Architecture: Not Just a Pile of Code Charts

If you're used to working with ASCII or other similar encodings designed for European languages, you'll find Unicode noticeably different from those other standards. You'll also find that when you're dealing with Unicode text, various assumptions you may have made in the past about how you deal with text don't hold. If you've worked with encodings for other languages, at least some characteristics of Unicode will be familiar to you, but even then, some pieces of Unicode will be unfamiliar.

Unicode is more than just a big pile of code charts. To be sure, it *includes* a big pile of code charts, but Unicode goes much further. It doesn't just take a bunch of character forms and assign numbers to them; it adds a wealth of information on what those characters mean and how they are used.

Unlike virtually all other character encoding standards, Unicode isn't designed for the encoding of a single language or a family of closely related languages. Rather, Unicode is designed for the encoding of *all* written languages. The current version doesn't give you a way to encode *all* written languages (and in fact, this concept is such a slippery thing to define that it probably never will), but it does provide a way to encode an extremely wide variety of languages. The languages vary tremendously in how they are written, so Unicode must be flexible enough to accommodate all of them. This fact necessitates rules on how Unicode is to be used with each language. Also, because the same

108_GILLAM.ch03.fm Page 62 Monday, August 19, 2002 1:58 PM

encoding standard can be used for so many different languages, there's a higher likelihood that they will be mixed in the same document, requiring rules on how text in the different languages should interact. The sheer number of characters requires special attention, as does the fact that Unicode often provides multiple ways of representing the same thing.

The idea behind all the rules is simple: to ensure that a particular sequence of code points will get drawn and interpreted the same way (or in semantically equivalent ways) by all systems that handle Unicode text. In other words, it's not so important that there should be only one way to encode "à bientôt," but rather a particular sequence of code points that represents "à bientôt" on one system will also represent it on any other system that purports to understand the code points used. Not every system has to handle that sequence of code points *in exactly the same way,* but merely must interpret it as meaning the same thing. In English, for example, you can follow tons of different typographical conventions when you draw the word "carburetor," but someone who reads English would still interpret all of them as the word "carburetor." Any Unicode-based system has wide latitude in how it deals with a sequence of code points representing the word "carburetor," as long as it still treats it as the word "carburetor."

As a result of these requirements, a lot more goes into supporting Unicode text than supplying a font with the appropriate character forms for all the characters. The purpose of this book is to explain all these other things you have to be aware of (or at least *might* have to be aware of). This chapter will highlight the things that are special about Unicode and attempt to tie them together into a coherent architecture.

■ The Unicode Character–Glyph Model

The first and most important thing to understand about Unicode is what is known as the *character–glyph model*. Until the introduction of the Macintosh in 1984, text was usually displayed on computer screens in a fairly simple fashion. The screen would be divided up into a number of equally sized *display cells*. The most common video mode on the old IBM PCs, for example, had 25 rows of 80 display cells each. A video buffer in memory consisted of 2,000 bytes,

108_GILLAM.ch03.fm Page 63 Monday, August 19, 2002 1:58 PM

one for each display cell. The video hardware contained a **character generator chip** that contained a bitmap for each possible byte value, and this chip was used to map from the character codes in memory to a particular set of lit pixels on the screen.

Handling text was simple. There were 2,000 possible locations on the screen, and 256 possible characters to put in them. All the characters were the same size, and were laid out regularly from left to right across the screen. There was a one-to-one correspondence between character codes stored in memory and visible characters on the screen, and there was a one-to-one correspondence between keystrokes and characters.

We don't live in that world anymore. One reason is the rise of the WYSIWYG ("what you see is what you get") text editor, where you can see on the screen exactly what you want to see on paper. With such a system, video displays have to be able to handle proportionally spaced fonts, the mixing of different typefaces, sizes, and styles, and the mixing of text with pictures and other pieces of data. The other reason is that the old world of simple video display terminals can't handle many languages, which are more complicated to write than the Latin alphabet is.

As a consequence, there's been a shift away from translating character codes to pixels in hardware and toward doing it in software. And the software for doing this has become considerably more sophisticated.

On modern computer systems (Unicode or no), there is no longer always a nice, simple, one-to-one relationship between character codes stored in your computer's memory and actual shapes drawn on your computer's screen. This point is important to understand because Unicode *requires* this flexibility—a Unicode-compatible system cannot be designed to assume a one-to-one correspondence between code points in the backing store and marks on the screen (or on paper), or between code points in the backing store and keystrokes in the input.[1]

Let's start by defining two concepts: character and glyph. A character is an atomic unit of text with some semantic identity; a glyph is a visual representation of that character.

---

1. Technically, if you restrict the repertoire of characters your system supports enough, you actually *can* make this assumption. At that point, though, you're likely back to being a fairly simplistic English-only system, in which case why bother with Unicode in the first place?

108_GILLAM.ch03.fm Page 64 Monday, August 19, 2002 1:58 PM

Consider the following examples:

13 thirteen 1.3×10^1
dreizehn $0C κγ 十三
١٢ 𝍸𝍸|||
treize ๑๓

These forms are 11 different visual representations of the number 13. The underlying semantic is the same in every case: the concept "thirteen." These examples are just different ways of depicting the concept of "thirteen."

Now consider the following:

g g *g* 𝔤

Each of these examples is a different presentation of the Latin lowercase letter g. To go back to our terms, these are all the same *character* (the lowercase letter g), but four different *glyphs*.

Of course, these four glyphs were produced by taking the small *g* out of four different typefaces. That's because there's generally only one glyph per character in a Latin typeface. In other writing systems, however, that isn't true. The Arabic alphabet, for example, joins cursively even when printed. This isn't an optional feature, as it is with the Latin alphabet; it's the way the Arabic alphabet is *always* written.

ه ـه ـهـ هـ

These are four different forms of the Arabic letter heh. The first shows how the letter looks in isolation. The second depicts how it looks when it joins only to a letter on its right (usually at the end of a word). The third is how it looks when it joins to letters on both sides in the middle of a word. The last form illustrates how the letter looks when it joins to a letter on its left (usually at the beginning of a word).

Unicode provides only one character code for this letter,[2] and it's up to the code that draws it on the screen (the **text rendering process**) to select the appropriate glyph depending on context. The process of selecting from among a set of glyphs for a character depending on the surrounding characters is called **contextual shaping**, and it's required to draw many writing systems correctly.

There's also not always a one-to-one mapping between character and glyph. Consider the following example:

ﬁ

This, of course, is the letter f followed by the letter i, but it's a single glyph. In many typefaces, if you put a lowercase f next to a lowercase i, the top of the f tends to run into the dot on the i, so the typeface often includes a special glyph called a ligature that represents this particular pair of letters. The dot on the i is incorporated into the overhanging arch of the f, and the crossbar of the f connects to the serif on the top of the base of the i. Some desktop-publishing software and some high-end fonts will automatically substitute this ligature for the plain f and i.

In fact, some typefaces include additional ligatures. Other forms involving the lowercase f are common, for example. You'll often see ligatures for a–e and o–e pairs (useful for looking erudite when using words like "archæology" or "œnophile"), although software rarely forms these automatically (æ and œ are actually separate letters in some languages, rather than combinations of letters), and some fonts include other ligatures for decorative use.

Again, though, ligature formation isn't just a gimmick. Consider the Arabic letter lam (ل) and the Arabic letter alef (ا). When they occur next to each other, you'd expect them to appear like this if they followed normal shaping rules:

لـا

2. Technically, this isn't true—Unicode actually provides separate codes for each glyph in Arabic, although they're included only for backward compatibility. These "presentation forms" are encoded in a separate numeric range to emphasize this point. Implementations generally aren't supposed to use them. Almost all other writing systems that have contextual shaping don't have separate presentation forms in Unicode.

108_GILLAM.ch03.fm Page 66 Monday, August 19, 2002 1:58 PM

Actually, they don't. Instead of forming a U shape, the vertical strokes of the lam and the alef actually cross, forming a loop at the bottom:

Unlike the f and i in English, these two letters *always* combine this way when they occur together. It's not optional. The form that looks like a U is just plain wrong. So ligature formation is a required behavior for writing many languages.

A single character may also split into more than one glyph. This happens in some Indian languages, such as Tamil. It's very roughly analogous to the use of the silent e in English. The e at the end of "bite," for example, doesn't have a sound of its own; it merely changes the way the i is pronounced. Since the i and the e are being used together to represent a single vowel sound, you could think of them as two halves of a *single* vowel character. Something similar happens in languages like Tamil. Here's an example of a Tamil split vowel:

This looks like three letters, but it's really only two. The middle glyph is a consonant, the letter ப. The vowel ◌ோ is shown with a mark on either side of the ப. This kind of thing is required for the display of a number of languages.

As we see, there's not always a simple, straightforward, one-to-one mapping between characters and glyphs. Unicode assumes the presence of a character rendering process capable of handling the sometimes complex mapping from characters to glyphs. It doesn't provide separate character codes for different glyphs that represent the same character, or for ligatures representing multiple characters.

Exactly how this process works varies from writing system to writing system (and, to a lesser degree, from language to language within a writing system). For the details on just how Unicode deals with the peculiar characteristics of each writing system it encodes, see Part II (Chapters 7 to 12).

■ Character Positioning

Another faulty assumption is the idea that characters are laid out in a neat linear progression running in lines from left to right. In many languages, this isn't true.

108_GILLAM.ch03.fm Page 67 Monday, August 19, 2002 1:58 PM

Many languages also employ diacritical marks that are used in combination with other characters to indicate pronunciation. Exactly where the marks are drawn can depend on what they're being attached to. For example, look at these two letters:

ä Ä

Each of these examples is the letter a with an umlaut placed on top of it. The umlaut needs to be positioned higher when attached to the capital A than when attached to the small a.

This positioning can be even more complicated when multiple marks are attached to the same character. In Thai, for example, a consonant with a tone mark might look like this:

อ้

If the consonant also has a vowel mark attached to it, the tone mark has to move out of the way. It actually moves up and becomes smaller when there's a vowel mark:

อื้

Mark positioning can get quite complicated. In Arabic, a whole host of dots and marks can appear along with the actual letters. Some dots are used to differentiate the consonants from one another when they're written cursively, some diacritical marks modify the pronunciation of the consonants, vowel marks may be present (Arabic generally doesn't use letters for vowels—they're either left out or shown as marks attached to consonants), and reading or chanting marks may be attached to the letters. In fact, some Arabic calligraphy includes marks that are purely decorative. There's a hierarchy of how these various marks are placed relative to the letters that can get quite complicated when all the various marks are actually being used.

بسم الله الرحمن الرحيم

108_GILLAM.ch03.fm Page 68 Monday, August 19, 2002 1:58 PM

Unicode expects that a text rendering process will know how to position marks appropriately. It generally doesn't encode mark position at all—it adopts a single convention that marks follow in memory the characters they attach to, but that's it.[3]

Diacritical marks are not the only characters that may have complicated positioning; sometimes the letters themselves do. For example, many Middle Eastern languages are written from right to left rather than from left to right (in Unicode, the languages that use the Arabic, Hebrew, Syriac, and Thaana alphabets are written from right to left). Unicode stores these characters in the order they'd be spoken or typed by a native speaker of one of the relevant languages, known as **logical order**.

Logical order means that the "first" character in character storage is the character that a native user of that character would consider "first." For a left-to-right writing system, the "first" character is drawn farthest to the left. (For example, the first character in this paragraph is the letter L, which is the character farthest to the left on the first line.) For a right-to-left writing system, the "first" character would be drawn farthest to the right. For a vertically oriented writing system, such as that used to write Chinese, the "first" character is drawn closest to the top of the page.

Logical order contrasts with **visual order**, which assumes that all characters are drawn progressing in the same direction (usually left to right). When text is stored in visual order, text that runs counter to the direction assumed (usually the right-to-left text) is stored in memory in the reverse of the order in which it was typed.

Unicode doesn't assume any bias in layout direction. The characters in a Hebrew document are stored in the order they are typed, and Unicode expects that the text rendering process will know that because they're Hebrew letters, the first one in memory should be positioned the farthest to the right, with the succeeding characters progressing leftward from there.

3. The one exception to this rule has to do with multiple marks attached to a single character. In the absence of language-specific rules governing how multiple marks attach to the same character, Unicode adopts a convention that marks that would otherwise collide radiate outward from the character to which they're attached in the order they appear in storage. This is covered in depth in Chapter 4.

108_GILLAM.ch03.fm Page 69 Monday, August 19, 2002 1:58 PM

This process gets really interesting when left-to-right text and right-to-left text are mixed in the same document. Suppose you have an English sentence with a Hebrew phrase embedded into the middle of it:

Avram said מזל טוב and smiled.

Even though the dominant writing direction of the text is from left to right, the first letter in the Hebrew phrase (מ) still goes to the right of the other Hebrew letters—the Hebrew phrase still reads from right to left. The same thing can happen even when you're not mixing languages: In Arabic and Hebrew, for example, even though the dominant writing direction is from right to left, numbers are still written from left to right.

This issue can be even more fun when you throw in punctuation. Letters have inherent directionality; punctuation doesn't. Instead, punctuation marks take on the directionality of the surrounding text. In fact, some punctuation marks (such as the parentheses) actually *change shape* based on the directionality of the surrounding text (called **mirroring**, because the two shapes are usually mirror images of each other). Mirroring is another example of how Unicode encodes meaning rather than appearance—the code point encodes the meaning ("starting parenthesis") rather than the shape (either "(" or ")" depending on the surrounding text).

Dealing with mixed-directionality text can become quite complicated, not to mention ambiguous, so Unicode includes a set of rules that govern just how text of mixed directionality is to be arranged on a line. The rules are rather involved, but are required for Unicode implementations that claim to support Hebrew or Arabic.[4]

The writing systems for the various languages used on the Indian subcontinent and in Southeast Asia have even more complicated positioning requirements. For example, the Devanagari alphabet used to write Hindi and Sanskrit treats vowels as marks that are attached to consonants (which are treated as "letters"). A vowel may attach not just to the top or bottom of the consonant, but

4. This set of rules, the Unicode Bidirectional Text Layout Algorithm, is explained in more detail later in this book. Chapter 8 provides a high-level overview, and Chapter 16 looks at implementation strategies.

108_GILLAM.ch03.fm Page 70 Monday, August 19, 2002 1:58 PM

also to the left or right side. Text generally runs left to right, but when a vowel attaches to the left-hand side of its consonant, you get the effect of a character appearing "before" (i.e., to the left of) the character it logically comes "after."

क + ि = कि

In some alphabets, a vowel can actually attach to the left-hand side of a group of consonants, meaning this "reordering" may actually involve more than just two characters switching places. Also, in some alphabets, such as the Tamil example we looked at earlier, a vowel might actually appear on *both* the left- and right-hand sides of the consonant to which it attaches (called a "split vowel").

Again, Unicode stores the characters in the order they're spoken or typed; it expects the display engine to do this reordering. For more on the complexities of dealing with the Indian scripts and their cousins, see Chapter 9.

Chinese, Japanese, and Korean can be written either horizontally or vertically. Again, Unicode stores them in logical order, and the character codes encode the semantics. Many of the punctuation marks used with Chinese characters have a different appearance when used with horizontal text than when used with vertical text (some are positioned differently, some are rotated 90 degrees). Horizontal scripts are sometimes rotated 90 degrees when mixed into vertical text and sometimes not, but this distinction is made by the rendering process and not by Unicode.

Japanese and Chinese text may also include annotations (called "ruby" or "furigana" in Japanese) that appear between the lines of normal text. Unicode includes ways of marking text as ruby and leaves it up to the rendering process to determine how to draw it. For more on vertical text and ruby, see Chapter 10.

■ The Principle of Unification

The bottom-line philosophy you should draw from the discussions on the character–glyph model and on character positioning is that *Unicode encodes semantics, not appearances*. In fact, the Unicode standard specifically states that the pictures of the characters in the code charts are for illustrative purposes only—

108_GILLAM.ch03.fm Page 71 Monday, August 19, 2002 1:58 PM

the pictures of the characters are intended to help clarify the meaning of the character code, not to specify the appearance of the character having that code.

The philosophy that Unicode encodes semantics and not appearances also undergirds the principle that Unicode is a plain-text encoding, which we discussed in Chapter 1. The fact that an Arabic letter looks different depending on the letters around it doesn't change what letter it is, and thus it doesn't justify having different codes for the different shapes. The fact that the letters lam and alef combine into a single mark when written doesn't change the fact that a word contains the letters lam and alef in succession. The fact that text from some language might be combined on the same line with text from another language whose writing system runs in the opposite direction doesn't justify storing either language's text in some order other than the order in which the characters are typed or spoken. In all of these cases, Unicode encodes the underlying meaning and leaves it up to the process that draws it to be smart enough to do so properly.

The philosophy of encoding semantics rather than appearance also leads to another important Unicode principle: the principle of **unification**.

Unlike most character encoding schemes, Unicode aims to be comprehensive. It aims to provide codes for all the characters in all the world's written languages. It also aims to be a superset of all other character encoding schemes (or at least the vast majority). By being a superset, Unicode can be an acceptable substitute for any of those other encodings (technical limitations aside, anyway), and it can serve as a pivot point for processes converting text between any of the other encodings.

Other character encoding standards are Unicode's chief source of characters. The designers of Unicode sought to include all the characters from every computer character encoding standard in reasonably widespread use at the time Unicode was designed. They have continued to incorporate characters from other standards as it has evolved, either as important new standards emerged, or as the scope of Unicode widened to include new languages. The designers of Unicode drew characters from every international and national standard they could get their hands on, as well as code pages from the major computer and software manufacturers, telegraphy codes, various other corporate standards, and even popular fonts, in addition to noncomputer sources. As an example of their thoroughness, Unicode includes code-point values for the glyphs that the

108_GILLAM.ch03.fm Page 72 Monday, August 19, 2002 1:58 PM

old IBM PC code pages would show for certain ASCII control characters. As another example, Unicode assigns values to the glyphs from the popular Zapf Dingbats typeface.

This wealth of sources led to an amazingly extensive repertoire of characters, but also produced redundancy. If every character code from every source encoding retained its identity in Unicode (say, Unicode kept the original code values and just padded them to the same length and prefixed them with some identifier for the source encoding), they would never fit in a 16-bit code space. You would also wind up with numerous alternative representations for things that anyone with a little common sense would consider to be the same thing.

For starters, almost every language has several encoding standards. For example, there might be one national standard for each country where the language is spoken, plus one or more corporate standards devised by computer manufacturers selling into that market. Think about ASCII and EBCDIC in American English, for example. The capital letter A encoded by ASCII (as 0x41) is the same capital letter A that is encoded by EBCDIC (as 0xC1), so it makes little sense to have these two source values map to different codes in Unicode. In that case, Unicode would have two different values for the letter A. Instead, Unicode *unifies* these two character codes and says that both sources map to the same Unicode value. Thus, the letter A is encoded only once in Unicode (as U+0041), not twice.

In addition to the existence of multiple encoding standards for most languages, most languages share their writing system with at least one other language. The German alphabet is different from the English alphabet—it adds ß and some other letters, for example—but both are really just variations on the Latin alphabet. We need to make sure that the letter ß is encoded, but we don't need to create a different letter k for German—the same letter k we use in English will do just fine.

A truly vast number of languages use the Latin alphabet. Most omit some letters from what English speakers know as the alphabet, and most add some special letters of their own. Just the same, there's considerable overlap between their alphabets. The characters that overlap between languages are encoded only once in Unicode, not once for every language that uses them. For example, both Danish and Norwegian add the letter ø to the Latin alphabet, but the letter ø is encoded only once in Unicode.

108_GILLAM.ch03.fm Page 73 Monday, August 19, 2002 1:58 PM

Generally, characters are not unified across writing system boundaries. For instance, the Latin letter B, the Cyrillic letter В, and the Greek letter Β are not unified, even though they look the same and have the same historical origins. This is partly because their lowercase forms are all different (b, б, and β, respectively), but mostly because the designers of Unicode didn't want to unify across writing-system boundaries.[5] It made more sense to keep each writing system distinct.

The basic principle is that, wherever possible, Unicode unifies character codes from its various source encodings, whenever they can be demonstrated beyond reasonable doubt to refer to the same character. One big exception exists: respect for existing practice. It was important to Unicode's designers (and probably a big factor in Unicode's success) for Unicode to be interoperable with the various encoding systems that came before it. In particular, for a subset of "legacy" encodings, Unicode is specifically designed to *preserve round-trip compatibility.* That is, if you convert from one of the legacy encodings to Unicode and then back to the legacy encoding, you should get the same thing you started with. Many characters that would have been unified in Unicode actually aren't because of the need to preserve round-trip compatibility with a legacy encoding (or sometimes simply to conform to standard practice).

For example, the Greek lowercase letter sigma has two forms: σ is used in the middle of words, and ς is used at the end of words. As with the letters of the Arabic alphabet, this example involves two different glyphs for the same letter. Unicode would normally just have a single code point for the lowercase sigma, but because all standard encodings for Greek give different character codes to the two versions of the lowercase sigma, Unicode has to do so as well. The same thing happens in Hebrew, where the word-ending forms of several letters have their own code point values in Unicode.

If a letter does double duty as a symbol, this generally isn't sufficient grounds for different character codes either. The Greek letter pi (π), for example, is still the Greek letter pi even when it's being used as the symbol of the

5. There are a few exceptions to this base rule: One that comes up often is Kurdish, which when written with the Cyrillic alphabet also uses the letters Q and W from the Latin alphabet. Because the characters are a direct borrowing from the Latin alphabet, they weren't given counterparts in Unicode's version of the Cyrillic alphabet, a decision that still arouses debate.

ratio between a circle's diameter and its circumference, so it's still represented with the same character code. Some exceptions exist, however: The Hebrew letter aleph (א) is used in mathematics to represent the transfinite numbers, and this use is given a separate character code. The rationale here is that aleph-as-a-mathematical-symbol is a left-to-right character like all the other numerals and mathematical symbols, whereas aleph-as-a-letter is a right-to-left character. The letter Å is used in physics as the symbol of the angstrom unit. Å-as-the-angstrom is given its own character code because some of the variable-length Japanese encodings did.

The business of deciding which characters can be unified can be complicated. Looking different is definitely not sufficient grounds by itself. For instance, the Arabic and Urdu alphabets have a very different look, but the Urdu alphabet is really just a particular calligraphic or typographical variation of the Arabic alphabet. The same set of character codes in Unicode, therefore, is used to represent both Arabic and Urdu. The same thing happens with Greek and Coptic,[6] modern and old Cyrillic (the original Cyrillic alphabet had different letter shapes and some letters that have since disappeared), and Russian and Serbian Cyrillic (in italicized fonts, some letters have a different shape in Serbian from their Russian shape to avoid confusion with italicized Latin letters).

By far the biggest, most complicated, and most controversial instance of character unification in Unicode involves the Han ideographs. The characters originally developed to write the various Chinese languages, often called "Han characters" after the Han Dynasty, were also adopted by various other peoples in East Asia to write their languages. Indeed, the Han characters are still used (in combination with other characters) to write Japanese (who call them *kanji*) and Korean (who call them *hanja*).

Over the centuries, many of the Han characters have developed different forms in the different places where they're used. Even within the same written language—Chinese—different forms exist: In the early 1960s, the Mao regime in the People's Republic of China standardized or simplified versions of many of the more complicated characters, but the traditional forms are still used in Taiwan and Hong Kong.

---

6. The unification of Greek and Coptic has always been controversial, because they're generally considered to be different alphabets, rather than just different typographical versions of the same alphabet. It's quite possible that Greek and Coptic will be disunified in a future Unicode version.

108_GILLAM.ch03.fm Page 75 Monday, August 19, 2002 1:58 PM

Thus the same ideograph can have four different forms: one each for Traditional Chinese, Simplified Chinese, Japanese, and Korean (and when Vietnamese is written with Chinese characters, you might have a fifth form). Worse yet, it's very often not clear what really counts as the "same ideograph" between these languages. Considerable linguistic research went into coming up with a unified set of ideographs for Unicode that can be used for both forms of written Chinese, Japanese, Korean, and Vietnamese.[7] In fact, without this effort, it would have been impossible to fit Unicode into a 16-bit code space.

In all of these situations where multiple glyphs are given the same character code, it either means the difference in glyph is simply the artistic choice of a type designer (for example, whether the dollar sign has one vertical stroke or two), or it's language dependent and a user is expected to use an appropriate font for his or her language (or a mechanism outside Unicode's scope, such as automatic language detection or some kind of tagging scheme, would be used to determine the language and select an appropriate font).

The opposite situation—different character codes being represented by the same glyph—can also happen. One notable example is the apostrophe ('). There is one character code for this glyph when it's used as a punctuation mark and another when it's used as a letter (it's used in some languages to represent a glottal stop, such as in "Hawai'i").

Alternate-Glyph Selection

One interesting blurring of the line that can happen from time to time is the situation where a character with multiple glyphs needs to be drawn with a particular glyph in a certain situation, the glyph to use can't be algorithmically derived, and the particular choice of glyph needs to be preserved even in plain text. Unicode has taken different approaches to solving this problem in different situations. Much of the time, the alternate glyphs are simply given different

7. One popular misconception about Unicode is that Simplified and Traditional Chinese are unified. This isn't true; in fact, it's impossible, because the same Simplified Chinese character might be used as a stand-in for several different Traditional Chinese characters. Most of the time, Simplified and Traditional Chinese characters have different code point values in Unicode. Only small differences that could be reliably categorized as font-design differences, analogous to the difference between Arabic and Urdu, were unified. For more on Han unification, see Chapter 10.

108_GILLAM.ch03.fm Page 76 Monday, August 19, 2002 1:58 PM

code points. For example, five Hebrew letters have different shapes when they appear at the end of a word from the shapes they normally have. In foreign words, these letters keep their normal shapes even when they appear at the end of a word. Unicode gives different code point values to the regular and "final" versions of the letters. Examples like this can be found throughout Unicode.

Two special characters, U+200C ZERO WIDTH NON-JOINER (ZWNJ for short) and U+200D ZERO WIDTH JOINER (ZWJ for short), can be used as hints of which glyph shape is preferred in a particular situation. ZWNJ prevents formation of a cursive connection or ligature in situations where one would normally happen, and ZWJ produces a ligature or cursive connection where one would otherwise not occur. These two characters can be used to override the default choice of glyphs.

The Unicode Mongolian block takes yet another approach. Many characters in the Mongolian block have or cause special shaping behavior to happen, but sometimes the proper shape for a particular letter in a particular word can't be determined algorithmically (except with an especially sophisticated algorithm that recognizes certain words). The Mongolian block includes three "variation selectors," characters that have no appearance of their own, but change the shape of the character that precedes them in some well-defined way.

Beginning in Unicode 3.2, the variation-selector approach has been extended to all of Unicode. Unicode 3.2 introduces 16 general-purpose variation selectors, which work the same way as the Mongolian variation selectors: They have no visual presentation of their own, but act as "hints" to the rendering process that the preceding character should be drawn with a particular glyph shape. The list of allowable combinations of regular characters and variation selectors is given in a file called StandardizedVariants.html in the Unicode Character Database.

For more information on the joiner, non-joiner, and variation selectors, see Chapter 12.

■ Multiple Representations

Having discussed the importance of the principle of unification, we must also consider the opposite property—the fact that Unicode provides alternate representations for many characters. As we saw earlier, Unicode's designers placed

108_GILLAM.ch03.fm Page 77 Monday, August 19, 2002 1:58 PM

a high premium on respect for existing practice and interoperability with existing character encoding standards. In many cases, they sacrificed some measure of architectural purity in pursuit of the greater good (i.e., people actually using Unicode). As a result, Unicode includes code point assignments for a lot of characters that were included solely or primarily to allow for round-trip compatibility with some legacy standard, a broad category of characters known more or less informally as **compatibility characters**. Exactly which characters are compatibility characters is somewhat a matter of opinion, and there isn't necessarily anything special about the compatibility characters that flags them as such. An important subset of compatibility characters *are* called out as special because they have alternate, preferred representations in Unicode. Because the preferred representations usually consist of more than one Unicode code point, these characters are said to **decompose** into multiple code points.

There are two broad categories of decomposing characters: those with **canonical decompositions** (these characters are often referred to as "precomposed characters" or "canonical composites") and those with **compatibility decompositions** (the term "compatibility characters" is frequently used to refer specifically to these characters; a more specific term, "compatibility composite," is better). A canonical composite can be replaced with its canonical decomposition with no loss of data: the two representations are strictly equivalent, and the canonical decomposition is the character's preferred representation.[8]

Most canonical composites are combinations of a "base character" and one or more diacritical marks. For example, we talked about the character positioning rules and how the rendering engine needs to be smart enough so that when it sees, for example, an a followed by an umlaut, it draws the umlaut on top of the a: ä. Much of the time, normal users of these characters don't see them as the combination of a base letter and an accent mark. A German speaker sees ä

8. I have to qualify the word "preferred" here slightly. For characters whose decompositions consist of a single other character ("singleton decompositions"), this is true. For multiple-character decompositions, there's nothing that necessarily makes them "better" than the precomposed forms, and you can generally use either representation. Decomposed representations are somewhat easier to deal with in code, though, and many processes on Unicode text are based on mapping characters to their canonical decompositions, so they're "preferred" in that sense. We'll untangle this terminology in Chapter 4.

108_GILLAM.ch03.fm Page 78 Monday, August 19, 2002 1:58 PM

simply as "the letter ä" and not as "the letter a with an umlaut on top." A vast number of letter–mark combinations are consequently encoded using single character codes in the various source encodings, and these are very often more convenient to work with than the combinations of characters would be. The various European character encoding standards follow this pattern—for example, assigning character codes to letter–accent combinations such as é, ä, å, û, and so on—and Unicode follows suit.

Because a canonical composite can be mapped to its canonical decomposition without losing data, the original character and its decomposition are freely interchangeable. The Unicode standard enshrines this principle in law: On systems that support both the canonical composites and the combining characters that are included in their decompositions, the two different representations of the same character (composed and decomposed) are required to be treated as identical. That is, there is no difference between ä when represented by two code points and ä when represented with a single code point. In both cases, it's still the letter ä.

Most Unicode implementations must be smart enough to treat the two representations as equivalent. One way to do this is by **normalizing** a body of text to always prefer one of the representations. The Unicode standard actually provides four different normalized forms for Unicode text.

All of the canonical decompositions involve one or more **combining marks**, a special class of Unicode code points representing marks that combine graphically in some way with the character that precedes them. If a Unicode-compatible system sees the letter a followed by a combining umlaut, it draws the umlaut on top of the a. This approach can be a little more inconvenient than just using a single code point to represent the a-umlaut combination, but it does give you an easy way to represent a letter with *more than one* mark attached to it, such as you find in Vietnamese or some other languages: Just follow the base character with multiple combining marks.

Of course, this strategy means you can get into trouble with equivalence testing even without having composite characters. There are plenty of cases where the same character can be represented multiple ways by putting the various combining marks in different orders. Sometimes, the difference in ordering can be significant (if two combining marks attach to the base character in the same place, the one that comes first in the backing store is drawn closest to the

108_GILLAM.ch03.fm Page 79 Monday, August 19, 2002 1:58 PM

character and the others are moved out of the way). In many other cases, the ordering isn't significant—you get the same visual result whatever order the combining marks come in. The different forms are then all legal and required—once again—to be treated as identical. The Unicode standard provides for a **canonical ordering** of combining marks to aid in testing such sequences for equivalence.

The other class of decomposing characters is **compatibility composites**, characters with compatibility decompositions.[9] A character can't be mapped to its compatibility decomposition without losing data. For example, sometimes alternate glyphs for the same character are given their own character codes. In these cases, a preferred Unicode code point value will represent the character, independent of glyph, and other code point values will represent the different glyphs. The latter are called **presentation forms**. The presentation forms have mappings back to the regular character they represent, but they're not simply interchangeable; the presentation forms refer to specific glyphs, while the preferred character maps to whatever glyph is appropriate for the context. In this way, the presentation forms carry more information than the canonical forms. The most notable set of presentation forms are the Arabic presentation forms, where each standard glyph for each Arabic letter, plus a wide selection of ligatures, has its own Unicode character code. Although rendering engines often use presentation forms as an implementation detail, normal users of Unicode are discouraged from using them and are urged to use the nondecomposing characters instead. The same goes for the smaller set of presentation forms for other languages.

Another interesting class of compatibility composites represent stylistic variants of particular characters. They are similar to presentation forms, but instead of representing particular glyphs that are contextually selected, they represent particular glyphs that are normally specified through the use of additional styling information (remember, Unicode represents only plain text, not styled text). Examples include superscripted or subscripted numerals, or letters with special styles applied to them. For example, the Planck constant is

9. There are a few characters in Unicode whose canonical decompositions include characters with compatibility decompositions. The Unicode standard considers these characters to be *both* canonical composites *and* compatibility composites.

108_GILLAM.ch03.fm Page 80 Monday, August 19, 2002 1:58 PM

represented using an italicized letter *h*. Unicode includes a compatibility character code for the symbol for the Planck constant, but you could also just use a regular h in conjunction with some non-Unicode method of specifying that it's italicized. Characters with adornments such as surrounding circles fall into this category, as do the abbreviations sometimes used in Japanese typesetting that consist of several characters arranged in a square.

For compatibility composites, the Unicode standard not only specifies the characters to which they decompose, but also information intended to explain what nontext information is needed to express exactly the same thing.

Canonical and compatibility decompositions, combining characters, normalized forms, canonical accent ordering, and related topics are all dealt with in excruciating detail in Chapter 4.

Flavors of Unicode

Let's take a minute to go back over the character-encoding terms from Chapter 2:

- An **abstract character repertoire** is a collection of characters.
- A **coded character set** maps the characters in an abstract character repertoire to abstract numeric values or positions in a table. These abstract numeric values are called *code points.* (For a while, the Unicode 2.0 standard referred to code points as "Unicode scalar values.")
- A **character encoding form** maps code points to series of fixed-length bit patterns known as *code units.* (For a while, the Unicode 2.0 standard referred to code units as "code points.")
- A **character encoding scheme,** also called a *serialization format*, maps code units to bytes in a sequential order. (This may involve specifying a serialization order for code units that are more than one byte long, specifying a method of mapping code units from more than one encoding form into bytes, or both.)
- A **transfer encoding syntax** is an additional transformation that may be performed on a serialized sequence of bytes to optimize it for some situation (transforming a sequence of 8-bit byte values for transmission through a system that handles only 7-bit values, for example).

108_GILLAM.ch03.fm Page 81 Monday, August 19, 2002 1:58 PM

For most Western encoding standards, the transforms in the middle (i.e., from code points to code units and from code units to bytes) are so straightforward that they're never thought of as distinct steps. The standards in the ISO 8859 family, for example, define coded character sets. Because the code point values are a byte long already, the character encoding forms and character encoding schemes used with these coded character sets are basically null transforms: You use the normal binary representation of the code point values as code units, and you don't have to do anything to convert the code units to bytes. (ISO 2022 does define a character encoding scheme that lets you mix characters from different coded character sets in a single serialized data stream.)

The East Asian character standards make these transforms more explicit. JIS X 0208 and JIS X 0212 define coded character sets only; they just map each character to row and column numbers in a table. You then have a choice of character encoding schemes for converting the row and column numbers into serialized bytes: Shift-JIS, EUC-JP, and ISO 2022-JP are all examples of character encoding schemes used with the JIS coded character sets.

The Unicode standard makes each layer in this hierarchy explicit. It comprises the following:

1. An abstract character repertoire that includes characters for an extremely wide variety of writing systems.
2. A single coded character set that maps each character in the abstract repertoire to a 21-bit value. (The 21-bit value can also be thought of as a coordinate in a three-dimensional space: a 5-bit plane number, an 8-bit row number, and an 8-bit cell number.)
3. Three character encoding forms known as Unicode Transformation Formats (UTF):
   - UTF-32, which represents each 21-bit code point value as a single 32-bit code unit. UTF-32 is optimized for systems where 32-bit values are easier or faster to process and space isn't at a premium.
   - UTF-16, which represents each 21-bit code point value as a sequence of one or two 16-bit code units. The vast majority of characters are represented with single 16-bit code units, making it a good general-use compromise between UTF-32 and UTF-8. UTF-16, the oldest Unicode encoding form, is the form specified by the Java and

108_GILLAM.ch03.fm Page 82 Monday, August 19, 2002 1:58 PM

JavaScript programming languages and the XML Document Object Model APIs.

- UTF-8, which represents each 21-bit code point value as a sequence of one to four 8-bit code units. The ASCII characters have exactly the same representation in UTF-8 as they do in ASCII, and UTF-8 is optimized for byte-oriented systems or systems where backward compatibility with ASCII is important. For European languages, UTF-8 is also more compact than UTF-16; for Asian languages, UTF-16 is more compact than UTF-8. UTF-8 is the default encoding form for a wide variety of Internet standards.

4. Seven character encoding schemes. UTF-8 is a character encoding scheme unto itself because it uses 8-bit code units. UTF-16 and UTF-32 each have three associated encoding schemes:

   - A "big-endian" version that serializes each code unit most-significant-byte first.
   - A "little-endian" version that serializes each code unit least-significant-byte first.
   - A self-describing version that uses an extra sentinel value at the beginning of the stream, called the "byte order mark," to specify whether the code units are in big-endian or little-endian order.

In addition, some allied specifications aren't officially part of the Unicode standard:

- UTF-EBCDIC is a version of UTF-8 designed for use on EBCDIC-based systems that maps Unicode code points to series of from one to five 8-bit code units.
- CESU-8 is a modified version of UTF-8 designed for backward compatibility with some older Unicode implementations.
- UTF-7 is a mostly obsolete character encoding scheme for use with 7-bit Internet standards that maps UTF-16 code units to sequences of 7-bit values.

108_GILLAM.ch03.fm Page 83 Monday, August 19, 2002 1:58 PM

- Standard Compression Scheme for Unicode (SCSU) is a character encoding scheme that maps a sequence of UTF-16 code units to a compressed sequence of bytes, providing a serialized Unicode representation that is generally as compact for a given language as that language's legacy encoding standards and that optimizes Unicode text for further compression with byte-oriented compression schemes such as LZW.
- Byte-Order Preserving Compression for Unicode (BOCU) is another compression format for Unicode.

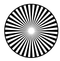
We'll delve into the details of these encoding forms and schemes in Chapter 6.

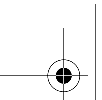
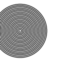
Character Semantics

Because Unicode aims to encode semantics rather than appearances, simple code charts aren't sufficient. After all, they merely show pictures of characters in a grid that maps them to numeric values. The pictures of the characters can certainly help illustrate the semantics of the characters, but they can't tell the whole story. The Unicode standard goes well beyond just pictures of the characters, providing a wealth of information on every character.

Every code chart in the standard is followed by a list of the characters in the code chart. For each character, an entry gives the following information:

- Its Unicode code point value.
- A representative glyph. For characters that combine with other characters, such as accent marks, the representative glyph includes a dotted circle that shows where the main character would go—making it possible to distinguish COMBINING DOT ABOVE from COMBINING DOT BELOW, for example. For characters that have no visual appearance, such as spaces and control and formatting codes, the representative glyph is a dotted square with some sort of abbreviation of the character name inside.
- The character's name. The name, and not the representative glyph, is the normative property (the parts of the standard that are declared to be



108_GILLAM.ch03.fm Page 84 Monday, August 19, 2002 1:58 PM

"normative" are the parts you have to follow exactly to conform to the standard; parts declared "informative" are there to supplement or clarify the normative parts and don't have to be followed exactly to conform). This reflects the philosophy that Unicode encodes semantics, although sometimes the actual meaning of the character has drifted since the earliest drafts of the standard and no longer matches the name. Such cases are very rare, however.

In addition to the code point value, name, and representative glyph, an entry may include the following:

- Alternate names for the character
- Cross-references to similar characters elsewhere in the standard (which helps to distinguish them from each other)
- The character's canonical or compatibility decomposition (if it's a composite character)
- Additional notes on its usage or meaning (for example, the entries for many letters include the languages that use them)

The Unicode standard also includes chapters on each major group of characters in the standard, with information that's common to all of the characters in the group (such as encoding philosophy or information on special processing challenges) and additional narrative explaining the meaning and usage of any characters in the group that have special properties or behavior that needs to be called out.

The Unicode standard actually consists of more than just *The Unicode Standard Version 3.0*. That is, there's more to the Unicode standard than just the book. The standard includes a comprehensive database of all the characters, a copy of which is included on the CD that's included with the book. Because the character database changes more frequently than the rest of the standard, it's usually a good idea to get the most recent version of the database from the Unicode Consortium's Web site at `http://www.unicode.org`.

Every character in Unicode is associated with a list of properties that define how the character is to be treated by various processes. The Unicode

108_GILLAM.ch03.fm Page 85 Monday, August 19, 2002 1:58 PM

Character Database comprises a group of text files that give the properties for each character in Unicode. Among the properties that each character has are the following:

- The character's code point value and name.
- The character's general category. All of the characters in Unicode are grouped into 30 categories. The category tells you things like whether the character is a letter, numeral, symbol, whitespace character, control code, and so forth.
- The character's decomposition, along with whether it's a canonical or compatibility decomposition, and for compatibility composites, a tag that attempts to indicate what data are lost when you convert to the decomposed form.
- The character's case mapping. If the character is a cased letter, the database includes the mapping from the character to its counterpart in the opposite case.
- For characters that are considered numerals, the character's numeric value (that is, the numeric value the character represents, not the character's code point value).
- The character's directionality (e.g., whether it is left-to-right, is right-to-left, or takes on the directionality of the surrounding text). The Unicode Bidirectional Layout Algorithm uses this property to determine how to arrange characters of different directionalities on a single line of text.
- The character's mirroring property. It says whether the character takes on a mirror-image glyph shape when surrounded by right-to-left text.
- The character's combining class. It is used to derive the canonical representation of a character with more than one combining mark attached to it (it's used to derive the canonical ordering of combining characters that don't interact with each other).
- The character's line-break properties. This information is used by text rendering processes to help figure out where line divisions should go.

For an in-depth look at the various files in the Unicode Character Database, see Chapter 5.

108_GILLAM.ch03.fm Page 86 Monday, August 19, 2002 1:58 PM

Unicode Versions and Unicode Technical Reports

The Unicode standard includes a group of supplementary documents known as Unicode Technical Reports. A snapshot of these is also included on the CD that comes with the book *The Unicode Standard*, *Version 3.0,* but the CD accompanying the Unicode 3.0 standard is now so out of date as to be nearly useless. You can always find the most up-to-date slate of technical reports on the Unicode Web site (`http://www.unicode.org`).

There are three kinds of technical reports:

- **Unicode Standard Annexes (UAX)** are actual addenda and amendments to the Unicode standard.
- **Unicode Technical Standards (UTS)** are adjunct standards related to Unicode. They're not normative parts of the standard itself, but carry their own conformance criteria.
- **Unicode Technical Reports (UTR)** include various types of adjunct information, such as text clarifying or expanding on parts of the standard, implementation guidelines, and descriptions of procedures for doing things with Unicode text that don't rise to the level of official Unicode Technical Standards.

Other types of reports exist as well:

- **Draft Unicode Technical Reports (DUTR).** Technical reports are often published while they're still in draft form. DUTR status indicates that an agreement has been reached in principle to adopt the proposal, but details must still be worked out. Draft technical reports don't have any normative force until the Unicode Technical Committee votes to remove "Draft" from their name, but they're published early to solicit comments and give implementers a head start.
- **Proposed Draft Unicode Technical Reports (PDUTR).** These technical reports are published before an agreement in principle has been reached to adopt them.

The status of the technical reports is constantly changing. Here's a summary of the slate of technical reports as of this writing (January 2002).

108_GILLAM.ch03.fm Page 87 Monday, August 19, 2002 1:58 PM

Unicode Standard Annexes

All of the following Unicode Standard Annexes are officially part of the Unicode 3.2 standard:

- **UAX #9: The Unicode Bidirectional Algorithm.** This report specifies the algorithm for laying out lines of text that mix left-to-right characters with right-to-left characters. It supersedes the description of the bidirectional layout algorithm in the Unicode 3.0 book. For an overview of the bidirectional layout algorithm, see Chapter 8. For implementation details, see Chapter 16.
- **UAX #11: East Asian Width.** It specifies a set of character properties that determine how many display cells a character takes up when used in the context of East Asian typography. For more information, see Chapter 10.
- **UAX #13: Unicode Newline Guidelines.** There are some interesting issues regarding how you represent the end of a line or paragraph in Unicode text, and this document clarifies them. This information is covered in Chapter 12.
- **UAX #14: Line Breaking Properties.** This document specifies how a word-wrapping routine should treat the various Unicode characters. Word-wrapping is covered in Chapter 16.
- **UAX #15: Unicode Normalization Forms.** Because Unicode has multiple ways of representing a lot of characters, it's often helpful to convert Unicode text to some kind of normalized form that prefers one representation for any given character over all the others. There are four Unicode Normalization Forms, described in this document. The Unicode Normalization Forms are covered in Chapter 4.
- **UAX #19: UTF-32.** It specifies the UTF-32 encoding form. UTF-32 is covered in Chapter 6.
- **UAX #27: Unicode 3.1.** This official definition includes the changes and additions to Unicode 3.0 that form Unicode 3.1. This document officially incorporates the other Unicode Standard Annexes into the standard and gives them a version number. It also includes code charts and character lists for all new characters added to the standard in Unicode 3.1.

108_GILLAM.ch03.fm Page 88 Monday, August 19, 2002 1:58 PM

Unicode Technical Standards

- **UTS #6: A Standard Compression Scheme for Unicode.** It defines SCSU, a character encoding scheme for Unicode that results in serialized Unicode text of comparable size to the same text in legacy encodings. For more information, see Chapter 6.
- **UTS #10: The Unicode Collation Algorithm.** It specifies a method of comparing character strings in Unicode in a language-sensitive manner and a default ordering of all characters to be used in the absence of a language-specific ordering. For more information, see Chapter 15.

Unicode Technical Reports

- **UTR #16: UTF-EBCDIC.** It specifies a special 8-bit transformation of Unicode for use on EBCDIC-based systems. This topic is covered in Chapter 6.
- **UTR #17: Character Encoding Model.** It defines a set of useful terms for discussing the various aspects of character encodings. This material was covered in Chapter 2 and reiterated in this chapter.
- **UTR #18: Regular Expression Guidelines.** It provides some guidelines for how a regular-expression facility should behave when operating on Unicode text. This material is covered in Chapter 15.
- **UTR #20: Unicode in XML and Other Markup Languages.** It specifies some guidelines for how Unicode should be used in the context of a markup language such as HTML or XML. This topic is covered in Chapter 17.
- **UTR #21: Case Mappings.** It gives more detail than the standard itself on the process of mapping uppercase to lowercase, and vice versa. Case mapping is covered in Chapter 14.
- **UTR #22: Character Mapping Tables.** It specifies an XML-based file format for describing a mapping between Unicode and some other encoding standard. Mapping between Unicode and other encodings is covered in Chapter 14.
- **UTR #24: Script Names.** It defines an informative character property: the script name, which identifies the writing system (or "script") to which each character belongs. The script name property is discussed in Chapter 5.

108_GILLAM.ch03.fm Page 89 Monday, August 19, 2002 1:58 PM

Draft and Proposed Draft Technical Reports

- **PDUTR #25: Unicode Support for Mathematics.** It gives a detailed account of the various considerations involved in using Unicode to represent mathematical expressions, including issues such as spacing and layout, font design, and interpretation of Unicode characters in the context of mathematical expressions. This technical report also includes a heuristic for detecting mathematical expressions in plain Unicode text, and proposes a scheme for representing structured mathematical expressions in plain text. We'll look at math symbols in Chapter 12.
- **DUTR #26: Compatibility Encoding Scheme for UTF-16: 8-bit (CESU-8).** It documents a UTF-8-like Unicode encoding scheme that's being used in some existing systems. We'll look at CESU-8 in Chapter 6.
- **PDUTR #28: Unicode 3.2.** Together with UAX #27, this official definition gives all the changes and additions to Unicode 3.0 that define Unicode 3.2, including revised and updated code charts with the new Unicode 3.2 characters. Unicode 3.2 is scheduled to be released in March 2002, so this technical report will likely be UAX #28 by the time you read these words.

Superseded Technical Reports

The following technical reports have been superseded by (or absorbed into) more recent versions of the standard:

- **UTR #1: Myanmar, Khmer, and Ethiopic.** Absorbed into Unicode 3.0.
- **UTR #2: Sinhala, Mongolian, and Tibetan.** Tibetan was in Unicode 2.0; Sinhala and Mongolian were added in Unicode 3.0.
- **UTR #3: Various Less-Common Scripts.** This document includes exploratory proposals for historical or rare writing systems. It has been superseded by more recent proposals. Some of the scripts in this proposal have been incorporated into more recent versions of Unicode: Cherokee (Unicode 3.0), Old Italic (Etruscan; Unicode 3.1), Thaana

108_GILLAM.ch03.fm Page 90 Monday, August 19, 2002 1:58 PM

(Maldivian; 3.0), Ogham (3.0), Runic (3.0), Syriac (3.0), Tagalog (3.2), Buhid (3.2), and Tagbanwa (3.2). Most of the others are in various stages of discussion, with another batch scheduled for inclusion in Unicode 4.0. This document is mostly interesting as a list of writing systems that will probably be in future versions of Unicode. For more information on this subject, check out `http://www.unicode.org/unicode/alloc/Pipeline.html`.

- **UTR #4: Unicode 1.1.** Superseded by later versions.
- **UTR #5: Handling Non-spacing Marks.** Incorporated into Unicode 2.0.
- **UTR #7: Plane 14 Characters for Language Tags.** Incorporated into Unicode 3.1.
- **UTR #8: Unicode 2.1.** Superseded by later versions.

The missing numbers belong to technical reports that have been withdrawn by their proposers, have been turned down by the Unicode Technical Committee, or haven't been published yet.

Unicode Versions

Many of the technical reports either define certain versions of Unicode or are superseded by certain versions of Unicode. Each version of Unicode comprises a particular set of characters, a particular version of the character-property files, and a certain set of rules for dealing with them. All of these things change over time (although certain things are guaranteed to remain the same—see "Unicode Stability Policies" later in this chapter).

This point brings us to the question of Unicode version numbers. A Unicode version number consists of three parts—for example, "Unicode 2.1.8." The first number is the **major version number**. It gets bumped every time a new edition of the book is released. That happens when the accumulation of technical reports becomes too unwieldy or when a great many significant changes (such as lots of new characters) are incorporated into the standard at once. A new major version of Unicode appears every several years.

The **minor version number** gets bumped whenever new characters are added to the standard or other significant changes are made. New minor versions of Unicode don't get published as books, but do get published as Unicode

108_GILLAM.ch03.fm Page 91 Monday, August 19, 2002 1:58 PM

Standard Annexes. There has been one minor version of Unicode between each pair of major versions (i.e., there was a Unicode 1.1 [published as UTR #4], a Unicode 2.1 [UTR #8], and a Unicode 3.1 [UAX #27]), but this pattern was broken with the release of Unicode 3.2 (PDUTR #28 at the time of this writing, but most likely UAX #28 by the time you read this).

The **update version number** gets bumped when changes are made to the Unicode Character Database. Updates are published as new versions of the database; there is no corresponding technical report.

The current version of Unicode at the time of this writing (January 2002) is Unicode 3.1.1. Unicode 3.2, which includes a whole slate of new characters, is currently in beta and will likely be the current version by the time this book is published.

One has to be a bit careful when referring to a particular version of the Unicode standard from another document, particularly another standard. Unicode changes constantly, so it's seldom a good idea to nail yourself to one specific version. It's generally best either to specify only a major version number (or, in some cases, just major and minor version numbers) or to specify an open-ended range of versions (e.g., "Unicode 2.1 or later"). Generally, this approach is okay, as future versions of Unicode will only add characters—because you're never required to support a particular character, you can pin yourself to an open-ended range of Unicode versions and not sweat the new characters. Sometimes, however, changes are made to a character's properties in the Unicode Character Database that could alter program behavior. Usually, this is a good thing—changes are made to the database when the database is deemed to have been *wrong* before—but your software may need to deal with this modification in some way.

Unicode Stability Policies

Unicode will, of course, continue to evolve. Nevertheless, you can count on some things to remain stable:

- Characters that are in the current standard will never be removed from future standards. They may, in unusual circumstances, be deprecated (i.e., their use might be discouraged), but they'll never be eliminated and their code point values will never be reused to refer to different characters.

108_GILLAM.ch03.fm Page 92 Monday, August 19, 2002 1:58 PM

- Characters will never be reassigned from one code point to another. If a character has ambiguous semantics, a new character may be introduced with more specific semantics, but the old one will never be taken away and will continue to have ambiguous semantics.
- Character names will never change. Occasionally a character with ambiguous semantics will get out of sync with its name as its semantics evolve, but this is very rare.
- Text in one of the Unicode Normalized Forms will always be in that normalized form. That is, the definition of the Unicode Normalized Forms will not change between versions of the standard in ways that would cause text that is normalized according to one version of the standard not to be normalized in later versions of the standard.
- A character's combining class and canonical and compatibility decompositions will never change, as that would break the normalization guarantee.
- A character's properties may change, but not in a way that would alter the character's fundamental identity. In other words, the representative glyph for "A" won't change to "B", and the category for "A" won't change to "lowercase letter." You'll see property changes only to correct clear mistakes in previous versions.
- Various structural aspects of the Unicode character properties will remain the same, as implementations depend on some of these things. For example, the standard won't add any new general categories or any new bi-di categories, it will keep characters' combining classes in the range from 0 to 255, noncombining characters will always have a combining class of 0, and so on.

You can generally count on characters' other normative properties not changing, although the Unicode Consortium certainly reserves the right to fix mistakes in these properties.

The Unicode Consortium can change a character's informative properties more or less at will, without changing version numbers, because you don't have to follow them anyway. Again, this type of change shouldn't happen much, except when consortium members need to correct a mistake of some kind.

These stability guarantees are borne out of bitter experience. Characters *did* get removed and reassigned and characters' names *did* change in Unicode 1.1 as

108_GILLAM.ch03.fm Page 93 Monday, August 19, 2002 1:58 PM

a result of the merger with ISO 10646, and it caused serious grief. This problem won't happen again.

Arrangement of the Encoding Space

Unicode's designers tried to assign the characters to numeric values in an orderly manner that would make it easy to tell something about a character just from its code point value. As the encoding space has filled up, this has become more difficult to do, but the logic still comes through reasonably well.

Unicode was originally designed for a 16-bit encoding space, consisting of 256 *rows* of 256 characters each. ISO 10646 was designed for a 32-bit encoding space, consisting of 128 *groups* of 256 *planes* containing 256 rows of 256 characters. Thus the original Unicode encoding space had room for 65,536 characters, and ISO 10646 had room for an unbelievable 2,147,483,648 characters. The ISO encoding space is clearly overkill (experts estimate that perhaps 1 million or so characters are eligible for encoding), but it was clear by the time Unicode 2.0 came out that the 16-bit Unicode encoding space was too small.

The solution was the **surrogate mechanism**, a scheme whereby special escape sequences known as **surrogate pairs** could be used to represent characters outside the original encoding space. It extended the number of characters that could be encoded to 1,114,112, leaving ample space for the foreseeable future (only 95,156 characters are actually encoded in Unicode 3.2, and Unicode has been in development for 12 years). The surrogate mechanism was introduced in Unicode 2.0 and has since become known as UTF-16. It effectively encodes the first 17 planes of the ISO 10646 encoding space. The Unicode Consortium and WG2 have agreed never to populate the planes above plane 16, so for all intents and purposes, Unicode and ISO 10646 now share a 21-bit encoding space consisting of 17 planes of 256 rows of 256 characters. Valid Unicode code point values run from U+0000 to U+10FFFF.

Organization of the Planes

Figure 3.1 shows the Unicode encoding space.

108_GILLAM.ch03.fm Page 94 Monday, August 19, 2002 1:58 PM

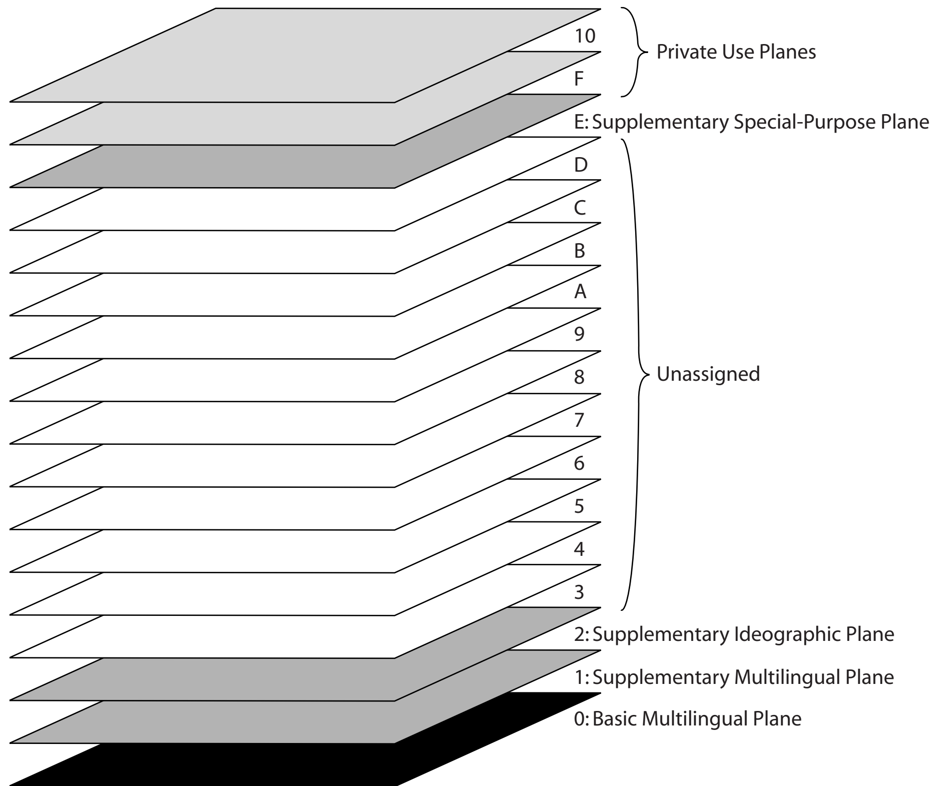


Figure 3.1 The Unicode Encoding Space

Plane 0 is the **Basic Multilingual Plane (BMP)**. It contains the majority of the encoded characters, including all of the most common ones. In fact, prior to Unicode 3.1, no characters were encoded in any of the other planes. The characters in the BMP can be represented in UTF-16 with a single 16-bit code unit.

Plane 1 is the **Supplementary Multilingual Plane (SMP)**. It is intended to contain characters from archaic or obsolete writing systems. Why encode them at all? They are here mostly for the use of the scholarly community in papers where they write about these characters. Various specialized collections of symbols will also go into this plane.

Plane 2 is the **Supplementary Ideographic Plane (SIP)**. This extension of the CJK Ideographs Area from the BMP contains rare and unusual Chinese characters.

Plane 14 (E) is the **Supplementary Special-Purpose Plane (SSP)**. It's reserved for special-purpose characters—generally code points that don't encode characters as such but are instead used by higher-level protocols or as signals to processes operating on Unicode text.

Planes 15 and 16 (F and 10) are the **Private Use Planes**, an extension of the Private Use Area in the BMP. The other planes are currently unassigned, and will probably remain that way until Planes 1, 2, and 14 start to fill up.

The Basic Multilingual Plane

The heart and soul of Unicode is plane 0, the BMP. It contains the vast majority of characters in common use today, and those that aren't yet encoded will go here as well. Figure 3.2 shows the allocation of space in the BMP.

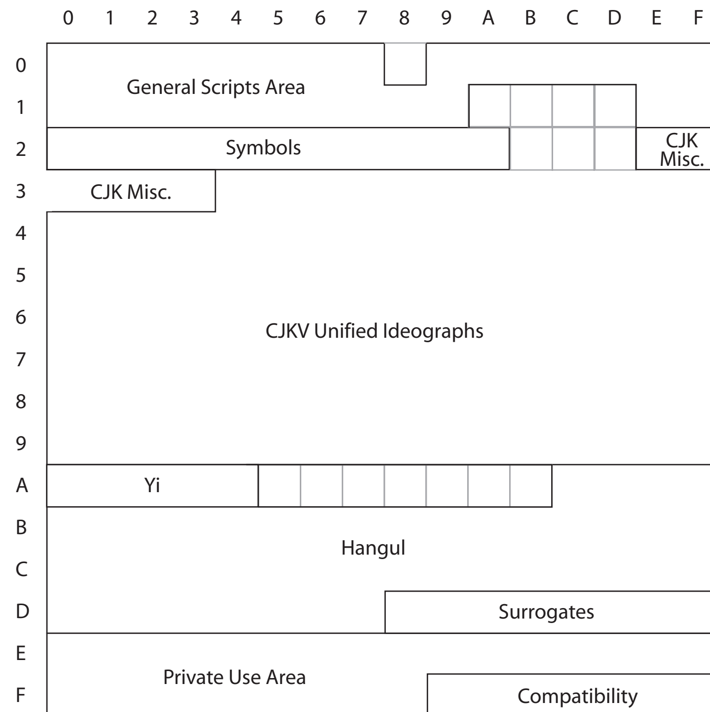


Figure 3.2 The Basic Multilingual Plane

108_GILLAM.ch03.fm Page 96 Monday, August 19, 2002 1:58 PM

The characters whose code point values begin with 0 and 1 form the **General Scripts Area**. This area contains the characters from all of the alphabetic writing systems, including the Latin, Greek, Cyrillic, Hebrew, Arabic, Devanagari (Hindi), and Thai alphabets, among many others. It also contains a collection of combining marks that are often used in conjunction with the letters in this area. Figure 3.3 shows how the General Scripts Area is allocated.

There are a few important things to note about the General Scripts Area. First, the first 128 characters (those from U+0000 to U+007F) are exactly the same as the ASCII characters with the same code point values. Thus you can convert from ASCII to Unicode simply by zero-padding the characters out to 16 bits (in fact, in UTF-8, the 8-bit version of Unicode, the ASCII characters have exactly the same representation as they do in ASCII).

	00	20	40	60	80	A0	C0	E0	00	20	40	60	80	A0	C0	E0
00/01	ASCII				Latin-1				Lat. Ext. A				Lat. Ext. B			
02/03	Lat. Ext. B		IPA			Mod. Ltrs.			Comb. Diac.			Greek				
04/05	Cyrillic									Armenian			Hebrew			
06/07	Arabic								Syriac				Thaana			
08/09									Devanagari				Bengali			
0A/0B	Gurmukhi				Gujarati				Oriya				Tamil			
0C/0D	Telugu				Kannada				Malayalam				Sinhala			
0E/0F	Thai				Lao				Tibetan							
10/11	Myanmar					Georgian			Hangul Jamo							
12/13	Ethiopic													Cherokee		
14/15	Canadian Aboriginal Syllablics															
16/17					Ogham	Runic			Phillipine Scripts				Khmer			
18/19	Mongolian															
1A/1B																
1C/1D																
1E/1F	Latin Extended Additional								Greek Extended							

Figure 3.3 The General Scripts Area

108_GILLAM.ch03.fm Page 97 Monday, August 19, 2002 1:58 PM

Second, the first 256 characters (those from U+0000 to U+00FF) are exactly the same as the characters with the same code point values from the ISO 8859-1 (ISO Latin-1) standard. (Latin-1 is a superset of ASCII; its lower 128 characters are identical to ASCII.) You can convert Latin-1 to Unicode by zero-padding out to 16 bits. (Note, however, that the non-ASCII Latin-1 characters have two-byte representations in UTF-8.)

For those writing systems that have only one dominant existing encoding, such as most of the Indian and Southeast Asian ones, Unicode keeps the same relative arrangement of the characters as their original encoding had. Conversion back and forth can be accomplished by adding or subtracting a constant.

We'll be taking an in-depth look at all of these scripts in Part II. The Latin, Greek, Cyrillic, Armenian, and Georgian blocks, as well as the Combining Diacritical Marks, IPA Extensions, and Spacing Modifier Letters blocks, are covered in Chapter 7. The Hebrew, Arabic, Syriac, and Thaana blocks are covered in Chapter 8. The Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Sinhala, Thai, Lao, Tibetan, Myanmar, Khmer, and Philippine blocks are covered in Chapter 9. The Hangul Jamo block is covered in Chapter 10. The Ethiopic, Cherokee, Canadian Aboriginal Syllables, Ogham, Runic, and Mongolian blocks are covered in Chapter 11.

The characters whose code point values begin with 2 (with a few recent exceptions) form the **Symbols Area.** This area includes all kinds of stuff, such as a collection of punctuation that can be used with many different languages (this block actually supplements the punctuation marks in the ASCII and Latin-1 blocks), collections of math, currency, technical, and miscellaneous symbols, arrows, box-drawing characters, and so forth. Figure 3.4 shows the Symbols area, and Chapter 12 covers the various blocks.

The characters whose code point values begin with 3 (with Unicode 3.0, this group has now slopped over to include some code point values beginning with 2) form the **CJK Miscellaneous Area**. It includes all of the characters used in the East Asian writing systems, except for the three very large areas immediately following. For example, it includes punctuation used in East Asian writing, the phonetic systems used for Japanese and Chinese, various symbols and abbreviations used in Japanese technical material, and a collection of "radicals," component parts of Han ideographic characters. These blocks are covered in Chapter 10 and shown in Figure 3.4.

108_GILLAM.ch03.fm Page 98 Monday, August 19, 2002 1:58 PM

Figure 3.4 The Symbols and CJK Miscellaneous Areas

The characters whose code point values begin with 4, 5, 6, 7, 8, and 9 (in Unicode 3.0, this area has slopped over to include most of the characters whose code point values begin with 3) constitute the **CJKV Unified Ideographs Area**. This is where the Han ideographs used in Chinese, Japanese, Korean, and (much less frequently) Vietnamese are located.

The characters whose code point values range from U+A000 to U+A4CF form the **Yi Area**. It contains the characters used for writing Yi, a minority Chinese language.

The characters whose code point values range from U+AC00 to U+D7FF form the **Hangul Syllables Area**. Hangul is the alphabetic writing system used (sometimes in conjunction with Han ideographs) to write Korean. Hangul can be represented using the individual letters, or jamo, which are encoded in the General Scripts Area. The jamo are usually arranged into ideograph-like blocks representing whole syllables, and most Koreans look at whole syllables as sin-

108_GILLAM.ch03.fm Page 99 Monday, August 19, 2002 1:58 PM

gle characters. This area encodes all possible modern Hangul syllables using a single code point for each syllable.

We look at the CJKV Unified Ideographs, Yi, and Hangul Syllables Areas in Chapter 10.

The code point values from U+D800 to U+DFFF constitute the **Surrogates Area**. This range of code point values is reserved and will never be used to encode characters. Instead, values from this range are used in pairs as code-unit values by the UTF-16 encoding to represent characters from planes 1 through 16.

The code point values from U+E000 to U+F8FF form the **Private Use Area (PUA)**. This area is reserved for the private use of applications and systems that use Unicode, which may assign any meaning they wish to the code point values in this range. Private-use characters should be used only within closed systems that can apply a consistent meaning to these code points; text that is supposed to be exchanged between systems is prohibited from using these code point values (unless the sending and receiving parties have a private agreement stating otherwise), as there's no guarantee that a receiving process would know what meaning to apply to them.

The remaining characters with code point values beginning with F form the **Compatibility Area**. This catch-all area contains characters that are included in Unicode simply to maintain backward compatibility with the source encodings. It includes various ideographs that would be unified with ideographs in the CJK Unicode Ideographs except that the source encodings draw a distinction, presentation forms for various writing systems, especially Arabic, and half-width and full-width variants of various Latin and Japanese characters, among other things. This section isn't the only area of the encoding space containing compatibility characters; the Symbols Area includes many blocks of compatibility characters, and some others are scattered throughout the rest of the encoding space. This area also contains a number of special-purpose characters and noncharacter code points. Figure 3.5 shows the Compatibility Area.

The Supplementary Planes

Planes 1 through 16 are collectively known as the Supplementary Planes. They include rarer or more specialized characters.

Figure 3.6 depicts plane 1. The area marked "Letters" includes a number of obsolete writing systems and will expand to include more. The area marked

108_GILLAM.ch03.fm Page 100 Monday, August 19, 2002 1:58 PM

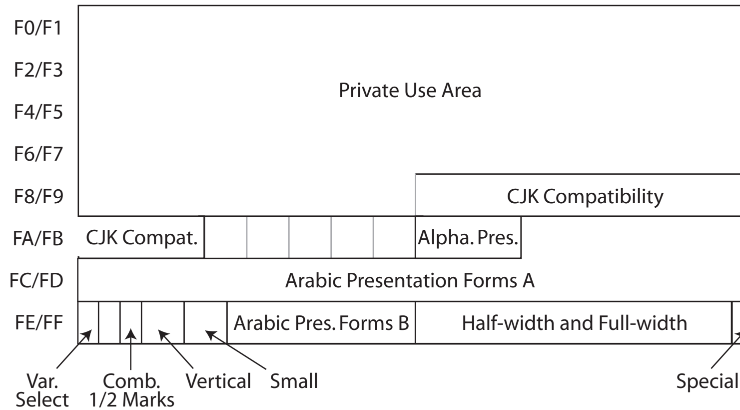


Figure 3.5 The Compatibility Area

"Music" includes a large collection of musical symbols, and the area marked "Math" includes a special set of alphanumeric characters intended to be used as symbols in mathematical formulas.

Figure 3.7 depicts plane 2. It's given over entirely to Chinese ideographic characters, acting as an extension of the CJKV Unified Ideographs Area in the BMP.

Figure 3.8 shows plane 14. It currently contains only a small collection of "tag" characters intended to be used for things like language tagging.

Although few unassigned code point values are left in the BMP, there are thousands and thousands in the other planes. Except for the Private Use Areas, Unicode implementations are not permitted to use the unassigned code point values for anything. All of them are reserved for future expansion, and they may be assigned to characters in future versions of Unicode. Conforming Unicode implementations can't use these values for any purpose or emit text purporting to be Unicode that uses them. This restriction also applies to the planes above plane 16, even though they may never be used to encode characters. It's also illegal to use the unused bits in a UTF-32 code unit to store other data.

Noncharacter Code Point Values

The code point values U+FFFE and U+FFFF, plus the corresponding code point values from all the other planes, are also illegal. They're not to be used in Unicode text at all. U+FFFE can be used in conjunction with the Unicode byte-

108_GILLAM.ch03.fm Page 101 Monday, August 19, 2002 1:58 PM

Figure 3.6 Plane 1: The Supplementary Multilingual Plane

order mark (U+FEFF) to detect byte-ordering problems (for example, if a Unicode text file produced on a Wintel PC starts with the byte-order mark, a Macintosh program reading it will read the byte-order mark as the illegal value U+FFFE and know that it has to byte-swap the file to read it properly).

U+FFFF is illegal for two main reasons. First, it provided a non-Unicode value that can be used as a sentinel value by Unicode-conformant processes. For example, the `getc()` function in C has to have a return type of `int` even though it generally returns only character values, which fit into a `char`. Because all `char` values are legal character codes, no values that are available to serve as the end-of-file signal. The `int` value `-1` is the end-of-file signal—you can't use the `char` value `-1` as end-of-file because it's the same as `0xFF`, which is a legal character. The Unicode version of `getc()`, on the other hand, could return `unsigned short` (or `wchar_t` on many systems) and still have a noncharacter value of that type—U+FFFF—available to use as the end-of-file signal.

108_GILLAM.ch03.fm Page 102 Monday, August 19, 2002 1:58 PM

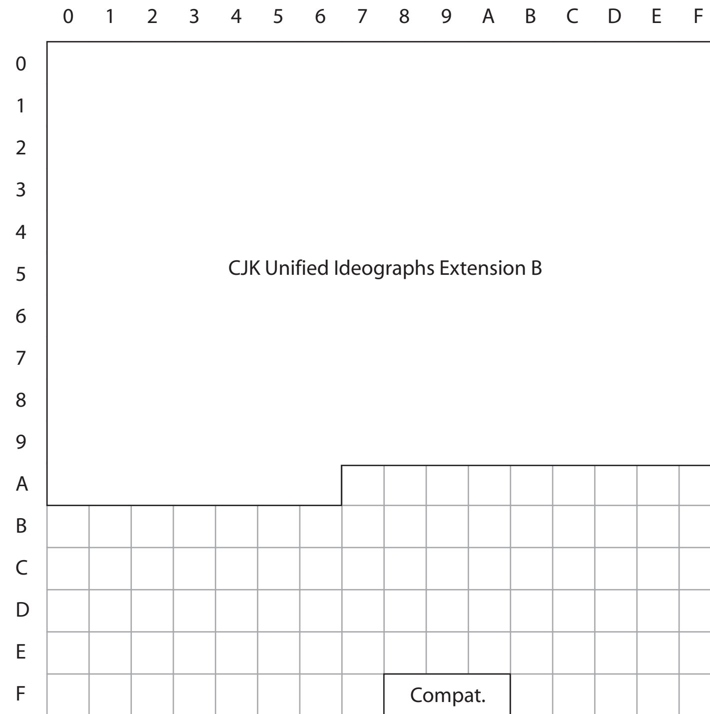


Figure 3.7 Plane 2: The Supplementary Ideographic Plane

Second, U+FFFF isn't a legal Unicode code point value for the reason given in the following example: Say you want to iterate over all of the Unicode code point values. You write the following (in C):

```
unsigned short c;
for (c = 0; c <= 0xFFFF; ++c) {
    // etc...
```

The loop will never terminate, because the next value after `0xFFFF` is 0. Designating U+FFFF as a non-Unicode value enables you to write loops that iterate over the entire Unicode range in a straightforward manner without having to resort to a larger type (and a lot of casting) for the loop variable or other funny business to make sure the loop terminates.

108_GILLAM.ch03.fm Page 103 Monday, August 19, 2002 1:58 PM

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Tags															
1																
2																
3																
4																
5																
6																
7																
8																
9																
A																
B																
C																
D																
E																
F																

Figure 3.8 Plane 14: The Supplementary Special-Purpose Plane

The corresponding code points in the other planes were reserved for the same reasons, although this is mostly a historical curiosity now. In the original design of ISO 10646, each plane was expected to function as a more or less independent encoding space. If you dealt with characters from only one plane, you might have had to represent them with 16-bit units (effectively chopping off the plane and group numbers) and encountered the same problem as described above.

Unicode 3.1 sets aside 32 additional code point values, U+FDD0 to U+FDEF, as noncharacter code points. This change makes these values available to implementations for their internal use as markers or sentinel values without the implementations having to worry about their being assigned to characters in the future. These values are *not* private-use code points and therefore aren't supposed to be used to represent characters. Like the other noncharacter code points, they're never legal in serialized Unicode text.

108_GILLAM.ch03.fm Page 104 Monday, August 19, 2002 1:58 PM

Conforming to the Standard

What does it mean to say that you conform to the Unicode standard? The answer to this question varies depending on what your product does. The answer tends to be both more and less than what most people think.

First, conforming to the Unicode standard does *not* mean that you have to be able to properly support every single character that the Unicode standard defines. The Unicode standard simply requires that you declare which characters you do support. For the characters you claim to support, *then* you have to follow all the rules in the standard. In other words, if you declare your program to be Unicode conformant (and you're doing that if you use the word "Unicode" anywhere in your advertising or documentation) and say "Superduperword supports Arabic," then you must support Arabic the way the Unicode standard says you should. In particular, you've got to be able to automatically select the right glyphs for the various Arabic letters depending on their context, and you've got to support the Unicode bidirectional text layout algorithm. If you don't do these things, then as far as the Unicode standard is concerned, you don't support Arabic.

Following are the rules for conforming to the Unicode standard. They differ somewhat from the rules as set forth in Chapter 3 of the actual Unicode standard, but they produce the same end result. There are certain algorithms that you have to follow (or mimic) in certain cases to be conformant. I haven't included those here, but will go over them in future chapters. There are also some terms used here that haven't been defined yet; all will be defined in future chapters.

General

For most processes, it's not enough to say you support Unicode. By itself, this statement doesn't mean very much. You'll also need to say:

- **Which version of Unicode you're supporting.** Generally, this declaration is just a shorthand way of saying which characters you support. In cases where the Unicode versions differ in the semantics they give to characters, or in their algorithms to do different things, you're specifying

108_GILLAM.ch03.fm Page 105 Monday, August 19, 2002 1:58 PM

which versions of those things you're using as well. Typically, if you support a given Unicode version, you also support all previous versions.[10]

Informative character semantics can and do change from version to version. You're not required to conform to the informative parts of the standard, but saying which version you support is also a way of saying which set of informative properties you're using.

It's legal and, in fact, often a good idea to say something like "Unicode 2.1.8 and later" when specifying which version of Unicode you use. This is particularly true when you're writing a standard that uses Unicode as one of its base standards. New versions of the standard (or conforming implementations) can then support new characters without going out of compliance. It's rarely necessary to specify which version of Unicode you're using all the way out to the last version number; rather, you can just indicate the major revision number ("This product supports Unicode 2.x").

- **Which transformation formats you support.** This information is relevant only if you exchange Unicode text with the outside world (including writing it to disk or sending it over a network connection). If you do, you must specify which of the various character encoding schemes defined by Unicode (the Unicode Transformation Formats) you support. If you support several, you need to specify your default (i.e., which formats you can read without being told by the user or some other outside source what format the incoming file is in). The Unicode Transformation Formats are discussed in Chapter 6.
- **Which normalization forms you support or expect.** Again, this point is important if you're exchanging Unicode text with the outside world. It can be thought of as a shorthand way of specifying which characters you support, but is specifically oriented toward telling people what characters can be in an incoming file. The normalization forms are discussed in Chapter 4.

10. Technically, this is guaranteed only as far back as Unicode 2.0— some radical changes, including removal and movement of characters, occurred between some of the earlier versions as Unicode and ISO 10646 were brought into sync with each other.

108_GILLAM.ch03.fm Page 106 Monday, August 19, 2002 1:58 PM

- **Which characters you support.** The Unicode standard doesn't require you to support any particular set of characters, so you need to say which sets of characters you know how to handle properly (of course, if you're relying on an external library, such as the operating system, for part or all of your Unicode support, you support whatever characters it supports). The ISO 10646 standard has formal ways of specifying which characters you support; Unicode doesn't. Instead, Unicode asks that you state these characters, but allows you to specify them any way you want, and you can specify any characters that you want.

  Part of the reason that Unicode doesn't provide a formal way of specifying which characters you support is that this statement often varies depending on what you're doing with the characters. Which characters you can display, for example, is often governed by the fonts installed on the system you're running on. You might also be able to sort lists properly only for a subset of languages you can display. Some of this information you can specify in advance, but you may be limited by the capabilities of the system you're actually running on.

Producing Text as Output

If your process produces Unicode text as output, either by writing it to a file or by sending it over some type of communication link, there are certain things you can't do. (Note that this constraint refers to *machine-readable* output; displaying Unicode text on the screen or printing it on a printer follow different rules, as outlined later in this chapter.)

- Your output can't contain any code point values that are unassigned in the version of Unicode you're supporting.
- Your output can't contain U+FFFE, U+FFFF, or any of the other noncharacter code point values.
- Your output *is* allowed to include code point values in the Private Use Area, but this technique is strongly discouraged. As anyone can attach any meaning desired to the private-use code points, you can't guarantee that someone reading the file will interpret the private-use characters in the same way you do (or interpret them at all). You can, of course, exchange things any way you want within the universe you control, but that

108_GILLAM.ch03.fm Page 107 Monday, August 19, 2002 1:58 PM

doesn't count as exchanging with "the outside world." You can get around this restriction if you expect the receiving party to uphold some kind of private agreement, but then you're technically not supporting Unicode anymore; you're supporting a higher-level protocol that uses Unicode as its basis.

- You can't produce a sequence of bytes that's illegal for whatever Unicode transformation format you're using. Among other things, this constraint means you have to obey the shortest-sequence rule. If you're putting out UTF-8, for example, you can't use a three-byte sequence when the character can be represented with a two-byte sequence, and you can't represent characters outside the BMP using two three-byte sequences representing surrogates.

Interpreting Text from the Outside World

If your program reads Unicode text files or accepts Unicode over a communications link (from an arbitrary source, of course—you can have private agreements with a known source), you're subject to the following restrictions:

- If the input contains unassigned or illegal code point values, you must treat them as errors. Exactly what this statement means may vary from application to application, but it is intended to prevent security holes that could conceivably result from letting an application interpret illegal byte sequences.
- If the input contains malformed byte sequences according to the transformation format it's supposed to be in, you must treat that problem as an error.
- If the input contains code point values from the Private Use Area, you can interpret them however you want, but are encouraged to ignore them or treat them as errors. See the caveats above.
- You must interpret every code point value you purport to understand according to the semantics that the Unicode standard gives to those values.
- You can handle the code point values you don't claim to support in any way that's convenient for you, unless you're passing them through to another process (see the following page).

108_GILLAM.ch03.fm Page 108 Monday, August 19, 2002 1:58 PM

Passing Text Through

If your process accepts text from the outside world and then passes it back out to the outside world (for example, you perform some kind of process on an existing disk file), you can't mess it up. Thus, with certain exceptions, your process can't have any side effects on the text—it must do to the text only what you say it's going to do. In particular:

- If the input contains characters that you don't recognize, you can't drop them or modify them in the output. You *are* allowed to drop illegal characters from the output.
- You *are* allowed to change a sequence of code points to a canonically equivalent sequence, but you're *not* allowed to change a sequence to a compatibility-equivalent sequence. This will generally occur as part of producing normalized text from potentially unnormalized text. Be aware, however, that you can't claim to produce normalized text unless the process normalizing the text can do so properly on any piece of Unicode text, regardless of which characters you support for other purposes.[11] (In other words, you can't claim to produce text in Normalized Form D if you only know how to decompose the precomposed Latin letters.)
- You *are* allowed to translate the text to a different Unicode transformation format, or a different byte ordering, as long as you do it correctly.
- You *are* allowed to convert U+FEFF ZERO WIDTH NO-BREAK SPACE to U+2060 WORD JOINER, as long as it doesn't appear at the beginning of a file.

Drawing Text on the Screen or Other Output Devices

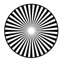
You're not required to be able to display every Unicode character, but for those you purport to display, you've got to do so correctly.

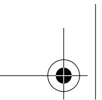
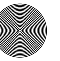
---

11. This caveat guarantees you produce normalized text according to whatever version of Unicode you support. If someone passes text that includes characters from *later* Unicode versions, you may still not normalize them properly. That is okay, as long as you're clear about what version of Unicode you support.

108_GILLAM.ch03.fm Page 109 Monday, August 19, 2002 1:58 PM

- You can do more or less whatever you want with any characters encountered that you don't support (including illegal and unassigned code point values). The most common approach is to display some type of "unknown character" glyph. In particular, you're allowed to draw the "unknown character" glyph even for characters that don't have a visual representation, and you're allowed to treat combining characters as non-combining characters. It's better, of course, if you don't do these things. Even if you don't handle certain characters, if you know enough to know which ones not to display (such as formatting codes) or can display a "missing" glyph that gives the user some idea of what kind of character it is, that's a better option.
- If you claim to support the non-spacing marks, they must combine with the characters that precede them. In fact, multiple combining marks should combine according to the accent-stacking rules in the Unicode standard (or in a more appropriate language-specific way). Generally, this consideration is governed by the font being used—application software usually can't influence this ability much.
- If you claim to support the characters in the Hebrew, Arabic, Syriac, or Thaana blocks, you have to support the Unicode bidirectional text layout algorithm.
- If you claim to support the characters in the Arabic block, you have to perform contextual glyph selection correctly.
- If you claim to support the conjoining Hangul jamo, you have to support the conjoining jamo behavior, as set forth in the standard.
- If you claim to support any of the Indic blocks, you have to do whatever glyph reordering, contextual glyph selection, and accent stacking is necessary to properly display that script. Note that the phrase "properly display" gives you some latitude—anything that is legible and correctly conveys the writer's meaning to the reader is good enough. Different fonts, for example, may include different sets of ligatures or contextual forms.
- If you support the Mongolian script, you have to draw the characters vertically.
- When word-wrapping lines, you have to follow the mandated semantics of the characters with normative line-breaking properties.





108_GILLAM.ch03.fm Page 110 Monday, August 19, 2002 1:58 PM

- You're not allowed to assign semantics to any combination of a regular character and a variation selector that isn't listed in the StandardizedVariants.html file. If the combination isn't officially standardized, the variation selector has no effect. You can't define ad hoc glyph variations with the variation selectors. (You can, of course, create your own "variation selectors" in the Private Use Area.)

Comparing Character Strings

When you compare two Unicode character strings for equality, strings that are canonically equivalent should compare as equal. Thus you're not supposed to do a straight bitwise comparison without normalizing the two strings first. You can sometimes get around this problem by declaring that you expect all text coming in from outside to already be normalized or by not supporting the non-spacing marks.

Summary

In a nutshell, conforming to the Unicode standard boils down to three rules:

- If you receive text from the outside world and pass it back to the outside world, don't mess it up, even if it contains characters you don't understand.
- To claim to support a particular character, you have to follow all the rules in the Unicode standard that are relevant to that character and to what you're doing with it.
- If you produce output that purports to be Unicode text, another Unicode-conformant process should be able to interpret it properly.