This file suffers from some oddities of formatting because it is a converted version of the original document. The original was written on a Xerox ViewPoint workstation system, and here the file has been converted to Microsoft Word (both ViewPoint and Word descend directly from the Bravo document editor created at Xerox PARC in the mid-1970's).

Unicode 88

Joseph D. Becker

August 29, 1988

A INTRODUCTION

1 Overview

1.1 Abstract

This document is a draft proposal for the design of an international/multilingual text character encoding system, tentatively called Unicode.

Unicode is intended to address the need for a workable, reliable world text encoding. Unicode could be roughly described as "wide-body ASCII" that has been stretched to 16 bits to encompass the characters of all the world's living languages. In a properly engineered design, 16 bits per character are more than sufficient for this purpose.

In the Unicode system, a simple unambiguous fixed-length character encoding is integrated into a coherent overall architecture of text processing. The design aims to be flexible enough to support many disparate (vendor-specific) implementations of text processing software.

A general scheme for character code allocations is proposed (and materials for making specific individual character code assignments are well at hand), but specific code assignments are not proposed here. Rather, it is hoped that this document may evoke interest from many organizations, which could cooperate in perfecting the design and in determining the final character code assignments.

1.2 Need for a new, world-wide ASCII

Electronic transmission and storage of the written word are based on standard numerical encoding of text characters. Currently much of the computing world relies on the character encoding for English text called 7-bit ASCII. ASCII (American Standard Code for Information Interchange) is defined by the standards ANSI X3.4-1977 and ISO 646-1973 (E). ANSI is the American National Standards Institute, Inc., and ISO is the worldwide International Organization for Standardization.

ASCII provides a common coinage for representing text content, permitting reliable exchange of English text among disparate software applications. Less obviously, sequences of ASCII characters form structural elements that interlink diverse computer systems. ASCII text files provide a widely-accepted file format among text-oriented programs (e.g. text editors, electronic mail), ASCII character streams provide one standard basis for file communication protocols, and ASCII text "filters" are capable of supporting an interesting class of text-processing applications.

The problem with ASCII is simply that the people of the world need to be able to communicate and compute in their own native languages, not just in English. Text processing systems designed for the 1990's and the 21st century must accommodate Latin-based alphabets for European languages such as French, German, and Spanish; and also major non-Latin alphabets such as Arabic, Greek, Hebrew, and Russian; and also "exotic" scripts of growing importance such as Hindi and Thai; not to mention the thousands of ideographic characters used in writing Chinese, Japanese, and Korean.

What is needed is a new international/multilingual text encoding standard that is as workable and reliable as ASCII, but that covers all the scripts of the world.

For reference, the table below ranks the world's writing systems roughly in order of commercial importance, as measured by the total GNP of countries using each system:

Rank	Writing System	Languages	% of World GNP
1	Latin	English, German, French, Spanish, Italian, Portuguese, Indonesian/Malay, ...	68
2	CJK ideographs	Chinese, Japanese, (Korean)	14
3	Cyrillic	Russian, Ukrainian, ...	14
4	Arabic	Arabic, Persian, ...	3
5	Devan_gar1 family	Hindi, Bengali, Punjabi, Marathi, ...	1
6	Korean (Hangul)	Korean	1
7	Dravidian family	Telugu, Tamil, ...	
8	Greek	Greek	
9	Khmer	Thai, Lao, Khmer	
10	Hebrew	Hebrew	

1.3 Technical summary of Unicode

The power of ASCII comes from two simple properties:

• Its *workability* in processing arises from a fixed length of character code (7 bits within an 8-bit byte)

• Its *reliability* in conveying text content arises from a fixed one-to-one correspondence with the characters of the English alphabet

Unicodes are the most straightforward multilingual generalization of ASCII codes:

• Fixed length of character code (16 bits)

• Fixed one-to-one correspondence with characters of the world's writing systems

That is, each individual Unicode code is an absolute and unambiguous assignment of a 16-bit number to a distinct character.

Since there are vastly more than $2^8 = 256$ characters in the world, the 8-bit byte has become a useless commodity in the context of modern international/multilingual character encoding. Stated otherwise, the evolution from ASCII to Unicode means precisely the

expansion of character codes from an 8-bit to a 16-bit basis. In Unicode, the 8-bit byte plays no role of any kind.

The name “Unicode” is intended to suggest a unique, unified, universal encoding. A sequence of Unicodes (e.g. text file, string, or character stream) is called “Unitext”.

ASCII text		Unitext	
01110100	t	0000000001110100	t
01101000	h	0000000001101000	h
01101001	i	0000000001101001	i
01110011	s	0000000001110011	s
00100000		0000000000100000	
01101001	i	0010011101110001	_
01110011	s	0010011101011011	_
00100000		0010011101101000	_
01110100	t	0000000000100000	
01100101	e	0100101000011011	_
01111000	x	0100101000001010	_
01110100	t	0100101010010101	_

The Unicode design includes major principles that support the pure 16-bit encoding:

• *characters vs. glyphs:* A clear and all-important distinction is made between *characters*, which are abstract text content-bearing entities, and *glyphs*, which are visible graphic forms. This model permits the resolution of many problems regarding variant forms, ligatures, and so on.

• *CJK ideograph unification:* The clear model of characters and glyphs permits unification of tens of thousands of equivalent ideographs that are currently given separate codes in China, Japan, and Korea.

• *public vs. private:* The design provides for the distinction between common-use encodings which are public, and other encodings which are kept private so as to enable vendor-specific implementations, vertical-market applications, and so on.

• *plain vs. fancy text:* A simple but crucial distinction is made between *plain text*, which is a pure sequence of Unicodes, and *fancy text*, which is any text structure that bears additional information beyond pure character content.

• *process-based design:* The design is founded on the fact a text encoding exists solely to support the various processes that act upon text. Thus processes such as rendering, filtering, and so on participate in the design.

Unicode may find its initial utility as a standard international/multilingual *interchange encoding*, but it is also designed to serve as the basis for efficient *internal text representation* (a.k.a. *process encoding*) in any text environment where more than 256 different characters are required.

1.4 Structure of this document

The many aspects of text character encoding are highly interrelated, and indeed each topic is best conceived in terms of a conception of all the others. Lacking hypertext or the ability to discuss all topics at once, the document is arranged as follows:

- Part A is an overview of the major concepts of Unicode.
- Part B is a detailed presentation of Unicode's architectural underpinnings.
- Part C applies the Unicode approach to major specific problems of character encoding.
- Part D contains reference lists of particular details, including suggested Unicode allocations and assignments.

Although the document is laid out from the general to the particular, solutions to the problems of character encoding actually evolve from the particular to the general. For example, the definitions in Part B are made only because they were found necessary to handle the particular problems described in Part C. Thus, the document may make an equal amount of sense if read backward.

2 The 16-bit Approach

The idea of expanding the basis for character encoding from 8 to 16 bits is so sensible, indeed so obvious, that the mind initially recoils from it. There must be a catch to it, otherwise why didn't we think of this long ago?

The major catch is simply that the 16-bit approach requires _____ *(perestroika)*, i.e. restructuring our old ways of thinking. Rather than struggling to salvage obsolete 8-bit encodings via horrendous "extension" contrivances, we need to recognize that the current absence of a standard international/multilingual encoding is a unique opportunity to rethink and revitalize the design concepts behind text encoding.

However, there do exist specific concerns that initially appear to be the "catch" to a 16-bit encoding. To some extent these concerns are overrated, and to some extent they are legitimate but inevitable. This section outlines how the Unicode 16-bit approach either provides for these concerns or trades them off against the greater good. A much longer list of detailed design issues is addressed in Section D.

2.1 Sufficiency of 16 bits

Are 16 bits, providing at most 65,536 distinct codes, sufficient to encode all characters of all the world's scripts? Since the definition of a "character" is itself part of the design of a text encoding scheme, the question is meaningless unless it is restated as: Is it possible to engineer a reasonable definition of "character" such that all the world's scripts contain fewer than 65,536 of them?

The answer to this is Yes. (Of course, the converse need not be true, i.e. it is certainly possible, albeit uninteresting, to come up with *unreasonable* definitions of "character" such

that there are more than 65,536 of them.) There are two main concepts in Unicode's approach to this fundamental question:

- The proper definition of character
- The distinction of "modern-use" characters from "obsolete/rare" ones

Proper definition of "character": Unicode does not confuse the notion of character with that of glyph. There are far more glyphs than characters because of the existence of variant forms, rendering forms, and fragment glyphs that can be used to compose graphic forms dynamically. Also, Unicode avoids tens of thousands of character replications by consolidating together the ideographic characters used in writing Chinese, Japanese, and Korean.

Distinction of "modern-use" characters: Unicode gives higher priority to ensuring utility for the future than to preserving past antiquities. Unicode aims in the first instance at the characters published in modern text (e.g. in the union of all newspapers and magazines printed in the world in 1988), whose number is undoubtedly far below 2^{14} = 16,384. Beyond those modern-use characters, all others may be defined to be obsolete or rare; these are better candidates for private-use registration than for congesting the public list of generally-useful Unicodes.

In other words, given that the limitation to 65,536 character codes genuinely does satisfy all the world's modern communication needs with a safety factor of about four, then one can decide up-front that preserving a pure 16-bit architecture has a higher design priority than publicly encoding every extinct or obscure character form. Then the sufficiency of 16 bits for the writing technology of the future becomes a matter of our active intention, rather than passive victimization by writing systems of the past.

2.2 Relation of Unicode to ASCII and other existing codings

Given two sequences of bits ("bit patterns") that supposedly represent the same series of text characters in two different encoding systems, either:

- the sequences are bit-for-bit identical, or
- they are not identical, in which case they require explicit software conversion.

Clearly, almost every possible pair of text encoding schemes require explicit software interconversion; that is, rarely is one encoding truly a pure "extension" of another. Once the inevitability of explicit conversion processes is recognized, the proper design goals for "compatibility" of a new encoding scheme with existing encodings become:

- to minimize the complexity of conversion processes
- to minimize the number of conversion processes

ASCII text		Unitext	
01110100	t	0000000001110100	t
01100101	e	0000000001100101	e
01111000	x	0000000001111000	x
01110100	t	0000000001110100	t

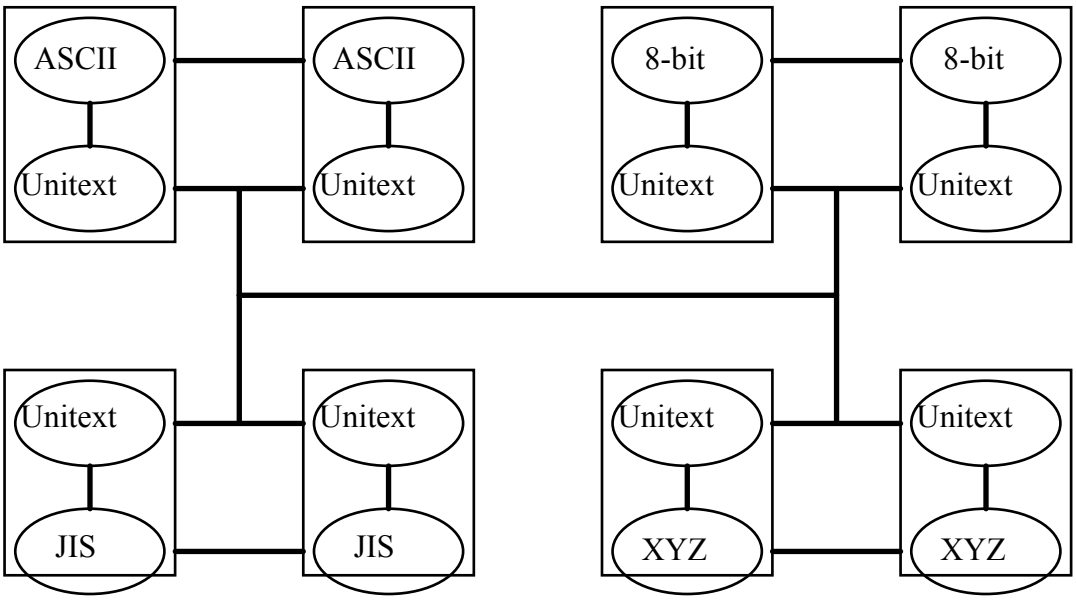
As an example of minimizing conversion complexity, interchange between ASCII text and Unitext is performed by a simple loop of the following operations:

- To convert a 7-bit ASCII character to a Unicode:
  - Preface it with the 9 bits 000000000.
- To convert a Unicode to a 7-bit ASCII character:
  - If the first 9 bits are 000000000, remove them.
  - Otherwise, assign it to a junk code, e.g. SUB (hex 1A).

Insofar as the above algorithms are quite trivial, ASCII text and Unitext may be said to be *conversion-compatible*. One of the design priorities in making the particular Unicode assignment of character codes is to preserve "conversion-compatibility", i.e. the simplicity of these conversion algorithms. This approach is consistent with the Unicode philosophy that processes should be explicitly taken into account as part of the encoding system, rather than being implicitly taken for granted.

Trivial interconversion with existing standards is easily attained for most alphabetic scripts. Unfortunately, straightforward conversion mappings are not possible when it comes to the CJK ideographic characters. Interconversion of these scripts is mainly a matter of indexing through a large table ... which after all is a trivial algorithm once the table is provided.

The goal of minimizing the number of conversion processes is attained simply by using Unitext as an interchange code among disparate encoding systems. Each system could be taught to speak Unitext as an interlingua, while optionally retaining its own "native language" for internal use and local communications. Such a world might be visualized as in the figure below (the rectangles represent systems, the ovals represent the text encodings they support). Evidently having each system implement only 1 conversion process to/from Unicode is vastly more efficient than implementing a number N of conversions that grows as new local encoding schemes are invented.



Since *any* new international/multilingual text encoding will inevitably require explicit conversion to/from existing encodings, this fact might as well be viewed as an opportunity. Within the bounds of "conversion-compatibility", it releases new designs from the need for strict conformity with designs of the past. With luck, the future of computing and electronic communications will be longer than the past. A text encoding design with hopes of serving the 1990's, and perhaps the 21st century, should be engineered primarily to best serve the future, not the past.

2.3 Twofold expansion of ASCII English text

Nothing comes for free, and the price of Unicode's fixed-length 16-bit character code design is the twofold expansion of ASCII (or other 8-bit-based) text storage, as seen in the figure on the previous page. This initially repugnant consequence becomes a great deal more attractive once the alternative is considered.

The only alternative to fixed-length encoding is a variable-length scheme using some sort of flags to signal the length and interpretation of subsequent information units. Such schemes require flag-parsing overhead effort to be expended for every basic text operation, such as get next character, get previous character, truncate text, etc. Any number of variable-length encoding schemes are possible (this fact itself being a major drawback); several that have been implemented are described in a later section.

By contrast, a fixed-length encoding is flat-out *simple*, with all of the blessings attendant upon that virtue. The format is unambiguous, unique, and not susceptible to debate or revision. It is a logical consequence of the fundamental notion of character stream. Since it requires no flag parsing overhead, it makes all text operations easier to program, more reliable, and (mainly) *faster*. It also greatly facilitates the process of unambiguously interpreting text received from other systems, and the deciphering of text that is found embedded within some unknown or extinct data structure.

Unquestionably the twofold expansion of ASCII text will engender increased storage space expense as Unitext is adopted. However, it may be argued that this expense will not prove intolerable. With regard to English text storage, systems may be divided into three categories:

• Software: Most system or application level software should contain little or no inherent English text. Indeed, the prevailing requirement is for program-internal text to be internationalized into message files that can be made multilingual .... precisely the purpose for which Unicode is designed.

• Compressor clients: A few text-system clients create and store vast quantities of English text, and therefore make use of explicit compression/expansion processes. For these systems, Unicode will have no impact at all, since Unicode English text compresses to precisely the same size as ASCII English text.

• Acceptor clients: Nearly all text-system clients create and store quantities of English text small enough that *it is not worth their while to use the currently available techniques for compressing ASCII English text by a factor of 2 or more*. These clients unquestioningly accept the "wasteful" storage of ASCII in order to receive the benefits of its simplicity in processing. There is no reason to alter this behavior when it comes to Unicode, given that the cost of storage media is still rapidly declining. It turns out that in designing a text encoding to serve for the 1990's and beyond, the expense of storage space may be the least important factor that could be brought into consideration.

Historically, computer and communication systems originally implemented 5-bit Baudot character encodings, but it was later discovered that these did not encompass lower-case letters. Then 7-bit ASCII/ISO encodings were implemented, but it was later discovered that these did not encompass European languages beyond English. Then 8-bit extended

ISO encodings were implemented, but it was later discovered that these did not encompass Japanese. Then 14-bit JIS and derivative encodings were implemented, but it was later discovered that these did not encompass Chinese.

The bottom line is that the world of computing has now become a fully international and multilingual one, in which 5-bit, 7-bit, 8-bit, and 14-bit text architectures are all extinct. The modern length of a computer word is 32 bits, and the ultimate length of a character code is 16 bits. All we have to do is recognize what is already true.

3 The Unicode Proposal

3.1 Background of the Unicode proposal

Unicode has evolved from a dozen years of practical experience in implementing multilingual computer systems, beginning at Xerox Palo Alto Research Center. This effort has included product or prototype implementation of the most important Latin-script languages (including Hausa, Hungarian, Polish, Turkish, Vietnamese, and many others), plus non-Latin-script languages including Amharic, Arabic, Armenian, Bulgarian, Chinese, Georgian, Greek, Hebrew, Hindi, Japanese, Korean, Persian, Russian, Ukrainian. This work involved the creation of over 100,000 ideographic character images in various sizes and styles for Chinese, Japanese, and Korean, plus tables cross-referencing the many "standard" encodings of these characters.

In 1978, the initial proposal for a set of "Universal Signs" was made by Bob Belleville at Xerox PARC. Many persons contributed ideas to the development of a new encoding design. Beginning in 1980, these efforts evolved into the Xerox Character Code Standard (XCCS) by the present author, a multilingual encoding which has been maintained by Xerox as an internal corporate standard since 1982, through the efforts of Ed Smura, Ron Pellar, and others.

Unicode arose as the result of eight years of working experience with XCCS. Its fundamental differences from XCCS were proposed by Peter Fenwick and Dave Opstad (pure 16-bit codes), and by Lee Collins (ideographic character unification). Unicode retains the many features of XCCS whose utility have been proved over the years in an international line of communicating multilingual system products.

3.2 Status of the Unicode proposal

This document is currently a conceptual exploratory draft only. It in no way represents the policy of Xerox Corporation, which currently uses the Xerox Character Code Standard in all of its systems products.

Many aspects of Unicode remain to be perfected, and the design itself calls for an ongoing organization devoted to its maintenance, particularly in determining the public registration of new characters.

If the idea of Unicode as a potential new ASCII does have validity, it should be of interest to many companies, standards bodies, and other organizations. The hope is that this document may form the nucleus of a cooperative effort to finish the development of

Unicode in a form satisfactory to all who have an interest in it. If this effort were to be successful, it might naturally lead to the formation of an appropriate Unicode maintenance organization.

Meanwhile, readers' comments for improving Unicode design or its presentation in this draft are avidly solicited.

B ARCHITECTURE

4 Text Processes

4.1 Basic text processes

A text character encoding is not an end in itself; the encoding exists solely to support various *processes* operating on text, ultimately serving the goals of a system's users.

Most computer systems provide low-level support for a relatively small number of *basic text processes*, out of which higher text-processing functionality is built. The following is a suggestive list of basic text processes; it may not be exhaustive, but the interesting point is that it is not far from exhaustive:

- Render visible
- render characters visible (incl. ligatures, contextual forms, etc.)
- break lines while rendering (incl. taboo & other such)
- justify lines
- compute directionality
- modify appearance, e.g. kern, underline, slant, bolden
- Determine units
- locate "character", "word", "sentence" unit
- deal with punctuation, esp. word-internal (e.g. don't)
- Interact with user
- resolve mouse selection
- highlight selected text
- Modify
- insert keyboard input
- transform keyboard input input (e.g. romaji-kana input)
- edit stored text (insert, delete)
- Compare
- determine sort-order of two strings
- filter strings by some criterion (e.g. force lower case, or [ü] => [ue])
- match by some criterion (e.g.content, appearance)
- Analyze text content
- spell check
- hyphenate
- parse morphology
- Treat text as bulk data
- compress/decompress
- truncate (e.g. to fit a string length limit)

- transmit/receive

  In the case of an English encoding like ASCII, the relationships between the encoding and the basic text processes built on it are so straightforward that they can be presumed implicitly without discussion. For example, it is presumed that characters are rendered visible one-by-one in little rectangles from left to right, that there is a linear alphabetical ordering, and so on.

  When it comes to designing an international/multilingual text encoding like Unicode, the relationship between the encoding and the implementation of basic text processes often needs to be considered explicitly, for some fundamental reasons:

• Nearly all of the implicit assumptions that hold for English turn out to fail for many writing systems: in general characters are *not* rendered visible one-by-one in little rectangles from left to right, there is *not* a linear alphabetical order, and so on. The basic text processes for some scripts are far from straightforward.

• The set of text characters appropriate for encoding a language is often debatable. For languages as familiar as French and German, there is disagreement over the identity of the text characters (e.g. ISO 8859 defines accented letters like "â" and "ü" to be individual characters, whereas ISO 6937 represents them by composition instead). The only way to resolve such cases is to explicitly understand how the basic text processes operate on the encoding.

• No encoding can support all basic text processes equally well, so tradeoffs are inevitable. For example, ASCII define separate codes for upper- and lower-case letters, makes some processes easier (e.g. rendering) and some processes harder (e.g. comparison). A different encoding design for English (e.g. case-shift control codes) would have reversed this tradeoff. In designing a new encoding for complex scripts, such tradeoffs must be evaluated explicitly rather than being made unwittingly.

The Unicode design does not specify particular basic text processing algorithms; rather, in most cases it is sufficient that the *existence* of appropriate processes can be presupposed. For example, the assignment of Unicode character code numbers cannot be assumed to provide an alphabetical character ordering for lexicographic string comparison, since in general no linear ordering exists and string comparison may be implemented by arbitrarily complex algorithms. Thus Unicode does not supply any particular string comparison process, but its design does presuppose the capability to implement sufficiently powerful algorithms.

There is no reason to expect text processes in general to be so simple as they are for English. Nevertheless, a computer system that can offer its users highly sophisticated operating and graphical windowing environments should also be sophisticated enough to support text in the user's native language.

4.2 Flexibility through modular process implementation

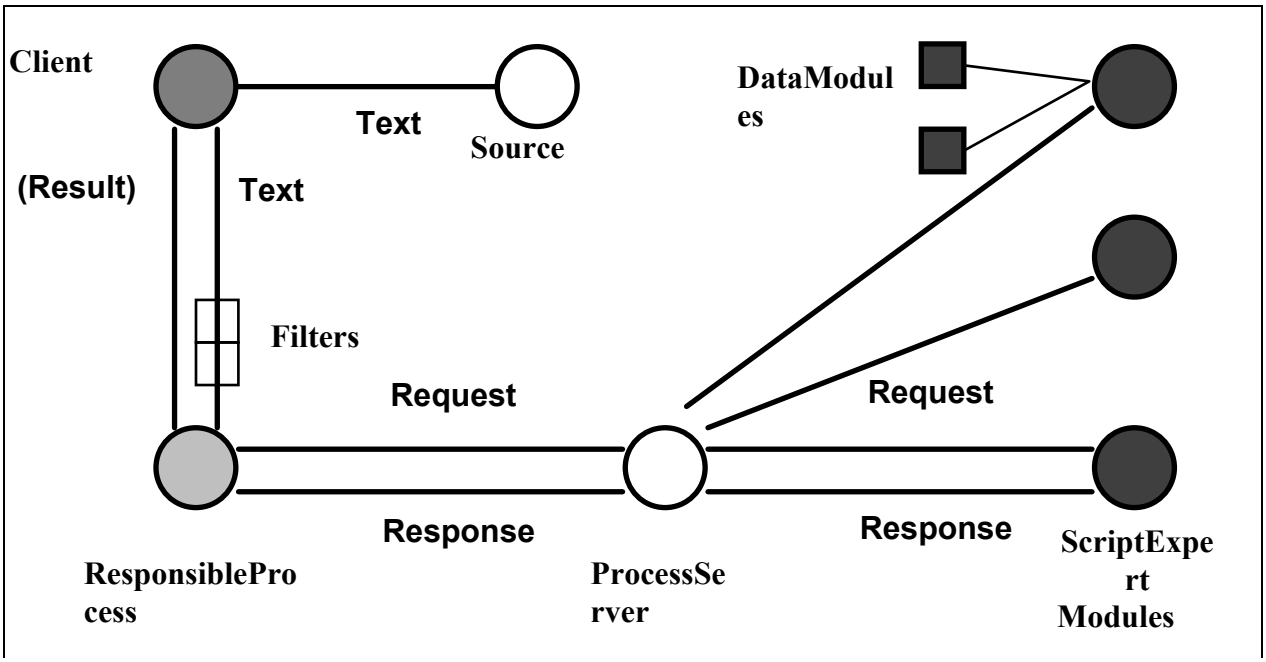
For many important writing systems (e.g. Arabic, Hindi), the optimal text processing implementations have yet to be discovered. Indeed, even for professional-quality English typography, the optimal implementation is still being pursued. A text encoding design like Unicode must afford organizations (from companies to countries) the flexibility to explore different implementation approaches, and even to retain proprietary private approaches for so long as they constitute a competitive advantage.

This goal of flexibility can be reconciled with the goal of standardization in two ways:

• distinguish character codes that are common, public, and standardized from those that are separate, private, and special-purpose

• presuppose modularity in the implementation of basic text processes

The latter point may perhaps be clarified by the generic process model illustrated in the figure on the following page, whose most noteworthy feature is the notion of a *process server* and its associated *script expert modules*. A responsible text process (e.g. rendering or string comparison) is not expected to know how to handle every script in the world. Instead, such detailed knowledge is concentrated in specialist expert modules, which make themselves available to a server that can supply their expertise (perhaps over a network) at the request of any responsible process that calls for help.



Apart from providing the flexibility to support disparate implementations of the same script, the modularity of the server/expert design has a major beneficial side-effect: it provides the flexibility to add new script capabilities incrementally to a given system. Even with a text encoding that supports all the world's languages, particular systems will actually implement only a few scripts at a time. A flexible, modular structuring of text processes is necessary to support the varying needs of user configurations.

4.3 The rendering process

The most important text process is the one for *rendering* a sequence of text character codes visible, mapping them to graphic forms seen on a display screen or paper. Other terms for rendering are *presentation* or *imaging*. From the Unicode point of view, rendering is indeed an explicit process like any other, capable of arbitrarily disparate, arbitrarily complex implementations. Provisions for the rendering process occupy a major part of the Unicode architectural design.

5 Characters and Glyphs

5.1 Characters

Unicode is fundamentally a one-to-one correspondence between 16-bit numbers and the characters of the world's writing systems. But what, precisely, is a character?

A *(text) character* is a unit which is traditionally enumerated as an element of some human writing system (e.g. alphabet). Two such elements are the same character if

there is no conventional enumeration that distinguishes them; otherwise they are different characters.

A "conventional enumeration" means primarily a schoolbook alphabet that is widely shared by members of a culture; for example, the schoolbook convention is that the English alphabet has 26 letters, A through Z. A secondary kind of enumeration is a computer encoding standard; for example, ASCII gives the English alphabet 26 more letters, a through z (however, the contents of computer encoding standards are subject to reconsideration in the design of Unicode).

This definition of character is not at all precise; on the contrary, it is intentionally founded on the terms "tradition" and "convention". The whole point is that there is no intrinsic property that defines a character as such. A character is a "unit of text content", but this notion is equally incapable of formal definition. The vital question of how this imprecise definition can be implemented is deferred to Section xxx.

A character is a totally abstract entity that *has no intrinsic visible form*. The fixed 16-bit length of Unicodes limits the number of characters to 2^{16} = 65,536. Of these numbers, some are to be publicly standardized Unicode assignments, and others are reserved so as to be available for private assignments.

5.2 Glyphs

In order to be seen, a character must be rendered visible as a graphic shape. In the electronic case, this shape is supplied by an explicit *rendering process*.

A *glyph* is a 2-Dimensional graphic shape that can be used in rendering a text character visible (not necessarily one glyph per one character). Two shapes are the same glyph if they can be made to coincide via translation and scaling; otherwise they are different glyphs.

As an example, the following may (arguably) be said to constitute three different characters:

char: first letter of the Latin alphabet

char: first letter of the Cyrillic alphabet

char: first letter of the Greek alphabet

Here the characters are denoted via abstract descriptions. Meanwhile, the following two slightly different shapes are distinct glyphs:

glyph: A

glyph: A

In this example, either of the above two glyphs (and many others such as A) may be used to render any of the above three characters.

There are more than 2^{16} = 65,536 distinct glyphs in use, so to give a unique standard numeric code to each glyph would require more than 16 bits. {xxx ISO 9541 and AFII} As with character code numbers, some glyph codes are to be publicly

standardized assignments, and others are reserved so as to be available for private assignments.

5.3 How the rendering process relates characters and glyphs

Basic English typography is so simple that English text can be rendered satisfactorily on the basis of quite primitive implicit assumptions:

- glyphs are rectangular,
- corresponding one-to-one with the coded text characters,
- arranged linearly along a baseline or path,
- from left to right,
- sometimes even constrained by a fixed lattice of positions.

All of these assumptions fail when it comes to rendering some of the world's major languages, for example Hindi. This is illustrated in the rendering of the Hindi word _____ (*p_rti*, meaning "fulfillment"), shown in the figure on the following page. The 1-Dimensional sequence of character codes on the left is rendered into the 2-Dimensional pattern of glyphs on the right. For convenience, a character is denoted here by a glyph in brackets, thus [_], while a glyph itself is enclosed in braces, thus {_}:

The example shows clearly enough that the rendering process mapping character codes into glyphs may in general be an *arbitrary algorithm*; it need not be one-to-one nor linear in any sense. In the long run, a preferred standard algorithm for each script may eventually emerge, but in the meantime Unicode assumes that competing private rendering implementations will exist, even for high-quality English typography.

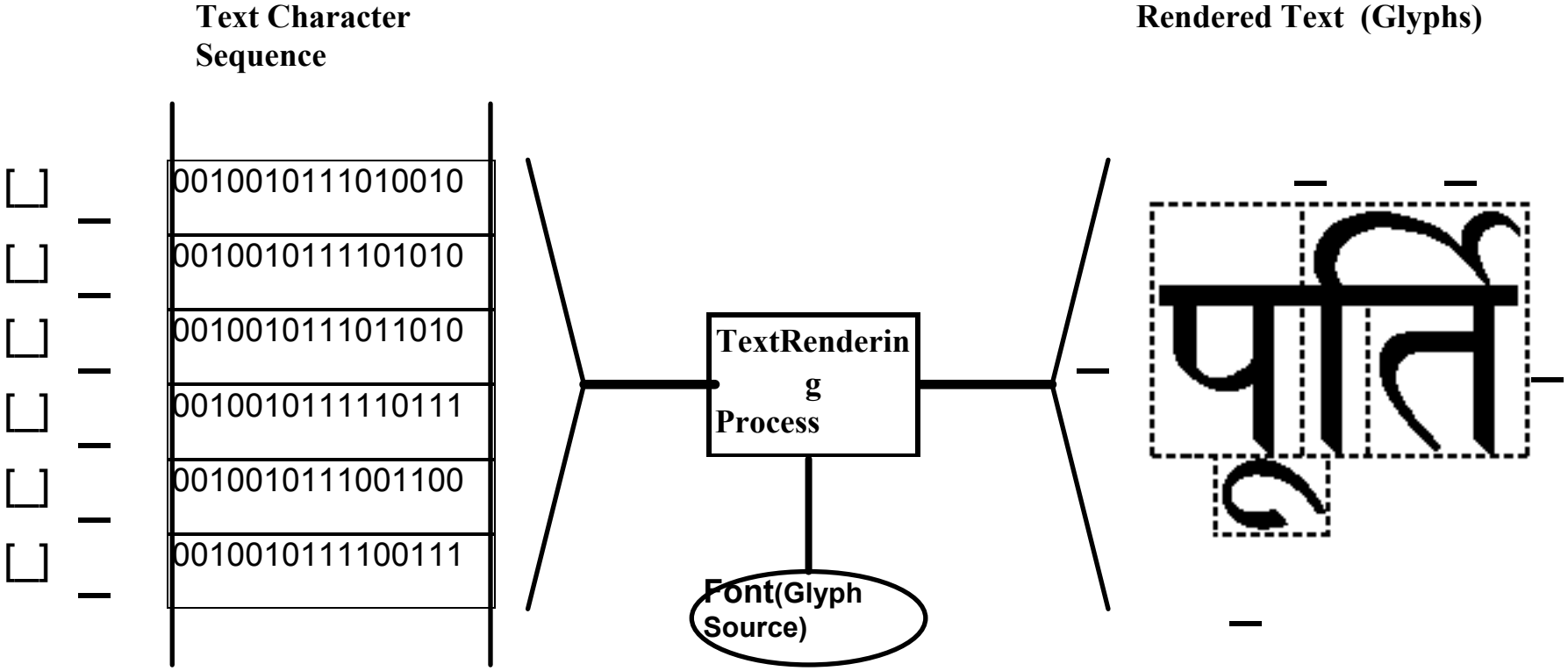
5.4 Stroke style of glyphs

Many different glyphs may be used to render any given character; for example, the glyphs {A, **A**, *A*, A} may all render the character "first letter of the Latin alphabet". The differences among these glyphs xxx ISO 9541; for the purposes of Unicode design, it is sufficient to combine all such differences into the property *stroke style*:

A *stroke style* is a category of traditional typography which defines a set of glyphs as harmonizing with each other for use in setting continuous text. That is, in rendering connected expository text other than ransom notes, most glyphs are selected to have the same stroke style.

Stroke style refers to the typographic design of the component curves and lines that make up the glyph form, not to the selection or arrangement of strokes. For example, the two glyphs {_} and {_} differ by a single dot, but all of their component strokes are drawn in the same style.

From the point of view of defining Unicodes, the only interest in the concept of stroke style is to be able to eliminate it as a variable. All of the relationships between characters and glyphs hold independently for *each* stroke style, so for the purposes of further discussion it may be assumed that all glyphs are in one given stroke style.



_ The character *pa* [_] is rendered by a glyph on the left, since the script generally runs from left to right_ The character _ [_] is rendered *below and a bit to the right of* the preceding

_ The character *ra* [_] is rendered by a *contextual form* glyph, namely a small hook {___ } that appears *high and to the right* according to certain rules

_ The character [_], which is a traditional mark (called *"halant"* in Hindi) that subtracts an implicit vowel from the preceding letter, *is not rendered at all*

_ The character *ta* [_] is rendered relatively "normally, i.e. to the right

_ The character *i* [_] is rendered *to the left* of the character that it follows, and optionally (in good typography), it is rendered by a *ligature* glyph {_} combining it with the character *three or more codes before it*

5.5 Particular relationships between characters and glyphs

It is worth having terms for certain important relationships between characters and glyphs that commonly recur:

An *independent form* glyph for a given character is a glyph that may normally be used to render that character in complete isolation. For example, the glyphs {A, **A**, *A*, A} are various independent forms for the character "first letter of the Latin alphabet". An independent form glyph is not to be confused with the abstract character itself.

A *contextual form* is a glyph that may be used to render a given character in some circumstances, but not normally in complete isolation. For example, the glyph {_} is an independent form for the character "letter *kaf* of the Arabic alphabet", while the glyph {___} is a contextual form for the same character that occurs only within words.

A *ligature* is a glyph that may be serve to render two or more characters at the same time. For example, the {ffi} ligature for English renders the three characters [f][f][i] simultaneously.

A *fragment* is a glyph that must be used in composition with other glyphs in order to render one or more characters. For example, the umlaut mark {_} is a fragment glyph, since the umlaut cannot render any character except in composition with other glyphs.

A *rendering form* is any glyph that is not an independent form of some character; this term thus includes contextual forms, ligatures, fragments, and perhaps other oddities.

A set of *variant forms* is a set of different glyphs in the same stroke style that can play the same roles in rendering the same character(s). For example, the glyphs {__, _} are variant forms for the character "letter *kaf* of the Arabic alphabet" (an Arabic-script font from a professional type house will contain both glyphs, confirming that they have the same stroke style). Significantly, the glyphs {_, _} are variant forms for the character "path". Note that the glyphs {A, **A**, *A*, A} are *not* variant forms, since they do not belong to the same stroke style.

5.6 Private fonts and glyphIDs

Any particular private computer implementation of a rendering process makes use of a particular private collection of glyphs. A *font* is an implemented collection of glyphs, all in the same stroke style, containing at most one glyph from each set of variants. The word *font* suffers from a considerable amount of abuse; what is important in the usage here is the restriction to at most one glyph from each set of variants.

A font implementation needs to index the glyphs within the font by some code number. A *glyphID* is a private code number used to index the glyphs within a font. There is no restriction on the length of glyphID codes, but for international/multilingual fonts the length 16 bits suggests itself, since there are far more characters than $2^8 = 256$, and far fewer glyphs in any one font than $2^{16} = 65{,}536$.

In general, a font and its associated rendering process define an arbitrary mapping between glyphIDs and Unicodes. Some of the glyphs in a font may be independent forms for individual characters, while others may be rendering forms that do not directly correspond to any one character. For those glyphs that are independent forms, it may be convenient for the glyphID to have the same numerical value as the Unicode, but this is not required.

In general, a font also defines an even more arbitrary mapping between glyph codes and glyphIDs.

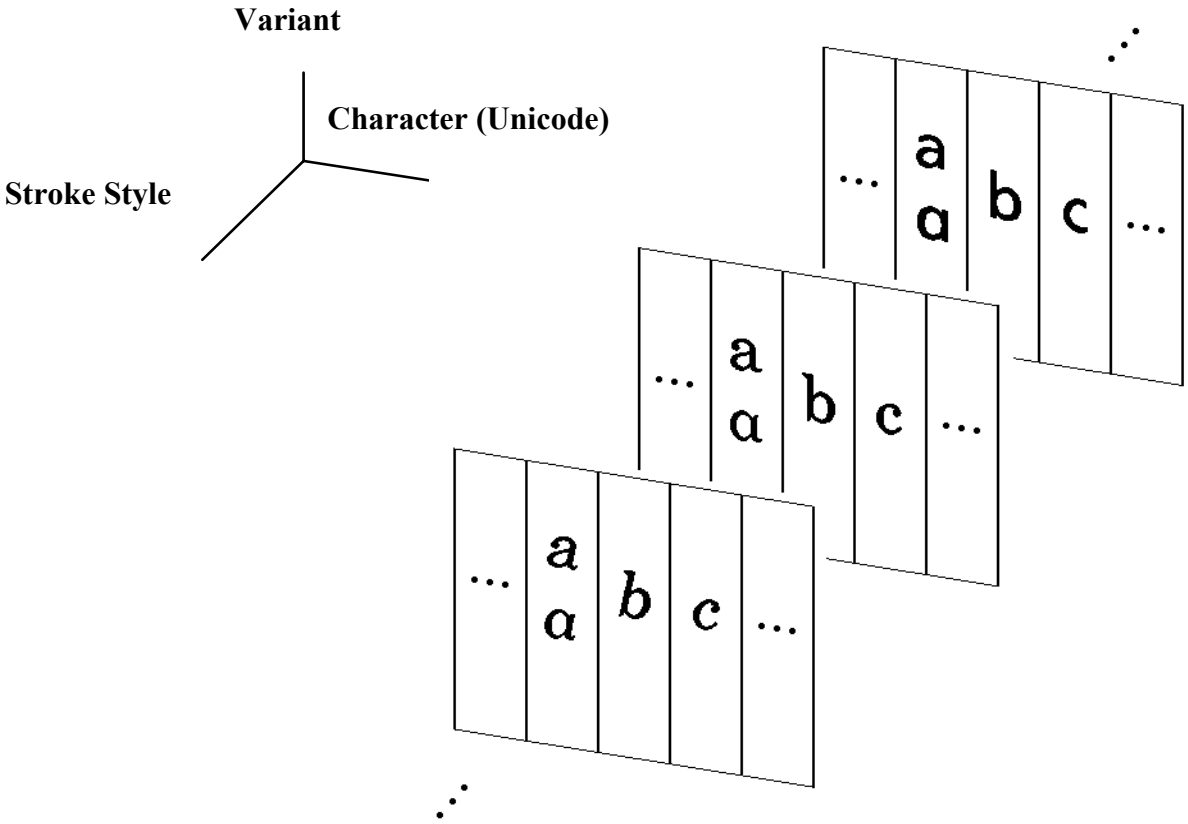
The figures on the following pages attempt to illustrate the relationships among characters, glyphs, and fonts as they have been defined. The first two figures are set in a "3-D pseudo glyph space", whose dimensions are:

- The text character that the glyph is used to render (i.e. Unicode)
- The stroke style of the glyph
- The glyph form variant (if any)

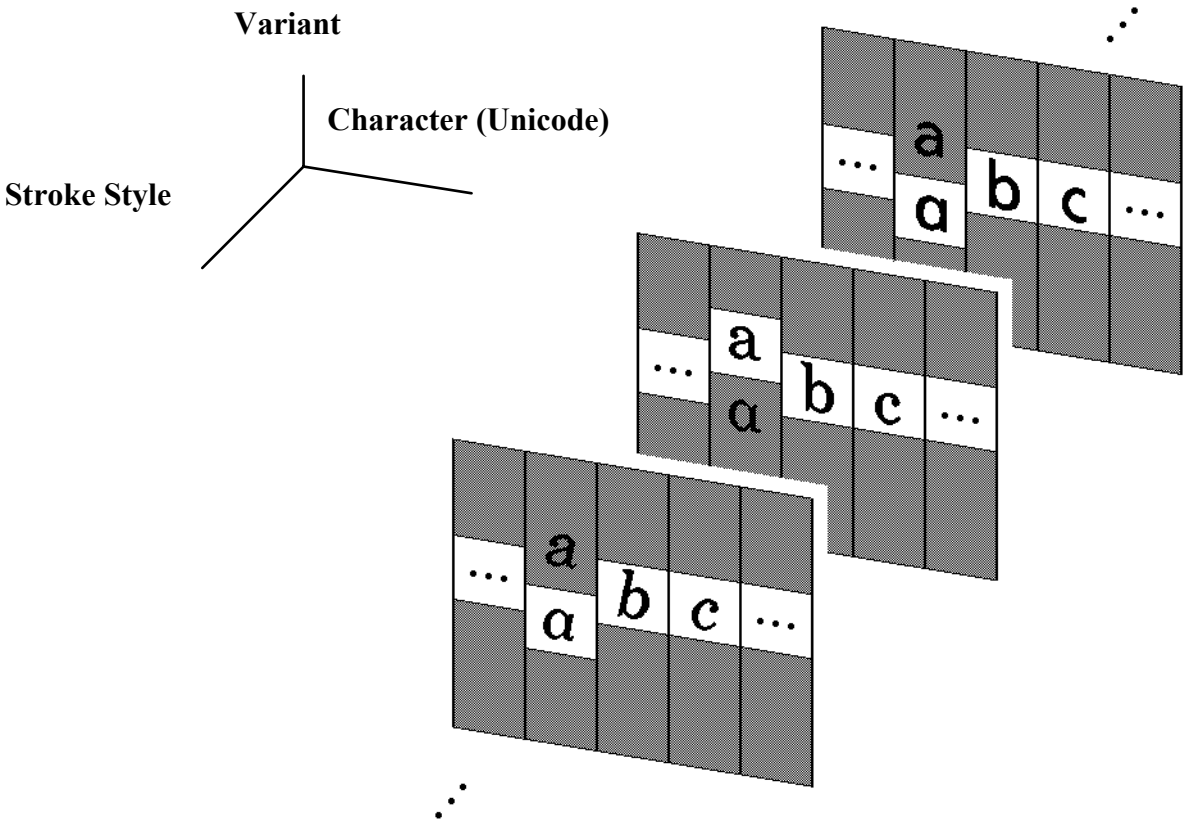
Actually, glyphs cannot in general be cleanly factored in this manner, since the relationship between text characters and glyphs is not generally one-to-one. However, it is worth briefly setting aside the fine points in order to gain a useful visualization.

The first figure shows several glyphs for the English letters [a][b][c] floating in pseudo glyph space. The stroke style planes are drawn in, roughly parallel to the plane of the page. The variant dimension is shown as vertical. The glyphs project down onto a horizontal dimension of text characters: this is the linear space of Unicode numbers.

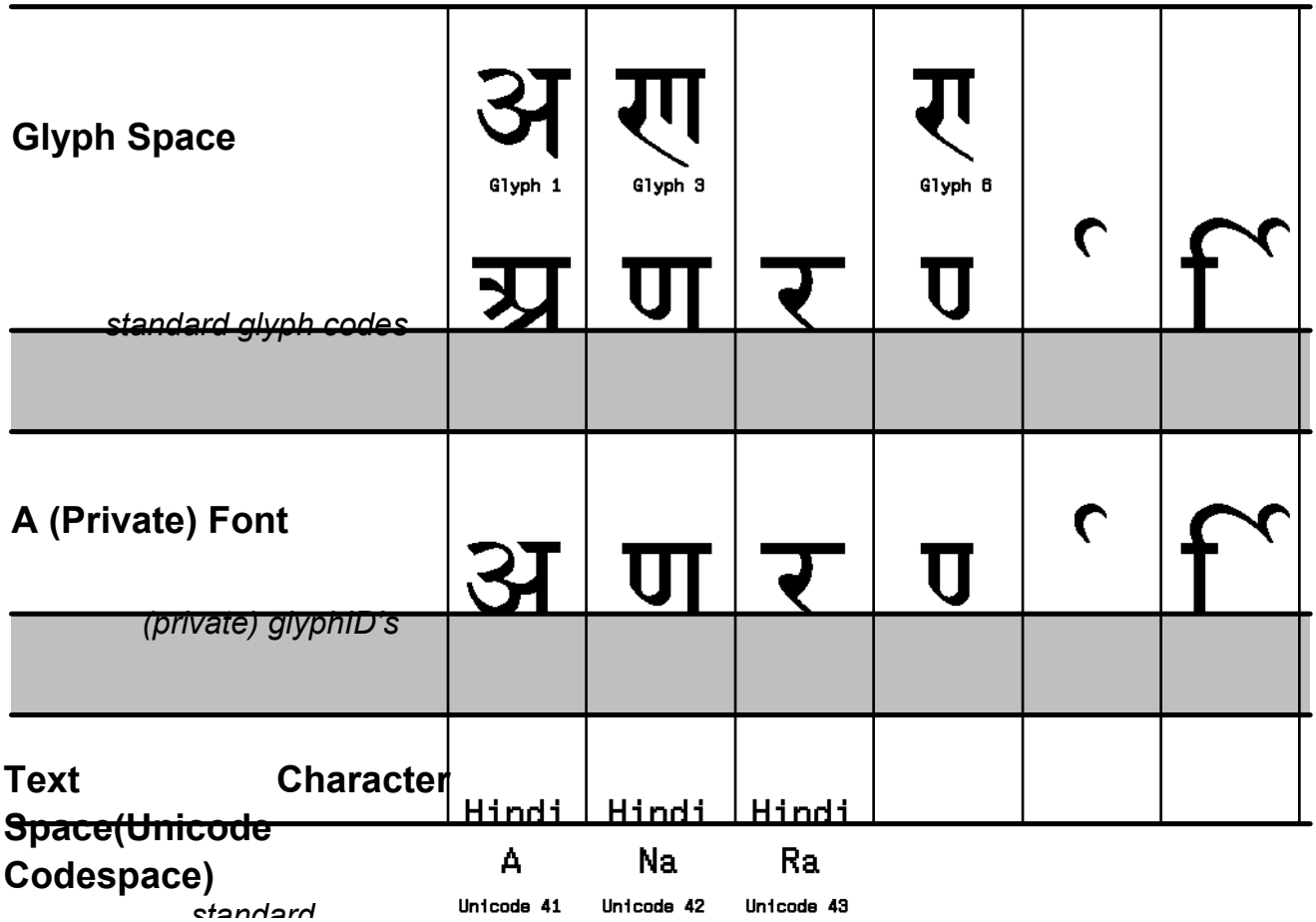
In each stroke style there are two variant form glyphs for the letter [a] (they even have names: "humanitarian" and "grotesque"). It would be a typographic error to use both of these forms in the same running text, so one form or the other must be selected for inclusion in a particular font.


Variant
Character (Unicode)
Stroke Style
a
a
b
c

The figure below illustrates the selection of a particular subset of the glyphs to form a font. The font is the white horizontal band; the rejected variant forms are grayed out.



The English alphabet does not offer enough complexities to illustrate the interesting features of this model. The figure below shows a more complicated case, that of Hindi. In this visualization, only a portion of a single stroke-style plane is shown, and the glyph space, character space, and private font are broken out as separate bands.



• The top band is a portion of (pseudo) glyph space, which is the total repertoire of graphic forms. Each glyph has a standard identifying code according to some system independent of Unicode. Glyphs that are vertically aligned are variant forms, i.e. free-choice alternative ways of writing the same character. The three glyphs {___ }, {_}, and {_} are rendering glyphs: they are not the independent form of any letter, and hence they do not correspond directly to any Unicode.

• The bottom band is a portion of 16-bit text character space, which is the total repertoire of content-bearing entities. To clarify this figure, characters are denoted by a content description such as"Hindi A", rather than via a graphic. Each text character has a 16-bit Unicode, independent from the glyph codes.

• The middle band is a portion of an instance of a private font, which is one of many possible private selections of glyphs. A font includes at most one choice from each variant form set. Each glyph has a private identifying code called a glyphID, which has no necessary relationship with either the standard glyph codes or the standard character codes.

The previous figure should make it clear that, even with regard to static encodings, the Unicode character code assignments are merely one aspect of a larger architecture. The Unicode design retains flexibility for disparate private implementations by providing for, but not specifying, several key elements:

• *available glyphs:* Especially the rendering glyphs may be privately designed, tailored to a specific and perhaps proprietary implementation.

• *glyphID encoding:* This and the mappings to it are implementation-private.

• *selection of glyphs:* The model does not specify who makes the selection of variants that go into a font. This may be done by the type designer, by the type vendor, by the system vendor, or perhaps (in the case of a desktop publishing system) even by the individual user building a personalized font.

6 Sequences of Characters

6.1 Plain text and fancy text

Plain text is a pure sequence of character codes. Plain Unicode text, i.e. a sequence of Unicodes, is called Unitext.

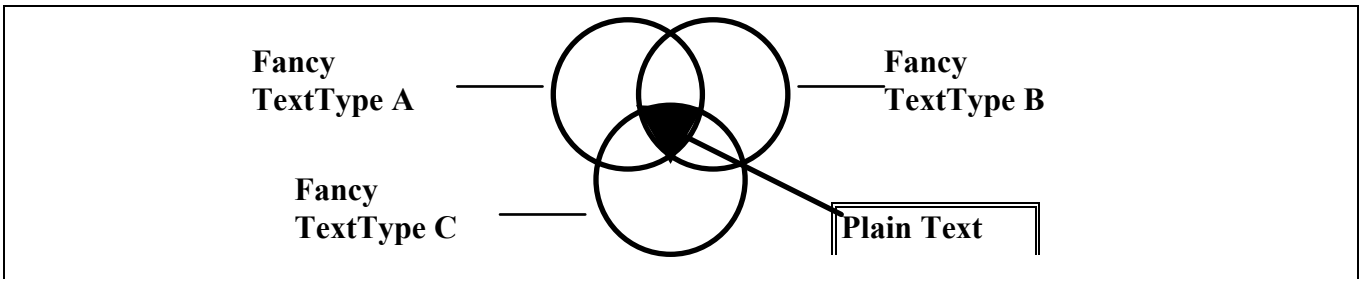
Fancy text is any text representation consisting of plain text plus added information. For example, multifont text as formatted by a desktop publishing system is fancy text.

The kinds of data structures that can be built into fancy text are limited only by the imagination. To give but one example, in fancy text containing ideographs it would be possible (and beneficial) to store the phonetic reading of each ideograph somewhere in the text structure. Other applications abound.

On the other hand, the simplicity of plain text gives it a natural role as a major structural element:

• Plain text arises inevitably from the notion of a character stream

• Plain text is the intersection or least common denominator of all fancy text:



Both plain and fancy text are already familiar constructs in ASCII-based systems. From experience with systems using both types of text, their relative functional roles are well known:

- Plain text, being inevitable, is public, standardized, universally readable
- Fancy text, being consciously designed for a particular purpose, is often intended to be private, implementation-specific, even proprietary

6.2 Content vs. appearance in plain vs. fancy text

The details of any particular fancy text design can be made public or standardized, but the fact remains that most fancy text designs are vehicles for particular implementations, not readable by other implementations. Since fancy text equals plain text plus added information, the extra information in fancy text can always be stripped away to reveal the "pure" text underneath. This operation is familiar, for example, in word processing systems that deal with both their own private fancy format and with the universal plain ASCII text file format. Thus by default:

- Plain text represents the basic interchangeable "content" of text

This is a suitable guideline, despite the fact that the term "content" appears to have no precise intrinsic definition.

Given that plain text represents content, then the interesting question becomes its relationship to appearance information. Since text characters are abstract entities that have no visible form, plain text *per se* has no appearance at all. It requires a rendering process to make it visible.

If the same plain text sequence is given to disparate rendering processes, there is no expectation that they should produce the identical text appearance; all that is required is that they should preserve the text content, i.e. that disparate rendering processes should make the text *legible* with the intended reading. Therefore, the relationship between appearance and the content of plain text may be stated as follows:

- *Plain text must contain enough information to permit the text to be rendered legibly, nothing more*

This conclusion is of the utmost importance in a text encoding design. It tells a great deal in general about what problems an encoding must be designed to solve, and it also answers a great many vital specific encoding issues. To give but a few examples:

• *Q:* Does the optional ligature glyph {ffi} have a corresponding Unicode?

A: No. Use of the ligature {ffi} is never *necessary to the legibility* of English text, therefore the basic content of the word "office" would never need to be represented by the sequence * [o][ffi][c][e]. It follows that any mechanism added to the text to call selectively for this ligature would be a *fancy* text feature: a nice touch, but one beyond the basics of plain English rendering.

• *Q:* Does the obligatory ligature {__} in Arabic have a corresponding Unicode?

A: No. Since this ligature is obligatory, the plain character sequence [_]_[_] contains enough information to permit usage of the ligature to be inferred, thus permitting the text to be rendered legibly.

• *Q:* Does the sequence of Hindi letters [_][_] require indicators, control characters, or other contrivances to specify that the {_} glyph is treated as a mark that is attached below the {_} glyph?

A: No. The rendering behavior of Hindi letters follows regular rules, so a plain character sequence contains enough information to permit the placement of the vowel marks to be inferred, thus permitting the text to be rendered legibly.

• *Q:* Does plain text containing characters of the Arabic or Hebrew alphabets require indicators, alternate forms of characters, or other contrivances to specify text directionality?

A: No, outside of rare exceptional cases. In nearly all cases, a plain character sequence contains enough information to permit the directional layout of text containing Arabic or Hebrew to be inferred, thus permitting the text to be rendered legibly.

6.3 Implicit spelling conventions and rendering conventions

A *spelling convention* is traditional rule for selecting and sequencing text characters to representing some particular text content.

A *rendering convention* is traditional rule for mapping from a conventionally-spelled text character sequence to a rendered text configuration.

Spelling conventions come primarily from the same widely shared schoolbook traditions that define the characters themselves. For example, the English word "dog" (canine) is traditionally spelled _[d], _[o], _[g]. (Spelling may in some cases have some correlation with pronunciation, but in general spelling rules are arbitrary, even in the case of the word "dog" which has many pronunciation variants in English.)

The plain character sequence _[d], _[o], _[g] *per se* does not contain within it enough information to specify how it is to be rendered, even if glyphs {d}, {o}, {g} are at hand. The plain text does not specify the relative positions of the glyphs, which might be:

dog god d
g
o

It would certainly be possible to create a fancy-text data structure that could specify the glyph positions precisely, but that is beside the point. According to the criterion given in

the previous section, it must also be possible for the plain character sequence alone, with no added information, to be rendered legibly (i.e. in this case as “dog”).

The only way that plain text rendering can be possible is via *rendering conventions* shared between the character encoding and the rendering process. There is nowhere else for specific rendering information to come from. For example, every basic English rendering process implicitly assumes that a sequence of English letters will be mapped one-by-one to glyphs arranged linearly from left to right along a baseline or path.

The Unicode model makes explicit the fact that rendering conventions like this are indispensable:

• Plain text rendering can be accomplished only through coordination of spelling conventions with rendering conventions

To give a contrasting example, the spelling conventions of Hindi are that the vowels (e.g. [_]) should come after their consonants (e.g. [_]) in phonetic sequence, but the rendering conventions specify that certain vowel glyphs are attached as marks below the preceding consonant glyphs. By the same token, it is a rendering convention that Arabic and Hebrew letters are arranged linearly from right to left (not to mention the complex contextual mutations of the Arabic letterforms).

The overall point here is that:

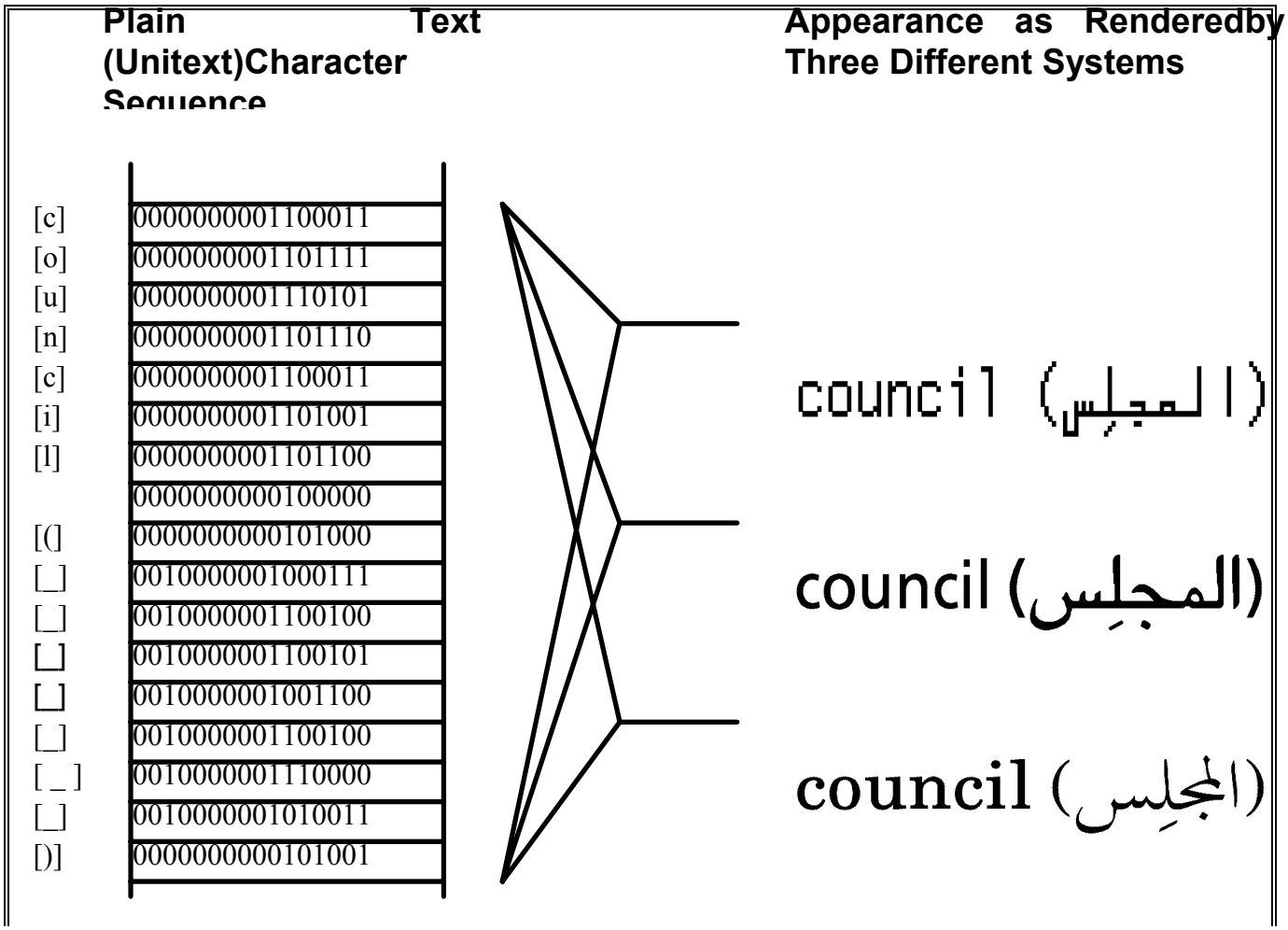
• Spelling conventions and rendering conventions are not explicitly encoded by any bit patterns at all in a plain text sequence

A major particular corollary is that:

• Basic rendering directionality is one of the rendering conventions, it is not explicitly encoded by any bit patterns at all in a plain text sequence

In other words, it is a widely-known convention that English letters are rendered from left to right, while Arabic and Hebrew letters are rendered from right to left. This information need not, and indeed cannot, be included explicitly in plain text.

The figure below illustrates an example of rendering mixed English/Arabic plain text, involving the word *majlis* (council). If disparate systems render the text, its final appearance may be quite different, but in each case its content is legibly the same. The rendering conventions include not only left/right directionality, but also the placement of a vowel mark below a letter, the use of contextual letterforms, and optional use of a ligature. None of this information is explicitly specified in the text sequence.



6.4 Explicit conventions and character properties

Spelling and rendering conventions cannot be encoded explicitly in plain text -- simply because plain text has no place to put them. They are therefore implicit in the sense that they must be shared by agreement between the creator of the text and the implementor of the rendering process. However, these conventions can be made explicit in the sense that they can be published and even standardized.

One useful format for making spelling and rendering conventions explicit is via lists of characters that share certain rendering properties. For example:

- Characters rendered from left to right: Latin alphabet, Cyrillic alphabet, etc.
- Characters rendered from right to left: Arabic alphabet, Hebrew alphabet
- Subscripted Hindi vowels: u, _, r, r, l, l

Some conventions may not be susceptible to such a simple format. For example, it is a spelling rule of the Indian national standard ISCII for Hindi that all vowel characters are to

be spelled in phonetic sequence after the consonants that they follow. (This rule is not vacuous, since in Hindi the glyph {_} for the vowel *i* is written and typewritten *before* the consonant that it follows.) It is not clear how algorithmic rules of this sort can be represented in a standardized fashion. Perhaps a simple statement such as the above is sufficient.

An awkward problem with publishing these conventions is that for some scripts (e.g. Arabic), the details of how to render plain text correctly are difficult to discover, and may currently constitute a proprietary advantage. Soon enough, however, all such techniques will become public knowledge susceptible to standardization.

The fact that spelling and rendering conventions are inevitable, and their eventual publication and standardization desirable, permits natural Unitext solutions for the two most difficult problems concerning the relation between spelling and rendering, namely: applied diacritical marks and Korean Hangul. These will be discussed in detail in their appropriate sections, but may be summarized as follows:

• *Applied diacritical marks:* A list of diacritical mark characters are defined (e.g. umlaut [_]), having the conventional rendering property that their glyphs are "applied" to the glyph of the character that precedes them in the text. The correlate spelling convention is that a diacritical mark character is sequenced after the character that it "applies" to.

• *Korean Hangul:* The spelling convention is that Korean Hangul text is spelled out in individual Hangul characters. The rendering convention is that such text is rendered by first parsing it into umcels (syllable groups), which are then realized via privately-defined rendering glyphs.

The publication of standardized spelling and rendering conventions is not an intrinsic part of the Unicode standard, but it is very closely affiliated. A map of the text elements associated with the Unicode architecture is provided at the end of this section.

6.5 Plain text filters

A *filter* is a process that takes some plain text as input, performs a simple transformation on it, and yields some plain text as output. Often the transformation consists of transliterating one set of characters (or character *n*-tuples) into another.

Filters are familiar from contexts like the UNIX® operating system, and adequately documented as such. Their power arises from the fact that they can be arbitrarily concatenated to build up useful complex transformations. Filters provide a natural compensation for some of the variability that inevitably arises in Unitext sequences:

• Filters can compensate for ambiguities caused by spelling conventions. For example, some "marked" characters like u-umlaut "ü" can be spelled two ways: either as the base character followed by the mark [u][_], or as a single Unicode [ü]. A system that wishes to normalize the text it receives to enforce one spelling convention or the other can easily do so via a filter.

• Filters can compensate for tradeoffs unavoidably made in the design of a text encoding standard. For example, ASCII makes the debatable tradeoff of building

"case" into the character encoding, thereby forcing many processes to normalize the text they receive into all upper-case or all lower-case. Such a transformation can be achieved naturally via a filter.

A general architecture for text processes might well assume that a library of useful filters is provided, available to be attached whenever a basic text process is applied to a stream of text character codes.

6.7 Control characters

Existing character encoding standards include a concept called "control character". The definition of ASCII control characters in the ANSI X3.4-1977 standard is:

"**Control Character.** A character whose occurrence in a particular context initiates, modifies, or stops an action that affects the recording, processing, transmission, or interpretation of data."

From the Unicode point of view, in which the role of text processes is dealt with explicitly, it is quite meaningless to say that "A character ... initiates ... an action ..." All characters are passively *acted upon* by processes. Recent standards, such as that for Office Document Architecture (ISO 8613/1: 1988), attempt to work around this problem by changing the wording of the definition:

"control function: An element of a character set that affects the recording, processing, transmission or interpretation of data."

But now this definition applies equally well to *all* text characters. It seems worthwhile to rethink the concept of "control character" from the beginning.

In terms of the Unicode model, the traditional ISO-standard control characters fall into four categories:

• *Punctuation:* The character "CR: Carriage Return" (hex 0D) may be regarded as a punctuation mark bearing text content, indicating the end of a paragraph in the same way that a punctuation mark like period indicates the end of a sentence or a comma the end of a phrase. The character "SP: space" (hex 20) is likewise indispensable punctuation. These two characters are members of Unicode.

• *Substitute:* The character "SUB: Substitute" (hex 1A) has transitory value in the history of Unicode. If multilingual Unitext is converted to a more restrictive encoding such as ASCII, the best that can be done with a non-ASCII character is to map it to SUB. If the resulting ASCII text is converted back to Unitext, the best that can be done with a SUB is to retain it as such. Therefore SUB may be said to bear the content "there was a character here, but its identity got lost". This function is unneeded in a pure Unitext world.

• *Meta-encoding mechanisms:* Characters such as "SO: Shift Out" (hex 0E) and "SI: Shift In" (hex 0F) are intended for meta-mechanisms that vary the interpretation of the text encoding format itself. Since Unitext is specifically designed to eliminate variable encoding, these mechanisms have no place in Unitext.

• *Device control, etc.:* The remaining control characters are sub-categorized by ISO as device control, transmission control, information separator, and format effector

characters. Some of these refer to obsolete functionality, and some might be useful in a fancy text extension; but none of them belong in plain text.

The Unicode model does not preclude control codes in general, except from plain text. There is nothing to prevent private implementations from agreeing to interpret a privately-assigned code value (presumably from the User section of the Unicode space) in any manner at all including so-called “control functions”. The only restriction is that such representations are not to be considered standard, public, plain Unitext.

Once it is realized that so-called “control functions” generally consist of adding extra information to plain text, the door is open to consider the best structures for such data. In general there may be alternative data structures for implementing “fancy” text functionality that may be preferable to embedding control codes in-line in the text ... at any rate, this is a matter beyond the concern of the plain Unitext level of encoding.

6.6 Layers of text representation

A great number of properties, structures, and operations are built on a text encoding model. These can be roughly organized into a layered structure, as suggested in the figure below.

	Level Name	Level	Examples
Fancy Text	Applications	higher applications	e-mail system, document editor
	Structures	higher data structures	string, e-mail msg, document
	Appearance ops	text appearance operations	line break, rendering, selection
	User items	user non-text item rendering	fields, user-drawn graphics
	Appear props	text appearance properties	font style, size
Plain Text	Sequence ops	plain Unitext content operations	filtering, sorting, matching
	Text rep	spelling conventions	diacritical marks, digraphs
	Lang props	lang-specific char props & classes	sort order, hyphenation
	Char props	inherent char props & classes	directionality, word-break, case
	Character ops	character-based operations	input, deletion, transmission
	User codes	user code assignments & escapes	rare or private-use characters
	Char codes	Unicode	

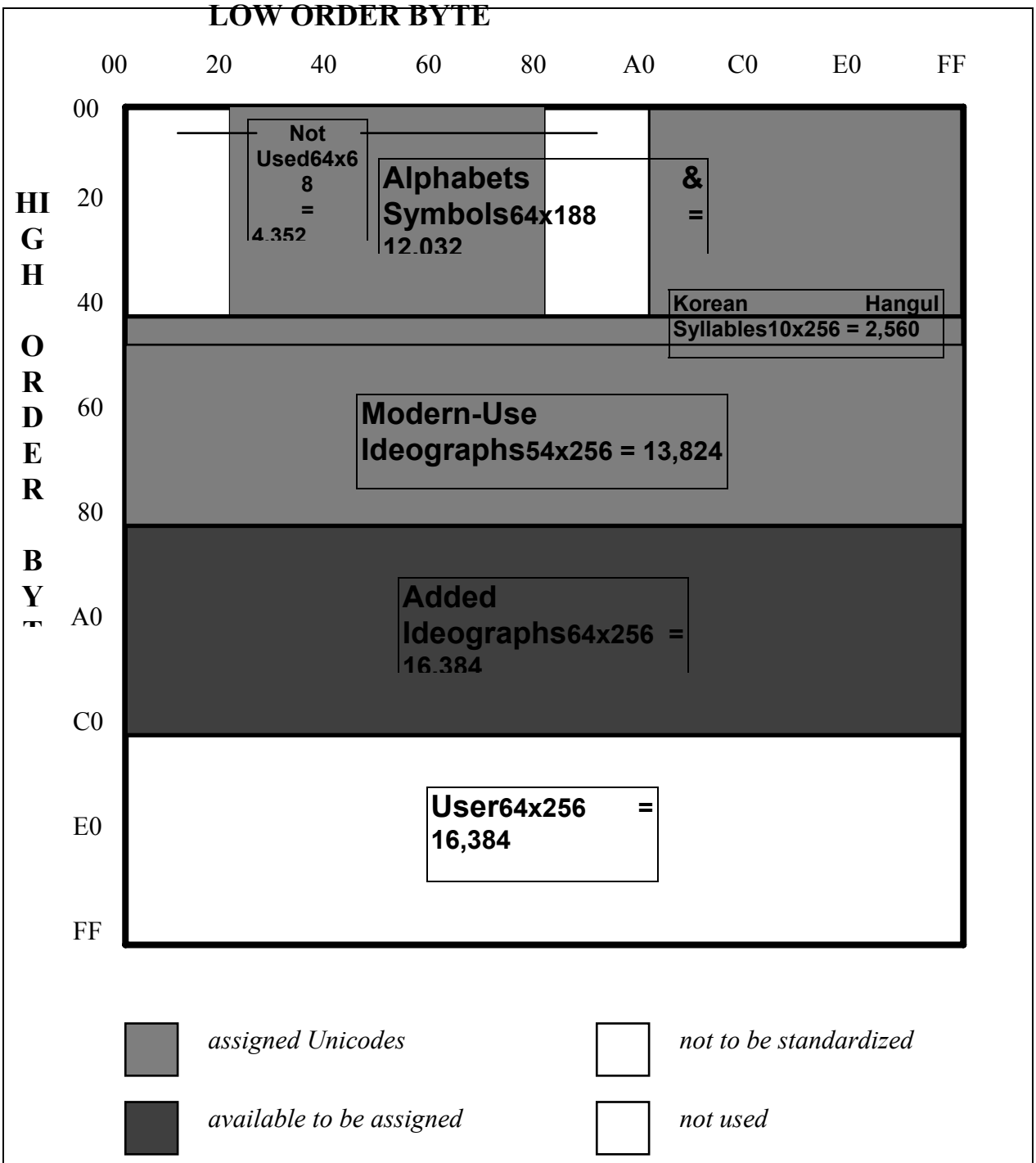
The primary division of layers is between *plain text* and *fancy text*. Within each of these levels, it is possible to distinguish a finer layered structure.

Within fancy text, the layers are rather general, since fancy text can take on arbitrary capabilities. However, the most important layers of fancy text operations are undoubtedly those that deal with providing high-quality text appearance, and then the applications built on that.

Within plain text, the layers correspond basically to the topics discussed in this document, although language-specific properties are not discussed here. The two most important layers of plain text operations are those focused on individual characters and those involving sequences of characters.

To be complete, a text encoding standard ideally would cover all of the layers shown in the figure. The present document is primarily addressed to only the bottom layer which defines Unicodes, but the Unicode design cannot be understood without an awareness of its relationship with the other layers of a complete text architecture.

[Figure **Unicode Codespace Allocation** goes here]



Unicode Codespace Allocation Overview