unicode-encoded text are starting to become available; for example, version 5 of the Tavultesoft Keyboard Manager (“Keyman”) is expected soon, and it will provide support for Unicode. Most applications that support Unicode are limited when running on Windows 95, 98 or Me, however, to allowing keyboard input only for characters that are defined in some Windows codepage. What can a user do when no tools are available for entering text using the particular character ranges that they need?

For example, there is an existing allocation in Unicode for Ethiopic, but to date MS has not provided any way to type Ethiopic-script text in any version of Windows. Or, you may be working with a script that is not yet defined in Unicode and have created a Private Use Area encoding for that script, but MS does not provide way to enter text in the Private Use Area (other than what we have described is happening with text formatted with a symbol font). For certain situations, there may be a real need to have some way of entering Unicode text when no such keyboarding facility is available.

While we don’t have a working solution to meet any such need, we do provide a macro that illustrates how temporary solutions to such needs might be met. The `EnterPUAText` macro is provided in `UnicodeWordMacros.dot` to illustrate how Visual Basic for Applications (VBA) might be used to implement an input method so that a user can key data in ranges of Unicode for which no existing input method is available. It presents a dialog box in which the user can type text; on closing, that text is inserted into the document.

For this illustration, we created a non-symbol font called “Latin1+E9xx SILDoulos” which contains the characters of codepage 1252 in their standard Unicode positions, and it also contains characters in the Unicode Private Use Area, specifically in the range U+E900–U+E9FF. The glyph for each character in that range is the Unicode value of that character. (E.g. the glyph for character U+E92B appears as “^{E9}2B”.) Thus, you will see exactly what Private Use Area values are being entered.

To use the macro, you must first have the Latin1+E9xx SILDoulos font installed on your system. (It may be necessary to install the font before Word is loaded for it to work properly.) When you are ready to run the macro, position the insertion point in a document at a place where you won’t mind having meaningless text added. Then run the macro. When the dialog appears, just begin typing. The text you enter will be displayed in the Latin1+E9xx SILDoulos font. At this point, the dialog will appear something like the screen shot in Figure 15:

¹⁹ Version 1.3 of this macro package has been made available in two variants specific to Word 97/98 and to Word 2000 and later. In the Word 2000 variant, the macro dialog is a modal dialog, so you can move the insertion point to a new location in your document while keeping the dialog open. This capability was not possible in Word 97 and Word 98.

²⁰ Once it becomes available, Office 10 will provide a means to enter any Unicode character by typing the Unicode value in hex and then pressing Alt-x. This is described in relation to searching in Word 10 below.

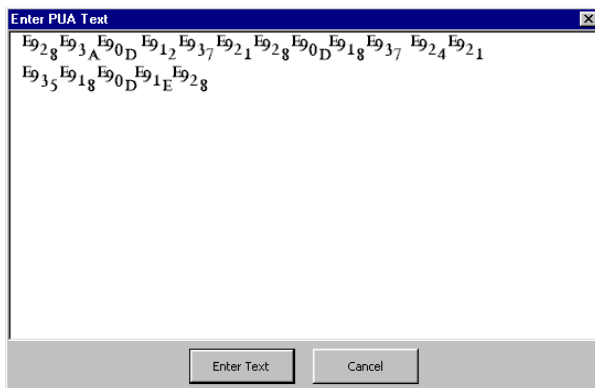


Figure 15: EnterPUAText macro dialog

After you have typed as much text as you want, press ENTER or click the Enter button, and the text you typed will be inserted into your document and formatted with the Latin1+E9xx SILDoulos font.²¹

This input method is extremely crude and simple. The dialog permits only the most minimal of editing capabilities: typing and backspacing. Rather more sophistication would be very desirable for a working implementation. Also, the keyboard layout will appear to be random. (The macro was adapted from something that was being done for another project without any attempt to make the keyboard layout more meaningful for this context.) If you look in the VBA code for the dialog form, you will see that the keyboard can be reconfigured very easily.

Feel free to experiment with this macro—that’s why it was made available.

Moving on, we now look at how you go about searching for a particular Unicode character in your document, or how to specify a particular Unicode character to be replacement text. In Word’s Find and Replace dialog, you can specify a particular Unicode character as the “Find what” or “Replace with” text by entering

`^uNNNNN`

where *NNNN* is the Unicode value in *decimal* representation. So, if you wanted to search for the character U+043C, you would calculate the decimal equivalent of 0x043C, which is 1084, and you would enter “^u1084” as the search string.

Word 2000 has another interesting way to specify arbitrary Unicode characters for searching or replacing. Try this: open the Find and Replace dialog box. In the “Find what” text box, enter the hex digits “5d0”, which is the Unicode value for the Hebrew letter aleph. Now, press ALT-X. You should see the hex digits that you entered disappear and be replaced by the corresponding Unicode character, aleph.

The “Find what” and “Replace with” text boxes in the Find and Replace dialog boxes are implemented using a control (a programmer’s user-interface building block) known as the RichEdit control. This control is also used for text boxes in other Office dialogs, such as the “File name” control in the File/Open dialog. The version of the RichEdit control that ships with Office 2000 has the interesting feature just described: you can enter the hex digits for any Unicode value and then press ALT-X, and it will replace the hex digits with the corresponding Unicode character.

If you have Word 97 and not Word 2000, however, you are limited to using a decimal representation, as described above.²² If you only need to search for a single character and don’t need to replace, and you don’t

²¹ In the Word 2000 variant of Version 1.3 of this macro package the dialog for this macro is a modal dialog. This means that you can move the insertion point in the document to a new location while keeping the dialog open. Also, when you click the ENTER button, the dialog remains open. In the variant for Word 97, this will close the dialog.

need to specify any search options, then you may find it easier to use the `FindUnicodeChar` macro provided in `UnicodeWordMacros.dot`. The main advantage of this macro is that it allows you to specify the character in hexadecimal rather than decimal, as is required by Word's Find and Replace dialog.

To use the macro, simply run it and specify the character to search for in hexadecimal. The controls on the dialog work exactly as those in the `EnterUnicodeCharacters` dialog described above.

We have looked at background issues on matters such as codepages, discovered how Word is making use of Unicode for encoding text, and explored some of the ways that we can work with Unicode characters in Word. It is now time to turn our attention to some of the problem areas mentioned at the outset.

6. Problem: Text appears as boxes

Many people have encountered the problem of having text appearing as boxes. The symptom of this problem is that text is displayed as a series of boxes rather than using the glyphs you expected from the font you selected.^{23,24}

As an example of how it can arise, consider a linguistics student writing a paper. In the paper, she is using Times New Roman (a non-symbol font) for body text, and the SILDoulos IPA93 font (a symbol font) for IPA characters. The latter font has all of the lower case English Roman characters in their usual ASCII positions, so after typing an IPA transcription, she continues typing prose text forgetting to first change the font back to the body text font. Once she types an upper case letter, however, she remembers that she needs to change the font to the body text font, and she tries to change the font for the prose text she has just typed. Instead of Roman characters, however, she gets boxes.

In earlier versions of Word, it was possible to change the font used for a run of text between symbol and non-symbol fonts without loss of information. In Word 97 and later, this is not always possible, and under certain conditions, changing the font selected from a symbol font to a non-symbol font may result in boxes appearing rather than the characters you expected.

The reason such font changes worked without any problems in earlier versions of Word was that Word didn't change the stored text data when a font change was made. If a character 0x54 was formatted with Times New Roman (a non-symbol or "UGL" font) and the font was changed to SILDoulos IPA93 (symbol), the stored 8-bit character value remained 0x54. If the font was changed again to Arial (UGL), the character again remained 0x54, as illustrated in Figure 16.

²² The mechanism described for Word 2000 will also work with Word 97 if you happen to have the appropriate version of the RichEdit control installed on your system. It might have been installed with another MS application, for instance. To find out if you have the necessary version of RichEdit, just try out this trick. If the right version of RichEdit is installed on your system, it will simply work.

²³ Certain fonts may display a different shape than a box. For example, many Postscript fonts display a middle dot: “•”

²⁴ There is another potential cause of unexpected boxes being displayed than what is discussed here. That situation applies to those using Windows 98 or who have the Windows 95 euro patch installed. This is discussed in §9.

Character codes in a Word 6.0 document:

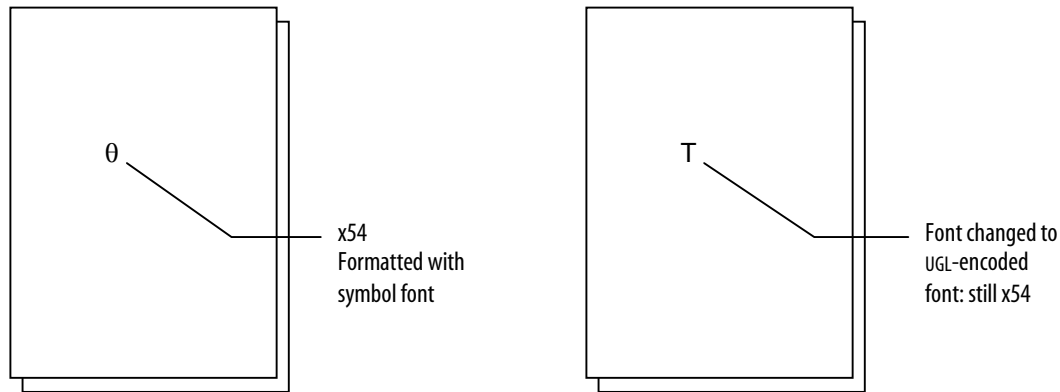


Figure 16: Font changes never affected data prior to Word 97

As of Word 97, all this has changed. Recall that when a character is entered, a translation takes place from 8-bit to Unicode, and Word stores the Unicode values. If a non-symbol font is in use at the time, the translation is determined by a codepage. If a symbol font is in use, however, a different translation process is used, and the resulting Unicode values are always in the range U+F020–U+F0FF. Recall also that similar translations also occur whenever a font change is made from a non-symbol font to a symbol font.

You can experiment with the `ShowUnicode` macro to see for yourself that characters formatted with a symbol font are always encoded in the range U+F020–U+F0FF.²⁵

Once Word 97 is at the point where it has a character formatted with a symbol font with a value in the range U+F020–U+F0FF, then if the font for the character is changed to a non-symbol font and Word doesn't have any prior information about the character set for that character, it won't have any way to know what character to really associate with that character value. Therefore it won't know if it is in any codepage supported by the non-symbol font, and therefore Word won't know what codepage to base the translation of the character value on. After the character is formatted with the non-symbol font, Word continues to display a character value in the range U+F020–U+F0FF. Again, you can experiment with `ShowUnicode` to verify this. Since most non-symbol fonts don't have any characters defined in this range, a box appears instead.

Let's consider an example: In Word 97, suppose you have the SILDoulos IPA93 font (symbol) selected and you press the "T" key. Word gets the 8-bit code 0x54 and translates this to U+F054. Next you select that character and change the font to Times New Roman (UGL). As illustrated in Figure 17, Word will continue to display the character at U+F054, but no such character exists in Times New Roman, and so you get a box.

²⁵ That is, Word makes it appear this way. Cf. note 11 for technical details.

Character codes in a Word 97 document:

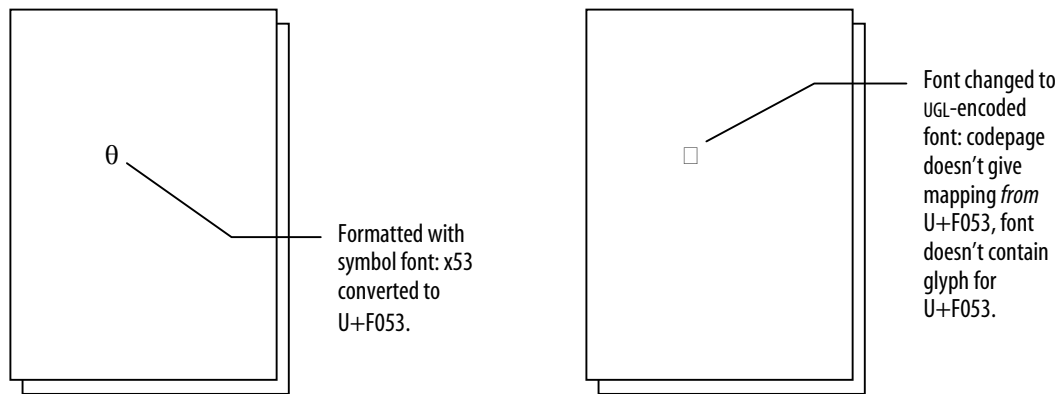


Figure 17: Fonts changes can result in boxes being displayed in Word 97

Now let's alter that example a little: Suppose you began with Arial and you press the "T" key. Assuming that codepage 1252 is active, Word will translate the 8-bit value 0x54 to U+0054. Now you change the font for that character to Wingdings, and Word translates the character to U+F054. At this point, as long as you have not saved the file since you made the change to Wingdings, Word remembers that the character was originally in the US English character set, and if you change the font back to a non-symbol font that supports codepage 1252, such as Times New Roman, the character will be translated back to U+0054. If, however, the font has been changed to Wingdings and you *save* the file, at the point the file is saved, Word forgets the earlier character set information. Now if you try to change the font from Wingdings to Times New Roman, Word will not know how to translate from U+F054, it will continue to display that value, and you will see a box rather than "T".

Word 2000 has addressed this problem to some extent. It only occurs if a file containing text that is formatted with a symbol font is saved and closed, later reopened, and the font for that text is then changed to a UGL font. In other words, it is doing a better job at remembering the history of character formatted with a symbol font for as long as the file is kept open, so as to reduce the likelihood of encountering this problem. The problem can still occur, however, in the situation just described.

In September, 1997, MS provided an article in their Knowledge Base (Microsoft 1997) providing a fix to this problem in the form of a VBA macro. This macro has some unfortunate weaknesses, most notably that it would alter any characters in the range U+8000–U+FFFF, which could affect characters in scripts such as Chinese or Korean, as well as symbols. I have written an improved macro to replace it called `ConvertSymbolToWestern`, which is available in the `UnicodeWordMacros.dot` template file.

To use the macro, simply select some text that is formatted with a non-symbol font and which is displaying as boxes, and run the macro.

This macro will only operate on characters in the range U+F020–U+F0FF. Characters will be displayed with boxes whenever they are formatted with a font that does not provide glyphs for those particular characters. For example, Times New Roman does not cover the Devanagari range of Unicode. If a text contains the character U+0915, DEVANAGARI LETTER KA, and that character is formatted using Times New Roman, a box will appear: "□". The `ConvertSymbolToWestern` macro does not attempt to fix that problem. The only solution is to find a font that covers the Devanagari range of Unicode.²⁶

²⁶ In future versions of Word or when using future versions of Windows, you may find that Word and/or Windows will automatically substitute a font that covers the Devanagari range. Word already does this for certain ranges of Unicode, as explained in §0.

The macro assumes that the desired codepage of the text is codepage 1252. Basically, all it does is to take the Unicode value of characters in the range U+F020–U+F0FF and subtract 0xF000. If you started with text in a different range of Unicode other than that which corresponds to codepage 1252, however, for example Cyrillic, and that text was formatted with a symbol font, and then later the font was changed back to a non-symbol font which caused boxes to appear, running this macro will return Roman characters, *not* the original Cyrillic text.

There is another solution available to this problem: It involves saving the document to Word 2.x, Word 6.0/95, or text file formats, as described in the following section. Details are left to be discussed in that section.

7. Problem: Text appears as question marks when file is saved to Word 6.0/95 or other formats

This is another common problem many users encountered after the introduction of Word 97. Typically, it will happen in situations when people work with multiple versions of Word or interact with others using older versions of Word. It can also be a problem for people who like to use Word to create plain text files (files that have only text and no formatting) because Word allows them to specify a font for displaying text and because it is a full-featured text editor.

Here's a typical scenario: Someone begins preparing a phonology exam for the course they are teaching using Word 97, and the document contains IPA characters formatted using the SILDoulos IPA 93 font (a symbol font). They want to work on the document at home using Word 6, so they save the document in Word 6.0/95 format. When they get home and open the document, all of the IPA characters appear as glottal stops.

Here's another scenario: Someone is working on minority-language text that will be used for CARLA processing, and they are using Word 97 to edit the text so they can see it displayed using their SILManuscriptBantu1 font, a symbol font which has several non-standard, extended-Roman characters. After editing their text, they save their file using the Text Only file format option. When they later look at the file, they discover, to their dismay, that the entire text has been converted to a string of question marks.

These scenarios all involve the use of symbol-encoded fonts. This problem did not occur on earlier versions of Word because, in those versions, Word only ever stored characters as 8-bit values, so it never had the need to do any translation from Unicode to 8-bit (see Figure 18). When exporting to a text file, all that was necessary was to strip away the layout and character formatting information, leaving only the original codepoint values.²⁷

²⁷ There are two important situations in which character information would be lost. One would be in the case of mixing text formatted using custom fonts with non-standard character sets together with text formatted using standard fonts. The other be in the case of mixing text formatted with two or more custom fonts that involve different character sets. In these situations, removing information about the font also removed information about character distinctions. In those situations, though, users had no expectation of all the character distinctions being retained in plain text. What is important in this discussion is that, in these earlier versions of Word, the *codepoints* did not change.

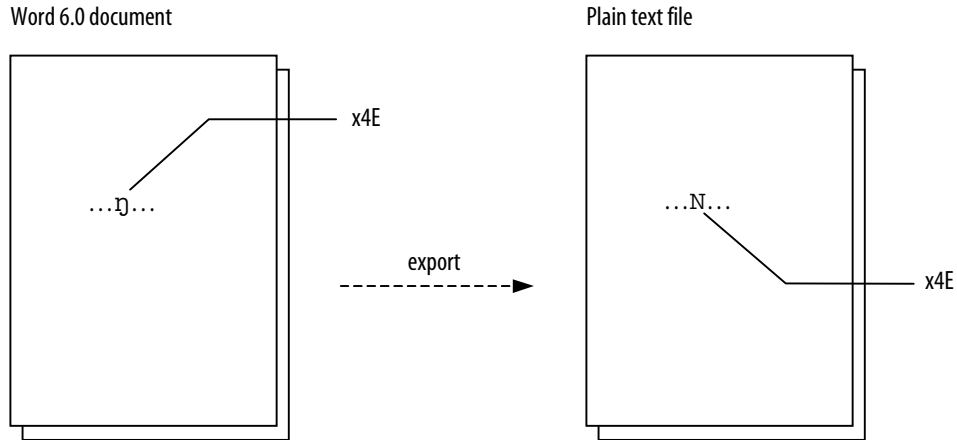


Figure 18: No character translation on export to text prior to Word 97

The reason this happens in Word 97 and later is that, whenever Word exports to an 8-bit text file or to a file format for an earlier version of Word, it is forced to translate every character from Unicode to an 8-bit value. Since the text was entered using a symbol font, it is stored by Word as Unicode characters in the range U+F020–U+F0FF. As mentioned above, when characters are formatted with a symbol font, and hence are in this range, Word has no way of knowing what codepage to associate with those characters. Nevertheless, when exporting to a text file, Word must make some decision about what codepage to use for the translation. For lack of any better alternative, Word simply uses the default system codepage, 1252 on a typical installation of US Windows.

Even though Word will make the translation using a system codepage, no Windows codepage supports characters in the range U+F020–U+F0FF in the domain of its Unicode-to-8-bit mapping table. As a result, for each character in that Unicode range it simply returns a default error value of 0x3F, “?”. (See Figure 19.) Thus, the user’s text gets converted to a sequence of question marks.

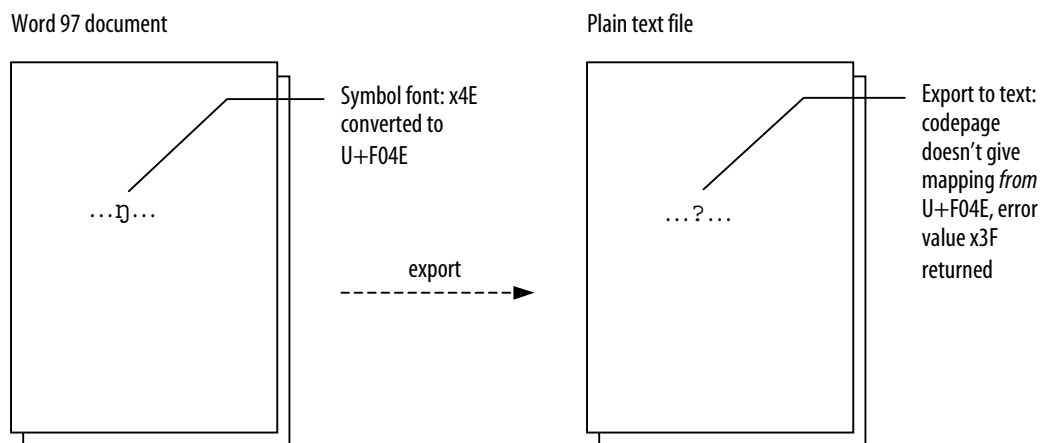


Figure 19: Export to text on Word 97: symbol characters converted to “?”

The only solution to this problem that we have available at this time requires that the system codepage for your installation of Windows is 1252 and that you have Martin Hosken’s modified codepage 1252 installed, and it assumes that the desired codepage of the text is 1252. (Note: this solution can also be applied to the problem described in the previous section, that of text appearing as boxes, since that issue also involves the need to convert from the range U+F020–U+F0FF down to 0x20–0xFF.)

To implement this solution, all that is needed is to install Martin's modified codepage 1252, and then to go back into Word and export your data to a text file again. With the modified codepage installed, it will all come out as Roman characters in the range 0x20–0xFF.

The reason this works is that Martin's modified codepage 1252 provides a mapping from U+F020–U+F0FF down to 0x20–0xFF. Thus, when Word attempts to use codepage 1252 to do the translation, the codepage is able to return 8-bit character values rather than the default error value, and so the text gets translated to Roman characters.

It should be noted that this solution to work around the problem requires some strong caveats. Altering system files always involves risks, since you have no way to know for certain what software might be detrimentally impacted. Also, it is not reliable since the customised codepage file could potentially be replaced with a stock version whenever the user installs new software or software updates on the system. In particular, both Windows 2000 and Windows Me are configured in such a way that they would automatically reinstall the original version of the codepage file. If you really need to find a way to work around this problem, however, and understand the risks, this is option is possible.

This discussion has focused on situations involving symbol-encoded fonts. As was seen in §4, there are other situations in which an export process may result in characters being converted into question marks. This will happen whenever the codepage used for conversion does not support the Unicode character in question.

Here is a user scenario for this case: Someone is creating a Thai document that is going to be distributed to a number of people (i.e. the Word DOC file will be distributed, not a printed version.) She prepares a draft using Thai Word 2000 on Thai Windows 98, and then gives the document to a co-worker to be checked. The co-worker is using the US versions of Word 2000 and Windows 98. He concludes that the document is ready for distribution, and decides to save the file in Word 6.0/95 format, thinking it might be more portable. After they start handing out copies, however, they get reports back that the document contains a lot of question marks, but no Thai characters. In this scenario, the Thai codepage should have been used for the conversion but was not. This might occur either because the codepage was not installed on the user's system, or because US Word 2000 was not making use of the codepage (possibly because support for Thai characters did not make it into this version). The solution, in this case, would have been for the file to be saved in Word 6.0/95 format using the Thai version of Word 2000.

In another scenario, a user is creating a Hindi document using the South Asia version of Word 2000 running on Windows 2000. They save the file in Word 6.0/95 format, however, and all of the Devanagari characters are converted to question marks. In this case, the Devanagari characters were lost because there is no Windows codepage for Devanagari. In this situation, there is no solution other than to stick with file formats and software that support Unicode.²⁸

8. Problem: Lines wrap at any character rather than only at spaces

With symbol fonts that contain minority-language orthographic characters or IPA symbols, Word has always given occasional problems with line wrapping and with word selection (i.e. what happens when you double-click on a character or when you hit the left- or right-arrow key while the CTRL key is depressed). Even so, a careful font developer could prevent a user using versions of Word prior to Word 97 from having frequent problems with lines that inappropriately wrapped in the middle of a word by making the right decisions about what characters to put at what codepoints

²⁸ Another possibility might be to save the file as plain text encoded in Unicode, and then run the file through an encoding-conversion processor to convert from Unicode to some 8-bit encoding that supports Devanagari, such as ISCII. The resulting file could then be opened in Word 2.x, Word 6 or Word 95, and the text formatted with a custom font designed for that Devanagari encoding. Of course, all of the style and formatting information would be lost in the process.

Word 97 and later behave quite differently with text that is formatted using a symbol font than did previous versions, however. As a result, in Word 97 and later, line wrapping with symbol fonts has become a serious problem.

In earlier versions, Word treated the characters in some ways as if they had the same properties as they would if a non-symbol font had been used. Specifically, earlier versions of Word made decisions about what characters are word-forming characters using the current codepage regardless of whether the text is formatted with a symbol font or a non-symbol font. This affects the behaviour of both line wrapping and word selection. Thus, suppose a font contains a LETTER BARRED L at 0x97 (d151, assuming codepage 1252 for sake of discussion; i.e. , the font has a glyph for a LETTER BARRED L encoded at U+2014). Word 95 and earlier would evaluate that character in terms of the current codepage (we're assuming codepage 1252) and would treat that character as though it were an em dash. The codepage tells Word that an em dash is not a word-forming character, and that it is acceptable to wrap a line after an em dash. If the LETTER BARRED L happened to fall in the middle of a word and that word was just too big to fit at the end of a line, Word would wrap the line right after the LETTER BARRED L. Likewise, if a user double clicked on a letter in the word to the left of the LETTER BARRED L, only the letters up to but not including the LETTER BARRED L would be selected rather than the entire word. If, however, the font developer avoided codepoints like 0x97 for any word-forming characters, such behaviours would not have been encountered by the user.

In Word 97 and later, the rules are different. Since the text is formatted with a symbol font, all the character values get translated up into the range U+F020–U+F0FF. Word no longer has any character set information to associate with the text that can tell it anything about which characters are word-forming. Accordingly, it simply applies a default rule which says, in effect, that every character is word-forming. The effect of this on line wrapping is that Word tries to keep all the text together on a line but, if it can't, it breaks the line after whichever character provides the best fit. For word selection, double-clicking on any character in a run of text that is formatted with a symbol font will cause that entire run of text to be selected.

What's important to note in this is that, with a symbol font selected, even the space character, 0x20, gets translated to U+F020, and U+F020 gets treated no differently than any other character in the range U+F020–U+F0FF. (Try using the ShowUnicode macro to verify that symbol "spaces" are actually U+F020.) Therein lies the basis for a solution which is conceptually very simple and gives very good results: Change the U+F020 characters to U+0020, i.e. to something that Word will recognise as spaces.

There are a couple of tricks to this: short of changing each "space" by hand, how do you get Word to find instances of U+F020, and how do you get it to replace them with U+0020? We have seen how we can search for U+F020. Consider the latter issue: Suppose we had a way to tell Word to insert a character U+0020. Even so, with a symbol font selected, Word will simply translate this back to U+F020. As a result, to make this work, we have to replace the U+F020 characters with U+0020 characters that are formatted with a non-symbol font.

So, the solution to this problem is to search for U+F020 and replace with U+0020 formatted with a non-symbol font. You can decide whether to do the search-and-replace throughout a document or only over a range of selected text. The decimal equivalent of U+F020 is 61462. Figure 20 shows how the Find and Replace dialog should appear (the Word 2000 dialog is shown):

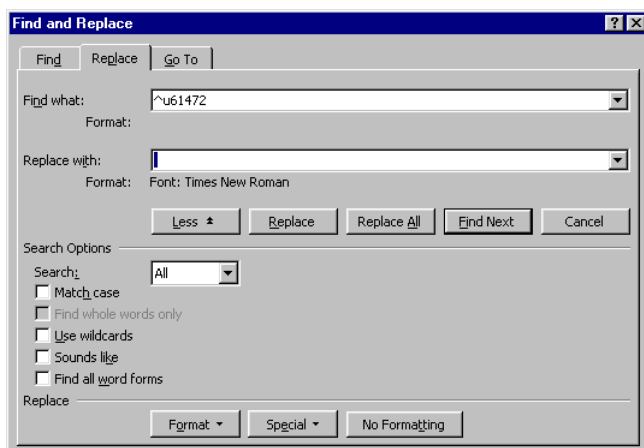


Figure 20: Find and Replace dialog for fixing symbol “spaces”

Note that the “Replace with” text is a space character and that it is formatted with a non-symbol font, Times New Roman.

If you forget the decimal equivalent for U+F020, you can specify the “Find what” text another way: select a symbol “space” character in your text, copy it to the clipboard (CTRL-C) and paste it (CTRL-V) into the “Find what” text box on the dialog. Or, in Word 2000, enter “f020” in the “Find what” text box and then press ALT-X.

To make things easier for you, we have provided a macro that will do this for you. The FixSymbolSpaces macro looks throughout a file for U+F020 and replaces it with U+0020 formatted in Times New Roman. The macro assumes that you want to make this change throughout the document. It also assumes that the document doesn’t use any non-symbol font that contains a character at U+F020. (If such characters occurred in a document, you might not want to change them to U+0020, but this macro will always make that change. It’s not very likely that you’d be using such a font, unless you are explicitly making use of the Unicode Private Use Area, however.) Also, in its current version, the macro doesn’t give any option of what font is used for the replacement spaces.

To use the macro on the active document, simply run it.

Whatever method you use to change the spaces, just remember that if you insert any new spaces into your document that are formatted with a symbol font, you may want to fix those as well.

9. Problem: Certain characters in fonts don’t work after switching from Windows 95 to Windows 98 (or later)

As users started switching to Windows 98, we starting getting reports of font “not working” in the new version. These situations all involved custom fonts, and always involved codepoints 0x80, 0x8E and 0x9E (d128, d142, d158) in codepage 1252.

In the spring of 1998, Microsoft revised various codepages, including codepage 1252, to add support for the euro currency symbol. In the case of codepage 1252, two other additional characters used for some European languages, “Ž” and “ž”, were also added. This was first introduced as a patch to Windows 95 (W95Euro.exe) and as part of service pack 4 for Windows NT 4, and it became a standard part of all subsequent versions of Windows, beginning with Windows 98.

The crucial effect of this update to codepage 1252 was to change the 8-bit-to-Unicode mapping for the three codepoints mentioned. The old and new mappings are shown in Table 2:

codepage 1252 codepoint	old Unicode value	new Unicode value
0x80	U+0080	U+20AC
0x8E	U+008E	U+017D
0x9E	U+009E	U+017E

Table 2: Changes in codepage 1252 when support for euro was added

This change affected some custom-encoded fonts that had been created in accordance with the old mappings. As mentioned above (see notes 2 and 8), TrueType fonts on Windows have always internally been encoded using Unicode. Thus, even when an application was working with 8-bit text, the operating system would have to use a codepage to convert the 8-bit codepoints to Unicode before accessing glyphs inside a font. For standard-encoded fonts, font developers would create the fonts so that the glyphs were accessed using the appropriate Unicode values for the characters being represented. In custom-encoded fonts, however, glyphs are in fact assigned according to some particular Windows codepage, usually codepage 1252. In other words, in order to access a glyph for (say) “u” using a codepoint of (say) 0xB5 (d181), the glyph would be encoded in the font using the Unicode value that corresponds to 0xB5 in codepage 1252, which is U+00B5.

So, a custom-encoded font is built assuming a certain set of codepage mappings to Unicode, and it is assumed that the font will be used on a system that has those same codepage mappings. If the codepage on a particular system is changed, however, so that it no longer matches the assumed mappings, the font will not work as intended. In the case of codepage 1252, this is exactly what happened: the codepage on users systems was changed.

For example, someone might have a Devanagari font that was designed to have d128 (0x80) display “ढ”. Since the font was originally designed for use with Windows 3.1, it assumed the original definitions for codepage 1252. Inside the font, this glyph is accessed via the Unicode value U+0080. On a Windows 3.x or original Windows 95 system, whenever a character d128 was entered in a document, this would get translated by the application or the operating system into U+0080 using codepage 1252. Suddenly, this user upgrades to Windows 98, however, and their system now has a new version of codepage 1252. When they enter d128 into a document, it no longer gets translated into U+0080; instead it is translated into U+20AC. But, this font does not contain any glyph that is accessed via U+20AC. The end result is that the default glyph, “□”, is displayed instead.

There are a couple of solutions to work around this problem, but the preferred fix is to modify the font so that both the old and new Unicode values present the same glyph. So, in the preceding example, the font would be changed so that the glyph “ढ” is accessed by both U+0080 and by U+20AC. This way, the font works as desired regardless of whether the old or the new version of the codepage is installed on a system. Details of the solutions, along with more detail regarding the problem, can be found in Hallissy (1998).

There is one implication specifically for Word 97 and Word 2000 that should be mentioned: since Word 97 arrived before the changes to the Windows codepages were made, you may have documents that were created using the original codepage definitions. Recall that, in Word 97 and later, 8-bit character codes are converted via a codepage as they are entered. So, for example, if you created a document in Word 97 running on (pre-euro patch) Windows 95 using a custom-encoded font, then you may have entered characters d128, d142 or d158 into the document. If so, then these codepoints were converted into Unicode values at that time.

This would only be an issue in using custom fonts, and it is not likely to be a problem in most cases since the old fonts can still be used to view the existing documents, even if they have not been modified to work with the new codepage definitions. In certain situations, this could be a concern. For example, the `AllChars97.dot` and `Codepg97.dot` templates that have been used by some within SIL to create charts that show the character sets of custom fonts were created in Word 97 on Windows 95 using the

original codepage definitions. As a result, they would not necessarily reflect the output that a user might get when using a custom font on Windows 98. Fortunately, this kind of problem is unlikely to affect many people. It can be frustrating and misleading for the few it does affect, though.

10. Problem: Word 2000 displays text with a different font than the font used to format the text

One of the consequences of using Unicode is that it is now much easier to provide true multilingual—and multi-script—support in applications. Nearly 50,000 different characters are defined in version 3 of Unicode, and documents created in programs like Word 2000 can potentially contain any number of them. This raises a potential concern, however: there are several problems with creating and using fonts that support the complete range of Unicode characters, and font developers generally have very little interest in doing so. But then, a user might receive a document or view a web page containing characters that are not supported in any of the fonts installed on their system. In that situation, text displayed with a default font would be illegible; that is, as nothing but empty boxes. Companies like Microsoft and Apple consider that to be an unacceptable user experience, and are working to find solutions to this problem.

The solution MS has adopted in Office 2000 involves a technique they refer to as *font linking*. They distribute their software with a collection of fonts that together cover a significant portion of Unicode (all of those ranges of Unicode that they have chosen to support). The idea is that, if the user's fonts don't support all of the characters in a document, then this collection of fonts can be used to at least make the document legible. When software gets some text to be displayed, they examine the characters in the text and then try to ascertain whether the font(s) that the user has chosen for displaying the text provide support for those characters. They apply various heuristic tests for this purpose, and if these tests conclude that the selected font does not support the given characters, then they will link up one of the fonts distributed with the software to display the text.

What this means is that text will not always be displayed with the font you want. It is a particular problem where custom-encoded fonts have been used: because these are custom fonts, there is probably a higher likelihood that they will fail the heuristic tests for some portions of the text. In that case, those portions of the text will be displayed using a stock font that will not have the custom character set, and the resulting text will not be completely legible. This is a feature that benefits 99.9% of Microsoft's users, but it can be frustrating for someone using custom-encoded fonts. What makes this problem particularly difficult is that the only workaround for using the custom font is to revert to earlier versions of software; for example, to stop using Word 2000 and use Word 95 instead.

Font linking can also affect some Unicode-conformant fonts as well, specifically fonts for ranges in Unicode that were not specifically part of the feature set for Word 2000. For example, Word 2000 was not designed to be used with Yi or Ethiopic characters, and so was not tested using characters from these scripts. In working on fonts for Yi and Ethiopic scripts using Unicode encoding, I found that Word 2000 was substituting other fonts. In fact, the fonts that Word was using in place of our Yi and Ethiopic fonts also did not support those character ranges, and so the text would display as empty boxes. Again, this simply was not a situation that they designed or tested for.

Font linking has been used in MS Office since Office 97, in Publisher at least since Publisher 98 (I think it was also used in Publisher 97), and in Internet Explorer at least since version 5.0. Thanks to influential contacts that we have in the MS Office development group, it will be possible to disable font linking in Office 10 using a registry setting. Also, we know that testing will be done for all Unicode ranges in the next version of Office, and so the problem should not arise in the case of fonts for Unicode ranges such as Ethiopic and Yi that are otherwise not yet fully supported by Microsoft.

Starting with Windows 2000, however, font linking has been introduced into Windows itself. Since the feature has been developed independently for Windows, disabling it for Office may not be enough if Windows itself is making font substitutions. It would still be an issue as well for Internet Explorer running

on any version of Windows. This problem will not likely go away entirely, though hopefully as Unicode support improves and as users adopt Unicode-based solutions, incidents will become infrequent.

11. Problem: Word 2000 does not allow text to be formatted with certain fonts

Users may find that Word 2000 does not allow text to be formatted with certain fonts. This behaviour is a minor issue and is related to font linking. It primarily affects Unicode-conformant fonts that support ranges of Unicode that are not specifically supported in Office 2000.

The various scripts supported in Word are of a few different types, each having particular characteristics, and Word provides some specific behaviours for each type. Depending upon what language support is enabled (set in the separate MS Office Language Settings applet), the Format/Font dialog may show three different list boxes to select fonts, one each for Latin script fonts, Asian fonts, and complex (right-to-left) script fonts. Thus, text can actually be formatted with three different fonts simultaneously, though each applying to distinct Unicode character ranges.

Word uses information inside fonts to determine which of its three categories a font fits in, and compares this with the selected text. When selecting a font from the font list on the toolbar, if Word's heuristic tests conclude that the font is for a different category than the selected text, it will not apply the change to the text. This may occur even if the font that was chosen does support the characters in the selected text. If you look in the font dialog, however, you will find that the font that was chosen has been applied for a different category of characters.

For example, the SIL Yi font supports Yi script but also the Latin characters of ASCII. Word categorises this font as an Asian font. If a string of English words is selected (i.e. the string contains only ASCII characters), we would expect that we should be able to apply that font to the text. Word will apply that font, but only for Asian characters, however. The English, Latin characters will still be formatted with the original font. To succeed in actually formatting the Latin text with the selected font, it is necessary to open the font dialog, select the SIL Yi font as the Asian font, and set the Latin font setting to "(Use Asian text font)".

This behaviour is not likely to be a serious problem for many users, and behaviour is likely to improve as Unicode support in Word matures with future versions.

12. Problem: Certain characters are converted to sequences like "(c)" when file is saved as text

This is not a specifically Unicode-related problem, though it does involve issues of codepages and symbol versus non-symbol fonts. It is also new in Word 97, and applies to Word 2000 as well.

With symbol fonts, we explained that, in exporting to text, the system has no way of knowing what the individual characters are and so everything gets translated to question marks (with a standard codepage, not with Martin's modified codepage 1252). You would think that, if text was formatted with a non-symbol font, that Word should be able to translate from the Unicode values back to 8-bit values without difficulty. In principle, this is true.

In certain particular situations, though, it may actually be preferable to get equivalent substitutes rather than the original character; for example, to replace the curly quotation characters with straight quotation marks. This might be true, for example, if a file had to be interpreted by processes designed for 7-bit ASCII. For whatever reason (probably because it was helpful to some significant group of customers), someone at Microsoft at some point decided that, when a file is saved to text, certain substitutions from codepage 1252 should occur, replacing characters using only characters in the ASCII range. Since this applies to codepage 1252, it affects only text formatted with non-symbol fonts. A complete list of changes that are made is given in Table 3.

Character	codepoint	description	gets converted to	character(s)	codepoint(s)	description
...	x85	ellipsis		...	x2E x2E x2E	sequence of three periods
'	x91	left single curly quotation mark		'	x27	apostrophe
'	x92	right single curly quotation mark		'	x27	apostrophe
"	x93	left double curly quotation mark		"	x22	straight double quotation mark
"	x94	right double curly quotation mark		"	x22	straight double quotation mark
—	x96	en dash		-	x2D	hyphen
—	x96	em dash		-	x2D	hyphen
™	x99	trademark sign		(tm)	x28 x74 x6D x29	sequence of ASCII characters
	xA0	non-breaking space			x20	space
©	xA9	copyright sign		(c)	x28 x63 x29	sequence of ASCII characters
®	xAE	registered trademark sign		(r)	x28 x72 x29	sequence of ASCII characters
¼	xBC	fraction one quarter		¼	x31 x2F x34	sequence of ASCII characters
½	xBD	fraction one half		½	x31 x2F x32	sequence of ASCII characters
¾	xBE	fraction three quarters		¾	x33 x2F x34	sequence of ASCII characters

Table 3: Word 97/Word 2000 translation of codepage 1252 characters on export to text

Of course, if you are using a custom font that has a y with a circumflex over it at codepoint 0x99 (assuming a codepage 1252), you probably don't want to have that codepoint replaced with "(tm)" when you export your document to text.

Apparently, there are factors that determine when this will occur of which we are unaware. I ran tests on two copies of Word 97, both with the same version number and file date; one made these translations, and the other did not. We have not been able to identify any setting in any dialog that controls this, though. That is exactly where the design weakness lay: not giving users a way to control whether or not these substitution should be made. This problem is being addressed in Word 10, however.

For now, there is a fairly simple way to get around this problem. If you have Martin Hosken's modified codepage 1252 installed on your system (see the previous section), then simply reformat all the text in your document with any symbol font, such as Wingdings, just before you export it to a text file. Martin's modified codepage will cause all characters formatted with a symbol font to export to text via a mapping that preserves 8-bit codepoint identity.

13. Problem: Certain characters do not work with Word 2000 when Arabic or Hebrew support is enabled

There are a few known bugs in Word 2000 when support for a right-to-left script is enabled.²⁹ The first of these can be a serious hindrance for users who work with Arabic or Hebrew scripts and who also use custom fonts or characters from the upper half of codepage 1252.

This first problem occurs if Arabic or Hebrew support is enabled in Word 2000 and you are working with settings for codepage 1252; that would be the case when using a keyboard layout for English or other languages that use codepage 1252, or if using Keyman. When, these conditions apply, then Word will convert the 8-bit codes d157, d253 and d254 as it receives them from the keyboard according to the Arabic codepage (codepage 1256) rather than codepage 1252. The mappings to Unicode that should result with codepage 1252 and the actual mappings that result in this situation are shown in :

²⁹ This set in the separate MS Office Language Settings applet. Note that enabling right-to-left support in Office 2000 does not require Arabic or Hebrew versions of Windows; right-to-left support in Office 2000 can be enabled on any version of Windows.

code from keyboard	codepage 1252 mappint to Unicode	actual mapping in this situation
d157 (0x9D)	U+009D	U+200C
d253 (0xFD)	U+00FD	U+200E
d254 (0xFE)	U+00FE	U+200F

Table 4: Incorrect codepage mappings with right-to-left support enabled

The implication of this is that a user with a custom font that is designed to use d157, d253 or d254 (i.e. has glyphs mapped from U+009D, U+00FD or U+00FE in the font) will not see the glyphs that they expect (unless the custom font was designed to work on an Arabic system using codepage 1256). This problem can affect users even if they are not using custom fonts: if you need characters d253 “ÿ” or d254 “þ” from codepage 1252 (in codepage 1252, d157 is undefined), these cannot be entered into a document.

The erroneous mapping appears to happen regardless of how the characters are input from the keyboard. In testing, it happened both when using the United States 101 keyboard layout and entering the characters with ALT-key combinations, and also when using the United States-International keyboard layout, which defines keystrokes for d253 and d254. On the other hand, the error appears not to happen when codepage 1252 is not active. In testing, it did not happen when using keyboard layouts for Polish or Vietnamese, which activate codepages 1250 and 1258 respectively. Unfortunately, that does not necessarily help users: codepages other than codepage 1252 will not provide the mappings to Unicode that are usually assumed by custom fonts. Also, for those that require the “ÿ” or “þ” characters in codepage 1252, using another codepage obviously will not necessarily provide the combination of characters that they need.

This problem pertains to character input from the keyboard only. It does not affect importing of data from text files, nor does it affect exporting to text files.

The following are possible workarounds at this point:

- Avoid these three codes.
- Devise some alternate way to enter characters, e.g. using Word macros, that enters the Unicode values directly.
- Disable Arabic support.
- Revert to an earlier version of Word.
- Redesign your custom font to work with a different codepage than 1252 (not recommended as there are several potential complexities that we have little experience with).

It should be reiterated that this problem will affect you only if Arabic or Hebrew support is enabled. Thus, many users can disregard this issue entirely.

14. Problem: Character code d211 does not work properly when running on Thai Windows 95/98

This is not actually a problem in Word 97 or Word 2000, but rather is a bug in Thai Windows. It relates to our general topic of encodings, codepages, Unicode and fonts, and it has presented problems for some using custom fonts in Word 97 or Word 2000 on Thai Windows. Thus, it seems appropriate to mention it here. The symptom is that, when codepoint d211 (0xD3) is entered into a document, a pair of glyphs appear that are different from the glyph expected for d211.

The source of the problem comes from certain code that was used in Thai Windows 3.x to handle rendering of Thai script. One of the vowels in Thai script, *sara am*, is a digraph that has one part written above the syllable-initial consonant, and another part that is written as a full character that sits on the

baseline: “๗”. Each of the components of this digraph is a distinct character in Thai script, and so each would have its own glyph in a font. To handle the display of sara am in Thai Windows 3.x, it was decided to replace the single character with the sequence of characters that represent the two components: d210, d237. This was all done internally to Windows, and so was invisible to the user.

Thai Windows 3.x implemented support for Thai script using custom-encoded fonts that assumed the codepage mappings defined in codepage 1252 (a surprising hack, actually, for software published by MS). To prevent the transformation of d211 from happening with standard fonts that conform to codepage 1252 and Unicode, a special flag was set inside the font that only Thai Windows would recognise. When that flag was set, it would apply the transformation; otherwise, it would not.

In the Thai version of Windows 95, the implementation of Thai script support was redone to make it conform to Unicode and the standard definition of codepage 1252. This meant that all of the code for handle the complex rendering rules of Thai script should have been replaced. Somehow, however, the bit of code that changed d211 into the sequence d210, d237 was overlooked. My understanding is that it is still there in Thai Windows 98.

I have not had opportunity to test the Thai versions of Windows 95 and Windows 98, so I do not know whether it affects all fonts or only certain fonts. I do know, though, that it is not limited to fonts that have the special flag that was used to indicate a Thai font in Windows 3.x.

Unfortunately, the only solution to this problem on Thai Windows 95/98 is either to avoid character code d211, or to switch to another operating system. For someone creating a custom-encoded font, they may be able to stay clear of this problem codepoint. That does not necessarily help someone using a font that was created by someone else, however. A good alternative for users to consider is to switch to Windows 2000, which provides support for Thai script (and several others), and does not suffer from this problem.

15. Working in a Unicode world

The advances that MS has made in Word 97 and, especially, Word 2000 offer important benefits for working with multilingual text. In Office 95, there were no real solutions for certain situation, such as mixing Chinese and Arabic text in a single file. In the span of a few years, we have gone from Word 95, with limited ability to create documents that mix languages and no real ability to mix significantly different scripts, to Word 2000, which makes it possible to mix text in a large number of languages involving several of the world’s major scripts. Of course, we have seen that there are still bugs, and the goal of one global product that supports the scripts of all of the markets they have targeted still has not been met. Nevertheless, progress has been considerable.

Even so, the world’s languages span thousands of varieties, and users in many cases still find that the writing systems in which they are interested are not adequately supported. Many users with multilingual needs also depend on software tools from other sources that do not yet match the multilingual capabilities of Office 2000. As a result, many users continue to work with custom-built solutions using non-standard character sets and encoding.

It is these users who are confronted by the most serious problems. Yet the very source of most of these problems comes from the fact that they are going against the encoding standards that are increasingly being assumed.

For the computer industry, Unicode and related technologies are seen to be the way of the future, and industry support for these technologies is growing at an increasing rate. There is no possible long-term solution to the symbol font problems. There is also no incentive for Microsoft to abandon font-linking techniques, which address real concerns in a way that helps the vast majority of their users. The only long-term solutions for us to avoid these serious problems will come as commercial software from companies like MS continues to adopt these standards and grow in multilingual capabilities, and as we abandon the

non-conformant practices we have come to depend upon and to begin adopting the same standards and technologies.

There are certainly concerns for us to consider in this. The biggest of these is that we are still in transition: some tools based on the newer technology paradigms are available now, but not all that we need. As long as we still need to depend on older software tools, we will face challenges of trying to move back and forth between the old and new, and of continuing to try to find multilingual solutions using those old technologies.

Fortunately, this course of action holds many more benefits for us than it does concerns. The new technologies available in Windows - Unicode and smart fonts - are some of the very things we have needed all along. They alone don't meet the needs of all minority languages, but they do provide the basis that will allow us to build effective and long-term solutions for each of these languages.

16. Contacting NRSI

To obtain copies of either of the files that were mentioned, the macros template or the modified version of codepage 1252, contact the NRSI at:

nrsi_ipub@sil.org

17. References

- Constable, Peter. 1997. Unicode capability in Microsoft Word 97. *NRSI Update #5*. Also available in International Publishing Services (1998) and in SIL International (2000).
- . 1998. Unicode Issues in Microsoft Word 97 and Word 98. Available in International Publishing Services (1998).
- . 2000a. Font issues in MS Windows & Office. *NRSI Update #13*. Also available in SIL International (2000).
- . 2000b. Understanding characters, keystrokes, codepoints and glyphs: Encoding and working with multilingual text. Available in SIL International (2000).
- . 2000c. Understanding multilingual software on MS Windows. Available in SIL International (2000).
- Hallissy, Bob. 1998. The BoxChar Mysteries presents... The Euro case. Available in International Publishing Services (1998), and also in SIL International (2000).
- Hosken, Martin. 1997. Windows and codepages. *NRSI Update #8*. Also available in International Publishing Services (1998), and in SIL International (2000).
- International Publishing Services, 1998. *Resource Collection 98 CD*. Dallas: Summer Institute of Linguistics.
- Kano, Nadine. 1995. *Developing international software for Window 95 and Windows NT*. Redmond, WA: Microsoft Press. Also available online at <http://msdn.microsoft.com/library/books/devintl/S24AE.HTM>.
- Microsoft Corporation. 1997. WD97: Symbol characters change to box characters. Microsoft Knowledge Base article, ID Q160022, available at <http://support.microsoft.com/support/kb/articles/q160/0/22.asp>.
- SIL International. 2000. *CTC Resource Collection 2000*. (CD-ROM.) Dallas: SIL International