NRSI: Computers & Writing Systems

SIL HOME | CONTACT US

Search

You are here: Encoding > Unicode

Search

Home

Contact us

General
Initiative
B@bel
WSI
Guidelines

Encoding
Principles
Unicode
Tutorials
PUA
Character stories
Conversion
Utilities
TECkit
Maps
Resources

Input
Principles
Utilities
Tutorials
Resources

Type Design
Design Tools
Formats
Resources
Font downloads
Gentium
Doulos
IPA

Rendering
Principles
Technologies
Graphite
XSEM
Resources
Font FAQ

Links

Glossary

Understanding Unicode™: A general introduction to the Unicode Standard (Sections 1-5)

Peter Constable, 2001-06-13; 2082 reads

Contents

Unicode is a hot topic these days among computer users that work with multilingual text. They know it is important, and they hear it will solve problems, especially for dealing with text involving multiple scripts. They may not know where to go to learn about it, though. Or they may have read a few things about it and perhaps have seen some code charts, but they are at a point at which they need to gain a firmer understanding so that they can start to develop implementations or create content. This introduction is intended to give such people the basic grounding that they need.

As a very succinct introduction to the subject, Unicode is an industry standard character set encoding developed and maintained by The Unicode® Consortium. The Unicode character set is able to support over one million characters, and is being developed with an aim to have a single character set that supports all characters from all scripts, as well as many symbols, that are in common use around the world today or in the past. Currently, the Standard supports over 94,000 characters representing a large number of scripts. Unicode also defines three encoding forms, each of which is able to represent the entire character set. The three encoding forms are based on 8-, 16- and 32-bit code units, providing a flexibility that makes Unicode suitable for implementation in a wide variety of environments. The benefits of a single, universal character set and the practical considerations for implementation that have gone into the design of Unicode have made it a success, and it is well on the way to becoming a dominant and ubiquitous standard.

This section will give an introduction to the Unicode Standard and the most important concepts that implementers and users need to be familiar with. No previous in-depth knowledge of Unicode is assumed, though I expect that the reader will probably have at least seen some code charts at some point. I will be assuming familiarity with basic concepts of character set encoding, though, as described in "Character set encoding basics". In particular, I will assume familiarity with the distinction between a coded character set and a character encoding form, and with the importance of character semantics for software processes, all of which I describe in that section.

I will go into some detail on several topics, and some sections are rather technical, which should be expected given the subject matter. I have put some of the more detailed and technical discussion into appendices. They are not essential reading for the beginner, but may be useful reading as you progress in your understanding and your use of the Standard.

This introduction is not intended as a substitute for the Standard itself. The only way to get authoritative information about Unicode is to read the source documents that comprise it. The Standard consists of a rather large book, however, together with a number of other electronic documents and data files. The volume of information can be somewhat daunting for a beginner. Moreover, the Standard was not written for instructional purposes. It is technical documentation. Neither the language nor the organisation was created with didactic purposes in mind. To learn about the Standard, you want something that will guide you through the important topics and point you to other sources that can take you into more depth at appropriate points. Hopefully this introduction achieves that.

On the other hand, there are topics that this introduction does not cover in any depth, and it does not provide an authoritative statement of the details of the Standard. For example, the Unicode bi-directional algorithm is introduced in Section 8, but it is not explained here in any detail. Similarly, the conformance requirements of the Standard are described in Section 12, but this document does not provide the precise wording of each conformance requirement and of the definitions for the terms and constructs that they refer to. In both cases, that level of detail is inappropriate for an introduction.

The terms **canonical equivalence**, **canonical decomposition**, **compatibility equivalence** and **compatibility decomposition** recur frequently in many parts of this section. These are important topics that affect many aspects of the Unicode Standard. Different aspects of these issues are covered in various sections. The nature and source of canonical and compatibility equivalence is discussed in Section 6 and in “Mapping codepoints to Unicode encoding forms”, and also in Section 8. The manner and location in which decomposition mappings are specified is covered in Section 7.5. The way in which these mappings are applied in relation to Unicode normalization forms is discussed in Section 10. Finally, some of the implications are considered in Section 11. Initially, it may seem unfortunate to have the coverage of these issues spread across several sections. In fact, these issues are so involved that they take up the bulk of these sections.

Before continuing, let me briefly explain some of the notational devices that are used:

- Character names are presented in caps; e.g. LATIN SMALL LETTER A.
- Unicode characters are always referenced by their Unicode scalar value (explained in Section 3.1), which is always given in hexadecimal notation and preceded by “U+”; e.g. U+20AC. In most cases, the character name is provided as well, as in U+20AC EURO SIGN, but may be omitted if the name was given in nearby, preceding text. In many cases, a representative glyph is also shown.
- A range of Unicode characters is denoted by the starting and ending scalar values separated by “..”, e.g. U+0000..U+FFFF.
- Numbers other than Unicode scalar values are generally cited in decimal notation. In any situation in which the base for a numerical representation is unclear, the base will be shown as a subscript; e.g. 1016.
- Sequences of encoded characters are shown inside angle brackets, with characters separated by commas; e.g. < U+0045 LATIN CAPITAL LETTER E, U+0301 COMBINING ACUTE ACCENT >.
- Where glyphs for combining marks are shown in isolation, a dotted circle is used to represent a base character with which the mark would combine in order to show the relative position of the mark; e.g. “◌̄”. (Combining marks are discussed in Section 5.6 and again in Section 9.)
- Numbered versions of The Unicode Standard are usually cited as “TUS *x.y*” where *x.y* is the version number; e.g. TUS 3.1.

1 A brief history of Unicode

In order to understand Unicode, it is helpful to know a little about the history of its development. Early on in the history of computing, character set encoding standards based on 8-bit technologies became the norm. As different vendors developed their systems and began to adapt them for use in markets that used several different languages and scripts, a large number of different encoding standards resulted.

This situation led to considerable difficulty for developers and for users working with multilingual data. Products were often tied to a single encoding, which did not allow users to work with multilingual data or with data coming from incompatible systems. Developers were also required to support multiple versions of their products to serve different markets, making development and deployment for multiple markets a difficult process. In order to support data created using others’ products, developers had to support a variety of different standards for a single language. In order to work with multilingual data, they needed to support several standards simultaneously since no one standard supported more than a handful of languages. In turn, it was impossible to support multilingual data in plain text. Developing software that had anything to do with multilingual text had become incredibly difficult.

By the early 1980s, the software industry was starting to recognise the need for a solution to the problems involved with using multiple character encoding standards. Some particularly innovative work was begun at Xerox. The Xerox Star workstation used a multi-byte encoding that allowed it to support a single character set with potentially millions of characters. Using this system, they implemented a word-processing system that had support for several scripts, including Roman, Cyrillic, Greek, Arabic, Hebrew, Chinese, Korean and the Japanese kana syllabaries.[1] The work at Xerox was a direct inspiration for Unicode.

The Unicode project began in 1988, with representatives from several companies collaborating to develop a single character set encoding standard that could support all of the world’s scripts. This led to the formation of the Unicode Consortium in January of 1991, and the publication of Version 1.0 of the Unicode Standard in October of the same year (The Unicode Consortium 1991).

There were four key original design goals for Unicode:

1. To create a universal standard that covered all writing systems.
2. To use an efficient encoding that avoided mechanisms such as code page switching, shift-sequences and special states.
3. To use a uniform encoding width in which each character was encoded as a 16-bit value.
4. To create an unambiguous encoding in which any given 16-bit value always represented the same character regardless of where it occurred in the data.

How well these goals have been achieved as the Standard has developed is probably a matter of opinion. There is no question, however, that some compromises were necessary along the way. To fully understand Unicode and the compromises that were made, it is also important to understand another, related standard: ISO/IEC 10646.[2]

In 1984, a joint ISO/IEC working group was formed to begin work on an international character set standard that would support all of the world’s writing systems. This became known as the **Universal Character Set** (UCS). By 1989, drafts of the new standard were starting to get circulated.

At this point, people became aware that there were two efforts underway to achieve similar ends, those ends being a comprehensive standard that everyone could use. Of course, the last thing anybody wanted was to have two such standards. As a result, in 1991 the ISO/IEC working group and the Unicode Consortium began to discuss a merger of the two standards. There were some obstacles to overcome in negotiating a merger, however. Firstly, at this point the ISO/IEC standard was at an advanced draft stage and the Unicode 1.0 had already been published, and there were incompatibilities in the two sets. Secondly, Unicode was being designed using a uniform, 16-bit encoding form, which allowed for up to 65,536 characters, but the ISO/IEC standard used a 31-bit codespace that allowed for over 2 billion characters.

The complete details of the merger were worked out over many years, but the most important issues were resolved early on. The first step was that the repertoires in Unicode and the draft 10646 standard were aligned, and an agreement was reached that the two character sets should remain aligned. This required some changes in Unicode, including several additions, a few deletions, and the reassignment of a significant number of characters to different codepoints.[3] These changes were reflected in TUS 1.1.[4]

As of Unicode 1.1, the potential repertoire supportable by Unicode was seen to be a proper subset of that for ISO/IEC 10646. In other words, any of the 65,536 characters that might eventually be defined in Unicode would also be in ISO/IEC 10646, but it was possible that characters could be defined in ISO/IEC 10646 in the vast portion of the codespace above 64K that was unavailable to Unicode.

At some point along the way, people involved in Unicode began to recognise that 65,536 codepoints was not going to be enough to cover all of the Chinese ideographs. A solution was needed, but it would require giving up the original goal of having a uniform, 16-bit encoding form. Obviously, the solution to this problem would have an impact on the discrepancy between the two standards in terms of the potential size of their character repertoires.

The solution came in the form of a revised 16-bit encoding that both standards adopted, known as UTF-16. This encoding form would allow for support of over a million characters. ISO/IEC 10646 still had a potential capacity for far more, but it was conceded that a million characters were more than enough. Eventually, the ISO/IEC standard was formally limited to that number of assignable characters by permanently reserving the rest of the codespace.[5]

There was also an implementation issue that arose that would affect how well either standard was accepted within industry: many 8-bit processes were in place that would not be able to handle 16-bit data properly, and it was not going to be practical to change them all. Thus, there was a practical need for a second encoding form based on 8-bit code units. An 8-bit encoding form was developed, known as UTF-8, which was able to support the full potential range of characters, and which was adopted by both standards.[6]

The replacement of the original single, uniform 16-bit encoding in Unicode with UTF-16 and UTF-8 was formalised in TUS 2.0 of Unicode (The Unicode Consortium 1996). The language of that version still treated the 16-bit representation as primary, but a fundamental change had occurred. The relationship between the Unicode character set and the alternate encoding forms was clarified with the approval in 1999 of Unicode Technical Report #17 (see Whistler and Davis 2000 for the current version).

Since it was first published, ISO/IEC 10646 has supported a 32-bit encoding form, known as UCS-4. Eventually, a 32-bit encoding form was introduced for Unicode, known as UTF-32. This was formally adopted in the Standard in TUS 3.1 (Davis et al 2001).[7]

Thus, the key results of the merger have been that Unicode is now kept synchronised with ISO/IEC 10646, and it now supports three encoding forms based on 8-, 16- and 32-bit code units.

The Unicode Standard has continued to be developed up to the present, and work is still continuing with an aim to make the Standard more complete, covering more of the world's writing systems, to correct errors in details, and to make it better meet the needs of implementers. Unicode Version 3.0 was published in 2000 (The Unicode Consortium 2000), introducing over 10,000 new characters. The most current version at this time, Version 3.1, was published this year (Davis et al 2001). This version added another 44,946 new characters, bringing the total number of characters to 94,140 encoded characters.

The most important points to be learned from this history lesson relate to the relationship with ISO/IEC 10646, the size of the Unicode codespace, and the fact that there are three encoding forms for Unicode. I will describe the codespace and the encoding forms in greater detail in Sections 3 and 4. First, though, I will take a brief look at who the Unicode Consortium is and how the Standard gets developed and maintained.

2 The Unicode Consortium and the maintenance of the Unicode Standard

In this section, I will describe the Unicode Consortium and the way in which the Unicode Standard is maintained. This will include looking at the versioning system used for the Standard, and also at **Unicode Technical Reports**, a set of satellite documents that have an integral relationship to the Standard.

2.1 The Unicode Consortium

The Unicode Consortium is a not-for-profit organisation that exists to develop and promote the Unicode Standard. Anyone can be a member of the consortium, though there are different types of memberships, and not everyone gets the privilege of voting on decisions regarding the Standard. That privilege is given only to those in the category of Full Member. There are two requirements for Full Membership: this category is available only for organisational members, not to individuals; and there is an annual fee of US$12,000. At the time of writing, there are currently 21 Full Members.

General information about the Unicode Consortium is available at http://www.unicode.org/unicode/consortium/consort.html. The current list of members is available at http://www.unicode.org/unicode/consortium/memblogo.html. (A text-only list is available at http://www.unicode.org/unicode/consortium/memblist.html.)

The work of developing the Standard is done by the Unicode Technical Committee (UTC). Every Full Member organization is eligible to have a voting position on the UTC, though they are not required to participate.

There are three other categories of membership: Individual Member, Specialist Member, and Associate Member. Each of these has increasing levels of privileges. The Associate and Specialist Member categories offer the privilege of being able to participate in the regular work of the UTC through an e-mail discussion list—the “unicore” list. All members are eligible to attending meetings.

The UTC maintains a close working relationship with the corresponding body within ISO/IEC that develops and maintains ISO/IEC 10646. Any time one body considers adding new characters to the common character set, those proposals need to be evaluated by both bodies. Before any new character assignments can officially be made, approval of both bodies is required. This is how the two standards are kept in synchronization.

2.2 Versions of the Unicode Standard

A three-level versioning system is used for the Unicode Standard. Major versions (e.g. from 2.1.9 to 3.0) are used for significant additions to the Standard, and are published as a book. Minor versions (e.g. from 3.0.1 to 3.1) are used for the addition of new characters or for significant normative changes that may affect implementations, and are published as Technical Reports on the Unicode Web site (see below). The key distinction between major and minor versions is one of degree: a major version will include a number of significant additions to the Standard—enough to warrant the production of a new book. At the third level, an update (e.g. from 3.0 to 3.0.1) is used for any other important changes that can affect implementations. (Note that updates never include addition of new characters.) These are reflected primarily in the form of revised data files; there has not always been prose documentation to accompany an update of the Standard. Minor corrections can be made at any time and are published as errata on the Web site (see http://www.unicode.org/unicode/uni2errata/UnicodeErrata.html).

The Unicode Standard is embodied in the form of three types of information:

- Firstly, there is the printed version of the most recent *major* version. At present, this corresponds to TUS 3.0 (The Unicode Consortium 2000).
- Secondly, the Unicode Consortium publishes a variety of documents known as **Unicode Technical Reports** (UTRs) on its Web site. These discuss specific issues relating to implementation of the Standard. (The following section provides a general overview of UTRs.) Some of the UTRs constitute normative parts of the Standard. A UTR with this normative status is identified as a **Unicode Standard Annex** (UAX). These annexes may include documentation of a minor version release, as in the case of UAX #27 (Davis et al 2001), or documents discussing specific implementation issues, as in the case of UAX #15: Unicode Normalization Forms (Davis and Dürst 2001).
- Thirdly, the Unicode Standard includes a collection of data files that provide detailed information about semantic properties of characters in the Standard that are needed for implementations. These data files are distributed on a CD-ROM with the printed versions of the Standard, but the most up-to-date versions are always available from the Unicode Web site. These can be found online at http://www.unicode.org/Public/UNIDATA/. Further information on the data files is available at http://www.unicode.org/unicode/onlinedat/online.html.

Thus, the current version of Unicode, TUS 3.1, consists of the published book for TUS 3.0, plus the UAX that describes the minor version for TUS 3.1, UAX #27, together with the current versions of the other annexes and data files.

For more information on the various versions of the Unicode Standard, see http://www.unicode.org/unicode/standard/versions/. A complete description of all of the items that constitute part of any version of Unicode is available at http://www.unicode.org/unicode/standard/versions/enumeratedversions.html.

2.3 Unicode Technical Reports, Unicode Standard Annexes, and Unicode Technical Standards

Unicode Technical Reports are satellite documents that complement the Standard in a variety of ways. As mentioned above, some embody portions of the Standard itself. Others discuss various implementation issues that relate to Unicode.

To date, there have been a total of 23 UTRs that have been published. Seven of these, all of them written prior to TUS 3.0, have been superseded. This has happened for one of two reasons:

- UTR #1 (Daniels et al 1992), UTR #2 (Becker and Daniels 1992), and UTR #3 (Becker and McGowan 1992–1993) were proposals for adding particular scripts to the Standard and so were inherently applicable for a limited time only.
- UTR #4 (Davis 1993), UTR #5 (Davis 1991), UTR #7 (Whistler and Adams 2001) and UTR #8 (Moore 1999) have been incorporated directly into the Standard itself.

Thus, there are 16 UTRs that are still current.

Of the UTRs that are current at this time, seven are presently designated as Unicode Standard Annexes and represent normative parts of TUS 3.1. These include the following:

- UAX #9 (Davis 2001a). This describes a normative algorithm for layout and rendering of bi-directional text (see IWS-Chapter04b#bi-di Section 8}). {ACRONYM:UAX
- #11 (Freytag 2001). This specifies definitions for values of an informative property of Unicode characters that is useful when interacting with East Asian legacy implementations.
- UAX #13 (Davis 2001b). This provides guidelines for handling the various character sequences, such as **CRLF**, that are used to represent line or paragraph breaks on different platforms.
- UAX #14 (Freytag 2000). This specifies definitions for normative and informative line breaking properties of Unicode characters (see Section 7.2).
- UAX #15 (Davis and Dürst 2001). This specifies four normalised representations that can be used with Unicode-encoded text (see IWS-Chapter04b#N13n Section 10}). {ACRONYM:UAX
- #19 (Davis 2001c). This defines the UTF-32 encoding form (see Section 4.3 and Section 1 of “Mapping codepoints to Unicode encoding forms”).
- UAX #27 (Davis et al 2001). This specifies Version 3.1 of Unicode.

Some UTRs are designated to be **Unicode Technical Standards** (UTSs). These are considered subsidiary standards that complement, but are independent from it. It is not necessary for software to conform to these other standards in order to be considered conformant to the Unicode Standard. Currently, there are two UTSs.

- UTS#6 (Wolf et al 2000). This specifies a compression scheme for use in storage and transmission of Unicode-encoded text and that can be used together with other compression algorithms.
- UTS#10 (Davis and Whistler 2001a). This describes a collation algorithm that can be used with multilingual, Unicode-encoded data, and that can be tailored to provide language- or locale-specific sorting.

Other UTRs discuss various implementation issues or provide supplementary information regarding certain aspects of the Standard:

- UTR #16 (Umamaheswaran 2001). This describes an encoding form that is used by some software vendors in contexts that require interoperation with the EBCDIC series of character set encoding standards.[8]
- UTR #17 (Whistler and Davis 2000). This describes the character encoding model assumed by Unicode. (See Section 4 or "Character set encoding basics" for further discussion.)
- UTR #18 (Davis 2000a). This describes guidelines for implementing regular expression engines that support Unicode-encoded data.
- UTR #20 (Dürst and Freytag). This discusses issues involved in implementing Unicode together with document markup languages, such as XML. (See Section 13.1 for related information.)
- UTR #21 (Davis 2001d). This discusses implementation issues related to case and case mapping.
- UTR #22 (Davis 2000b). This specifies an XML format (DTD) that can be used in documenting the mapping between various legacy character encodings and Unicode.
- UTR #24 (Davis 2001e). This documents the format of a data file that describes the complete set of Unicode characters that are related to each of various scripts. (See Section 3.3 for further information.) It also describes the relationship between script names used in Unicode and script identifiers defined in the draft ISO standard, DIS 15924.

A complete list of current UTRs can be found online at http://www.unicode.org/unicode/reports/index.html. This also lists documents that are being considered as possible UTRs ("proposed draft UTRs"), as well as UTRs that are in the process of being drafted. Presently, there are no proposed draft or draft UTRs.

3 Codepoints, the Unicode codespace and script blocks

We have covered some important background on Unicode. Now we need to begin exploring the technical aspects of the Standard in more depth. In this section, I will describe the Unicode coded character set in greater detail. I will give particular focus to how the codespace and character set are organised, and will also introduce the code charts and other resources that provide information about the characters themselves.

3.1 Codepoints and the Unicode codespace

The Unicode coded character set is coded in terms of integer values, which are referred to in Unicode as **Unicode scalar values** (USVs). By convention, Unicode codepoints are represented in hexadecimal notation with a minimum of four digits and preceded with "U+"; so, for example, "U+0345", "U+10345" and "U+20345". Also by convention, any leading zeroes above four digits are suppressed; thus we would write "U+0456 **і** CYRILLIC SMALL LETTER BYELORUSSIAN-UKRAINIAN I" but not "U+03456 !!unknown USV!!".

Every character in Unicode can be uniquely identified by its codepoint, or also by its name. Unicode character names use only ASCII characters and by convention are written entirely in upper case. Characters are often referred to using both the codepoint and the name; e.g. U+0061 LATIN SMALL LETTER A. In discussions where the actual characters are unimportant or are assumed to be recognisable using only the codepoints, people will often save space and use only the codepoints. Also, in informal contexts where it is clear that Unicode codepoints are involved, people will often suppress the string "U+". I will continue to write "U+" in this document for clarity, however.

The Unicode codespace ranges from U+0000 to U+10FFFF. Borrowing terminology from ISO/IEC 10646, the codespace is described in terms of 17 **planes** of 64K codepoints each. Thus, Plane 0 includes codepoints U+0000..U+FFFF, Plane 1 includes codepoints U+10000..U+1FFFF, etc.

In the original design of Unicode, all characters were to have codepoints in the range U+0000..U+FFFF. In keeping with this, Plane 0 is set apart as the portion of the codespace in which all of the most commonly used characters are encoded, and is designated the **Basic Multilingual Plane** (BMP). The remainder of the codespace, Planes 1 to 16_{10}, are referred to collectively as the **Supplementary Planes**.[9] Up to and including TUS 3.0.1, characters were assigned only in the BMP
. In TUS 3.1, characters were assigned in the Supplementary Planes for the first time, in Planes 1, 2 and 14.

There are gaps in the Unicode codespace: codepoints that are permanently unassigned and reserved as non-characters. These include the last two codepoints in each plane, U+*n*FFFE and U+*n*FFFF (where *n* ranges from 0 to 10_{16}). These have always been reserved, and characters will never be assigned at these codepoints. One implication of this is that these codepoints are available to software developers to use for proprietary purposes in internal processing. Note, however, that care must be taken not to transmit these codepoints externally.

Unassigned codepoints can be reserved in a similar manner at any time if there is a reason for doing so. This was recently done in TUS 3.1, reserving 32 codepoints from U+FDD0..U+FDEF as non-characters. This was done specifically in order to make additional codes available to programmers to use for internal processing purposes. Again, these should never appear in data.

There is another special range of 2,048 codepoints that are reserved, creating an effective gap in the codespace.

These occupy the range U+D800..U+DFFF and are reserved due to the mechanism used in the UTF-16 encoding form (described in Section 4.1). In UTF-16, codepoints in the BMP are represented as code units having the same integer value. The code units in the range 0xD800–0xDFFF, serve a special purpose, however. These code units, known as **surrogate code units** (or simply **surrogates**), are used in representing codepoints from Planes 1 to 16. As a result, it is not possible to represent the corresponding codepoints in UTF-16. Hence, these codepoints are reserved.

The overall range U+0000..U+10FFFF includes 1,114,112 codepoints. Subtracting 66 for the non-character positions and 2,048 for the range reserved for surrogates, we find that the Unicode codespace includes 1,111,998 assignable codepoints.

3.2 Script blocks and the organisation of the Unicode character set

As has been mentioned, the Basic Multilingual Plane is intended for those characters that are most commonly used. This implies that the BMP is primarily for scripts that are currently in use, and that other planes are primarily for scripts that are not in current use. This is true to a certain extent.

As development of Unicode began, characters were first taken from existing industry standards. For the most part, those included characters used in writing modern languages, but also included a number of commonly used symbols. As these characters were assigned, they were added to the BMP. Assignments to the BMP were done in an organised manner, with some allowances for possible future additions.

The overall organisation of the BMP is illustrated in Figure 1.

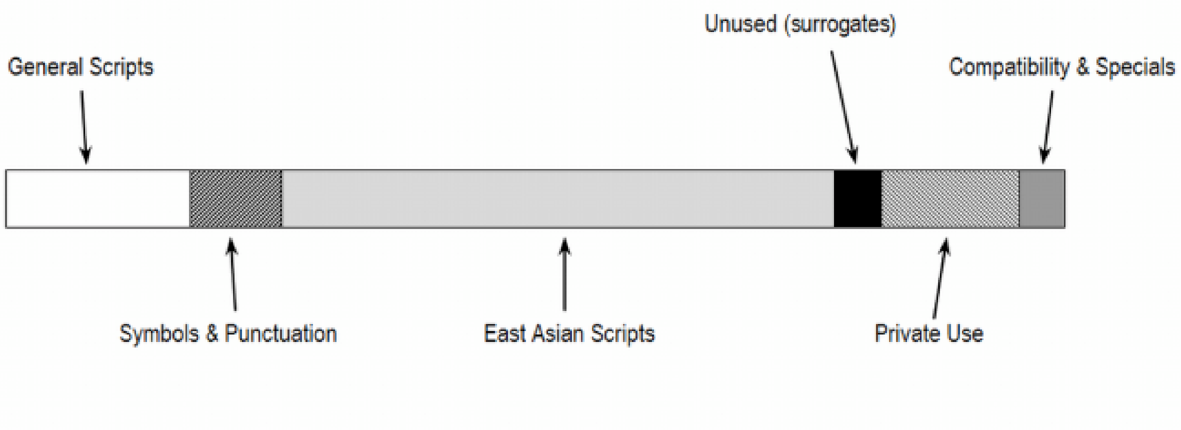


Figure 1. Organisation of the BMP

There are a couple of things to be noted straight away. Firstly, note the range of unused codepoints. This is the range U+D800..U+DFFF that is reserved to allow for the surrogates mechanism in UTF-16, as mentioned above (and described in more detail in Section 4).

Secondly, notice the range of codepoints designated "Private Use". This is a block of codepoints called the **Private Use Area** (PUA). These codepoints are permanently unassigned, and are available for custom use by users or vendors. This occupies the range U+E000..U+F8FF, giving a total of 6,400 private-use codepoints in the BMP. In addition, the last two planes, Plane 15 and Plane 16, are reserved for private use, giving an additional 131,068 codepoints. Thus, there are a total of 137,468 private-use codepoints that are available for private definition. These codepoints will never be given any fixed meaning in Unicode. Any meaning is purely by individual agreement between a sender and a receiver or within a given group of users.

Before commenting further on the characters in the BMP, let me briefly outline the organisation of the supplementary planes. As just mentioned, Planes 15 and 16 are set aside for private use. Prior to TUS 3.1, no character assignments had been made in the supplementary planes. In TUS 3.1, however, a number of characters were assigned to Planes 1, 2 and 14. Plane 1 is being used primarily for scripts that are no longer in use or for large sets of symbols used in particular fields, such as music and mathematics. Plane 2 is set aside for additional Han Chinese characters. Plane 14 is designated for special-purpose characters; for example, characters that are required for use only in certain communications protocols. No characters or specific purposes have yet been assigned to Planes 3 to 13.

In any of the planes, characters are assigned in named ranges referred to as **blocks**. Each block occupies a contiguous range of codepoints, and generally contains characters that are somehow related. Typically, a block contains characters for a given script. For example, the Thaana block occupies the range U+0780..U+07BF and contains all of the characters of the Thaana script.

In Section 2, I mentioned that the Standard includes a set of data files. One of these, Blocks.txt, lists all of the assigned blocks in Unicode, giving the name and range for each. The current version may be found at http://www.unicode.org/Public/UNIDATA/Blocks.txt.

While a given language may be written using a script for which there is a named block in Unicode, that block may not contain all of the characters needed for writing that language. Some of the characters for that language's writing system may be found in other blocks. For example, the Cyrillic block (U+0400..U+04FF) does not contain any punctuation characters. The writing system for a language such as Russian will require punctuation characters in the Basic Latin block (U+0020..U+007F)[10] as well as the General Punctuation block (U+2000..U+206F).

Also, the characters for some scripts are distributed between two or more blocks. For example, the Basic Latin block (U+0020..U+007F) and the Latin 1 Supplement block (U+00A0..U+00FF) were assigned as separate blocks because of the relationship each has to source legacy character sets. There are a number of other blocks also containing Latin characters. Thus, if you are working with a writing system based on Latin script, you may need to become familiar with all of these various blocks. Fortunately, only a limited number of scripts are broken up among multiple blocks in this manner. There is also a data file, Scripts.txt, which identifies exactly which Unicode codepoints are associated with each script. The format and contents of this file are described in UTR #24 (Davis 2001e). You are best off simply familiarising yourself with the character blocks in the Unicode character set, but it you need some help, these files are available.

Tables 1 and 2 summarise the scripts covered in the general scripts and East Asian scripts regions of the BMP. (Note that there are a large number of additional Han ideographs in Plane 2.) In addition, there are a number of blocks containing various dingbats and symbols, such as arrows, box-drawing characters, mathematical operators and Braille patterns. Apart from the Braille patterns, most symbols were taken from various source legacy standards.

Arabic	Georgian	Lao	Sinhala
Armenian	Greek	Latin	Syriac
Canadian Aboriginal Syllabics	Gujurati	Malayalam	Tamil
	Gurmukhi	Mongolian	Telugu
Cherokee	Hebrew	Myanmar (Burmese)	Tibetan
Cyrillic	IPA	Ogham	Thaana
Devanagari	Kannada	Oriya	Thai
Ethiopic	Khmer	Runic	

Table 1. Scripts in the general scripts region of the BMP (in alphabetical order)

Bopomofo	Hiragana	Katakana
Han Chinese ideographs	Kanbun	Yi and Yi radicals
Hangul (Korean)	Kang Xi radicals	

Table 2. Scripts in the Asian scripts region of the BMP (in alphabetical order)

There are currently six blocks of assigned characters in Plane 1: Old Italic, Gothic, Deseret, Byzantine Musical Symbols, Musical Symbols (for Western musical notation), and Mathematical Alphanumeric Symbols. These new blocks are described in UTR #27 (Davis et al 2001).

I will not take up space describing each of the scripts and blocks here. They are all described in the Standard. The Standard contains individual chapters (Chapters 6–13) that describe groups of related scripts. For example, Chapter 9 discusses scripts of South and Southeast Asia. If you need to learn how to implement support for a given script using Unicode, then the relevant chapter in the Standard for that script is essential reading.

3.3 Getting acquainted with Unicode characters and the code charts

In addition to the chapters in the Standard that describe different scripts, the Standard also contains a complete set of code charts, organised by block. The best way to learn about the characters in the Unicode Standard is to read the Standard and browse through its charts.

The code charts are included in the printed editions of the Standard and are also available online at http://www.unicode.org/charts/. You can also download the free UniBook™ Character Browser program, which is a very handy chart viewer. In fact, this program was originally created in order to produce the charts used in the production of the Standard. It is available at http://www.unicode.org/unibook/index.html.

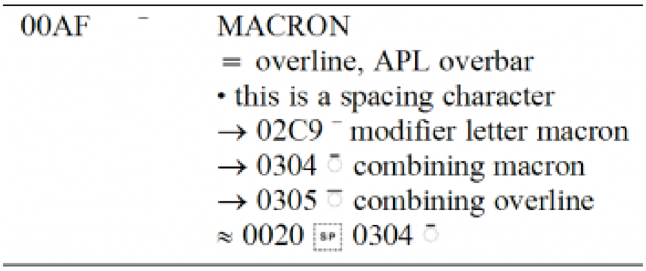
The code charts include tables of characters organised in columns of 16 rows. The column headings show all but the last hexadecimal digit of the USVs; thus, the column labelled "21D" shows glyphs for characters U+21D0..U+21DF. Within the charts, combining marks are shown with a dotted circle that represents a base character with which the mark would combine (as explained at the start of this paper). Also, unassigned codepoints are indicated by table cells that are shaded in grey or that have a diagonal lined pattern fill.

The regular code charts are organised in the numerical order of Unicode scalar values. There are also various other charts available online that are organised in different orders. In particular, there are a set of charts available at http://www.unicode.org/unicode/reports/tr24/charts/ that show all characters from all blocks for a given script, sorted in a default collating order. This can provide a useful way to find characters that you are looking for.[11] Note that these other charts do not necessarily use the same presentation as the regular code charts, such as using tables with 16 rows. Also, you will probably find it helpful to use both these charts that are organised by scripts as well as the regular charts that are organised by blocks. Because the text describing characters and scripts and the code charts in the Standard itself are organised around blocks, it is important that you not only become familiar with the individual characters used in the writing systems that you work with but also with the blocks in which they are located.

One important note about using the online charts: some are Adobe Acrobat PDF documents, which contain all of the glyphs that are used in the charts. Others are HTML pages, however, and rely on you having appropriate fonts installed in your system and on having your web browser set up to use those fonts. The Unibook program also makes use of fonts installed on your system. If you use Microsoft Office 2000 or individual applications from that suite, you may already have the Arial Unicode MS font installed on your system. This font includes glyphs for all of the characters that were assigned in TUS 2.1. Also, James Kass has made available a pair of fonts named Code2000 (shareware) and Code2001 (freeware) that include glyphs for a large portion of TUS 3.1. These are available from his web site: http://home.att.net/~jameskass/. These fonts may be useful to you in viewing some of the online charts or in using the UniBook program.[12]

Each of the regular code charts, both in the printed book and online, is accompanied by one or more pages of supporting information that is known as the **names list**. (The UniBook program gives options for viewing the charts with or without the names list.) The names list includes certain useful information regarding each character. This includes some of the normative character properties, specifically the character name and the canonical or compatibility decompositions.[13] In addition, it includes a representative glyph for each character as well as some additional notes that provide some explanation as to what this character is and how it is intended to be used.

If you take a quick glance at the names list, you will quickly note that certain special symbols are used. The organisation and presentation of the names list is fully explained in the introduction to Chapter 14 (the code charts) in TUS 3.0. As a convenience, I will briefly describe the meaning of some of the symbols. To illustrate, let us consider a sample entry:



00AF ¯ MACRON
= overline, APL overbar
• this is a spacing character
→ 02C9 ˉ modifier letter macron
→ 0304 ̄ combining macron
→ 0305 ̅ combining overline
≈ 0020 SP 0304 ̄

Figure 2. Names list entry for U+00AF MACRON

This example is useful as it contains each of the different types of basic element that may be found in an names list entry.

The first line of an entry always shows the Unicode scalar value (without the prefix “U+”), the representative glyph, and the character name.

If a character is also known by other names, these are given next on lines beginning with the equal sign “=”. If there are multiple aliases, they will appear one per line. These are generally given in lower case. In some cases, they will appear in all caps; that indicates that the alternate name was the name used in TUS 1.0, prior to the merger with ISO/IEC 10646.

Lines beginning with bullets “•” are informative notes. Generally, these are added to clarify the identity of the character. For example, the note shown above helps to prevent confusion with an over-striking (combining) macron. Informative notes may also be added to point out special properties or behaviour, such as a case mapping that might otherwise not have been expected. Notes are also often added to indicate particular languages for which the given character is used.

Lines that begin with arrows “→” are cross-references. The most common purpose of cross-references is to highlight distinct characters with which the given character might easily be confused. For example, U+00AF is similar in appearance to U+02C9, but the two are distinct. Another use of cross-references is to point to other characters that are in some linguistic relationship with the given character, such as the other member of a case pair (see, for instance, U+0272 LATIN SMALL LETTER N WITH LEFT HOOK), or a transliteration (see, for instance, U+045A CYRILLIC SMALL LETTER NJE).

Lines that begin with an equivalence sign “≡” or an approximate equality sign “≈” are used to indicate canonical and compatibility decompositions respectively.[14]

At first, you might not always remember the meaning of these symbols in the names list. If you own a copy of the book, you may want to put a tab at the beginning of Chapter 14 for quick reference.

The complete names list is available online as an ASCII-encoded plain text file (without representative glyphs for characters) at http://www.unicode.org/Public/UNIDATA/NamesList.txt. The format used for this file is documented at http://www.unicode.org/Public/UNIDATA/NamesList.html.

As you look through the code charts and the names list, bear in mind that this is not all of the information that the Standard provides about characters. It is just a limited amount that is generally adequate to establish the identity of each character. That is the main purpose they are intended for. If you need to know more about the intended use and behaviour of a particular character, you should read the section that describes the particular block containing that character (within Chapters 6–13), and also check the semantic character properties for that character in the relevant parts of the Standard.[15]

There are additional things you need to know in order to work with the characters in Unicode, particularly if you are trying to determine how the writing-system of a lesser-known language should be represented in Unicode. Before we look further at the details of the Unicode character set, however, we will explore the various encoding forms and encoding schemes used in Unicode.

4 Unicode encoding forms and encoding schemes

In Section 1, I briefly introduced the three encoding forms that are part of Unicode: UTF-8, UTF-16 and UTF-32. In this section, I will describe each of these in greater detail. (Explicit specifications are provided in “Mapping codepoints to Unicode encoding forms”.) We will also look at two encoding forms used for ISO/IEC 10646, and consider the relative merits of these various alternatives. Finally, I will describe the various encoding schemes defined as part of Unicode, and also the mechanism provided for resolving byte-order issues.

4.1 UTF-16

Because of the early history of Unicode and the original design goal to have a uniform 16-bit encoding, many people today think of Unicode as a 16-bit-only encoding. This is so even though Unicode now supports three different encoding forms, none of which is, in general, given preference over the others. UTF-16 might be considered to have a special importance, though, precisely because it is the encoding form that matches popular impressions regarding Unicode.

The original design goal of representing all characters using exactly 16 bits had two benefits. First it made processing efficient since every character was exactly the same size, and there were never any special states or escape sequences. Secondly, it made the mapping between codepoints in the coded character set and code units in the encoding form trivial: each character would be encoded using the 16-bit integer that is equal to its Unicode scalar value. Although it is no longer possible to maintain this fully in the general case, there would still be some benefit it this could be maintained in common cases. UTF-16 does this.

As mentioned earlier, the characters that are most commonly used, on average, are encoded in the Basic Multilingual Plane. Thus, for many texts it is never necessary to refer to characters above U+FFFF. If a 16-bit encoding form were used in which characters in the range U+0000..U+FFFF were encoded as 16-bit integers that matched their scalar values, this would work for such texts, but fail if any supplementary-plane characters occurred. If, however, some of the codepoints in that range were permanently reserved, perhaps they could somehow be used in some scheme to encode characters in the supplementary planes. This is precisely the purpose of the **surrogate code units** in the range 0xD800–0xDFFF.

The surrogate range covers 2,048 code values. UTF-16 divides these into two halves: 0xD800–0xDBFF are called **high surrogates**; 0xDC00–0xDFFF are called **low surrogates**. With 1,024 code values in each of these two sub-ranges, there are 1,024 x 1,024 = 1,048,576 possible combinations. This matches exactly the number of codepoints in the supplementary planes. Thus, in UTF-16, a pair of high and low surrogate code values, known as a surrogate pair, is used in this way to encode characters in the supplementary planes. Characters in the BMP are directly encoded in terms of their own 16-bit values.

So, UTF-16 is a 16-bit encoding form that encodes characters either as a single 16-bit code unit or as a pair of 16-bit code units, as follows:

Codepoint range	Number of UTF-16 code units
U+0000..U+D7FF	one
U+D800..U+DFFF	none (reserved—no characters assignable)
U+E000..U+FFFF	one
U+10000..U+10FFFF	two: one high surrogate followed by one low surrogate

Table 3. Number of UTF-16 code units per codepoint

It should be pointed out that a surrogate pair *must* consist of a high surrogate followed by a low surrogate. If an unpaired high or low surrogate is encountered in data, it is considered ill-formed, and must not be interpreted as a character.

The calculation for converting from the code values for a surrogate pair to the Unicode scalar value of the character being represented is described in Section 2 of "Mapping codepoints to Unicode encoding forms".

One of the purposes of Unicode was to make things simpler than the existing situation with legacy encodings such as the multi-byte encodings used for Far East languages. On learning that UTF-16 uses either one or two 16-bit code units, many people ask how this is any different from what was done before. There is a very significant difference in this regard between UTF-16 and legacy encodings. In the older encodings, the meaning of a code unit could be ambiguous. For example, a byte 0x73 by itself might represent the character U+0073, but it might also be the second byte in a two-byte sequence 0xA4 0x73 representing the Traditional Chinese character 山 'mountain'. In order to determine what the correct interpretation of this byte should be, it is necessary to backtrack in the data stream, possibly all the way to the beginning. In contrast, the interpretation of code units in UTF-16 is never ambiguous: when a process inspects a code unit, it is immediately clear whether the code unit is a high or low surrogate. In the worst case, if the code unit is a low surrogate, the process will need to back up one code unit to get a complete surrogate pair before it can interpret the data.

4.2 UTF-8

The UTF-8 encoding form was developed to work with existing software implementations that were designed for processing 8-bit text data. In particular, it had to work with file systems in which certain byte values had special significance. (For example, 0x2A, which is "*" in ASCII, is typically used to indicate a wildcard character). It also had to work in communication systems that assumed bytes in the range 0x00 to 0x7F (especially the control characters) were defined in conformance to certain existing standards derived from ASCII. In other words, it was necessary for Unicode characters that are also in ASCII to be encoded exactly as they would be in ASCII using code units 0x00 to 0x7F, and that those code units should never be used in the representation of any other characters.

UTF-8 uses byte sequences of one to four bytes to represent the entire Unicode codespace. The number of bytes required depends upon the range in which a codepoint lies.

The details of the mapping between codepoints and the code units that represent them is described in Section 3 of "Mapping codepoints to Unicode encoding forms". An examination of that mapping (see Table 3) reveals certain interesting properties of UTF-8 code units and sequences. Firstly, sequence-initial bytes and the non-initial bytes come from different ranges of possible values. Thus, you can immediately determine whether a UTF-8 code unit is an initial byte in a sequence or is a following byte. Secondly, the first byte in a UTF-8 sequence provides a clear indication, based on its range, as to how long the sequence is.

These two characteristics combine to make processing of UTF-8 sequences very efficient. As with UTF-16, this encoding form is far more efficient than the various legacy multi-byte encodings. The meaning of a code unit is always clear: you always know if it is a sequence-initial byte or a following byte, and you never have to backup more than three bytes in the data in order to interpret a character.

Another interesting by-product of the way UTF-8 is specified is that ASCII-encoded data automatically also conforms to UTF-8.

It should be noted that the mapping from codepoints to 8-bit code units used for UTF-8 could be misapplied so as to give more than one possible representation for a given character. The UTF-8 specification clearly limits which representations are legal and valid, however, allowing only the shortest representation. This matter is described in detail in Section 3 of "Mapping codepoints to Unicode encoding forms".

4.3 UTF-32

The UTF-32 encoding form is very simple to explain: every codepoint is encoded using a 32-bit integer equal to the scalar value of the codepoint. This is described further in Section 1 of "Mapping codepoints to Unicode encoding forms".

4.4 ISO/IEC 10646 encoding forms: UCS-4 and UCS-2

It is also useful to know about two additional encoding forms that are allowed in ISO/IEC 10646. UCS-4 is a 32-bit encoding form that supports the entire 31-bit codespace of ISO/IEC 10646. It is effectively equivalent to UTF-32, except with respect to the codespace: by definition UCS-4 can represent codepoints in the range U+0000..U+7FFFFFFF (the entire ISO/IEC 10646 codespace), whereas UTF-32 can represent only codepoints in the range U+0000..U+10FFFF (the entire Unicode codespace).[16]

UCS-2 is a 16-bit encoding form that can be used to encode only the Basic Multilingual Plane. It is essentially equivalent to UTF-16 but without surrogate pairs, and is comparable to what was available in TUS 1.1. References to UCS-2 are much less frequently encountered than was true in the past. You may still come across the term, though, so it is helpful to know. Also, it can be useful in describing the level of support for Unicode that certain software products may provide.

4.5 Which encoding is the right choice?

With three different encoding forms available, someone creating content is faced with the choice of which encoding they should use for the data they create. Likewise, software developers need to consider this question both for what they use as the internal memory representation of data and what they use when storing data on a disk or transmitting it over a wire. The answer depends on a variety of factors, including the nature of the data, the nature of the processing, and the contexts in which it will be used.

One of the original concerns people had regarding Unicode was that a 16-bit encoding form would automatically double file sizes in relation to an 8-bit encoding form.[17] Unicode's three encoding forms do differ in terms of their space efficiency, though the actual impact depends upon the range of characters being used and on the proportions of characters from different ranges within the codespace. Consider the following:

Codepoint range	Number of bytes: UTF-8	Number of bytes: UTF-16	Number of bytes: UTF-32
U+0000..U+007F	one	two	four
U+0080..U+07FF	two	two	four
U+0800..U+D7FF, U+E000..U+FFFF	three	two	four
U+10000..U+10FFFF	four	four	four

Table 4. Bytes required to represent a character in each encoding form

Clearly, UTF-32 is less efficient, unless a large proportion of characters in the data come from the supplementary planes, which is usually not likely. (For supplementary-plane characters, all three encoding forms are equal, requiring four bytes.) For characters in the Basic Latin block of Unicode (equivalent to the ASCII character set), i.e. U+0000..U+007F, UTF-8 is clearly the most efficient. On the other hand, for characters in the BMP used for Far East languages (all are in the range U+2E80..U+FFEF), UTF-8 is less efficient than UTF-16.

Another factor particularly for software developers to consider is efficiency in processing. UTF-32 has an advantage in that every character is exactly the same size, and there is never a need to test the value of a code unit to determine whether or not it is part of a sequence. Of course, this has to be weighed against considerations of the overall size of data, for which UTF-32 is generally quite inefficient. Also, while UTF-32 may allow for more efficient processing than UTF-16 or UTF-8, it should be noted that none of the three encoding forms is particularly inefficient with respect to processing. Certainly, it is true that all of them are much more efficient than are the various legacy multibyte encodings.

For general use with data that includes a variety of characters mostly from the BMP, UTF-16 is a good choice for software developers. BMP characters are all encoded as 16-bits, and testing for surrogates can be done very quickly. In terms of storage, it provides a good balance for multilingual data that may include characters from a variety of scripts in the BMP, and is no less efficient than other encoding forms for supplementary-plane characters. For these reasons, many applications that support Unicode use UTF-16 as the primary encoding form.

There are certain situations in which one of the other encoding forms may be preferred, however. In situations in which a software process needs to handle a single character (for example, to pass a character generated by a keyboard driver to an application), it is simplest to handle a single UTF-32 code unit. On the other hand, in situations in which software has to cooperate with existing implementations that were designed for 8-bit data only, then UTF-8 may be a necessity. UTF-8 has been most heavily used in the context of the Internet for this reason.

On first consideration, it may appear that having three encoding forms would be less desirable. In fact, having three encoding forms based on 8-, 16- and 32-bit code units has provided considerable flexibility for developers and has made it possible to begin making a transition to Unicode while maintaining operability with existing implementations. This has been a key factor in making Unicode a success within industry.

There is another related question worth considering here: Given a particular software product, which encoding forms does it support? Some software may be able to handle "16-bit" Unicode data. Note, however, that this may actually mean UCS-2 data and not UTF-16; in other words, it is able to handle characters in the BMP, but not supplementary-plane characters encoded as surrogate pairs. For example, Microsoft Word 97 handles 16-bit Unicode data fairly well, but knows nothing about surrogates. (If you insert a surrogate pair into a Word 97 document, it will simply assume these are two characters having unknown properties.) Likewise, Unicode support in Microsoft Windows 95/98/Me/NT has actually been support for UCS-2; Windows 2000 is the first version that has any support for supplementary-plane characters.

The question of support for supplementary-plane characters does not necessarily apply only to UTF-16. For example, many Web browsers are able to interpret HTML pages encoded in UTF-8, but that does not necessarily mean that they can handle supplementary-plane characters. For example, the software may convert data in the incoming file into 16-bit code units for internal processing, and that processing may not have been written to deal with surrogates correctly. Or, that application may have been written with proper support for supplementary-plane characters, but may depend on the host operating system for certain processing, and the host operating system on a given installation may not have the necessary support.

In general, when choosing software, you should verify whether it supports the encoding forms you would like to use. For both UTF-8 and UTF-16, you should explicitly verify whether the software is able to support supplementary-plane characters, if that is important to you. Until recently, many developers were ignoring supplementary-plane characters since none had been assigned within Unicode. This has changed as of TUS 3.1, however. Hence, we are likely to start seeing full support for UTF-16 and for supplementary-plane characters in a growing number of products.

4.6 Byte order: Unicode encoding schemes

As explained in "Character set encoding basics", 16- and 32-bit encoding forms raise an issue in relation to byte ordering. While code units may be larger than 8-bits, many processes are designed to treat data in 8-bit chunks at some level. For example, a communication system may handle data in terms of bytes, and certainly memory addressing with personal computers is organised in terms of bytes. Because of this, when 16- or 32-bit code units are involved, these may get handled as a set of bytes, and these bytes must get put into a serial order before being transmitted over a wire or stored on a disk.

There are two ways to order the bytes that make up a 16- or 32-bit code unit. One is to start with the high-order (most significant) byte and end with the low-order (least significant) byte. This is often referred to as **big-endian**. The other way, of course, is the opposite, and is often referred to as **little-endian**.[18] For 16- and 32-bit encoding forms, the specification of a particular encoding form together with a particular byte order is known as a **character encoding scheme**.

In addition to defining particular encoding forms as part of the Standard, Unicode also specifies particular encoding schemes. Before explaining these, however, I need to make a distinction between the actual form in which the data is organised (what it really is) versus how a process might describe the data (what gets said about it).

Clearly, for data in the UTF-16 encoding form, it can only be serialised in one of two ways. In terms of how it is actually organised, it must be either big-endian or little-endian. However, Unicode allows three ways in which the encoding scheme for the data can be described: big-endian, little-endian, or unspecified-endian. The same is true for UTF-32.

Thus, Unicode defines a total of seven encoding schemes:

- UTF-8[19]
- UTF-16BE
- UTF-16LE
- UTF-16
- UTF-32BE
- UTF-32LE
- UTF-32

Note that the labels "UTF-8", "UTF-16" and "UTF-32" can be used in two ways: either as encoding form designations or as encoding scheme designations. In most situations, it is either clear or irrelevant which is meant. There may be situations in which you need to clarify which was meant, however.

Before a software process can interpret data encoded using the UTF-16 or UTF-32 encoding forms, the question of byte order does need to be resolved. Clearly, then, it is always preferable to tag data using an encoding scheme designation that overtly indicates which byte order is used. As Unicode was being developed, however, it was apparent that there would be situations in which existing implementation did not provide a means to indicate the byte order. Therefore the ambiguous encoding scheme designations "UTF-16" and "UTF-32" were considered necessary.

When the ambiguous designators are applied, however, the question of byte order still has to be resolved before a process can interpret the data. One possibility is simply to assume one byte order, begin reading the data and then check to see if it appears to make sense. For example, if the data were switching from one script to another with each new character, you might suspect that it was not being interpreted correctly. This approach is not necessarily reliable, though some software vendors have developed algorithms that try to detect the byte order, and even the encoding form, and these algorithms work in most situations.

To solve this problem, the codepoint U+FEFF was designated to be a **byte order mark** (BOM). When encountered at the start of a file or data stream, this character can always make clear which byte order is being used. The reason is that the codepoint that would correspond to the opposite byte order, U+FFFE, is reserved as a non-character.

For example, consider a file containing the Thai text "ความจริง". The first character "ค" THAI CHARACTER KHO KHWAI has a codepoint of U+0E04. Now, suppose that the file is encoded in UTF-16 and is stored in big-endian order, though the encoding scheme is identified ambiguously as "UTF-16". Suppose, then, that an application begins to read the file. It encounters the byte sequence 0x0E 0x04, but has no way to determine whether to assume big-endian order or little-endian order. If it assumes big-endian order, it interprets these bytes as U+0E04 THAI CHARACTER KHO KHWAI; but if it assumes little-endian order, it interprets these bytes as U+040E CYRILLIC CAPITAL LETTER SHORT U. Only one of these interpretations is correct, but the software has no way to know which.

But suppose the byte order mark, U+FEFF, is placed at the start of the file. Thus, the first four bytes in sequence are 0xFE 0xFF 0x0E 0x04. Now, if the software attempts to interpret the first two bytes in little-endian order, it interprets them as U+FFFE. But that is a non-character and, therefore, not a possible interpretation. Thus, the software knows that it must assume big-endian order. Now it interprets the first four bytes as U+FEFF (the byte-order mark) and U+0E04 THAI CHARACTER KHO KHWAI, and it is assured of the correct interpretation.

It should be pointed out that the codepoint U+FEFF has a second interpretation: ZERO WIDTH NO-BREAK SPACE. Unicode specifies that if data is identified as being in the UTF-16 or UTF-32 encoding *scheme* (not *form*) so that the byte order is ambiguous, then the data should begin with U+FEFF and that it should be interpreted as a byte order mark and not considered part of the content. If the byte order is stated explicitly, using an encoding scheme designation such as UTF-16LE or UTF-32BE, then the data should not begin with a byte order mark. It may begin with the character U+FEFF, but if so it should be interpreted as a ZERO WIDTH NO-BREAK SPACE and counted as part of the content.

The use of the BOM works in exactly the same way for UTF-32, except that the BOM is encoded as four bytes rather than two.

Note that the BOM is useful for data stored in files or being transmitted, but it is not needed for data in internal memory or passed through software programming interfaces. In those contexts, a specific byte order will generally be assumed.[20]

The byte order mark is often considered to have another benefit aside from specifying byte order: that of identifying the character encoding. In most if not all existing legacy encoding standards, the byte sequences 0xFE 0xFF and 0xFF 0xFE are extremely unlikely. Thus, if a file begins with this value, software can infer with a high level of confidence that the data is Unicode, and also be able to deduce the encoding form. This also applies for UTF-32, though in that case the byte sequences would be 0x00 0x00 0xFE 0xFF and 0xFF 0xFE 0x00 0x00. It is also applicable in the case of UTF-8. In that case, the encoded representation of U+FEFF is 0xEF 0xBB 0xBF.

When the BOM is used in this way to identify the character set encoding of the data, it is referred to as an **encoding signature**.

5 Design principles for the Unicode character set

We have taken a careful look at how Unicode characters are encoded. We have not looked as closely at the characters themselves, however. For many situations in which people are implementating Unicode, it is familiarity with the characters and an understanding of the design of the character set that matters. Hence, we will turn to focus on that now.

Several design principles were applied in the development of the Unicode character set. These represent ideals. One of the principles was in conflict with some of the others, however. In order to make Unicode practical within the context of existing implementations, there was a requirement involving backward compatibility that caused many of the ideals to be violated numerous times. In the remainder of this section, I will describe the ideal design principles. Then in Section 6 we will look at why and how many of these ideals have not always been upheld.

5.1 Unicode encodes plain text

A lot of the text data we work with is formatted text created within a word-processing or desktop publishing application. Such text includes the character information of the content, as well as other information specifying the formatting parameters or the document structure. The formatting parameters determine such things as fonts, paragraph alignments and margins; the document structure identifies things such as sections, pages and running headers. This kind of text is known as **rich text** or **fancy text**. It stands in contrast with text data that does not include such additional information but consists of the character information of the content only. The latter is known as **plain text**. Unicode encodes plain text.

Some fancy text data uses text-based markup schemes such as XML or HTML. These use character sequences as mechanisms for controlling formatting or for delimiting higher-level structures as part of the document encoding. For instance, the title of a document might be identified within the structure of a document by using string-based tags <title> and </title> before and after the title content.

Text-based mark-up schemes blur the distinction between plain and fancy text. For example, whereas a file format used by a word-processing application will usually use binary (i.e. non-character) data, an HTML file contains only character data. Not all of the character data is intended to be exposed to users, but it could be read as text if it were exposed to users. Such a file, in a sense, has a dual interpretation. It can be viewed as a plain text file in the sense that the entire content is human-readable character data. But there is a clear difference between the data that is intended for viewing by users and data that is not.

Generally, it is most useful to consider only the content of such files to be plain text. Whether only the content or the entire file is considered plain text, however, the data is character data, and as such is data to which Unicode is applicable. That is, both the content and the textual markup can be encoded in terms of Unicode. Note, however, that Unicode applies to the markup only to the extent to which it is considered text. Unicode says nothing about higher-level protocols for specifying document structure or formatting.

The relationship between Unicode and higher-level protocols is discussed further in Section 13.1.

5.2 Unification

One of the design principles of Unicode was to avoid unnecessary and artificial duplication of characters by unifying characters across languages within a given script. For example, even though the English letter *b* and the Spanish letter *be grande* are part of different writing systems, they have the same shape, function and behaviour and so can be considered a single character of the Latin script. Thus, Unicode encodes a single Latin script character, U+0062 LATIN SMALL LETTER B, which is used in hundreds of different writing systems around the world.

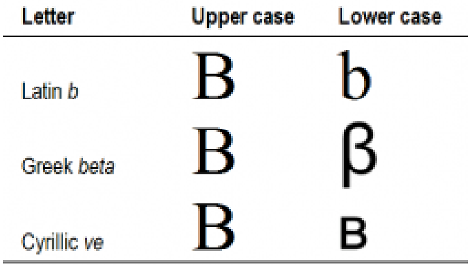
The principle of unification has been particularly important in relation to Han ideographs used in Chinese, Japanese, Korean and Vietnamese writing systems. There are variations between characters as used for these different writing systems, but wherever shapes are historically related (i.e. are cognate) and have the same basic structure, then they can, in principle, be unified.

Unification can apply even in cases where one might initially consider a distinct function to be involved. For instance, Latin and Greek letters have often been used in technical domains to denote various units of measurement, such as “mA” for *milliamperes*. These letters do not behave differently in relation to text processes, however, when used with this different function. In practical terms, there is no need to make a character distinction, and so unification can apply.

Such unification makes sense in relation to users. Consider the problems that would occur, for instance, if there were distinct characters “A” and “m” to be used to represent *ampere* and *milli*. Many users would not be aware of the difference, and two types of problems would result. Firstly, a user might search in existing data for “mA” using the regular Latin letters and would be frustrated when the results do not include the text they expect. Secondly, when creating data, some users will enter “mA” using the regular Latin letters. The result will be inconsistent data in which the regular Latin letters and the special characters for technical usage are both used. Unification of characters across these different functions avoids these problems.

Not all things that look the same get unified, however. As mentioned, characters are unified across languages, but not across scripts. For instance, Latin capital *B*, Greek capital *beta* and Cyrillic capital *ve* look the same[21] and are historically related, but they are not unified in Unicode.

If there were a situation in which it might have made sense to unify across scripts, this is one of a few such cases. Overall, however, it is much simpler both to develop and to implement the standard to handle each script separately. For example, sorting data containing text in multiple scripts is going to be much simpler if you do not have to design a process to judge whether a string such as “CAB” should be considered Latin, Cyrillic or Greek. Also, these characters may behave differently for some processes. For example, in case mapping, these three characters map to three clearly distinct lower case letters:



Letter	Upper case	Lower case
Latin *b*	B	b
Greek *beta*	Β	β
Cyrillic *ve*	В	в

Figure 3. Different lower case mappings

Unification across scripts does happen, however, for general punctuation characters that are used across scripts. Some punctuation characters are specific to a particular script, such as U+0700 SYRIAC END OF PARAGRAPH. In these cases, the punctuation character is encoded within the range for that script. Many writing systems from several scripts use a common set of punctuation, however, including the comma, period and quotation marks. These are unified in Unicode. Most are in the Basic Latin and Latin-1 Supplement ranges (U+0020..U+007F, U+00A0..U+00FF), and others are in the General Punctuation block (U+2000..U+206F).

5.3 Logical order

Unicode stores characters in logical order, meaning roughly “in reading order”. In most mono-directional text, what this means is self evident, whether the script used in that text is written from left-to-right, right-to-left, or vertically. It may be less obvious in cases of mixed-direction text, however. There are also cases of mono-directional scripts in which the visual order of characters does not match their logical order. I will describe each of these two situations in greater detail.

Firstly, for mixed-direction text, the main point to bear in mind is that the text is stored from beginning to end in reading order, regardless of visual order. This is illustrated in Figure 4:

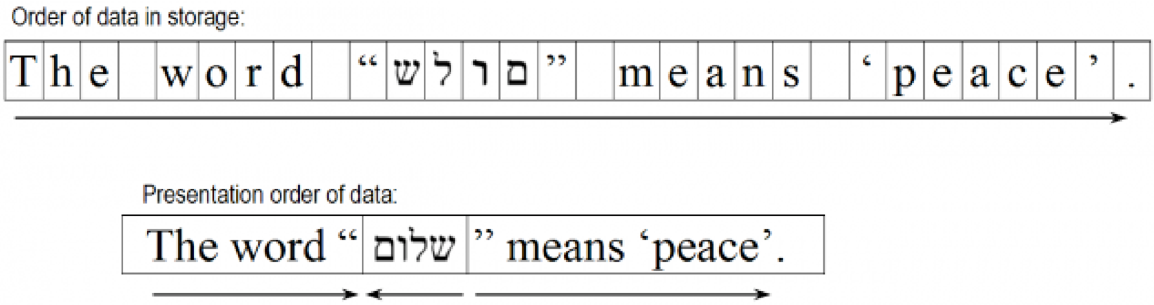

Order of data in storage:
T h e w o r d “ ש ל ו ם ” m e a n s ‘ p e a c e ’ .
Presentation order of data:
The word “ שלום ” means ‘peace’.

Figure 4. Text stored in logical order

Logical order also applies to numbers occurring in text, if numbers are thought of as starting with the most-significant digit. Thus, the order in which numbers are stored within Hebrew text does not match the right-to-left visual order:

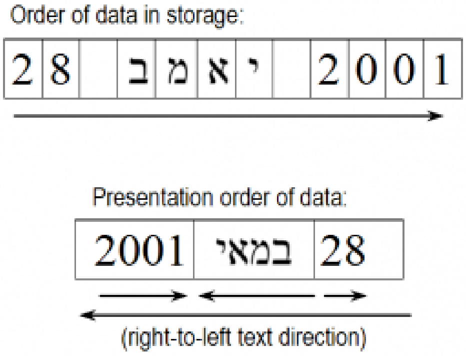
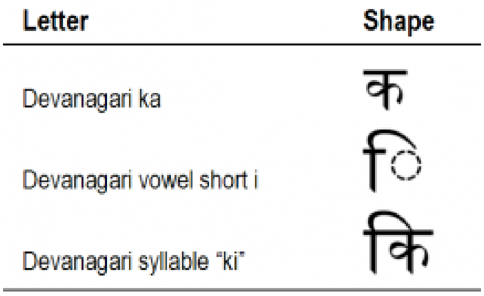

Order of data in storage:
2 8 ב מ א י 2 0 0 1
Presentation order of data:
2001 במאי 28
(right-to-left text direction)

Figure 5. Left-to-right numbers in right-to-left text

Since Unicode stores data in logical order, software that supports mixed-direction text needs to handle the visual order of text as part of the text rendering process. As part of the Standard, Unicode includes a specification of how horizontal, bi-directional text should be processed. This will be discussed further in Section 8.

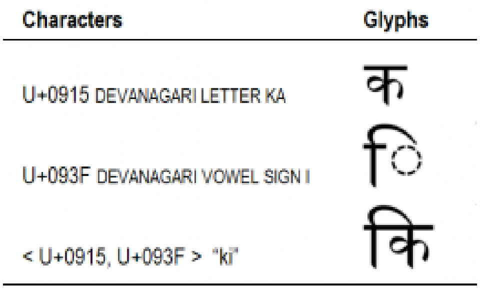
The second situation in which logical order is relevant pertains to the Indic scripts of South and Southeast Asia. One characteristic feature of these scripts is that certain vowels are written before a consonant but pronounced after it:



Letter	Shape
Devanagari ka	क
Devanagari vowel short i	ि◌
Devanagari syllable "ki"	कि

Figure 6. Devanagari vowel written to left of consonant

In these cases, the storage order of Unicode characters is in their logical (reading) order:



Characters	Glyphs
U+0915 DEVANAGARI LETTER KA	क
U+093F DEVANAGARI VOWEL SIGN I	ि◌
< U+0915, U+093F > "ki"	कि

Figure 7. Devanagari vowel stored in logical order

Among the Indic scripts, Thai and Lao are exceptions to this. Because of existing practices, Thai and Lao vowels that are written to the left of the consonant are stored in their visual order.

5.4 Characters, not glyphs

A fundamental principle in the design of the Unicode character set is the distinction between **characters** and **glyphs**.[22] Within the domain of information technology, the term "character", or "abstract character", is used to refer to an element of textual information. The emphasis is on abstract information content, and has nothing to do with how information is presented. The term "glyph", on the other hand, has only to do with presentation. Unicode defines these two terms as follows:

> **Abstract character**. A unit of information used for the organization, control, or representation of textual data.
>
> **Glyph**. (1) An abstract form that represents one or more glyph images. (2) A synonym for glyph image.

The term *glyph image*, in turn, is defined as follows:

> **Glyph image**. The actual, concrete image of a glyph representation having been rasterised or otherwise imaged onto some display surface.

So, a character is a piece of textual information, and a glyph (or glyph image) is a graphical object used to display a character.

Now, it may seem like this is nothing but terminological nit-picking, and that we could simplify our discussions by simply talking about characters. That would assume a one-to-one relationship between characters and glyphs, however, and it is that very assumption which Unicode rejects. Let me give some examples to illustrate this.

First of all, there is the obvious issue that different typefaces can be used to present a given character using different glyphs. For example, U+0041 LATIN CAPITAL LETTER A can be represented using a wide variety of graphic shapes:

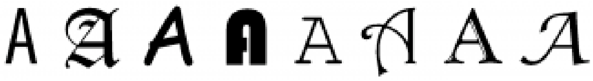
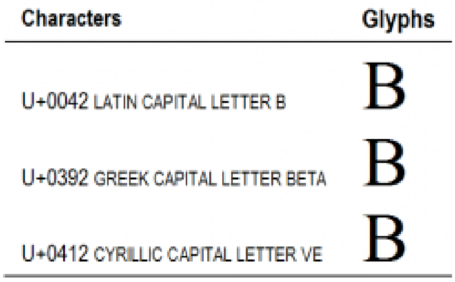


Figure 8. One character, multiple glyphs

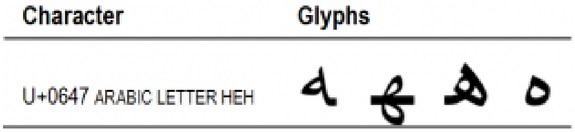
The opposite can also apply: it may be possible for distinct characters to be represented using a single glyph.



Characters	Glyphs
U+0042 LATIN CAPITAL LETTER B	B
U+0392 GREEK CAPITAL LETTER BETA	B
U+0412 CYRILLIC CAPITAL LETTER VE	B

Figure 9. Multiple characters, one glyph

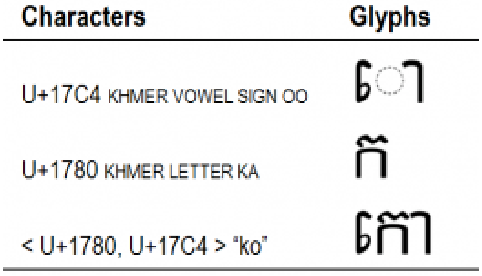
The more interesting cases of mismatch between characters and glyphs relate to the writing behaviours associated with different scripts. The process of presenting characters from many scripts often requires a mismatch between characters and glyphs. So, for example, different instances of a single character may need to be presented using alternate glyphs according to their contexts. This can be illustrated from Arabic script: because Arabic script is written cursively, the shape of a character can change according to the attachments required:



Character	Glyphs
U+0647 ARABIC LETTER HEH	ـه ـهـ هـ ه

Figure 10. One character, contextually-determined glyphs

In some cases, a character may need to be presented using multiple glyphs, regardless of context. This would be the case, for example, with U+17C4 KHMER VOWEL SIGN OO. This character is presented using a combination of two shapes that are written on either side of the symbol representing the syllable-initial consonant:

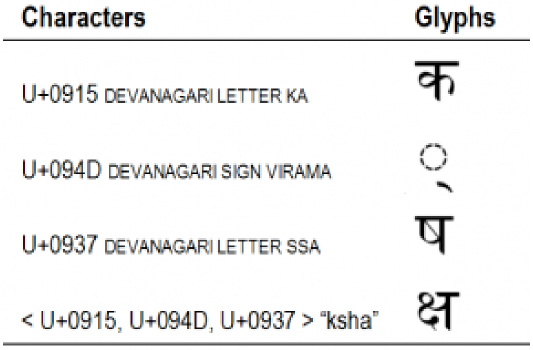


Characters	Glyphs
U+17C4 KHMER VOWEL SIGN OO	◌ោ
U+1780 KHMER LETTER KA	ក
< U+1780, U+17C4 > "ko"	កោ

Figure 11. Single character requiring multiple glyphs

In the process of rendering a character sequence < U+1780, U+17C4 >, the single character U+17C4 would normally be translated in the domain of glyphs into a pair of glyphs that can be positioned before and after the glyph for the consonant.

Finally, there are cases in which the opposite also occurs: a sequence of characters may correspond to a single glyph. Such glyphs are referred to as **ligatures** or (particularly in relation to Indic scripts) **conjuncts**. Many-to-one character to glyph relationships can be illustrated by a Devanagari conjunct:



Characters	Glyphs
U+0915 DEVANAGARI LETTER KA	क
U+094D DEVANAGARI SIGN VIRAMA	◌्
U+0937 DEVANAGARI LETTER SSA	ष
< U+0915, U+094D, U+0937 > "ksha"	क्ष

Figure 12. Multiple characters rendered as one glyph

Characters and glyphs belong within separate domains within a text-processing system: characters are part of the *text-encoding* domain, while glyphs are part of the distinct domain of *rendering*. In the process of presenting data, sequences of characters need to be translated into sequences of positioned glyphs, but the relationship between characters and glyphs is, in general, many-to-many. The Unicode Standard assumes that it will be used in conjunction with software implementations that are able to handle the many-to-many mapping from characters to glyphs. This is handled in modern software using specialised technologies for this purpose. Such technologies are often referred to as *smart-font rendering* or *complex-script rendering*. These are discussed in greater detail in Guidelines for Writing System Support: Technical Details: Smart Rendering: Part 3 and in Rendering technologies overview.

5.5 Characters, not graphemes

As users work with text, they are likely to think in terms of minimal units that correspond to the "letters of the alphabet" for their particular language; that is, the units that are enumerated in that language's orthography. The functional units within orthographies are often known as **graphemes**. It is important to note that Unicode encodes characters, not graphemes.

For example, in the orthographies of many African languages, the digraph "gb" functions as a unit: it represents a phoneme, and is listed separately in the sort order (for instance, "gb" might sort after "gu" and before "ha"). In such orthographies, "gb" would be considered a grapheme. Unicode does not assign a codepoint for graphemes such as this, however. Rather, Unicode assumes that they can be represented in the computer as a sequence of characters. In this case, "gb" would be encoded in Unicode using the sequence of characters U+0067 LATIN SMALL LETTER G and U+0062 LATIN SMALL LETTER B. It is then up to software developers to design processes such as sorting with an understanding of the rules for the given language. In that way, the software emulates the behaviour expected by the user, even though the characters in the encoded representation of the data do not directly reflect the units that the user is assuming.

Note that the character sequences that need to be interpreted as a single unit vary from one writing system to another. For instance, the accented vowel "ô" may be considered a unit in some writing systems, as in the case of French. For other writing systems, however, the accent may be thought of as an independent unit. This might be a more appropriate interpretation in the case of the writing system for a tonal language, for example, such as the languages of West Africa.

Also, the sequences that need to be processed as units may be different from one process to another within a single language. For example, the combination "ck" in German functions as a unit requiring special processing in relation to hyphenation (when hyphenated, it becomes "k-k"), but not in relation to sorting.

In general terms, processes for manipulating text data operate in terms of **text elements**:

> **Text element**. A minimum unit of text in relation to a particular text process, in the context of a given writing system.

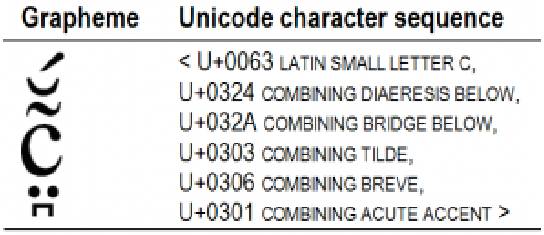
Note that the definition of a text element depends both upon a given process and a given writing system. The key point in relation to Unicode is that Unicode assumes that the mapping between characters and text elements is, in general, many-to-many.

5.6 Dynamic composition

Unicode distinguishes between two general classes of graphic characters: **base characters** and **combining marks**. Combining marks are characters that are typographically dependent upon other characters; for example, the acute and grave accent marks. Base characters are graphic characters that are *not* dependent on other characters in this way, and thus do not combine.

Using these two classes, Unicode applies a principle of dynamic composition by which combining marks can be productively and recursively applied to base characters in arbitrary combinations to dynamically create new text elements. In practice, what this means is that a text element may be encoded as a sequence of a base character followed by one or more combining marks. Such a sequence is known as a **combining character sequence**. The benefit of this dynamic composition is that it enables the character set to support a wide variety of graphemes that might occur in writing systems around the world using a reasonably limited character repertoire. This avoids having to assign a large number of characters for specific combinations, and it automatically provides support for combinations that have not been anticipated in advance.

A good illustration for dynamic composition would be to imagine a hypothetical grapheme consisting of a base character with numerous diacritics, for instance "c" with a tilde, breve and acute above as well as a diaeresis and bridge below. This unlikely combination is readily represented in Unicode:



Grapheme	Unicode character sequence
	< U+0063 LATIN SMALL LETTER C, U+0324 COMBINING DIAERESIS BELOW, U+032A COMBINING BRIDGE BELOW, U+0303 COMBINING TILDE, U+0306 COMBINING BREVE, U+0301 COMBINING ACUTE ACCENT >

Figure 13. Dynamically composed character sequence

For some graphemes, a user might suppose that they must correspond to a single character due to the way the graphic elements that are used to present the character interact. For instance, it might not occur to someone that the Latin c-cedilla "ç" could correspond to more than one encoded character. Unicode has a rich set of combining marks, however, including some that may normally be presented as graphically contiguous with the base character. Typographically, the best results might be obtained by presenting the combination using a single, pre-composed glyph. That has no bearing on the encoded representation of the data, however: as seen in Section 5.4, characters and glyphs do not correspond in a one-to-one manner, and Unicode assumes that software is able to handle rendering complexities that include cases such as this.

There are combining marks in many Unicode blocks, though one block is dedicated entirely to combining marks: the Combining Diacritical Marks block (U+0300..U+036F). We will revisit the topic of combining marks again in Section 9.

Note

To complete this article go to: Understanding Unicode™: A general introduction to the Unicode Standard (Sections 6-15) .

[1] For a fuller discussion of the multilingual capabilities of the Xerox Star system, see Becker (1984). This article is also an excellent introduction to some of the basic issues in working with multilingual text on any system.

[2] The abbreviations ISO and IEC stand for *The International Organization for Standardization* and *The International Electrotechnical Commission*. Each of these organizations is responsible for the development of international standards. They collaborate in the development of standards for information technology.

[3] One of the fundamental rules of character encoding standards is that, once a character has been assigned to a given codepoint, it is never deleted or re-assigned. Thus, these changes in Unicode represented an unprecedented step. This will never be repeated in Unicode.

[4] Version 1.1 of Unicode is formally described in Davis (1993). This document is not easily accessible, however. The changes in Unicode required for the merger are described in Volume 2 of Unicode 1.0 (The Unicode Consortium 1992).

[5] This decision, which represented the last major area of difference between the two standards, was not made until the year 2000.

[6] There is a slight difference between Unicode and ISO/IEC10646 in how they define UTF-8: the ISO/IEC definition supports the larger

codespace of that standard. This is described in "Mapping codepoints to Unicode encoding forms".

7 There is a noteworthy distinction between UTF-32 and UCS-4^: UTF-32 is limited to Unicode's codespace of a little over a million characters, while UCS-4 supports the full 31-bit codespace of ISO/IEC10646. For practical purposes, though, this distinction is not really important.

8 Note that UTF-EBCDIC is not a part of the Unicode Standard. This UTR merely documents a particular vendor implementation used in connection with Unicode.

9 The planes in Unicode are most often referred to using decimal notation rather than hexadecimal. I will continue to use decimal notation in referring to planes, and will suppress the base indicator. Thus, "Plane 10" will mean the tenth supplementary plane.

10 The Basic Latin block is identical to the set of graphic characters in ASCII.

11 Note, however, that these charts will also show certain characters known as **compatibility characters** that you may not want to use in your data. The issues related to this are the topic of several sections, in particular Sections 6 and 11, and also "A review of characters with compatibility decompositions". You will need to understand some of the material in prior sections first, however.

12 Of course, fonts that cover a wide range of Unicode characters may also be available from commercial font vendors. For example, Agfa Monotype have fonts in their library containing glyphs for all of the characters in TUS 3.0.

13 Character properties in general are discussed in Section 7. Character decompositions in particular are discussed in sections 6, 7.5 and 10. For the moment, all that you need to know about these is that normative properties are part of the specification of the Standard that must be followed in conformant implementations, as opposed to informative properties that are merely provided to guide implementers but that may not be applicable in all situations and that do not have to be followed.

14 Different aspects of canonical and compatibility decompositions are discussed in section 6, section 7.5 and section 10.

15 See section 7 for details on semantic character properties.

16 Note that UTF-32 is defined in Unicode, but is not part of the ISO/IEC 10646 standard. It isn't really needed in that standard since UCS-4 is already defined. On the other hand, UCS-4 was not appropriate in the context of Unicode since it permits representation of codepoints that are outside the Unicode codespace.

17 This concern was raised in relation to single-byte legacy encodings. It clearly doesn't apply for multi-byte encodings.

18 I found it very difficult to remember which was big-endian and which was little-endian until I learned the etymology of these terms. They were based on *Gulliver's Travels*, by Jonathan Swift. In the course of his travels, Gulliver comes to a land in which the residents are divided into two camps based on how they eat their hard-boiled eggs: big end first or little end first.

19 For UTF-8, byte order is not an issue since the code units are already bytes. Therefore BE and LE variants are not necessary.

20 Of course, there may be exceptions within a set of programming interfaces; for example, in functions for reading or writing to disk that deal with byte order issues.

21 These characters look the same within certain typeface limitations. For instance, a Fraktur design is appropriate for Latin *b* but not for Greek *beta*.

22 For additional information, this distinction is discussed in section 2.2 of TUS 3.0, as well as in "Understanding characters, keystrokes, codepoints and glyphs". It is also the topic of an ISO/IEC technical report, ISO/IEC TR15285 (ISO/IEC 1998).

Backlinks (20 most popular; affiliated sites and popular search engines removed)

http://alumnae.mills.edu/horde/imp/message.php?index=2268

Add a response to this article

Note: the opinions expressed in submitted contributions below do not necessarily reflect the opinions of our website.

Provided by SIL's Non-Roman Script Initiative. Contact us at nrsi@sil.org.