

# Guidelines for Writing System Support

*Victor Gaultney (Editor),  
SIL Non-Roman Script Initiative (NRSI)  
2003-10-31*

## Table of Contents

|   |           |
|---|-----------|
| <b>Section 1 Components of a Writing System Implementation.....</b> | <b>4</b>  |
| 1.1 Writing system implementations.....                             | 4         |
| 1.2 Goals for writing system support.....                           | 5         |
| 1.2.1 Minimal — type, see, print, store.....                        | 5         |
| 1.2.2 Reasonable — interchange, process, analysis.....              | 5         |
| 1.2.3 Ideal — parity.....   | 6         |
| 1.2.4 Completeness.....   | 6         |
| 1.2.5 Accessibility.....  | 6         |
| 1.3 Components of a WSI — the SIL NRSI Model.....                   | 6         |
| 1.3.1 Encoding.....   | 7         |
| 1.3.2 Input.....  | 7         |
| 1.3.3 Rendering.....  | 8         |
| 1.3.4 Analysis.....   | 8         |
| 1.3.5 Conversion.....   | 9         |
| 1.4 WSI example.....  | 9         |
| 1.5 Summary.....  | 10        |
| 1.6 References.....   | 10        |
| <b>Section 2 The Process of WSI Development.....</b>                | <b>11</b> |
| 2.1 Determine needs.....  | 11        |
| 2.1.1 What do users need to do?.....                                | 11        |
| 2.1.2 What components are needed?.....                              | 11        |
| 2.1.3 What information is needed?.....                              | 12        |
| 2.1.4 What model of distribution is appropriate?.....               | 12        |
| 2.2 Identify and apply development resources.....                   | 12        |
| 2.3 Develop components.....   | 13        |
| 2.4 Gather components into solutions.....                           | 14        |
| 2.4.1 Integration.....  | 15        |
| 2.4.2 Testing.....  | 15        |
| 2.4.3 Documentation.....  | 15        |
| <b>Section 3 Roles and Actors.....</b>                              | <b>16</b> |
| 3.1 Introduction.....   | 16        |
| 3.2 Roles.....  | 16        |
| 3.2.1 Language expert.....  | 16        |
| 3.2.2 Researcher.....   | 16        |
| 3.2.3 Sample implementer.....                                       | 16        |
| 3.2.4 Standardizer.....   | 17        |
| 3.2.5 WSI provider.....   | 17        |
| 3.2.6 Encourager.....   | 17        |
| 3.3 Actors.....   | 17        |
| 3.3.1 Language community.....                                       | 17        |
| 3.3.2 Industry.....   | 18        |
| 3.3.2.1 Software vendors.....                                       | 18        |
| 3.3.2.2 Open source community.....                                  | 19        |
| 3.3.3 Standards organizations.....                                  | 19        |
| 3.3.3.1 International Standards Organization (ISO).....             | 19        |

**Guidelines for Writing System Support**

2003-10-31

Page 2 of 80

|   |           |
|---|-----------|
| 3.3.3.2 Unicode Consortium .....  | 19        |
| 3.3.3.3 World Wide Web (W3C) .....  | 20        |
| 3.3.3.4 Internet Engineering Task Force (IETF).....                                   | 20        |
| 3.3.4 Non-governmental organizations (NGOs) .....                                     | 20        |
| 3.3.4.1 SIL International.....  | 20        |
| 3.3.5 Governments .....   | 20        |
| 3.3.5.1 National standards bodies .....   | 21        |
| 3.3.5.2 Technical working groups .....  | 21        |
| 3.3.5.3 Legislation .....   | 21        |
| 3.3.5.4 UNESCO .....  | 21        |
| 3.3.6 Academics .....   | 22        |
| 3.3.7 Professional communities .....  | 22        |
| 3.3.8 Press .....   | 22        |
| 3.4 Conclusion .....  | 23        |
| <b>Section 4 Keys to Success .....</b>  | <b>24</b> |
| 4.1 Understanding of writing system needs .....                                       | 24        |
| 4.2 Understanding of technical issues .....   | 25        |
| 4.3 Consensus between stakeholders.....   | 26        |
| 4.4 Adequate software applications.....   | 26        |
| 4.5 Inclusion of characters in Unicode.....   | 27        |
| <b>Section 5 Technical Details: Characters, Codepoints, Glyphs.....</b>               | <b>28</b> |
| 5.1 Characters .....  | 28        |
| 5.1.1 Orthographies, characters and graphemes.....                                    | 28        |
| 5.1.2 Characters as elements of textual information .....                             | 29        |
| 5.2 Codepoints and Glyphs.....  | 30        |
| 5.2.1 Codepoints.....   | 30        |
| 5.2.2 Glyphs .....  | 31        |
| 5.2.3 From codepoints to glyphs .....   | 32        |
| 5.3 Keystrokes and codepoints.....  | 34        |
| 5.3.1 From keystrokes to codepoints .....   | 34        |
| 5.4 Further reading.....  | 36        |
| 5.5 References .....  | 37        |
| <b>Section 6 Technical Details: Encoding and Unicode.....</b>                         | <b>38</b> |
| 6.1 An Introduction to Encodings .....  | 38        |
| 6.1.1 Text as numbers .....   | 38        |
| 6.1.2 Industry standard legacy encodings .....  | 38        |
| 6.2 An Introduction to Unicode.....   | 39        |
| 6.2.1 A brief history of Unicode .....  | 39        |
| 6.2.2 The Unicode Consortium and the maintenance of the Unicode Standard .....        | 40        |
| 6.2.3 Unicode and ISO/IEC 10646.....  | 41        |
| 6.2.4 Types of information .....  | 41        |
| 6.3 Adding new characters and scripts to Unicode .....                                | 42        |
| 6.3.1 Evaluating whether a writing system is already supported by Unicode.....        | 42        |
| 6.3.2 Determining the best encoding solution.....                                     | 43        |
| 6.3.3 Qualifying what it takes to get new characters or scripts added to Unicode..... | 43        |
| 6.3.4 The formal process for adding new characters and scripts to Unicode.....        | 44        |
| 6.3.5 Preparing a proposal to add new characters or scripts.....                      | 45        |
| 6.4 Encoding conversion.....  | 46        |
| <b>Section 7 Technical Details: Data Entry and Editing .....</b>                      | <b>47</b> |
| 7.1 Types of data interaction.....  | 47        |
| 7.2 Keyboard layout .....   | 47        |
| 7.2.1 Mnemonic .....  | 48        |
| 7.2.2 Positional .....  | 48        |
| 7.3 Extending the keyboard .....  | 48        |
| 7.3.1 Modifier keys.....  | 49        |
| 7.3.2 Dead keys .....   | 49        |
| 7.3.3 Operator keys .....   | 49        |
| 7.3.4 Input method editors: composition and candidate windows.....                    | 50        |
| 7.4 Analytic keyboards .....  | 50        |
| 7.4.1 Sequence checking .....   | 50        |
| 7.4.2 Using keyboards in place of rendering systems.....                              | 51        |
| 7.4.3 Using keyboards to enter non-visual data .....                                  | 51        |
| 7.5 Tools for data entry and editing .....  | 52        |

**Guidelines for Writing System Support**

2003-10-31

Page 3 of 80

|                   |  |           |
|-------------------|--|-----------|
| 7.5.1             | Keyman keyboards.....                            | 52        |
| 7.5.2             | Macintosh Unicode keyboard layouts.....          | 52        |
| 7.5.3             | Reference info on OS-supplied layouts.....       | 52        |
| 7.5.4             | Copyleft, FLOSS and open source tools.....       | 52        |
| <b>Section 8</b>  | <b>Technical Details: Glyph Design .....</b>     | <b>53</b> |
| 8.1               | Basic font requirements .....                    | 53        |
| 8.2               | The need for quality.....                        | 53        |
| 8.3               | Sources for fonts .....                          | 54        |
| 8.3.1             | Commercial companies.....                        | 54        |
| 8.3.2             | Public and not-for-profit.....                   | 55        |
| 8.3.3             | Freeware/shareware.....                          | 55        |
| 8.3.4             | Intellectual property and copyright issues ..... | 56        |
| 8.4               | Process of font development .....                | 56        |
| 8.4.1             | Initial research and planning .....              | 56        |
| 8.4.2             | Glyph design procedures .....                    | 57        |
| 8.4.3             | Testing .....                                    | 58        |
| 8.5               | Conclusion.....                                  | 58        |
| 8.6               | Sources for information .....                    | 58        |
| 8.6.1             | Tools .....                                      | 58        |
| 8.6.2             | Web sites .....                                  | 59        |
| 8.6.3             | Publications.....                                | 59        |
| 8.7               | References .....                                 | 59        |
| <b>Section 9</b>  | <b>Technical Details: Smart Rendering .....</b>  | <b>61</b> |
| 9.1               | The Rendering Process .....                      | 61        |
| 9.1.1             | Glyph identification .....                       | 62        |
| 9.1.2             | Glyph processing.....                            | 62        |
| 9.1.3             | Glyph rendering.....                             | 62        |
| 9.1.4             | User interaction.....                            | 62        |
| 9.2               | Glyph processing—Dumb Fonts.....                 | 62        |
| 9.3               | Glyph Processing—Smart Fonts .....               | 63        |
| 9.3.1             | Substitution .....                               | 64        |
| 9.3.2             | Bidirectional Ordering.....                      | 64        |
| 9.3.3             | Positioning .....                                | 65        |
| 9.3.3.1           | Kerning .....                                    | 65        |
| 9.3.3.2           | Shifting.....                                    | 66        |
| 9.3.3.3           | Attachment Points.....                           | 66        |
| 9.3.4             | Justification .....                              | 67        |
| 9.3.5             | Line Breaking.....                               | 68        |
| 9.3.6             | Features .....                                   | 68        |
| 9.4               | User Interaction .....                           | 68        |
| 9.4.1             | Hit Testing.....                                 | 68        |
| 9.4.2             | Selection .....                                  | 68        |
| 9.4.3             | Arrow Keys.....                                  | 69        |
| 9.5               | References .....                                 | 69        |
| <b>Section 10</b> | <b>Glossary.....</b>                             | <b>70</b> |
| <b>Index</b>      | <b>.....</b>                                     | <b>79</b> |

## Section 1

# Components of a Writing System Implementation

*Authors: Victor Gaultney, Melinda Lyons*

Communication is a basic component of society. The ability of a group of people to communicate with one another and with the rest of the world is critical to community and nation development. Economic growth, education and cultural development depend on rich spoken and written communication.

Computers have enabled the diverse peoples of the world to communicate in powerful ways, whether that be in printed media or digital packets of information. Language communities can potentially communicate faster, farther, and in much greater volume. But there is a problem — most computer applications and operating systems were initially designed to handle English. They soon broadened enough to support the major languages of Europe and the Pacific Rim.

There are still, however, millions of people who are cut off from global communication. Their languages and writing systems are not supported on standard computers. The long-term viability of their culture and language is increasingly dependent on the ability to communicate through the digital medium, but many groups are hindered from even print publishing because there are no broadly available means to type and print their language in their alphabet.

This paper is intended as a guide to the planning and management of projects that seek to solve this problem. It provides a basic framework for the development of computer software components that support the diverse languages of the world. It is written primarily for policy makers and professionals, but includes some introductory technical material in later sections.

The scope of this document is necessarily limited. It does not seek to recommend particular software applications, or prescribe a single type of solution. Rather, it is intended to stimulate creative approaches to writing system implementations and encourage cooperation between commercial industries, governmental bodies and non-governmental organizations (NGOs).

## 1.1 Writing system implementations

Language is primarily a spoken means of communication. **Writing systems** (also called **orthographies**) are ways of communicating on printed or visual media, and are usually, but not always, based on spoken languages. A **script**, sometimes referred to using terms such as **alphabet** and **syllabary**, is a collection of symbols (typically with accompanying rules of behavior) that forms the basis for writing many languages. This leads to a basic definition of a writing system, as used in this paper: **the use of one or more scripts to form a complete system for writing a particular language**. Note that a writing system is unique to a specific language, or language family. Although Russian and Ukrainian share the same Cyrillic script, they represent two distinct writing systems.

A **writing system implementation (WSI)** refers to a set of software components that allow computer users to process textual data in that script and language; making it possible, for instance, to enter data using a keyboard and display it on the screen. Because writing systems are language-specific, there is no guarantee that an implementation of a certain script on a computer will work for all languages that use that script. A WSI that works for Farsi, for example, may not work for Sindhi, although they share the same Arabic script.

Common examples of WSIs would be those computer systems found in newspaper publishing houses around the world. WSIs can be very simple, such as for English, for which only a simple font is needed. They can also be complex and include expensive, dedicated software applications. For instance, to use

Chinese on a computer requires a combination of fonts, input systems, and publishing software that can write both horizontally and vertically.

## 1.2 Goals for writing system support

Language communities need WSIs in order for them to express their languages using computers. A WSI is needed to do even the most basic communications, such as printing out a letter for mass reproduction. Because of the growing importance of cyberspace — the realm of digital communications — WSIs will continue to be critical.

Every language deserves to have its writing system working on computers. But what does this mean? WSIs can differ in their breadth and depth of support. *So what level of support does a WSI need to provide for a language in order to be sufficient? At what point can the need be considered to be met? And what should be the goal of a WSI development program?*

### 1.2.1 Minimal — type, see, print, store

At a foundational, minimal level, people must be able to:

- **Type their language in a manner that is sensible to them.** Aspects of the keyboarding experience, such as the key layout or typing order, should usually match up with the way people think about writing their language, and not require a shift in linguistic understanding. For writing systems with hundreds or thousands of letters, this means that the correct symbol should be accessible in a reasonably intuitive manner.
- **See the results of their typing on the screen and view the text as it will be printed.** A WSI needs to provide basic feedback to users and allow them to catch and change mistakes.
- **Print their document with text rendered correctly.** Lettershapes need to be correct for the language, with accents and diacritics positioned appropriately. Words need to be formed using the right word-formation rules. For writing systems that require ligatures — the joining of two or more letters to form a combined shape that is different than the individual letters — the ligatures need to be present, and correct.
- **Store their data in a format that retains all important information.** This should include all information needed to present the text content legibly and facilitate common text processes (editing, sorting, spell-checking, etc.) in a manner that is appropriate to that writing system. For example, if a language is written with special tone marks, those marks need to be stored as an integral part of the text, not as separate graphics or manually positioned marks.

This allows for basic functionality, particularly for publishing on paper. If any one of these four attributes are missing, the WSI should be considered foundationally inadequate.

### 1.2.2 Reasonable — interchange, process, analysis

In today's digital world, print publishing is not really enough. People need to be able to share information in their language with others — through email, Web sites, portable documents, even interactive presentations. It is reasonable to expect that every language needs the ability to communicate in a wider world.

A WSI that enables broader communication would need to support:

- **Transmission of adequate language data via email.** Information needs to be encoded so that it could be sent to another computer without loss of important language information.
- **Preparation of Web pages that display the language correctly.** This is, however, highly dependent on individual Web browsers. Development of a WSI that supports Web page design that displays correctly in all browsers may not be feasible, and WSIs may need to be prepared that depend on a single browser platform.

- **Preparation of portable documents.** The current standard for this type of document is Adobe's PDF (portable document format). WSIs need to allow the embedding of language data into documents that can be viewed on a variety of operating system platforms.

There are also other means of electronic communication (Web logs, chat systems) that should be considered, as well as the broad realm of interactive documents.

In addition to information sharing, a WSI ought to provide for an appropriate level of language processing or analysis. This can include word sorting (for dictionaries and directories), spell-checking, hyphenation and other algorithmic processes.

The technology needed to support these activities already exists in some cases, but is often in an early state of development.

### 1.2.3 Ideal — parity

Ideally, computing in any language should be as easy as it currently is for English or other European languages. This is sometimes technologically or economically unfeasible. Nevertheless, there is no reason to be satisfied with a minimal solution for a writing system. As a language community grows in its technical ability and desire for communication, there should be no artificially imposed barriers on their use of computers.

### 1.2.4 Completeness

There remains, however, a question of completeness. For many activities, there may be a range of WSI support that is sufficient. In the end, WSIs can only be deemed to be complete if they fully meet the need for which they were intended.

Publishing, for example, is a term that covers the production of everything from a leaflet to a glossy airline magazine. For anything but the most basic publishing, a wider range of support is needed. For example, people typesetting Hebrew text (which runs right-to-left on the page), often need to mix words or phrases from left-to-right writing systems into their text. There is a whole set of paragraph and line construction rules that need to be applied when mixing such diverse scripts. A WSI may be sufficient for basic Hebrew publishing without supporting these rules, but more complex publishing requires them.

### 1.2.5 Accessibility

Finally, a WSI needs to be accessible to those who need it. There are a number of barriers to this:

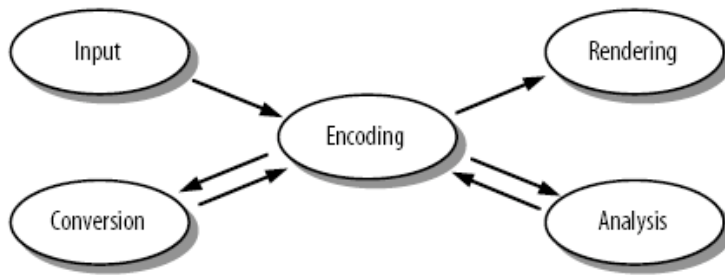
- **System complexity.** A WSI may meet all the technical needs for a writing system, but be too complex for the general public to use. It may meet the need for a smaller segment of the community, but generally useful WSIs are still needed.
- **Available software platforms.** To be adequate for a broad population, WSIs need to be usable on operating system and application platforms that are available to the public. For example, a WSI that only works on a Macintosh computer would not be sufficient in a location where only Windows computers are available.
- **Economic cost.** Although commercial investment is often necessary for the development of WSIs, solutions that are affordable by only a small segment of the community may not be sufficient. It will take creativity and cooperation to develop WSIs for the general public.

These three issues need consideration in the planning, development and evaluation of WSIs.

## 1.3 Components of a WSI — the SIL NRSI Model

Successful WSIs depend on a wide variety of software, including operating systems, standalone applications, fonts, conversion tools and other utilities. Some of these, such as the underlying operating systems themselves, are outside the control of most WSI developers, and cannot be altered. Others are

fully in the hands of independent companies and individuals. Despite these differences, it is possible to construct a general architectural model for WSIs.



The SIL NRSI Text Encoding Model

SIL International, through its Non-Roman Script Initiative (NRSI) department<sup>1</sup>, has developed a model for implementing writing systems that encompasses text input, storage, processing, and output. Because of the variety and breadth of locations where it works, SIL has one of the greatest needs for flexibility in these various functions. During its early days a model was developed for dealing with complex scripts that encompassed all of these needed functions.

### 1.3.1 Encoding

The model begins with data **encoding** — how language information is stored on the computer. Any information within a computer system is given a digital representation, meaning that it is encoded in terms of binary numerical values. For text data, the **encoding** refers to a set of rules by which the sequence of characters that make up a text are represented in terms of a sequence of numbers.

All WSIs are based upon an encoding, whether public or private. An adequate encoding needs to store all the information needed to represent the text in a given writing system. For example, an encoding for English needs to maintain a difference between *capital A* and *lowercase a*, because capitalization is an important writing system feature.

The current international standard for encoding is Unicode<sup>2</sup>, a system with almost unanimous support from software companies and government bodies. This standard assigns a distinct numerical range to individual scripts, and unique numeric identifiers to each character of each script. This allows data in a variety of languages to be stored together without confusion. For more information, consult the section on [Technical Details: Encoding and Unicode](#)<sup>3</sup>. It can be assumed that any adequate WSI stores its data according to *The Unicode Standard*.<sup>4</sup>

Examples of encoding components:

- A Unicode set defined for a given script that includes all needed characters (letters).
- A storage format that allows data to be encoded according to Unicode.

### 1.3.2 Input

Data needs to be entered into the computer somehow. This process involves input, whether by the computer keyboard or other input methods. The technical process of keyboarding involves translating

<sup>1</sup> SIL International's Non-Roman Script Initiative (NRSI) is focused on enabling WSI development by making WSI components, linguistic information, development resources and training materials available to the computing community. Website: [scripts.sil.org](http://scripts.sil.org)<sup>1</sup>

<sup>2</sup> See [UNI2003]. Details on the current Unicode version, can be obtained from [www.unicode.org](http://www.unicode.org)<sup>2</sup>

<sup>3</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_6\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_6_1)

<sup>4</sup> There are, however, some writing systems that have not yet been incorporated into Unicode.

keystroke sequences into character data in some given encoding. Since most physical computer keyboards are designed around the entry of Roman script data, the routines needed to interpret keystrokes into language data for non-Roman scripts can be very complex, and are commonly built directly into computer operating systems. The section on [Technical Details: Data Entry and Editing](#)<sup>5</sup> addresses these issues in greater detail.

Examples of keyboarding components:

- A simple keyboard definition for the script, developed by Microsoft and included in distributions of Windows.
- A Macintosh keyboard definition file, created by someone outside of Apple Computer, but that plugs into the standard Mac OS system.
- A complex keyboard definition created for use with the Keyman program. This would be separately installed, along with Keyman, onto computers running Windows.

### 1.3.3 Rendering

**Rendering** takes stored text in a given encoding and presents it visibly on a screen or printed page. If the data stored on the computer exactly parallels the individual letters in a line of text, this is a simple process: one code on the computer per letter. Even in this situation it is important that the letter be shaped correctly, and harmonize with the rest of the alphabet. Computer font design is a subtle but important process, and is briefly addressed in [Technical Details: Glyph Design](#)<sup>6</sup>.

In complex Roman and most non-Roman scripts, however, data must be interpreted to display or print properly. In the end, users want to view their data in an easy, trouble-free manner that accurately reflects the spelling and word formation rules of their writing system. This can require 'smart' rendering components that perform sophisticated interpretations of the data before sending it to the computer screen; see [Technical Details: Smart Rendering](#)<sup>7</sup>.

Examples of rendering components:

- A simple font, such as for English, that does not require special 'smart' code for rendering.
- An *OpenType* font, also intended for English, that has 'smart' code in order to display typographic ligatures required for fine typesetting.
- A non-Roman font to which special *SIL Graphite* 'smart' code has been added to support sophisticated diacritic positioning.

### 1.3.4 Analysis

Many WSIs include **analysis** components. This refers to a variety of actions in which data is processed or analyzed so that:

- the data can be transformed in some way specific to the writing system,
- the data can be presented to the user in some useful manner, or
- operations related to the editing, management or other processing of text data can be performed.

Such processes include sorting, word demarcation, hyphenation, and spell-checking, as well as more complex systems such as speech synthesis. Analysis may, at first, seem less important, but can be critical to language support. For example, a means of sorting language data is needed to produce dictionaries and telephone books. The brevity of this document, and the breadth of the topic, does not allow for further discussion here.

---

<sup>5</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_7](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_7)

<sup>6</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_8](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_8)

<sup>7</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_9\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_9_1)



Examples of analysis components:

- Standard sorting routines built into operating systems.
- Word demarcation (word breaking) rules built into a Thai typesetting program.
- Systems that turn text into a series of phonemes for input into a speech synthesis program.
- A Spanish spell-checking module that works within a word processor.

### 1.3.5 Conversion

Basic **conversion** components transform data from one encoding into another. Until the advent of Unicode, WSIs used hundreds of different encodings for their data. Some of these encodings were official standards, but others were proprietary and unique to a specific application program. Any time more than one encoding exists for a given language, there needs to be conversion routines and tools as a 'bridge' between them. This is especially relevant in the transition from older encodings to Unicode.

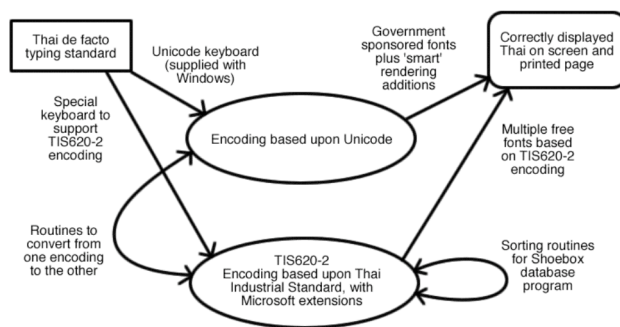
Without reliable conversion, users will hesitate to migrate their data to newer systems. A lack of good conversion tools may hinder the uses that could be made of older data. This is, again, a very important topic, but there is not room for further discussion in this document.

Examples of conversion components:

- A small program that converts text files back and forth between two encodings (such as the old MacRoman encoding and Unicode).
- Definition files that document the differences and correspondences between encodings, and that could serve as input for a conversion program.

## 1.4 WSI example

A real-world example can make this model, and the various components, easier to understand. There is a long history of computing for the Thai writing system. There is now a movement toward use of Unicode, away from old encodings, and that transition requires a variety of components. The diagram below describes, in brief, one system that has been developed as part of this new movement. It is one of many WSIs for Thai.



An example WSI for Thai

This WSI for Thai is based upon not one, but two central encodings: Unicode and an older Thai Industrial Standard (TIS620) with extensions by Microsoft (now known as TIS620-2). This reflects the current transitional state of the industry. Support for dual encodings requires conversion routines that can take text in either encoding and convert it to the other one.

There are two software keyboard mechanisms, one for each encoding. The keyboard layout for both is the de facto typewriter standard for Thai. Users can pick which keyboard to use and switch between them depending on what encoding they wish to produce at the time. One of these is distributed by Microsoft and the other was developed by a third party.

There are many fonts available, especially for the older encoding, that translate text in the encodings into symbols on the screen or page. Because the Unicode encoding is simpler, it relies on 'smart' fonts to position diacritics.

Finally, a set of sorting routines have been developed for use with a linguistic database program called Shoebox. This is currently only available for the TIS encoding, but a Unicode version is in development. General sorting routines, built into Thai versions of operating systems, have been available for many years, but are not incorporated into this illustration.

Although this system architecture describes a WSI for Thai, it is very similar to WSIs used for many encodings throughout the world.

## 1.5 Summary

The ability to communicate using one's own language is a basic human right, and use of an accompanying writing system can be an important part of that communication. To do this using a computer requires a writing system implementation (WSI). There is no standard set of requirements for WSIs that are applicable to every situation. Each language will require different capabilities, although the ability to type, see, print and store information is a basic minimum.

Although some WSIs are very simple in design, most involve a wide variety of components that must be coordinated. The choice of encoding is a critical one, as all the other components directly depend on it. A minimum system must include at least encoding, keyboarding and rendering components.

The following three sections focus on the process of developing WSIs, the important roles of companies, organizations and governments in WSI development, and some basic keys to successful WSIs. Later sections go into more detail on technical topics and are particularly appropriate to those initiating or managing WSI development.

## 1.6 References

[UNI2003] The Unicode Consortium, The Unicode Standard, Version 4.0 Reading, MA: Addison-Wesley, 2003.

## Section 2

# The Process of WSI Development

---

*Author: Victor Gaultney*

Writing system implementations do not just appear, they are developed. They can, however, seem to take shape in haphazard ways, without strong focus and direction. This is due to the diverse nature of WSI components. The people who design fonts are not always the same people who understand keyboard development.

Despite this diverse nature, it is possible to initiate, coordinate and complete a WSI development program. It requires a basic understanding of WSI components and their relationships to one another — which were addressed in the previous section. In addition, it needs a clear knowledge of the process of WSI development. This holds true for both the development of individual components as well as a complete integrated system. This section describes the WSI development process, and in doing so, recommends a basic procedure for planning such projects.

## 2.1 Determine needs

The first step in addressing writing system computing needs is to *determine the needs*. A focused approach based on problem solving can help to limit a development project, give it reasonable objectives and a means for evaluation. Many such approaches are available, and the process can be rather informal, such as making a list of needs, or can be formalized using systems such as Logical Framework Analysis<sup>8</sup>.

### 2.1.1 What do users need to do?

This step begins with an effort to *concretely identify what users need to do*. This could be writing down their cultural history, publishing a community newspaper, setting up a Web site on health education, or communicating through email. This may seem obvious, but is often neglected, and is a major cause of failure in WSI development projects.

### 2.1.2 What components are needed?

This information will help *determine what software applications and WSI components are needed*. This can start with an evaluation of applications suitable for the tasks related to what users need to accomplish (see above). In the case of a community newspaper, the community might look at a number of word processors and desktop publishing applications and decide which application best meets their need. Once that is decided, they can review what WSI components are available for that application, and determine what additional pieces need to be developed.

It would be nice if this process always worked in this way. In reality, the choice of application is usually determined by what WSI components exist, and which application supports those components. For example, in India, a particular word processing application might be chosen solely because it supports a particular writing system, such as Gujarati.

An extended example of this step may be helpful. The Tai Dam people of Southeast Asia have a writing system that is beautiful, but has numerous accents that must be carefully aligned with each letter. To publish a dictionary of their language would require:

---

<sup>8</sup> For more information on Logical Framework Analysis, see <http://www.usaid.gov/ausguide/ausguidelines/1.cfm>.

- A database program that can manage the large amounts of data required.
- A word processor or desktop publishing package that can import information from the database program, and supports paragraph and character styles. This is needed for proper formatting.
- A standard encoding scheme for their data that works in both applications.
- A keyboarding component that, at a minimum, works in the database.
- A font that works in both applications and is complete and accurate.
- A rendering system that can position the accents properly. This may be optional for the database, but is required for the final layout.
- A set of sorting routines, most likely built into the database, that respects the commonly understood order of letters in the alphabet.
- A set of encoding conversion routines if there is already data in another encoding that the community wishes to integrate into the database.

All but the first two items on this list would be considered WSI components.

### 2.1.3 What information is needed?

At this point it becomes important to *gather relevant linguistic and technical information*. A language community might have a very complete understanding of their writing system, but WSI development requires that information to be documented. Since computers behave in objective, analytical ways, the linguistic information needs to be similarly objective and analytical. This is so important that it is the first Key to Success listed in [Section 4](#)<sup>9</sup>.

Technical information, sometimes unrelated to the writing system may also be needed. For example, a word processor under consideration may support a variety of rendering methods. Details on each of those methods, and what sort of rendering behaviors they support, is important.

The ultimate purpose in gathering information is to provide the developers — those who will create WSI components — the information they need. Details need to be communicated clearly and unambiguously so that developers know exactly how components ought to behave. A cooperative dialogue between experts on the given writing system and developers will likely be needed in the information-gathering process.

### 2.1.4 What model of distribution is appropriate?

Early in the planning process, it is wise to *think through how WSI components ought to be distributed*. The model of distribution for a community-wide email project would need to be more broad and less costly for the individual, whereas a dictionary project completed by a team of two people only needs two copies of each component, and would likely have a source of special funding. This issue is increasingly relevant as intellectual property (IP) concerns gain more visibility around the world. For a detailed analysis of the difficulties and issues related to WSIs and IP, see the companion document: [“Intellectual Property Concerns in the Development of Complex Script and Language Resources”](#)<sup>10</sup>.

## 2.2 Identify and apply development resources

The second step in WSI planning is to *identify and apply appropriate development resources*. This refers to the companies, organizations and individuals that contribute to the project, whether that be in the form of specific deliverable components or consultant expertise in an area. It also refers to the non-personnel needs: funding, materials, equipment, etc., although that is not discussed in detail here.

<sup>9</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_4](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_4)

<sup>10</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=UNESCOIP](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=UNESCOIP)

Coordinated planning of WSIs does not imply that components should come only from one source. In fact, there are great benefits from distributed development. Components can be developed by whoever has the expertise. Projects that involve volunteers and charitable organizations can limit their involvement to what can be reasonably expected. The need for communication between parties motivates everyone to document their work.

The key to matching up WSI needs with resources is to identify who has the skills to complete each task, then consider what would motivate their involvement. For example, an independent, volunteer font designer might be willing to design a font, but not have the time or expertise to add smart rendering capabilities. In this case, a separate company could be contracted to add the additional routines. This would motivate the volunteer, and in so doing, reduce the funding needed to only what the separate company would charge.

A slightly more involved example might be where a minority language community has significant understanding about their writing system, but does not have the technical knowledge needed to implement it in major software applications. Major software developers depend on industry encoding standards, and are very hesitant to implement support for scripts for which no standard exists. Hence, a key to application support would be inclusion of the script in an international standard — Unicode. Let us suppose, though, that the community does not understand the process for adding their script to Unicode. A non-governmental organization (NGO) might have great technical expertise in this process, but lack details of how the writing system works.

None of these groups can remedy the situation themselves, but could work together. The NGO could work with the language community to prepare a proposal for the Unicode Consortium to consider. Once accepted, the major software developer could add support. The community would be motivated by the desire to see their language working in major applications, the NGO would be motivated out of humanitarian concern, and the software developer would be motivated by a desire to make their software fully support that portion of Unicode, which could have commercial benefits.

[Section 3](#)<sup>11</sup>, on the roles of various people and organizations in WSI development, addresses these issues in greater depth.

## 2.3 Develop components

Once the needs have been determined, and the resources identified, the third step is to develop the actual components. This document has already discussed some issues around component development, with greater detail in later sections, but it can be helpful to review the general milestones in approximate order. Note that these actions do not necessarily have to be done in this order. Most could be done in parallel to one another. The actions, which closely parallel the SIL NRSI model, are:

- **Choose the supported encoding(s).** This must be done first, as it affects all other components and is the bridge between them. In most cases the encoding should be Unicode, although there are situations when this is not completely applicable (such as if a script is not yet accepted into Unicode). For an introduction to the technical sections, which gives an overview of key topics, see [Section 5](#)<sup>12</sup>. For detailed information on encodings and Unicode, see [Section 6](#)<sup>13</sup>.
- **Obtain or design font(s).** There are many sources for fonts, from commercial companies to freeware developers. It may, however, be necessary to design an original font. Details on font requirements, sources for fonts and the process of original design can be found in [Section 8](#)<sup>14</sup>. Fonts are also the most commonly 'pirated' types of software, so IP issues need to be carefully considered. See "[Intellectual Property Concerns in the Development of Complex Script and Language Resources](#)"<sup>15</sup> for an in-depth look at the problems.

---

<sup>11</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_3](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_3)

<sup>12</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_5\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_5_1)

<sup>13</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_6](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_6)

<sup>14</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_8](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_8)

<sup>15</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=UNESCOIP](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=UNESCOIP)

- **Add smart behaviors to the font(s), if necessary.** This will depend on the behavior of the writing system and the support for smart rendering in applications. This is currently a highly technical task, and the number of available experts are few. See [Section 9](#)<sup>16</sup> for more details.
- **Develop keyboard(s).** This is often the most important component from the user's perspective. They need to be able to type in an intuitive manner that fits their writing system well. It is also common to have to support multiple keyboarding standards, since not all users are coming to the computer with the same history of computer literacy. [Section 6.3](#)<sup>17</sup> discusses general keyboard design theory and related issues.
- **Enable data conversion.** Unless a WSI is the very first implementation of a certain writing system on a computer, there is likely to be a large body of text that needs to be converted from older encodings to new ones. This may also not be just a one-time effort, as many older computer systems will remain in use even when new WSIs are available.
- **Add analysis tools.** As illustrated earlier, these peripheral components can be important, but are not discussed in any more detail here.

## 2.4 Gather components into solutions

The final step in WSI development is to *gather the diverse components into functional solutions*. Some examples:

- [LaoScript for Windows](#)<sup>18</sup> is a shareware package that includes fonts, keyboards, conversion and other utilities (word demarcation, spell-checking, Thai-Lao translation). It supports Unicode as well as older encodings and works in most Windows applications. This is an example of a very complete, full-featured WSI, developed by a single developer.
- The [SIL Dai Banna Fonts](#)<sup>19</sup> is a free package of fonts and keyboards for the New Tai Lue (Xishuangbanna Dai) script. It works in all standard Macintosh and Windows applications. The script does not need smart rendering, nor analysis tools, and as the first implementation of the writing system, the package does not include text conversion tools. It is not a complete solution on its own, as it depends on a separate program for keyboard handling (that must be downloaded separately).

In the second example, people would need to download and install two separate packages (main package and keyboard handler) to create a functioning WSI. This is becoming more common, particularly among free or inexpensive solutions.

The ideal is, of course, a single package (like LaoScript for Windows) that can be easily installed and used, with common documentation. This can be expensive for a company to develop and expensive for the user to purchase. It can require licensing fonts and software from different sources, and is not always accessible to the general public.

Good, inexpensive WSIs can also be assembled from a variety of sources. For example, there is no complete Unicode-based solution for ancient Greek from a single vendor. The nature of standards-based solutions, however, allows components from multiple vendors to be interoperable. As a result, a completely free working system for personal use in Windows can be put together by combining:

- [Gentium](#)<sup>20</sup> fonts (SIL International)
- [Classical Greek keyboard definition file](#)<sup>21</sup> (Manuel Lopez)

---

<sup>16</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_9\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_9_1)

<sup>17</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_6\\_3](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_6_3)

<sup>18</sup> <http://www.tavultesoft.com/lswin/>

<sup>19</sup> <http://www.sil.org/computing/fonts/sildb/index.htm>

<sup>20</sup> <http://www.sil.org/~gaultney/gentium/>

<sup>21</sup> <http://members.aol.com/AtticGreek/>

- A keyboard manager program, [Keyman](http://www.tavultesoft.com/keyman/)<sup>22</sup> (Tavultesoft)
- A [Web-based encoding converter](http://www.jiffycomp.com/smr/unicode/)<sup>23</sup> that converts text from many older encodings to Unicode (Sean Redmond)

This involves interacting with four different Web sites and managing three separate software installations — but it works and is free for personal use. In other words, it meets most basic needs for ancient Greek computing.

### 2.4.1 Integration

One key to success of this final step in WSI development is integration. If a WSI is produced by a single developer, integration is relatively easy. Diverse WSIs made up of many components run the risk that assumptions made for one component will impact other components, sometimes in surprising ways.

Communication between developers is the best insurance against these incompatibilities. The primary role of a WSI project manager that involves components from multiple sources is to foster communication on technical issues. Deliberate mechanisms for developer communication (Web forums, email discussion lists, etc.) can be very helpful.

It can also be important to follow international standards, such as Unicode. If each component developer agrees to use Unicode as the primary encoding, then components can be prepared separately and assembled later. This is the case for the Greek example discussed earlier. Even though it is very likely that the four developers never spoke to one another, the system works because of uniform Unicode support.

### 2.4.2 Testing

Another important part of putting WSIs together is testing. This is often ignored or neglected due to limited resources, but has a strong impact on the usefulness and acceptance of a WSI. There is not enough room to discuss testing strategies in this document, but a single principle is most important — do it.

### 2.4.3 Documentation

Whether a WSI is simple or complex, or developed by a single developer or a broad group, documentation is important. Two types of documentation (at a minimum) should be delivered:

- **End user documentation.** This would consist of basic installation steps and a brief instruction to use, such as a keyboard layout.
- **Technical reference material.** This would include discussion and a listing of the encoding used, a full reference to what fonts contain, detailed listing of smart font behaviors, etc. A technically-capable person ought to be able to pick up this reference and develop additional components that work in harmony with the rest of the package. They should also be able to help an end user who has difficulties, or who needs some additional features.

Again, this holds true even if the WSI is a collection of diverse components. Technical reference material should likely come from the primary developer of each component, but users would appreciate a single document that describes the installation of complex packages.

There are creative ways in which this documentation can be delivered. It is not always necessary to produce commercial-style printed documentation. A site of Web pages can serve just as well, if not better, because it is always available on the internet and could be regularly updated to reflect changes and problem resolutions.

---

<sup>22</sup> <http://www.tavultesoft.com/keyman/>

<sup>23</sup> <http://www.jiffycomp.com/smr/unicode/>

## Section 3

## Roles and Actors

---

*Authors: Martin Hosken, Lorna A. Priest*

### 3.1 Introduction

Crossing the digital divide brings us into the realm of bridge building. Many of the analogies with traditional bridge building — bridge types, bridge building roles and actors, etc. — hold when thinking about the digital world. Writing system implementations correspond to a bridge where the computer takes the place of the construction site and its environs.

This section examines the main roles that need to be filled in order to build the bridge and then examines various actors who directly and indirectly take on those roles in crossing the divide. In different contexts, different actors may take on different roles. So by separating roles and actors we can see more clearly the activities that need to take place and then find appropriate actors to take on those roles.

### 3.2 Roles

Roles arise from activities. The overall activity is building WSIs suitable for use by the user community, but there are many different specific activities that are necessary to that process.

#### 3.2.1 Language expert

The role of being the language expert involves providing information about the language and orthography. Those involved in this role also need to be able to answer questions about cultural aspects that relate to their writing system. For example, the shape of some special characters may be important to the community since they help identify a text as being from *their* language. Other important questions may involve such issues as keyboard layouts and font styles suitable for new readers — though the list of all possible questions is endless. Helpful information can range from detailed answers to “it doesn’t matter”, “we don’t care”, or “either will do”.

#### 3.2.2 Researcher

The role of researcher is taken up by someone or some group interested in the language and orthography sufficiently to study it. In conjunction with the language community, they study and analyze the orthography (and with it the language). Good research includes anthropological aspects of reading and writing, printing and computer use within the community. The researcher then takes the results and makes them available, in the appropriate language, to WSI implementers. This may be in the form of published papers or via private correspondence. One of the most important contributions a researcher can make is to provide a clear description of the orthography in question.

#### 3.2.3 Sample implementer

The first WSI for a writing system is often a proof of concept. Although this is not necessarily required, it is unlikely that the first bridge built across a river is going to be a three-lane highway. As such, the task of



the sample implementer is to create as good an initial bridge as they can — even if it is only one made of rope.

A sample implementation has a number of benefits. It proves that the orthography can be implemented, thus increasing the awareness among the language community. It also validates the WSI description, and proves that it is sufficient for a WSI to be created. It can also provide a model for other WSI implementers to follow. It is this latter role which encourages the sample implementer to do the best job they can.

One of the characteristics of the sample implementation is that it is tested within the language community with close feedback between the community and the implementer. Due to its reliance upon primary contact, in effect, the sample implementation is the completion of the research. As a tool for future development, therefore, the sample implementation should be as complete and thorough as possible. This often results in two types of sample implementations: a rough solution that allows for basic data entry and printing within a limited context; and a more thorough, standards-based solution that may be used as a basis for future development.

### 3.2.4 Standardizer

The standardization process for WSIs is not necessarily one of creating new standards. Often an existing standard is enhanced to include the new orthography. For example, instead of creating a new encoding standard for a particular orthography, the extra characters needed can be proposed as additions to the Unicode standard.

The standardizer also has the task of deciding what needs to be standardized across WSIs. For example, there is no standard keyboard layout for the Myanmar script beyond the limited typewriter layout. As a result, there are many different layouts, causing difficulties for users. This is an area ripe for standardization.

### 3.2.5 WSI provider

Sample implementations are usually not sufficient to meet all the needs of the language community. Higher quality solutions are needed. The development of these solutions may be tied into a particular product or combined with a number of other WSI components into a wider solution. The final result is a higher quality solution for the targeted users.

The WSI provider role acknowledges that a single WSI may not meet all needs. There are specialist needs (high speed databases, high quality typesetting, etc.) which may need specialist solutions.

### 3.2.6 Encourager

The Encourager stimulates the whole process of WSI creation to take place. Encouragement takes two forms. Promotion involves actively encouraging actors to be involved in the process, via funding, recognition, etc. Regulation is sometimes used to encourage standardization.

## 3.3 Actors

Having examined the various roles and activities in the WSI creation process, we now look at the various actors that may get involved in the process. We will examine which role is most appropriate for each actor to take. Some actors may be involved in more than one role.

### 3.3.1 Language community

Different language communities take different approaches to the role of being language experts. Some communities are very passive in their role. It is up to researchers and others to find out the community's needs. This is common among pre-literate societies or those with little interaction with more technologically advanced groups. It is only later in the whole process that interest may develop.

Some groups have strong feelings about aspects of their orthography, and may even regard such information as private. Identity is sometimes related to orthography and certain features of the writing system or printing style are important in identifying a text as being in *their* language. Language communities may also take an active role in promoting a certain orthography.

An example of this happening comes from Indonesia, among the Batak people. Between 1851 and 1857, H.N. van der Tuuk investigated the Batak language and script. He discovered that only the *datu* (religious leader/healer) and his students were able to read the script. Van der Tuuk composed a standard grammar and script based on the material he received from the *datu*. Years later, in 1882, when a missionary was in the Batak region, a number of headmen got together to ask the missionary if they could give instruction in reading and writing the Batak script as well as the Latin script. The resurgence of the use of the Batak script came about because the people themselves wanted it.<sup>24</sup>

Even if the language community is not pursuing standardization efforts, they need to be approached and invited to participate throughout the whole process.

### 3.3.2 Industry

The computing industry has a long and complex relationship with the WSI creation process. The most natural role for the computing industry is that of WSI provider. Industry often sees the needs for standards and may be the ones who facilitate standards proposals through [ISO](#) or [national standards bodies](#). They must know the established usage of a particular script before they try to implement it. They need to know what the expectation is from the users of how things should work. What industry does well is to provide finished solutions that work for a particular set of users or a specific software need.

Ideally, a company prefers to be the only one to have a bridge across the river and so reap the benefits of a monopoly. Unfortunately, this desire for uniqueness has made WSI creation by others difficult. The proprietary nature of these specialist solutions means that they work well in only a limited range of contexts, and are less open to refinement. The return on investment of developing a unique, specialized WSI for a small language group is low. This has resulted in expensive solutions that can be relatively poor in quality.

Larger software developers, who want to see their products used by many different language communities, often see WSI development as a cost rather than as a revenue source. As a result, increasing emphasis has been placed on producing globalized software that can work in many languages. The primary concern of this globalization process has been to cover the major language markets — those that have the most sales potential. Support for minority languages and the ability to add extra WSIs are often ignored or actively locked out. Sales potential is not the only concern for these companies — they do not want to bear the burden of supporting third-party additions or modifications to their product.

The result of all this is that there are two types of industry actors: those who see WSIs as a direct revenue source and so are interested in making their WSI unique in some way to encourage sales; and those who see WSIs as a cost necessary to enable their main product to be sold in a new market. The latter group is more likely to embrace international standards and try to see issues globally, while the former is more likely to keep WSI elements proprietary in order to encourage people to pay for them.

#### 3.3.2.1 Software vendors

Software vendors can be multinational, such as Microsoft Corporation. Multinational organizations are highly motivated to help set standards. They cannot afford to implement software solutions for multiple standards and will often work with governments to stimulate standards development. In most cases, they will not even attempt to implement a writing system until there is a standard which is recognized by the government.

Another group of software vendors are those interested in solutions for a particular country or region. [Mithi \(Mithi Software Technologies Private Limited\)](#)<sup>25</sup> of India could be considered an example of a regional software vendor. Mithi has made a name for itself by developing local language solutions. Their desire is that the non-English computer user not be left out. One example of software Mithi has developed is an email application which allows the user to exchange email in 11 different Indian languages (plus English).

<sup>24</sup> <http://www.library.uu.nl/digiarchief/dip/diss/1922364/full.pdf>

<sup>25</sup> <http://www.mithi.com/>

A third type of vendors are local. Their software could be for one particular script only, or even for just one particular language using that script. Sometimes software which is developed for a particular local need is seen outside of the local community and adopted. For example, [Keyman](#)<sup>26</sup> is a program which allows a user to type in other languages and scripts using their existing keyboard. It is easily customizable. Keyman was developed for one solution — inputting Lao. Others saw the program and wanted something similar for their language. Keyman is now used worldwide for handling complex languages including Amharic, Nepalese, Lao, and Farsi.

### 3.3.2.2 Open source community

The open source community is rapidly growing, and is very interested in supporting minority language communities unreached by the wider industry. They often are motivated to provide inexpensive solutions for people who cannot afford expensive software. They also want to implement solutions for minority language communities in which industry has little interest.

The open source community creates freely available solutions. This challenges the traditional computing industry model of selling expertise and focuses more towards selling services. For the open source community, software is a cost, so the community embraces standards wherever it can. The result is that open source software provides an alternative framework and environment for WSI developers. If those developers are not interested in providing WSIs for profit, they may find that the open source model is more appropriate for them. An example of an open source project is the [Graphite/Mozilla project](#)<sup>27</sup>, begun by SIL International because of needs that minority language groups had for rendering their writing system that remained unmet by industry software. Mozilla has now implemented Graphite rendering in their open source project.

### 3.3.3 Standards organizations

Standards organizations naturally take on the standardization role, but each tends to have a particular domain of interest. This means that they will be concerned with a particular aspect of a WSI across all languages within their domain.

Standards are only effective if people use them. The result is that a standards body has a synergetic relationship with other actors in the process.

#### 3.3.3.1 International Standards Organization (iso)

The [International Standards Organization \(ISO\)](#)<sup>28</sup> can be described as the United Nations of standards organizations. It provides an international forum for the standards process across a wide range of standards. As such, orthography issues are well within its area of responsibility. The difficulty, as with any large organization, is knowing where and how to relate to the appropriate parts of the organization, with its sections and working groups. An important encouragement activity is to enable researchers and sample implementers to relate to the appropriate standardizers within the ISO structure. Generally, the need for a standard is brought by an industry sector to a national member body (of ISO) where the process of definition and approval is carried out.

#### 3.3.3.2 Unicode Consortium

The [Unicode Consortium](#)<sup>29</sup> is the place to go for encoding standards. Without getting something into Unicode it is unlikely that multinational software companies will provide support for individual needs. The Unicode Consortium has a set of [guidelines to follow for proposing new scripts or characters](#)<sup>30</sup>. Individuals may submit proposals, but if the proposal involves the encoding of a whole script (not just additional

---

<sup>26</sup> <http://www.tavultesoft.com/>

<sup>27</sup> <http://sila.mozdev.org/>

<sup>28</sup> <http://www.iso.org/>

<sup>29</sup> <http://www.unicode.org/>

<sup>30</sup> <http://www.unicode.org/pending/proposals.html>

characters), sponsorship by a relevant academic or government organizations may be helpful. [Section 6.3](#)<sup>31</sup> of this document is a guide to this process.

### 3.3.3.3 World Wide Web (w3c)

The [World Wide Web Consortium \(w3c\)](#)<sup>32</sup> is the *de facto* standards organization establishing web content and structure standards including HTML and XML. As such, it is not directly concerned with orthography issues, but has an indirect interest in WSI needs, and can benefit from orthography descriptions.

### 3.3.3.4 Internet Engineering Task Force (IETF)

The [IETF](#)<sup>33</sup> is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the internet architecture and the smooth operation of the internet. Amongst other things, they are involved in setting standards for language identification — the way that language data is identified in internet documents.

## 3.3.4 Non-governmental organizations (NGOs)

The primary concerns of non-governmental organizations and other specialist groups are the language communities they serve. In order to ensure a wide user base for their WSIs, and increase the viability of their solutions, NGOs are interested in communicating what they are doing to a broad audience. They are also interested in training, transferring skills to the local community, facilitating workshops and facilitating processes (like setting standards). They enter the technological realm out of necessity in order to obtain technology for their needs and the those of the language community. There are, therefore, two types of NGOs we consider here.

Most NGOs do not have the technical expertise to implement or manage the creation of a WSI for a particular language. Instead, they search for any possible solution that they can use. As such, they constitute another user community alongside the language community. What they bring to the WSI creation process is the ability to communicate the need to possible solution providers. They can do this very effectively and take on an encourager role, through raising awareness and funding.

A few NGOs do have the technical expertise to implement or manage the creation of a WSI, and these often take on the research and sample implementer roles. With their close relationship to the language communities concerned, they have excellent access to the information needed and can describe it well to other implementers.

### 3.3.4.1 SIL International

[SIL International](#)<sup>34</sup> is one such organization that functions much like an NGO. It has the expertise and access to language communities that enable it to fulfill the researcher and sample implementer roles. Due to its interest in so many languages and orthographies, it has to take a global view. As a result, it also gets involved in the standardization process and with the encouragement of various actors to enable access to technology by minority language communities.

## 3.3.5 Governments

The primary role of a government is as encourager. They encourage by:

- Funding and promoting WSI development within the country.
- Identifying and recognizing WSIs that have been implemented well.

---

<sup>31</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_6\\_3](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_6_3)

<sup>32</sup> <http://www.w3.org/>

<sup>33</sup> <http://www.ietf.org/>

<sup>34</sup> <http://www.sil.org/>

- Encouraging the language communities to get involved in the process.
- Encouraging and regulating the use of standards where competing solutions are not being reconciled and are causing confusion.
- Promoting national standards, both their use and acceptance within the country and also on the international stage.

A common role has been for a government to encourage standards for encoding, keyboards, and more recently, sorting. For example, the Thai government based their national encoding standard (TIS620-2533) on an existing *de facto* industry standard. This was then taken up, in its entirety, into Unicode.

### 3.3.5.1 National standards bodies

There are sometimes national standards bodies whose goals can include promoting and coordinating standardization at all levels in the country. An example of a government national standards body is the [Quality and Standards Authority of Ethiopia \(QSAE\)](#)<sup>35</sup>. It would be natural for a national standards body to be a member of [ISO](#), as the QSAE is.

### 3.3.5.2 Technical working groups

There are also technical working groups sponsored by governments. These often do software development. The [National Electronics and Computer Technology Center of the Royal Thai Government \(NECTEC\)](#)<sup>36</sup> is an example of this. As stated on its website, NECTEC's mission is to "ensure Thailand's competitiveness in Electronics and Computer and the use of IT to stimulate economic and social impact through own [sic] R&D programs as well as R&D funding services to universities". NECTEC is involved in open source projects and has implemented Thai support for OpenOffice.

### 3.3.5.3 Legislation

Governments may even legislate trade based on standards support. For instance, the Government of the People's Republic of China introduced [GB18030](#)<sup>37</sup>, a new standard for the support of its national characters. GB18030 mandates that new products released in China after January 1, 2001 must support certain character sets. For Microsoft, this meant that in order to sell its software in China they must support the standard.

Governments are often interested in standardizing orthographies and so may have already been involved in that process. This can, however, be controversial. For instance, in 1941 the Mongolian government (under apparent pressure from Stalin) passed a law to outlaw the traditional Mongolian script and replaced it with a Cyrillic-based alphabet. This may have been an attempt to wipe out all things Mongolian and replace them with Soviet institutions. After Mongolia had free elections and instituted a new constitution in the early 1990s, the Mongolian government has been encouraging the use of the traditional vertical Mongolian script.

Nevertheless, governments most often have good reasons for the standards they set, and may be the ones who are involved with the whole process. They may work through the academic communities or Ministry of Education, down to the language community level.

### 3.3.5.4 UNESCO

As a para-governmental organization, [UNESCO](#)<sup>38</sup> has an important role to play in the whole process. It can encourage governments to not ignore the needs of minorities in their concern to unify their nations. It can encourage WSI development and provide resources to governments and other NGOs. They can encourage NGOs directly and help in all areas of the WSI development process, especially for those languages which may not be receiving the attention they deserve.

---

<sup>35</sup> <http://www.qsae.org/>

<sup>36</sup> <http://www.nectec.or.th/home/>

<sup>37</sup> <http://www-3.ibm.com/software/globalization/topics>

<sup>38</sup> <http://www.unesco.org/>

UNESCO is especially interested in making needs and specifications known. They can be very helpful in bringing interested parties together for setting standards. Recently, UNESCO collaborated with the Ministry for Capacity Building of Ethiopia and the Economic Commission for Africa (ECA) to discuss how to move forward to create [standards for the Ethiopic script](#)<sup>39</sup>. Besides encoding standards, there was discussion on the need for standardization of keyboard layout and transliteration.

### 3.3.6 Academics

The academic community may include universities, academic associations, and even the individual linguist. They have a strong interest in minority languages and want to be involved in the standards process. As with the linguist, they may already have been involved in the orthography development. They will often support research and development within language communities, provide training, facilitate standards development, etc.

It is natural for academics to fall into the researcher role. Academic research can fall into two types: language-directed and technology-directed research. Language-directed research arises as a linguist studies a particular language and its community. Technology-directed research is interested in the interaction between computers and languages and tends to take a more global approach. Language-directed research tends to provide better orthography descriptions, while the technology-directed research results in sample implementations.

Linguists also have an interest in the writing system. They are likely to be the ones who have spent months and years analyzing the language and possibly know more about the technical features of the language than even mother-tongue speakers. The linguist is also one who will understand the local needs for standards from living with the language community for long periods of time. They may have been involved with the development of an orthography for that language community.

One potential source of conflict is the immediacy and flexibility of the linguist's needs. The linguist is often working alone, needs a solution right away, and needs to be able to alter that solution easily in light of new discoveries. As a result, linguists may choose to work out their own solutions without discussing options with others. This can lead to odd orthographies and computer solutions which do not follow established standards.

Even if the linguist is not the one pursuing standards, they would certainly be a source of information and expertise. They may also know which members of the language community would be best to approach for further information.

### 3.3.7 Professional communities

Professional communities act as a particular type of encourager. They stimulate people with similar skills and interests to work together. Since they are not financially motivated, they are often highly motivated to support minority languages. Although professional communities rarely make standards proposals (Unicode, language identification, etc), they will often be involved in promoting or implementing standards.

As an example, the [International Font Technology Association \(IFTA\)](#)<sup>40</sup>, a spinoff from [Association Typographique Internationale \(ATypI\)](#)<sup>41</sup>, is a new organization dedicated to technical aspects of font design and production. Thus they enable the industry to produce fonts more easily, an important part of WSI creation. They also foster communication. Though a professional organization, they see themselves having an active role in standards formation as a natural extension of their role as industry encourager.

### 3.3.8 Press

Although the press will likely not be involved in developing standards, they can have an active role as encouragers. Not only do they communicate needs and results, but also motivate implementers and conduct research to make their results known. The primary actor in this area is the academic press, who

---

<sup>39</sup> <http://www.uneca.org/aisi/docs/ethiopic.doc>

<sup>40</sup> [http://www.atypi.org/news\\_tool/news\\_html?newsid=79&from=/](http://www.atypi.org/news_tool/news_html?newsid=79&from=/)

<sup>41</sup> <http://www.atypi.org>

encourages the publication of papers. Although there is no natural means through which to publish orthography descriptions, papers are an important way of getting technical details to a broad audience.

The more popular, but still technical, press can also help raise awareness of the needs among potential WSI developers. For example, the magazine [Multilingual Computing & Technology](http://www.multilingual.com)<sup>42</sup>, published by Multilingual Computing, Inc. encourages those working in the area of multilingual computing to broaden their horizons both technically and linguistically.

### 3.4 Conclusion

Creating a WSI is hard work. The whole process involves many different actors taking on a variety of roles. Although this section discusses the various roles and actors in generalized terms, the precise details of who does what and even what gets done may well vary when it comes to a particular language group in a specific sociolinguistic context. But the variation is unlikely to be radically different from the model presented here.

One of the key points that arises from this model is that it is very unlikely that a complete solution will be provided by a single actor. Interaction between the various actors involved in a particular orthography is essential. As each actor acknowledges the value of the others to the process, better WSIs will result, and users will be better served.

---

<sup>42</sup> <http://www.multilingual.com>

## Section 4

### Keys to Success

*Author: Victor Gaultney*

WSI development can be very complicated, both technically and organizationally. It requires a team of people or companies to work toward a single goal — enabling a certain group of people to use their language and writing system on computers. Their knowledge, background and skills may be very different, and they will likely make unique contributions to the end solution. The technologies involved are also challenging and continue to evolve.

Because of this complexity, it can be very easy for a WSI development effort to fail. There are ways, however, to reduce the chance of failure and build a strong foundation for development. Each of the keys to success discussed here is important. A serious weakness in any one can cause a WSI effort to break down. At the same time, even modest efforts to improve in any area can greatly enhance the end product and streamline the development process.

#### 4.1 Understanding of writing system needs

The first key to success is an *adequate understanding of writing system needs*. Before the project starts, there needs to be an assessment of the end goal: what users want to accomplish. A statement of writing system needs should be:

- **Clear.** It should state what users want to do (exchange email, publish a community newsletter, prepare educational materials, etc.) in very direct terms. If the WSI needs to operate within a specific application or operating system, those constraints should be listed. The style should be simple — an outline with bullet points can be ideal. This will also aid in building consensus (see [Section 4.3](#)).
- **Complete.** It needs to address the full scope of the need, both technically and linguistically. For example, details about the writing system should include a full list of symbols and their spoken equivalents, as well as shaping behaviors (such as spelling rules, required ligatures, details on diacritic alignment, etc.). This will take considerable time to complete, but will make the development phase much more rapid and less likely to be delayed due to ambiguous goals or language behavior.
- **Detailed.** It should be written with computing in mind, and give detailed instructions. For example: 'the iota subscript is found only after lower- and upper-case vowels: alpha, eta and omega. For lower case, it should be centered under the previous glyph, except under the eta, where it should be aligned under the left leg. Following upper-case, it is usually written as a normal iota after the vowel.' An even more useful description could be:

```
alpha + iotasub = alpha with iotasub centered beneath
eta + iotasub = eta with iotasub centered under left leg
omega + iotasub = omega with iotasub centered beneath
Alpha + iotasub = Alpha, iota
Eta + iotasub = Eta, iota
Omega + iotasub = Omega, iota
```

- **User-focused.** The statement should also specify how users expect the system to work, particularly for keyboarding and editing. In writing systems where the glyph order is different from the spoken order (such as with many Indic scripts), which order should be used for typing? And what should happen if the user clicks in the middle of a sequence of characters and hits the



backspace key? In the case of an Indic script where reordering is happening, the desired effect will likely be different than for, say, English.

- **Broad.** It should address how this specific project should relate to other WSIs. If the writing system is based upon the Khmer script, but with some additional conjuncts, there ought to be a clear statement of those differences. It should also state those things that are to be the same as existing Khmer WSIs, such as keyboard layout and typing order. This exercise can also make it easier for a single WSI to serve multiple language communities.

The objective of this key is to produce a needs statement. This does not have to follow a specific model or framework, but must clearly state the need and outline how the WSI ought to work in the end. There is no requirement to have all the information in a single document or form, as long as all needs are addressed in adequate detail.

For example, one component should be an orthography statement — a description of how the individual written shapes express the language — and how they interrelate with one another. Without this it will be difficult to build a system that takes the full breadth of the written language into account.

## 4.2 Understanding of technical issues

The second key is an *adequate understanding of technical issues*. A WSI development team must include people with enough technical ability to research, digest and understand the technologies that are needed to reach their goal. It is rare that a single person will be able to grasp all the technical issues, especially for a complex project. The team then needs to understand:

- **The basic model for complex script computing.** This model, introduced in [Section 1](#)<sup>43</sup>, can be very helpful in planning the architecture of a system and explaining it to others.
- **Foundational concepts.** These would be concepts such as the distinction between characters, codepoints, and glyphs (see [Section 5](#)<sup>44</sup>), and the basic principles of Unicode (see [Section 6](#)<sup>45</sup>). This is important for every member of the technical development team, whatever their role might be.
- **The logical framework behind key technologies.** If a team member needs to address complex script rendering, then they will have to understand the similarities and differences between the various smart font technologies (see [Section 9](#)<sup>46</sup>). Keyboard developers need to understand the difference between modifier keys and dead keys (see [Section 7](#)<sup>47</sup>).
- **Specific limitations of operating systems and applications.** For example, an application might advertise that it supports OpenType fonts, but that does not guarantee that it will support the specific OpenType behavior required. The developer may have to read documentation, interact with experts on Web forums, and experiment with the applications themselves.
- **Data transfer and conversion issues.** WSI developers must understand how their data will be encoded, and how that data will be moved in and out of applications. This is especially important if a WSI already exists for the writing system, and data will need to migrate back and forth between different WSIs. Detailed discussion of this topic is out of the scope of this limited introduction, but more information is available on the NRSI Web site: [scripts.sil.org](http://scripts.sil.org)<sup>48</sup>.

Here the objective is to gather the necessary personnel and link them with resources that will enable them to be technically successful. The remaining main sections of this document go into detail regarding many of the technical issues involved in WSI development, and can be considered an introductory guide.

---

<sup>43</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_1)

<sup>44</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_5\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_5_1)

<sup>45</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_6\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_6_1)

<sup>46</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_9\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_9_1)

<sup>47</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_7](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_7)

<sup>48</sup> <http://scripts.sil.org/>

## 4.3 Consensus between stakeholders

The third key is *consensus between stakeholders* — people who have some sort of interest or investment in the project. Communication is often a neglected and overlooked component of WSI projects. Yet it is this which is the greatest cause of delays and unfinished efforts. The stakeholders need to make a commitment to:

- **Agreement upon goals.** It is important for those planning WSIs to work toward clearly understood and agreed upon goals and development timelines. Assumptions should be stated clearly, as should known limitations on what can be expected from the WSI.
- **Communication on technical issues.** There ought to be regular, scheduled meetings to review project status, identify roadblocks and solutions, and discuss technical details. Experience has shown that without such meetings, many WSI projects can flounder and lose momentum. Recent advances in communication technologies can be a great help — tele-and video-conferencing, instant messaging, web logs, and the like can make communication easier and more frequent. Even a simple telephone call to a fellow developer can save hours of research.
- **Early agreement regarding distribution and IP issues.** WSI planners tend to avoid discussion of these topics, especially if the WSI is being built up from small parts contributed by a variety of sources. Early discussion of ownership and revenue may seem to spoil the cooperative spirit, but if those issues are left until later, serious disagreement can result and threaten the whole project.

The objective should be to have a written record of any agreements, however minor, and to develop a working procedure that promotes and encourages regular interaction between stakeholders and between technical personnel.

## 4.4 Adequate software applications

The fourth key is the *existence of adequate software applications*. WSIs do not exist in a vacuum, and must work within specific software applications. So it is important that those operating systems and applications provide adequate support for the technologies that have been chosen. Applications chosen for WSIs need to:

- **Support necessary behaviors.** The application should support all basic functions needed for the writing system, or allow other WSI components to override default behaviors. For example, an application for processing text in an Arabic-based script obviously needs to support right-to-left text direction, but it should also manage right-to-left paragraph shaping and (usually) mixed direction line layout.
- **Be reasonably extensible in how they handle text.** Proprietary “black boxes” should be avoided, as they often do not allow enough control over script behaviors. An example of this would be a Hindi word processor that uses its own special routines to handle vowel rearrangement and conjunct formation, and does not document those routines, or allow them to be modified. Such restrictions can make WSI development for minority languages difficult if not impossible. In the Hindi example, it might be better to use an application that is based upon more open and well-defined technologies such as OpenType and Uniscribe.
- **Allow for adequate import/export.** Even if a certain WSI is the first and only one for a writing system, and data import and export are not important, it is very likely that it will not remain the only WSI in the future. Care must be taken to ensure that textual data does not get imprisoned within a single application. Hardware and software both become obsolete with time, and the inability to migrate data out of (and back into) a system can lead to unrecoverable data loss.

This key is usually outside the control of the WSI developers, so the objective is not to modify the applications, but to have a clear understanding of their capabilities (as highlighted in [Section 4.2](#)). It can be productive to actively discuss needs with application developers, but it is very risky to assume that any changes will be made until those changes appear in released versions of the software.

## 4.5 Inclusion of characters in Unicode

The final key to success is the *inclusion of all needed characters in the Unicode Standard*. It is still possible to create WSIs that do not use Unicode, but more and more applications are becoming Unicode-based, and require text to be in that encoding. Both Windows and Macintosh operating systems now use Unicode internally, and the major application development frameworks do as well.

If even a single character is missing, the cost to the WSI development process can be great — applications may not support shaping behaviors, import and export of text data may be compromised, etc. For example, if a special character from an Arabic-based script is not yet in Unicode, it may be impossible to get an application to recognize that character and give it the appropriate initial, medial and final forms.

The final objective and goal of this key is to verify that every character in the writing system has been accepted into the Unicode Standard, and if it has not, to navigate through the Unicode proposal process until that is the case. The good news is that thanks to the hard work of many people over the years, the great majority of scripts and characters that will be needed for WSIs are already in place. If that is not the case for a WSI need, then a strong commitment to the process — and a good deal of patience — will be required.<sup>49</sup>

---

<sup>49</sup> For more information on Unicode in general, see [Section 6.2](#)<sup>49</sup>. For details of how to add new characters or scripts to Unicode see [Section 6.3](#)<sup>49</sup>.

## Section 5

# Technical Details: Characters, Codepoints, Glyphs

*Author: Peter Constable*

Software systems used for working with multilingual data are evolving, and it is increasingly important for users and support personnel to have an understanding of how these systems work. This section serves as an introduction to remaining technical sections, and explains some of the most basic concepts involved in working with multilingual text: characters, keystrokes, codepoints, and glyphs. Each notion is explained, as is the way they relate to one another and interact within a computer system.

## 5.1 Characters

There are, in fact, different senses of the word character that are important for us. In common usage, though, the distinctions are not typically recognized. These differences must be understood in working with multilingual software technologies.

### 5.1.1 Orthographies, characters and graphemes

The first and most common sense of the term character has to do with orthographies and writing systems: languages are written using orthographies,<sup>50</sup> and a character in this first sense, an orthographic character, is an element within an orthography. For example, the lower case letter “a” used to write English, the letter “ၵၵ” used for Tai Lue, and the IPA symbol for a voiced, inter-dental fricative, “ð”, are characters.

It is easy to provide clear examples of characters in this sense of the word. Providing a formal definition is not so easy, though. To see why, let’s consider some of the things that can occur in an orthography.

Some orthographies contain elements that are complex, using multiple components to write a single sound. For example, in Spanish as well as in Slovak, “ch” functions as a single unit. This is an example of what is sometimes called a **digraph**. Some languages may have orthographies with even more complex elements. For instance, the orthographies of some languages of Africa have elements composed of three letters, such as “ngb”. Such combinations of two or more letters or written symbols that are used together within an orthography to represent a single sound are sometimes referred to as **multigraphs** or **polygraphs**.

Also, many languages use dependent symbols known as **accents** or **diacritics**. These are common in European languages; for example, in Danish “e” and “æ”, or French “e” and “ô”.

So, are multigraphs one character or several characters? And are the diacritics by themselves considered characters? There are not always clear answers to these kinds of questions. For a given written symbol, different individuals or cultures may have different perceptions of that symbol based on their own use of it. Speakers of English would not recognize the dot in “i” as a character, but they also would not hesitate to acknowledge the components of the digraph “th” as characters since “t” and “h” function independently in English orthography. The case of “th” might not be as clear to users of another language if, suppose, that

---

<sup>50</sup> The familiar term **orthography** is used here in place of the more correct and more specialized but less well-known term **writing system**. Writing systems include not only conventional systems of graphic objects used for written linguistic communication—commonly known as **orthographies**, but also systems of written notation used to describe or transcribe language and linguistic utterances, such as IPA or shorthand.

language does not make independent use of “h”. Likewise, English speakers would probably not be as confident in commenting about the ring in “a”.

We might avoid this uncertainty by using a distinct term: **grapheme**. A *grapheme is anything that functions as a distinct unit within an orthography*. By this definition, the status of multigraphs are clear: multigraphs, such as Spanish “ch”, and “ngb” in the orthography of some Bantu languages, are all graphemes.<sup>51</sup> Diacritics, either by themselves or in combination with base letters, may or may not be graphemes, depending on whether they function as distinct units with an orthography.

The notion of **grapheme** is important for us. Obviously, though, it would still be helpful to be able to talk about things like the “h” in “th” or the ring diacritic in general terms, even if they don’t correspond to a grapheme in a given orthography. The best we can do for the moment is to have an approximate, informal definition: when speaking in terms of writing systems and orthographies, a **character** (or **orthographic character**) is a *written symbol that is conventionally perceived as a distinct unit of writing in some writing system*.

### 5.1.2 Characters as elements of textual information

A second sense of the term **character**, important to WSI development, is particularly applicable within the domain of information systems and computers: *a minimal unit of textual information that is used within an information system*. In any given case, this definition may or may not correspond exactly with either our informal sense of the term **character** (i.e. **orthographic character**) or with the term **grapheme**. This will be made clearer as we consider some examples.

Note that this definition for **character** is dependent upon a given system. Just as the definition we gave for **grapheme** was dependent upon a given orthography, such that something might be a grapheme in one orthography but not in another, so also something may exist as a character in one information system but not in another.

For example, a computer system may represent the French word “hôtel” by storing a string consisting of six elements with meanings suggested by the sequence <h, o, ^, t, e, l>. Each of those six component elements, which are directly encoded in the system as minimal units, is a character within that system.

Note that a different system could have represented the same French word differently by using a sequence of five elements, <h, o, t, e, l>. In this system, the O-CIRCUMFLEX is a single encoded element, and hence is a character in that system. This is different from the first system, in which O and CIRCUMFLEX were separate characters.

Up to now the characters we have considered are all visible, orthographic objects (or are direct representations of such graphical objects within an information system). In using computers to work with text, we also need to define other characters of a more abstract nature that may not be visible objects, such as “horizontal tab”, “carriage return” and “line feed”.

In technical discussions related to information systems, in talking about multilingual software, for example, it is the sense of the term **character** discussed in this section that is usually assumed. From here on, we will adopt that usage, referring to (**abstract**) **characters** as meaning units of textual information in a computer system, and using the term **grapheme** when talking about units within orthographies. Thus, we might say something like, “The Dutch grapheme ‘ij’ is represented in most systems as a character sequence, <i, j>, but in this system as a single character, <ij>.” Where we wish to speak of (**orthographic**) **characters** in the informal sense discussed above, we will state that explicitly.

In developing a system for working with multilingual text, it is important to understand the distinction between abstract characters and graphemes. We implement systems to serve the needs of users, and users think in terms of the concrete objects with which they are familiar: the graphemes and orthographic characters that make up orthographies. They do not need to be aware of the internal workings of the system. In other words, it does not matter what abstract characters are used internally to represent text, just so long as users get the behavior and results they expect.

<sup>51</sup> Note that graphemes are not necessarily related to phonemes. For example, the English phoneme /θ/ is written as “th”, but “th” does not function as a unit in terms of the behaviors of English orthography.

## 5.2 Codepoints and Glyphs

Potentially, though, the characters may be somewhat different from the elements of the orthography, just so long as the system can be implemented to give the right behavior. To explain how this might be possible, though, we first need to understand the key components that are used in a computer system for working with text.

Within a computer system, we work with text and writing systems primarily in terms of three main components:

- input methods: how we create the data, typically using keyboards
- encodings: how the data are stored, and
- fonts & rendering: how the data is displayed.

We have talked about graphemes and abstract characters as individual units. We need to look at their counterparts in these three components, and understand how they interact. These counterparts are **keystrokes**, **codepoints**, and **glyphs**. In this section, we will introduce **codepoints** and **glyphs**, and look at how they interact within a system.

### 5.2.1 Codepoints

Computers store information internally as numbers. A **codepoint** is merely *the number that is used to store an abstract character in the computer*.<sup>52</sup> When working with text, each abstract character of the text (including control characters) is stored as a number with a unique number assigned to each different character. For example, whenever you enter **[SHIFT]+[F]** on an English keyboard, the computer will (on most systems) insert the number 70 at the current position in the data. This is the codepoint that is used on those systems for storing the character LATIN CAPITAL LETTER F.

An **encoding** (or **character encoding**) is a system of numbers used to represent a set of characters within a digital information system, such as a computer. There is, in principle, nothing special about the numbers that are used. For instance, in the example above, there is no *a priori* reason that the number 42 could not have been used to represent LATIN CAPITAL LETTER F. The actual numbers that are used are specified by the encoding designer. There are only two necessary restrictions:

- Each abstract character that is represented in the encoding system must have exactly one numerical representation—one codepoint.
- In order for users to exchange data, their computer systems must agree on what the meaning is for a given number.

To achieve the latter end, encoding standards are devised, either by individual vendors, or across the entire industry. Two important examples of encoding standards are ASCII and Unicode. Every DOS, Windows and Macintosh computer understands the ASCII encoding standard and would know, for example, that the codepoint 104 corresponds to the character LATIN SMALL LETTER H (“h”).

The numerical value of codepoints can be expressed in different ways. Most computer users are aware that computers store numbers in binary rather than decimal. Thus, 70 to us would be 01000110 to a computer. Programmers often use a system known as **hexadecimal**, or **hex**. Thus, decimal 70 would also be x46 (hex) to a programmer. Anyone involved in writing system implementation ought to be at least familiar with hex notation. For example, in any discussion of Unicode, codepoint values are almost always expressed using hex.

---

<sup>52</sup> There is more that can be said about codepoints and encoding schemes than we have space to discuss here, particularly concerning multi-byte encodings. For more information regarding encoding schemes, see The Unicode Standard, Version 3.0, and especially Unicode Technical Report #17. For information on multi-byte encodings that are used for East Asian character sets, see Lunde [LUN1999].

## 5.2.2 Glyphs

Glyphs are the graphical elements used to visually represent characters. Because of their graphical nature, a user is likely to associate them closely with the (relatively) concrete objects in the domain of writing and orthographies. For our purposes, the notion of **glyph** has an additional, specific characteristic: glyphs are *graphic objects that are stored within a font*. Basically, they are the shapes for displaying characters that you see on a screen or a printer. In a simple sense, then, a font is simply a collection of glyphs, usually with common design characteristics. Since glyphs are contained within fonts, which are part of a computer system, glyphs are therefore a component within the domain of information systems, like abstract characters.

The notions of **character** and **glyph** are different. For example, LATIN SMALL LETTER A “a” can be displayed using any of a number of different glyphs:



Figure 1: different fonts: one character, different glyphs

In some scripts, characters can have more than one shape due to certain behaviors of the script. This has nothing to do with changing fonts. For example, in Greek script, the sigma has two different shapes, according to its position within a word. Sigma can be displayed by more than one glyph, but in each instance only one glyph is used.



Figure 2: Greek sigma: one abstract character, two glyphs

There can also be situations in which a single character is displayed by multiple glyphs in every instance of its use. For example, Indic scripts of South and Southeast Asia are well known for having vowels that are written using multiple shapes that are distributed around the shape for the initial consonant. So, for example, in the following Bengali-script example, the two highlighted shapes represent a single instance of one character (one grapheme), the vowel o:



Figure 3: Bengali VOWEL O: one character displayed using two discontinuous glyphs

We have seen that one character can have many glyphs. The opposite is also possible: one glyph for multiple characters. In Lanna script, when the character ๒ is followed by the character ๓, they may be written as ๒๓. In this case, we have two characters that are presented by a single shape, forming what is known as a **ligature**.

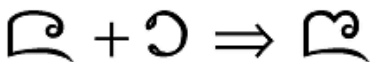


Figure 4: Lanna ligature: two characters, one glyph

These examples suggest that the number of glyphs is determined by the character elements in an orthography and by their behaviors. That is largely true, but not necessarily so, however. The glyphs used in a font are determined by the font designer, and a font designer may choose to implement behaviors in different ways. For example, for Greek, a font designer may choose to present using a single, composite glyph, or by using two glyphs, one for alpha and another for the over-striking diacritic:



single, composite glyph

alpha glyph + overstriking oxia



Figure 5: Alternate glyph implementations for Greek alpha with oxia

Some font implementations may even use glyphs that only represent a portion of the written symbols in the orthography:

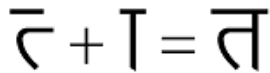


Figure 6: Glyphs for portions of a Gujarati character

These examples raise some important questions: Does this mean that, within a computer system, there can be a mismatch between the characters that are stored and the glyphs that are needed to display them? If so, how is this handled? This brings us to the general issue of how glyphs relate to characters within an information system.

### 5.2.3 From codepoints to glyphs

Textual information is stored within a computer as codepoints that directly correspond to abstract characters. In a process known as **rendering**, software will take a sequence of codepoints and transform that into a sequence of glyphs displayed on some presentation device (a monitor or a printer).

Consider a simple example: the English word “picture”. As this document was created, that word was stored on the computer as a string of seven characters, <p, i, c, t, u, r, e>, and was displayed on the monitor using seven glyphs selected from a particular font and arranged horizontally (using information also found in the font to control relative positioning). In this case, there was a simple one-to-one mapping between the codepoints in the data and the glyphs in the font.

That much was fairly obvious. What is more interesting is what happens in the more complicated situations described above, in which there is not a one-to-one relationship between “characters” and glyphs. In general, the answer is that it depends upon the given system. But to see what might possibly happen, let’s consider the same English example again, yet with a twist.

Suppose I am a font designer, and I want to create a font that closely mimics my handwriting. Of course, I will write English letters in many different ways, and I can’t capture every subtle variation. If I am willing to stay within some limits, though, perhaps I can have my font show each letter the way I might typically write it within a certain set of combinations. So, for example, I might write “c” with different types of connection on the left: in some instances with no connection (at the beginning of words, say), or with a connection near the bottom (after letters that terminate near the bottom, such as “a”), or in other instances with a connection near the top (for instance, after “o”).

As I work through all the details, I might actually decide that the only way to really get the word “picture” to display the way I want is to create it as a single, complex glyph for the entire word. (This may seem unusual, but such things are possible in fonts, and some fonts have even done things like this.) So, I have a single glyph for “picture”, but this is stored in data as a sequence of seven characters. What I need, then, is for some process to intervene between the stored data and the glyphs that will recognize this particular sequence of characters and select the single, combined glyph, rather than seven separate glyphs.

This is precisely the kind of processing that happens in modern systems that are designed to support complex scripts. These systems are sometimes referred to as “smart font” or “smart rendering” systems. Examples include Apple’s **TrueType GX**, which has more recently been renamed as **Apple Advanced Typography (AAT)**; the **OpenType** font standard, developed by Microsoft and Adobe; and SIL’s **Graphite** rendering system. It would go beyond the scope of this discussion to examine how these systems work in any detail. The main point to grasp is that they mediate between characters that are



stored and the glyphs used to display them, and allow there to be complex processes that give many-to-many mappings between sequences of characters and sequences of positioned glyphs.

So let's revisit the more typical examples presented above, and consider how the rendering process might apply in those cases. First, we saw that the Greek sigma is displayed using different shapes according to word position. Within a system, the single grapheme sigma can be represented as a single character, SIGMA, and a rendering process will determine when it does or does not occur at the end of a word and select glyphs on that basis:

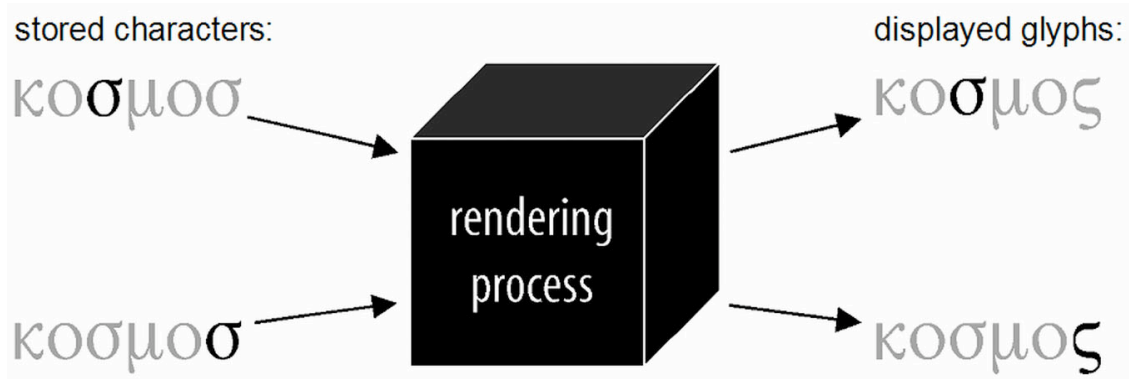


Figure 7: Complex rendering process for Greek SIGMA

In the case of Bengali, a similar process may occur: the system may store a sequence of two characters, <LETTER KA, VOWEL O>, and the rendering process will somehow transform that into the appropriate sequence of glyphs. The actual number of glyphs would be dependent upon a particular font implementation. It could be one composite glyph for the entire combination of characters. More likely, though, it would be rendered using three glyphs. Note, though, that the ordering of the three glyphs does not correspond to the ordering of the stored characters.

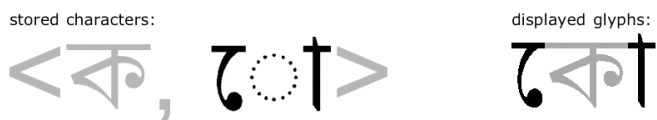


Figure 8: Complex rendering process for Bengali <LETTER KA, VOWEL O>

Similar processing could occur in rendering the Lanna ligature. In that case, an implementation will likely involve two stored characters displayed using a single glyph.

In these examples, we have described one way in which support for each of these examples can be implemented. But, as has been mentioned, the actual glyphs and number of glyphs can vary from one implementation to another. In the case of Bengali, for instance, we have seen that the grapheme for the /o/ vowel ("৩") can be represented in terms of a single character, VOWEL O. But another system could perhaps implement support for this grapheme using a pair of characters, <৩, ৩>. This might make particular sense if each of these corresponded to other graphemes in the orthography or script being implemented. For Bengali script, it turns out that these do have separate identities. Thus, many systems would represent "৩" using a character sequence of <VOWEL E, VOWEL AA>. Of course, such a difference would have an affect on how the rendering process needs to operate in order to generate the correct sequence of glyphs. The point here is that a smart rendering system that can support many-to-many mappings between characters and glyphs makes it possible to have different implementations for a given writing system. This flexibility can provide alternatives for a developer, or can also be used to provide special functionality for particular purposes.

Smart rendering systems that can handle many-to-many transformations from characters to glyphs are relatively new. In the past, computer software has typically used rendering systems that support only one-to-one relationships between characters and glyphs. Such systems are sometimes known as "basic", or "dumb" rendering systems. For a writing system like that of English, for which the standard behaviors are

very simple, a “dumb” rendering system is adequate for most use. For complex scripts, however, this limitation presents a problem.

For instance, if a Greek SIGMA requires context-based glyph selection but the system is limited to only one glyph per character, then the only possible solution is to have more than one SIGMA character: one character for each of the two glyphs. Since the mapping from characters to glyphs is a simple, one-to-one mapping, the rendering process becomes essentially transparent:

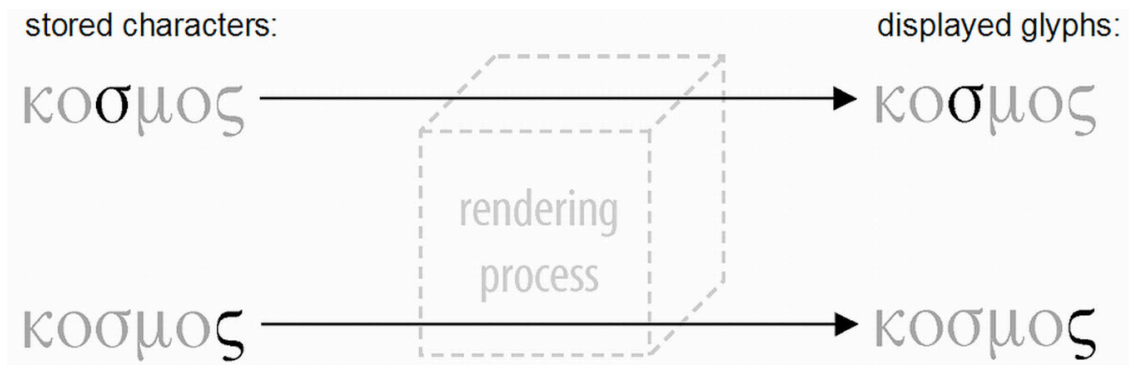


Figure 9: Greek SIGMA: presentation-form encoding and rendering

This approach to implementation is important for us to understand. It is important not because it reflects good practice or good technology—it is neither. Rather, it is important because it has been used for many years in a large number of implementations to support writing systems that involved complex behaviors, ranging from Arabic to IPA to Thai. This way of implementing a writing system imposes requirements not on the glyphs, but on the abstract characters: since there is a one-to-one mapping from characters to glyphs, one abstract character is required for every glyph that is needed. For this reason, encoding systems that are designed to work in this way are often referred to as “glyph encodings”, “display encodings”, or “presentation-form encodings”.

In general, an encoding should be devised to accommodate the needs of all processes that need to be performed on the text: rendering, input, sorting, searching, etc. In the case of a presentation-form encoding, however, the encoding is designed to accommodate the needs of rendering alone. If any other processes can still be performed without additional processing, that is coincidental. In most situations, however, other processes are made significantly more difficult, or are considered expendable.

## 5.3 Keystrokes and codepoints

Input methods represent the third of three basic components for working with text data on a computer that were introduced earlier. In general, input methods can include things like voice- or handwriting-recognition. Keyboards are the most common form of input, however, and also the only one that is easily extended or modified. This discussion will therefore focus on keyboard input.<sup>53</sup>

### 5.3.1 From keystrokes to codepoints

Just as codepoints and glyphs are the counterparts to characters in the encoding and rendering components, keystrokes are the counterpart to characters in the keyboarding component. Whereas characters (or codepoints) get transformed into glyphs in the rendering process, keystrokes are transformed into codepoints in the input process.

All computer operating systems include software to handle keyboard input, and many provide more than one keyboard layout; that is, more than one set of mappings from keys to characters. Many keyboard

<sup>53</sup> It will not, however, cover systems for Far Eastern scripts (Chinese, Japanese, Korean) that typically require sophisticated **input method editors (IMEs)**.

input methods use a strictly one-to-one mappings from keystrokes to characters: for each keystroke, there is one particular character that is generated.<sup>54</sup> But some keyboards provide alternative mappings based on a previously-entered “dead key” (a key that does not enter a character, but rather changes the character entered by the following key). For example, typing “” followed by “a” to get “á”, but “” followed by “o” to get “ó”.

Just as the mapping from characters to glyph might involve complex, many-to-many mappings, the same is potentially true for keyboard input. For example, it would be possible to have a single keystroke that generated a sequence of several characters, such as <n, g, b>, or <GREEK SMALL ALPHA, COMBINING ROUGH BREATHING MARK, COMBINING ACUTE, COMBINING IOTA SUBSCRIPT>. Similarly, it would be possible for a sequence of keystrokes to generate single characters, perhaps with each keystroke in the sequence changing the previous output.















| keystroke sequence:   | resulting character output: |
|---|-----------------------------|
|  ,    | <k, a>                      |
|  ,  ,    | <k, á>                      |
|  ,  ,  ,    | <k, o>                      |
|  ,  ,  ,  ,  | <k, ñ>                      |

Figure 10: Possible keystroke sequences mapping to single characters (hypothetical)

Different input methods might generate exactly the same characters, though in different ways. For example, one may use a single keystroke to generate a given character, while another uses two or more keystrokes (the first, perhaps, being a **dead key**) to generate that character.












|                                  | keystroke sequence:  | resulting character output: |
|----------------------------------|--|-----------------------------|
| system A: two keystroke sequence |  ,  | < á >                       |
| system B: single keystroke       |                                      | < á >                       |

Figure 11: Two input methods: same character, different number of keystrokes

Complex input methods can map a single keystroke to different characters, depending on context. For example, in the case of Greek sigma, an input method may use a single key, such as the  key, to enter all forms of sigma, with the input method generating either a FINAL SIGMA or NON-FINAL SIGMA according to the context:

<sup>54</sup> We will use the term **keystroke** to refer to the pressing of any basic (non-modifier) key in combination with zero or more modifier keys. By modifier keys, we mean keys such as , , , , , and .

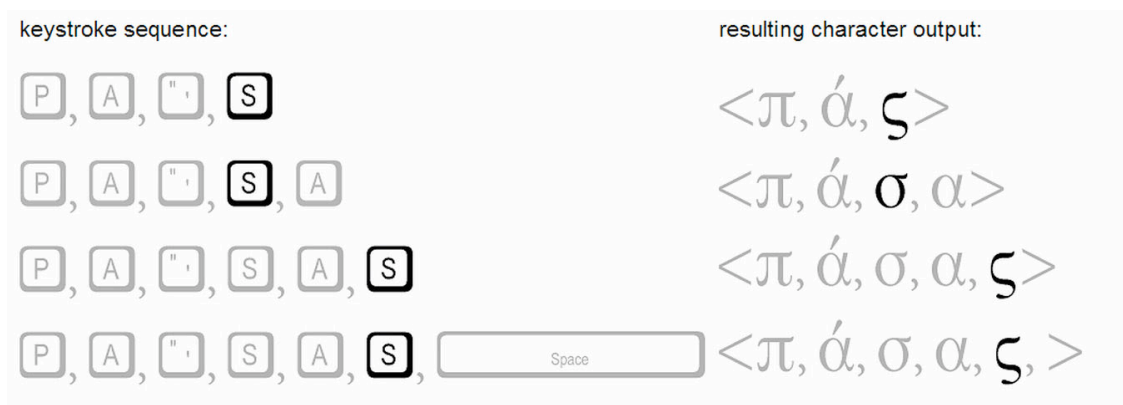


Figure 12: Greek sigma: keyboard input of presentation-form-encoded data

Note that, when the [S] key is pressed, the sigma is at that point word-final. It is the next character that is entered that determines whether or not the sigma will remain word-final. If another word-forming character is entered, then the FINAL SIGMA is changed to NON-FINAL SIGMA.<sup>55</sup>

The point to see in this discussion of input methods is that the mapping of keystrokes into characters is potentially a complex process involving many-to-many mappings, just as in the rendering process. Users familiar with systems designed to support English would certainly be familiar with keyboards that use one-to-one mappings only. But keyboard processing need not be limited in this manner, and many keyboard implementations are not.

## 5.4 Further reading

There are many related topics that are not addressed in this brief introduction. The next four sections go into more detail on encodings (including Unicode), data entry, glyph design and rendering. The following resources may also provide greater depth into particular topics:

To learn more in detail about codepoints and the ways in which abstract characters are encoded in terms of bytes, see Unicode Technical Report #17. This discusses these issues specifically in relation to the Unicode standard, but that can be helpful for understanding the issues in general since the Unicode standard actually has multiple ways to encode its characters in terms of byte sequences. The introduction to Graham [GRA2000] is also somewhat useful in this regard (although this has some unfortunate typographic errors). Section 6 of this document also discusses these issues.

For more information on character encodings used with Microsoft Windows, see Constable [CON2000b], Kano [KAN1995], or Hosken [HOS1997]. For information on multi-byte character encoding standards for Chinese, Japanese, Korean and Vietnamese, see Lunde [LUN1999]. Information on a variety of character encoding standards is available in the reference section of the Microsoft Global Software Development Web site: <http://www.microsoft.com/globaldev/default.asp>. The introduction to Graham [GRA2000] is also somewhat useful.

For further information on implementing software to work with multilingual text on Microsoft Windows, see Kano [KAN1995], Constable [CON2000a], and Hosken [HOS1997]. Other useful information is available at the Microsoft Global Software Development Web site (see above), and in the Microsoft Developer Network Library, which is available online at <http://msdn.microsoft.com/library/default.asp>.

For greater depth on a variety of topics related to WSIs, see Lyons [LYO2001].

<sup>55</sup> As to be expected, there are many possible implementations for a Greek keyboard. Another implementation might provide separate keystrokes for final versus non-final characters.

## 5.5 References

[CON2000a] Constable, Peter. Understanding multilingual software on MS Windows. Dallas, TX: SIL International, 2000. Available in CTC Resource Collection 2000 CD-ROM, by SIL International.

[CON2000b] Constable, Peter. Unicode issues in Microsoft Word 97 and Word 2000. Dallas, TX: SIL International, 2000. Available in CTC Resource Collection 2000 CD-ROM, by SIL International.

[GRA2000] Graham, Tony. Unicode: A primer. Foster City, CA: M&T Books, 2000.

[HOS1997] Hosken, Martin. Windows and codepages. NRSI Update #8. Dallas, TX: SIL International, 1997. Available in Resource Collection 98 CD, by International Publishing Services. Dallas, SIL International. Also available in the CTC Resource Collection 2000 CD-ROM, by SIL International.

[KAN1995] Kano, Nadine. Developing international software for Windows® 95 and Windows NT™. Redmond, WA: Microsoft Press, 1995. Also available online at <http://msdn.microsoft.com/library/books/devintl/S24AE.HTM>.

[LUN1999] Lunde, Ken. CJK Information Processing. Sebastopol, CA: O'Reilly, 1999.

[LYO2001] Lyons, Melinda, ed.. Implementing Writing Systems: An Introduction. Dallas, TX: SIL International, 2001.

[WHI2000] Whistler, Ken, and Mark Davis. Unicode Technical Report #17: Character Encoding Model. Available online at <http://www.unicode.org/unicode/reports/tr17/>

## Section 6

### Technical Details: Encoding and Unicode

---

*Author: Peter Constable*

#### 6.1 An Introduction to Encodings

Computer systems employ a wide variety of character encodings. The most important of these is Unicode. It is also important for us to understand other encodings, however, and how they relate to Unicode. This section introduces basic encoding concepts, briefly mentions legacy encodings, and gives an introduction to Unicode. It also describes the process of interacting with the Unicode Consortium to get new characters and scripts accepted into the standard.

##### 6.1.1 Text as numbers

**Encoding** refers to the process of representing information in some form. In computer systems, we encode written language by representing the **graphemes** or other **text elements** of the writing system in terms of sequences of **characters**, units of textual information within some system for representing written texts. These characters are in turn represented within a computer in terms of the only means of representation the computer knows how to work with: binary numbers.

A **character set encoding** (or **character encoding**) is such a system for doing this. Any character set encoding involves at least these two components: a set of characters and some system for representing these in terms of the processing units used within the computer.

##### 6.1.2 Industry standard legacy encodings

Encoding standards are important for at least two reasons. First, they provide a basis for software developers to create software that provides appropriate text behaviors. Secondly, they make it possible for data to be exchanged between users.

The ASCII standard was among the earliest encoding standards, and was minimally adequate for US English text. It was not minimally adequate for British English, however, let alone fully adequate for English-language publishing or for most any other language. Not surprisingly, it did not take long for new standards to proliferate. These have come from two sources: standards bodies and independent software vendors.

Software vendors have often developed encoding standards to meet the needs of a particular product in relation to a particular market. Among personal computer vendors, Apple created various standards that differed from IBM and Microsoft standards in order to suit the distinctive graphical nature of the Macintosh product line. Similarly, as Microsoft began development of Windows, the needs of the graphical environment led them to develop new codepages—ways of encoding character sets. These are the familiar Windows codepages, such as codepage 1252, alternately known as “Western”, “Latin 1” or “ANSI”.

The other main source of encoding standards is national or international standards bodies. A national standards body may take a vendor’s standard and promote it to the level of a national standard, or they may create new encoding standards apart from existing vendor standards. In some cases, a national standard may be adopted as an international standard, as was the case with ASCII.

It is important to understand the relationship between industry standard encodings and individual software products. Any commercial software product is explicitly designed to support a specific collection of character set encoding standards.

Of course, the problem with software based solely on standards is that, if you need to work with a character set that your software does not understand, then you are stuck. This happens because software vendors have designed their products with specific markets in mind, and those markets have essentially never included people groups that are economically undeveloped or are not accessible to the vendor. This is not unfair on the part of software vendors; they can only support something they know about and that is predictable, implying a standard.

When the available software does not support the writing systems they need to work with, linguists and others create their own solutions. They define their own character set and encoding, they “hack” out fonts that support that character set using that encoding so that they can view data, they create input methods (keyboards) to support that character set and encoding so that they can create data, and then go to work.

Such practice is quite a reasonable thing to do from the perspective of doing what it takes to get work done. People who have needed to resort to this have been quite resourceful in creating their own solutions. There is a dark side to this, however. Although the user has defined a custom codepage, the software they are using is generally still assuming that some industry standard encoding is being used.

The most serious problem with custom codepages, which affects data archiving and interchange, is that the data is useless apart from a custom font. Dependence on a particular font creates lots of hassles when exchanging data: you always have to send someone a font whenever you send them a document, or make sure they already have it.

One context in which this is a particular problem is the Web. People often work around the problem by using Adobe Acrobat (PDF) format, but for some situations, including the Web, this is a definite limitation. If data is being sent to a publisher, who will need to edit and typeset the document, Acrobat is not a good option<sup>56</sup>. Custom codepages are especially a problem for a publisher who receives content from multiple sources since they are forced to juggle and maintain a number of proprietary fonts. Furthermore, if they are forced to use these fonts, they are hindered in their ability to control the design aspects of the publication.

The right way to avoid all of these problems is to follow a standard encoding that includes these characters. This is precisely the type of solution that is made possible by Unicode, which is being developed to have a universal character set that covers all of the scripts in the world.

## 6.2 An Introduction to Unicode

Unicode is an industry standard character set encoding developed and maintained by The Unicode® Consortium. The Unicode character set has the capacity 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 96,000 characters representing a large number of scripts. 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.

### 6.2.1 A brief history of Unicode

In order to understand Unicode, it is helpful to know a little about the history of its development. By the early 1980s, the software industry was starting to recognize the need for a solution to the problems involved with using multiple character encoding standards. Inspired by early innovative work by Xerox, 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.

---

<sup>56</sup> Once a document is typeset and ready for press, however, Acrobat format is generally a good option.

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.<sup>57</sup>

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 international 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.

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.

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. The most current version at this time, version 4.0, was published in 2003 (The Unicode Consortium, 2003). As of this version, the Standard includes a total of 96,447 encoded characters.

## 6.2.2 The Unicode Consortium and the maintenance of the Unicode Standard

The [Unicode Consortium](http://www.unicode.org/unicode/consortium/consort.html)<sup>58</sup> is a not-for-profit organization 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 organizational members, not to individuals; and there is an annual fee of US\$12,000. At the time of writing, there are currently [15 Full Members](http://www.unicode.org/unicode/consortium/memblogo.html)<sup>59</sup>.

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 attend meetings.

---

<sup>57</sup> The abbreviations ISO and IEC stand for *The International Organization for Standardization (ISO)* 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.

<sup>58</sup> <http://www.unicode.org/unicode/consortium/consort.html>

<sup>59</sup> <http://www.unicode.org/unicode/consortium/memblogo.html>



### 6.2.3 Unicode and ISO/IEC 10646

The UTC maintains a liaison 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.

Because the process of developing the Unicode Standard involves interaction with ISO/IEC and the international standard ISO/IEC 10646, it is worth mentioning briefly the workings of the international standards body as it relates to Unicode. The Joint Technical Committee 1 (JTC1) of ISO and IEC is responsible for standards related to information technologies, and the work of this technical committee is divided among multiple sub-committees. Sub-committee 2 (JTC1/SC 2) is responsible for standards related to character encoding, and that work is divided among various working groups. Among these, working group 2 (JTC 1/SC 2/WG 2) is responsible for the development of ISO/IEC 10646.

The combined standards body ISO/IEC is an international standards organization — the members of which are national standards bodies from various countries. Standards bodies from any country are potentially eligible to participate in the work of any ISO or ISO/IEC technical committee or sub-committee, including work on ISO/IEC 10646.

In order to ensure quality standards that facilitate domestic and international commerce without providing unfair advantage to certain countries over others, a very formal process is used that includes several stages of review and balloting before something is published as part of an international standard. Thus, if a standards institute of a given country wishes to influence the development of ISO/IEC 10646 (and, in turn, Unicode), they should become a member of ISO/IEC, become a participating member of JTC 1/SC 2, and then actively contribute to the work by voting on ballots, preparing and commenting on draft revisions, and attending meetings of JTC 1/SC 2/WG 2 whenever possible.

### 6.2.4 Types of information

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 The Unicode Standard (TUS) 4.0 (The Unicode Consortium, 2003).
- 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, and some even become parts of the Standard. A UTR with this status is identified as a **Unicode Standard Annex** (UAX). These annexes may include documentation of a minor version release, or information concerning specific implementation issues.
- 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](http://www.unicode.org)<sup>60</sup>. Further information on the data files is available at <http://www.unicode.org/unicode/onlinedat/online.html>.

Thus, the previous [version](#)<sup>61</sup> of Unicode, TUS 3.2, consists of the published book for TUS 3.0, plus the [UAXes](#)<sup>62</sup> that describes the minor versions for TUS 3.1 and TUS 3.2, UAX #27 and UAX #28 respectively, together with the current versions of the other annexes and data files.

---

<sup>60</sup> <http://www.unicode.org/Public/UNIDATA/>

<sup>61</sup> <http://www.unicode.org/unicode/standard/versions/>

<sup>62</sup> <http://www.unicode.org/unicode/standard/versions/enumeratedversions.html>

## 6.3 Adding new characters and scripts to Unicode

If a language community wants to have their writing system implemented for use in information technologies, then it will be very much to their advantage to see that the writing system is supported in key industry standards. Doing so will greatly increase the likelihood that the writing system is supported in off-the-shelf products from vendors around the world. In particular, it is important to have the characters of the writing system included in the Unicode Standard.

The Unicode Standard is a work in progress, and new characters and scripts are being added on an on-going basis. The Standard is also intended to be universal in coverage; thus, small size or limited economic/market potential of a community is not an obstacle.

In this section, we will discuss how one can know when a character or script needs to be proposed for addition to the Unicode Standard, and what steps should be taken in order to get characters or scripts added to the Standard.

### 6.3.1 Evaluating whether a writing system is already supported by Unicode

Before making a decision to request that new characters be added to the Unicode Standard, it is important first to ascertain that the characters are not, in fact, already supported by the Standard. This requires good familiarity with the Unicode Standard. If possible, someone with a thorough knowledge of the writing system should acquire a good understanding of the Standard. If that is not practical, then they should seek assistance from others with expertise in Unicode. Useful resources in either case include the [Unicode Web site](http://www.unicode.org)<sup>63</sup> and the [Unicode email discussion list](mailto:unicode-discuss@unicode.org)<sup>64</sup>; you can also contact the [Unicode Consortium](http://www.unicode.org)<sup>65</sup> for assistance.

Due to limited understanding of Unicode, novices in the workings of the Standard often conclude incorrectly that the language they are interested in is not supported. A few key concepts will help to avoid these mistakes.

The first is that Unicode encodes **characters**, not **phonemes**, and not **graphemes**. For instance, the character LATIN SMALL LETTER P is used to write a bilabial consonant phoneme in English and other European languages. If, suppose, there were a language for which this symbol were used to write a vowel phoneme, that would not imply that a distinct character is required: the same character LATIN SMALL LETTER P can be used, regardless of what type of phoneme is being represented for a given language.

A more frequent misconception involves graphemes: the orthography of a language might use digraphs or trigraphs to represent a particular phoneme. So, for instance, a combination of letters such as “kp” might be considered a distinct member of the alphabet for some language; it might even have its own place in the alphabetic order. This does not imply, however, that the combination “kp” needs to be encoded in Unicode as a distinct, atomic unit: Unicode assumes that multigraphs are encoded as sequences of characters, and that if they need to be recognized as a unit in relation to some text process for some language, that it is the job of software to provide that language-specific behavior.

Another key concept is that Unicode encodes [characters, not glyphs](http://www.unicode.org)<sup>66</sup>. For instance, the scripts of South Asia often use ligature forms (often referred to in the South-Asian context as **conjuncts**), but these are not encoded as distinct elements in Unicode. So, for instance, a Devanagari conjunct ksha “क्ष” is not represented directly in Unicode as an atomic element. Rather, it is represented as a sequence of characters, < ka “क”, virama “ँ”, ssha “ष” >. To avoid confusion regarding issues such as these, it is especially important to read the sections of the Unicode Standard that describe the particular script in question.

A third key concept is the principle of dynamic composition. The writing systems of many languages include elements that consist of base-diacritic combinations. Unicode includes many such combinations

---

<sup>63</sup> <http://www.unicode.org>

<sup>64</sup> <http://www.unicode.org/consortium/distlist.html>

<sup>65</sup> <http://www.unicode.org/contacts.html>

<sup>66</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_5\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_5_1)

as individual characters, but certainly not every possible combination. If the writing system for some language requires a particular combination that does not exist in Unicode as a separate character, that does not imply that this new combination should be added. Rather, Unicode follows a design principle of dynamic composition, meaning that such composite text elements can be represented using character sequences involving combinations of a base character followed by a character for the combining diacritic. So for instance, a grapheme w-tilde would be represented by the character sequence < w, combining tilde >.<sup>67</sup>

A final consideration worth mentioning is that the location of characters in the Unicode code charts is generally organized by scripts, but this is not an absolute rule. For practical reasons, the characters in a given writing system may be located in separate areas of the Unicode code space. For this reason, it is important in determining if and how a writing system can be represented in Unicode to become familiar with the many different portions of the Unicode code space.

The problem of determining whether or not something is already supported in Unicode is discussed on the Unicode Web site page [Where is my Character?](http://www.unicode.org/standard/where/)<sup>68</sup> Before preparing any proposal to add new characters to Unicode, it is recommended that this page be read, as well as the [FAQ pages](http://www.unicode.org/faq/)<sup>69</sup> on the Unicode Web site.

When investigating whether a writing system can be represented adequately using existing characters in Unicode, there may be situations in which it is unclear whether or not existing characters would be appropriate to use for a given purpose. It is recommended that questions regarding such issues be asked on the [Unicode email discussion list](mailto:unicode-discuss@unicode.org)<sup>70</sup>.

With these considerations in mind, it is certainly possible that a user may find that some or all of the characters in their writing system are not adequately supported in Unicode. If that is the case, the next steps will be to determine what would be an appropriate approach to encoding these characters or script, and then to submit a concrete proposal for addition of characters to the Standard.

### 6.3.2 Determining the best encoding solution

When a writing system cannot be adequately represented in Unicode, it is necessary to determine what revision to the Standard would constitute the most appropriate solution. In most cases, this will simply involve the addition of new characters to the Standard. In some situations, though, there may be alternate possible solutions that need to be evaluated, or there may be special considerations that require more careful consideration. It is recommended that solutions being considered are first discussed on the Unicode email discussion list. This will provide opportunity for review and may bring to light important issues that had been overlooked. It will also provide opportunity to learn whether there are others who have been working toward a proposal dealing with the same or similar problems.

### 6.3.3 Qualifying what it takes to get new characters or scripts added to Unicode

Once it has been determined that new characters or scripts need to be added to the Unicode Standard, it is time to begin the formal processes that are required to bring about a revision in an international industrial standard. This assumes that there is already a clear idea of what characters should be proposed, and how those characters should be used in representing text elements of the given writing system.

International industry standards need to be designed with care to ensure successful and problem-free implementation and utilization. For this reason, development of such standards typically proceeds slowly

---

<sup>67</sup> The reader may wonder why certain base-diacritic combinations exist in Unicode rather than being handled dynamically in the same manner. These were included only out of necessity for backward compatibility with pre-existing industry-standard character set encodings. Moreover, in each case, the same text element can be represented using either the precomposed character or the dynamically-composed character sequence, and it is expected that these different representations will be considered equivalent by software.

<sup>68</sup> <http://www.unicode.org/standard/where/>

<sup>69</sup> <http://www.unicode.org/faq/>

<sup>70</sup> <http://www.unicode.org/consortium/distlist.html>

and methodically. In order to get characters or scripts added to the Unicode Standard, three essential elements are needed:

- Individuals that understand the requirements of the given writing system, that also understand technical aspects of implementing support for writing systems in information technologies, and who can communicate such information clearly.

The Unicode Standard involves many technical details, and a clear understanding of what is required, in full technical detail, is needed in order to make additions to the Standard. Such requirements are derived from a combination of a thorough knowledge of the writing system in question together with a solid grasp of the technical aspects in implementing writing systems on computers. That information must be communicated clearly to the committees that will be making decisions regarding the proposed additions and, if approved, to those actually making detailed changes in the Standard.

- Committed participation of such individuals in the formal processes involved in revising the Standard.

While development of Unicode and ISO/IEC 10646 involves industry bodies, standards agencies and major corporations, ultimately the actual work is undertaken by a relatively small group of individuals that has remained largely the same over the course of several years. There is more work involved in development of the standards than these individuals can do on their own. Thus, it is important that a stakeholder in the development of this writing system—one of the individuals mentioned above—takes responsibility for seeing the proposed changes followed through to completion. They need to become the champion for the cause, ensuring that necessary documents are submitted and distributed to committee members, and that the issues are added to meeting agendas. If there isn't such a person promoting the changes that have been proposed, then those changes will not happen on their own.

- Time.

Getting from the preliminary evaluation of possible solutions through to publication of a new version of the Standard that includes the needed changes typically takes around two years. Those seeking the changes must set appropriate expectations in terms of the length of time required, and must be willing to remain committed to the process for the duration.

### 6.3.4 The formal process for adding new characters and scripts to Unicode

As mentioned earlier, the Unicode Standard is maintained in synchronization with the corresponding international standard ISO/IEC 10646, and any changes to either requires approval of both the Unicode Technical Committee (hereafter, UTC) and ISO/IEC JTC 1/SC 2/WG 2 (hereafter, WG2). The formal process to add new characters or scripts to these standards can be initiated with either body. It does not matter which body you approach first. If a proposal is submitted to WG2, then in the course of the proposal being approved as part of an amendment to ISO/IEC 10646 it will be reviewed by UTC. If a proposal is submitted first to UTC, they can approve it, but then they must submit it for consideration by WG2. Hence, a proposal will eventually be considered by both bodies either way.

WG2 members consist of national standards bodies from various countries. For this reason, initiating the process with WG2 is best done through a national standards body that is a participating member of JTC 1/SC 2. The list of participating member bodies is available from the [JTC 1/SC 2 Web site](#)<sup>71</sup>.

If your country is not represented in JTC 1/SC 2 by a national standards institute, an institute from your country can become a member. Alternately, you may find that one of the existing member bodies would be willing to assist in submitting and sponsoring the proposal. For instance, the national bodies from Ireland<sup>72</sup> and Japan have sponsored numerous proposals for characters and scripts from other parts of the world.

<sup>71</sup> [http://lucia.itsci.ipsj.or.jp/itsci/servlets/ScmMem10?Com\\_Id=02&PER\\_F=1](http://lucia.itsci.ipsj.or.jp/itsci/servlets/ScmMem10?Com_Id=02&PER_F=1)

<sup>72</sup> Mr. Michael Everson (Everson Typography) is the representative from the Irish national body to WG2, and is also a contributing editor of ISO/IEC 10646. He has submitted proposals on behalf of many different user communities for characters and scripts from many parts of the world. His contact information is available from the [Everson Typography Web site](#)<sup>72</sup>.

The other approach that may be taken is to initiate the process through UTC. Membership in the Unicode Consortium is not a requirement in order to submit a proposal to UTC. Information on [how to submit a proposal to UTC](#)<sup>73</sup> is available on the Unicode Web site. Whether a proposal is being submitted through UTC or WG2, it is recommended that you first read this page.

There are some third-party agencies that may be able to offer assistance in preparing and submitting proposals to UTC. If you have a working relationship with one of the [members of the Unicode Consortium](#)<sup>74</sup>, they may be able to offer assistance.

Also, the Department of Linguistics at the University of California, Berkeley has established the [Script Encoding Initiative](#)<sup>75</sup> for the express purpose of assisting in getting the remaining characters and scripts not yet in Unicode added to the Standard. Information on the *Script Encoding Initiative*, including how to contact them, is available from their Web site.

The Non-Roman Script Initiative (NRSI) of SIL International has also been involved in preparing proposals to get new characters and scripts added to Unicode, and may be able to provide assistance. Contact information is available from the [NRSI Web site](#)<sup>76</sup>.

Once a proposal has been submitted to either UTC or WG2, they may accept it, they may request clarification on various issues before they make any decision, or they may request certain revisions be made. They may also respond by rejecting the proposal, with reasons why; if there has been preliminary interaction as suggested in earlier sections, however, a situation resulting in a formal rejection is unlikely to occur.

UTC typically meets quarterly, and WG2, semi-annually. Formal action on proposals is taken during these meetings, but not between. If a proposal is accepted at a given meeting, the new characters are not yet formally part of either Unicode or ISO/IEC 10646. This will not happen in the case of Unicode until the next major or minor [version](#)<sup>77</sup> is released. For ISO/IEC 10646, the new characters must be published as part of an amendment to the international standard, and such amendments require multiple stages of review and balloting by member bodies.

New versions of the Unicode Standard are published a little more frequently than are amendments to ISO/IEC 10646. Depending on timing, it's conceivable that the span of time from when a proposal is submitted to UTC to the point at which those characters appear in a published version of Unicode could be as little as six months. More typically, though, you should anticipate that this will require 1 1/2 to 2 1/2 years.

### 6.3.5 Preparing a proposal to add new characters or scripts

Proposals for adding new characters or scripts must be communicated in written proposal documents. A proposal to add new characters or scripts must include a Proposal Summary Form,<sup>78</sup> and this document must use a specific form prepared by WG2. A template for the Proposal Summary Form is available on the WG2 Web site. WG2 also provides a document ("Principles and Procedures") with information that is useful in completing the Proposal Summary Form. You can also browse through a directory of previously-submitted proposals for examples you can follow.<sup>79</sup>

The purpose of the proposal documents is two-fold: to communicate clearly exactly what changes or additions to the Standard are being requested, and to provide reasonable justification for those changes or additions.

---

<sup>73</sup> <http://www.unicode.org/pending/proposals.html>

<sup>74</sup> <http://www.unicode.org/consortium/memblogo.html>

<sup>75</sup> <http://www.linguistics.berkeley.edu/~dwanders/>

<sup>76</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=CatContact](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=CatContact)

<sup>77</sup> <http://www.unicode.org/standard/versions/>

<sup>78</sup> A complete proposal may consist of this one document alone, or it may consist of this plus other documents.

<sup>79</sup> Templates for the Proposal Summary Form, the "Principles and Procedures" document, and the directory of previously-submitted proposals is available at <http://std.dkuug.dk/JTC1/SC2/WG2/docs/summaryform.html>

Conveying exactly what characters are proposed for addition can be as simple as creating a list that includes a character name and representative glyph for each. Additional information may also be needed, however, particularly if an entire script is being proposed. For instance, if characters have special writing system behaviors such as taking on special forms in particular contexts, such details should be included so that the committees can determine that the proposal is appropriate and will actually provide what is required. If it is not clear whether information of this sort is needed in your situation, you should seek the help of experts familiar with the Standard<sup>80</sup>.

When approving a proposal, the committees need to be confident that they have made a good decision: that the addition to the Standard will serve a useful purpose, and that it will meet the needs of the intended user community. A proposal, therefore, needs to provide justification—a demonstration that there is a real user community needing the additional characters, and that what is proposed will meet the needs of that community.

It is not necessary to demonstrate that the writing system in question represents a large community with substantial market potential. The size of the user community is generally not an issue, but there are limits. There must, at least, be a real potential for public usage of the characters or scripts. For instance, if two people invented a coding system, whether for their personal recreation or for some internal business purpose, and proposed that this be added to Unicode, their proposal would be rejected on the basis that they do not constitute an acceptable user community. On the other hand, if there was evidence available of usage by a significant number of independent third parties, that could provide a sufficient basis for encoding their characters. To use a real example, [a recent proposal](#)<sup>81</sup> that was accepted by UTC was for a pair of characters for which the known user community was 700–1500 people.

The most important aspect of justification is the ability to demonstrate that each of the proposed characters is in use or required by some community of users. A recommended way to demonstrate usage is to include images (e.g. photos or photocopies) showing the characters in usage in actual publications.

Justification can also involve demonstrating that the proposed characters reflect what is desired by the given user community. Suppose, for instance, a traditional script was formerly in use for a given language and is being revived, and that this script is being proposed for addition to Unicode. Suppose also, though, that there is some disagreement within the community regarding a particular letterform as to whether it constitutes a separate element or is merely a calligraphic variant of some other form. In this situation, the committees will very likely require that a consensus first be reached within the community regarding the status of this particular letterform. In most situations in which characters need to be added to Unicode, issues of this sort will not be a concern. If such do exist, though, it would be best to get these resolved before attempting to initiate the formal process to get characters added to the Standard.

## 6.4 Encoding conversion

Although Unicode offers many benefits to WSI developers and users worldwide, most systems still use other encoding schemes. So there is a great need for tools that can convert existing textual data to and from Unicode. Though important to WSIs, this complex topic is not covered in this document.

---

<sup>80</sup> See the suggestions for obtaining assistance mentioned in sections [6.3.1](#), [6.3.2](#) and [6.3.4](#).

<sup>81</sup> <http://scripts.sil.org/cms/sites/nrsi/media/YupikCyrillicProposal.pdf>



## Section 7

### Technical Details: Data Entry and Editing

---

*Authors: Martin Hosken, Victor Gaultney*

When people enter text into a computer, or edit text that already exists, they most commonly use the keyboard. The layout and function of that keyboard can make a WSI easily accepted—or rejected. This section briefly discusses the two types of interaction, and focuses primarily on the design of software keyboards.

#### Note

[Technical Details: Characters, Codepoints, Glyphs](#)<sup>82</sup> should be read before this section.

## 7.1 Types of data interaction

**Entry** of textual data is a process of translating physical key presses into individual chunks of data. This is relatively easy: you just allocate codepoints to keystrokes. The difficulty comes when trying to decide what codepoints to assign to what keystrokes. Do you design based around the characters on the keytops of a user's physical keyboard hardware or the relative position of the keys? What do you do if you want to be able to type more characters than there are keys in your keyboard?

**Editing** of data is even more complicated. On top of the issues regarding what happens when keys are pressed, there are issues of cursor position, text selection, directionality. For example, when a user clicks in the middle of a word and starts typing, what should happen? This is easy for text that is linear (one letter after another), but what if tone marks and vowel symbols are involved, some of which could be typed after a consonant, but appear before? The details of this are too complex for this brief paper, but need to be considered in addition to basic keystroke to codepoint translations.

This rest of this section will examine various design tradeoffs for software keyboards and look at some of the different approaches used in different situations and technologies. Note that, as yet, there is no technology which can support all the different approaches presented here. So in addition to making decisions based on the particular keyboard behavior required, you will also need to take into consideration the limitations of your keyboarding software.

First we consider issues of keyboard layout and then issues of large keyboards (those where there are more characters to be typed than keys to type them). Note that this section contains information relevant to those designing keyboards with very few characters as well. Then we look at sequence checking and the whole issue of how different types of keyboards are expressed as rule systems. At the end is a listing of tools that can be used to create software keyboards.

## 7.2 Keyboard layout



The primary consideration in keyboard design is the layout of the keys. Whether the physical keyboard is a variant of a standard European keyboard, or one localized for the script, there are ways to adjust the layout to accommodate new or rearranged letters. Much of what we need to consider can be brought out

---

<sup>82</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_5\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_5_1)

in the simple example of adding support for the character “y” to a Latin keyboard. There are two ways of doing this.




### 7.2.1 Mnemonic

The first way is to consider the “y” as two components: “y” + *umlaut*. In this case we would have a special keystroke to add the *umlaut* on top of the “y”, for example the keystroke  following a  might add the *umlaut*. In other words we are using existing information on the keys of a user's keyboard to help the user remember the keying of the character. We use the term **mnemonic keyboard** for this type of keyboard. Mnemonic keyboards are commonly used with Latin-based scripts, since there is a close correspondence between what people want to type and what they see printed on the keyboard in front of them.

### 7.2.2 Positional

The second approach to the design of a keyboard is to reconsider the position of the letters on the keyboard. Of course, this means that there may be differences between the letters that appear on the keytops and the results when typing. There are physical ways to assist users in using rearranged keyboards, but those are not discussed here.

The easiest way to implement “y” in this case is to consider “y” as a unit. We might want a single key to press to type this character. In addition, we would like it on the periphery of the keyboard, since it is a rarely typed letter, and goes just as well on the right, where there are punctuation characters we can use. So we might specify that it is associated with the second row right-most key on the keyboard, which on some keyboards represents the “]” key. This is a departure from the standard keyboard, but one that users can readily accept.

A more complex, but possibly better, method is to still give “y” a special key, but arrange it positionally close to the “y”. This would be helpful if “y” was a very common letter combination. For example, the UIOP keys could be shifted right (with the  replacing the  key), and the new letter placed in the place of .

This type of rearranged layout is a **positional keyboard**, where keys are defined positionally in relation to each other. Thus it does not matter what is printed on the keytops of the keys; what is important is what character is generated by each position in the grid of keys. This approach is most commonly used when implementing a keyboard based on a typewriter layout or some other standard.

This approach of defining keyboards in terms of the relative positions of keys is less common for the addition of a single character to an existing keyboard than it is for the implementation of a whole keyboard, particularly if that keyboard emulates an existing keyboard layout. For example, there is no mnemonic relationship between the keytops on a physical keyboard created for the UK and the *de facto* standard for typing Thai. The Thai keyboard is designed in terms of the old typewriter layout, which is a good layout. That is, the most commonly typed letters are positioned in easy-to-reach locations, and rarely-typed letters are more difficult to reach, without respect to the traditional QWERTY layout.

When designing a complete new layout for a keyboard purely in terms of relative positioning, it is useful to do some analysis of common letters and combinations. The keyboard designer can then allow typists to type quickly by placing commonly typed letters on the keys in the middle of the keyboard. This is a radical departure from QWERTY, which was originally designed to slow typists down so that the typewriter would be less likely to jam. The DVORAK layout for English keyboards was an attempt to provide a keyboard layout which allows typists to type English faster.

## 7.3 Extending the keyboard

In many cases, the requirement is to design a keyboard which supports more characters than can be accessed simply from a single press of the 101 or 102 keys on a normal physical keyboard. There are numerous approaches which are used to extend keyboards and we present most of them here.



### 7.3.1 Modifier keys

The most common approach is to extend a keyboard using modifier keys such as **Shift** (a traditional modifier key) or **Ctrl**, **Alt**, **Alt-Gr**<sup>83</sup>, **Command**, **Option**, etc., depending upon what type of keyboard you have. The Macintosh, for example, allows access to all the 8-bit codes directly through use of combinations of the option and shift keys.

The problem is that modifier keys are often used by applications for speed keys or for controlling the application, and requiring their use for typing can preclude their use by the application. Or worse, the application may take precedence over the modified key and not allow that combination to be used for typing a character. The Macintosh, at least, is reasonably tidy: combinations including the **Command** key are for “speed keys”, those without aren’t (except in non-text-oriented applications or situations such as games). So combinations including the **Shift** and **Option** modifiers are safe for generating characters. On Windows, **Alt** is typically used for accessing menu items, whereas **Alt-Gr** may be used for entering extra characters.

### 7.3.2 Dead keys

**Dead keys** are a popular approach to extending the keyboard for Latin keyboards. This approach allows the user to type a single character as a sequence of two or more keys on the keyboard. All but the last key do not result in anything being displayed, but change the state of the keyboard for subsequent keystrokes.

For example, on the International English keyboard, pressing the **’** key results in nothing being displayed. Following that by the **a** key results in “a” being output. If a key, such as **b** were to follow, then “b” would be output.

Dead keys work well where there is a very strong mnemonic relationship between the key being pressed and its function. Dead key sequences should be obvious and short, so the user is less likely to forget where they are. The problem with dead keys is that they easily confuse users since pressing a dead key results in no initial visual feedback.

### 7.3.3 Operator keys

A different approach to dead keys is to place the modifier after the key it modifies. Thus we might type **a** **’** to get “a”. On pressing the **a**, an “a” would be output. Then when the **’** is pressed, the “a” preceding the cursor is replaced by a “a”. This is a very powerful approach in that it allows the user to always have feedback regarding what they are typing.

The major difficulty with this approach is the implementation. You first need a system which can handle complex editing as well as typing. Thus, if I were to click in a document following an “a” and then press **’**, I would expect that “a” to change. But that might well not be possible, technically. Tools such as [Keyman](#)<sup>84</sup> work hard to emulate this behavior, but even then have limitations.

Secondly, all intermediate output characters need to be supported by the system. When implementing a keyboard for the International Phonetic Alphabet, which has hundreds of symbols, it may be nice to use combinations that begin with the **;** key to enter certain symbols. Using an operator keys approach, it would be ideal for the system to display an intermediate “;” as the first key is pressed. But the “;” might not exist in the IPA font, and so could not be displayed. Thus, you can only rely on intermediate output for codes that your keyboard needs to generate anyway.

---

<sup>83</sup> Some keyboards, particularly European keyboards, have an extra key called **Alt-Gr** (Alternative Graphics) which may be used to access a 3<sup>rd</sup> character on a key. The **Alt-Gr** key can be simulated on keyboards without it, by using **Ctrl-Alt**.

<sup>84</sup> <http://www.tavultesoft.com>

### 7.3.4 Input method editors: composition and candidate windows

One approach to the very large keyboard problem, for such languages as Chinese, is to use a type of keyboard input system known as an **input method editor** (IME). IMEs are characterized by special pop-up windows used to facilitate input. One type of window, known as a composition window, allows an intermediate string to be edited. For instance, a phonetic representation using Latin characters may appear temporarily, and then be replaced in the composition window by a Chinese character when a valid syllable is recognized. When the desired Chinese characters appear, a final keystroke causes this completed result to be transferred into the document.

A second type of window, known as a candidate window, can supplement a composition window. For example, as a phonetic representation using Latin characters is entered in the composition window, there can be several Chinese characters that could be intended. A candidate window can appear, showing the different candidate characters, allowing the user to select the desired Chinese character via the mouse, pressing the initial key or using the arrow keys. As the user types more keys, the possible selection list changes, homing in on appropriate character to input. For example, a Chinese keyboard results in the following windows being displayed:



In the screen shot above, the left hand window is the composition window, which contains temporary output from keystrokes that have been pressed. The right hand window contains candidate Chinese characters that are possibilities for what the user might be wanting to type. There are various other controls to allow the user to page through the list.

Composition and candidate windows can be used in combination, but either can be used alone. A candidate window provides a powerful mechanism for selecting characters from a large list, or even a shorter list, if a user is liable to have difficulty remembering the keying for a particular character. Its weakness is the amount of screen space it takes up, although with increasingly large screens, this is becoming less of a problem.

Unfortunately, there are few, if any, tools that allow such candidate systems to be customized or created.

## 7.4 Analytic keyboards

Modern keyboard technologies open up many new possibilities for complex keyboard behavior—beyond dead keys and operator keys. It is now possible to build analytical routines into systems that intercept the key presses, perform some analysis, and deliver the results to the application.

### 7.4.1 Sequence checking

Simple keyboards, where one keystroke equals one letter, and each letter follows directly after the previous one on the line, can usually assume that people will type data correctly. Other situations — more prevalent in scripts which use diacritics — is the issue of ensuring that people, for example, do not type the same diacritic twice. This enforcement of valid keying sequences is known as **sequence checking**.

This can be as simple as checking that people don't type the same diacritic key twice in succession, and possibly generating an error (such as a system beep) when they do.

Sequence checking can also be used to ensure that only valid base character and diacritic combinations are typed, and that they are typed in the correct order. Some software specifies a combining order for diacritics, and it helps if the keyboard can encourage data entry in this order. The order can be enforced, either by not allowing an earlier diacritic to be typed after a later one, or by allowing the keystroke and then re-ordering the data before it is stored in the computer.

Editing data that requires sequence checking presents a similar problem to that of operator keys. It is not easy for the keyboarding utility to manage the editing of a document, especially if a cursor is positioned randomly within a string of text. For this reason, applications cannot assume that the keyboarding utility will ensure that data is entered in any particular standard form or combining order. This can be troublesome.

While the keyboard does not interact with rendered text, but with the underlying stored text, the two are related in that they are both concerned with the visual representation of the text. Thus, the rendering subsystem may handle 'illegal' sequences by, for example, displaying illegal diacritics over a dotted circle, which can alleviate work from the keyboard. But there is no harm in programming defensively, with both the renderer able to display illegal sequences and the keyboard endeavoring not to allow users to type illegal sequences.

With dumb rendering systems, which are particularly unable to work with illegal sequences (for example, duplicates of the same diacritic), then more of the burden for ensuring valid data is placed upon the keyboarding subsystem.

## 7.4.2 Using keyboards in place of rendering systems

Keyboards can also be used to take the place of sophisticated rendering systems. For example, a writing system may rely upon a dot, placed over certain letters, to indicate some linguistic function. For most letters, the dot is placed at a normal height. For tall letters, however, it must be raised. This is most appropriately handled using smart rendering systems. If such a system is unavailable, then the same effect can be produced by encoding two variants of the dot—one normal height and one higher—as distinct characters, and the keyboard can determine which to use based upon the last letter typed.

There is a danger to this use, however. The resulting data stored on the computer now has two different codes in use for the dot, and the chosen encoding will need to include both. This makes further analysis on the text more difficult and error prone, and hence is not considered good practice in information systems. In particular, it is completely contrary to Unicode design principles. This is not needed as much with newer rendering systems, but was once the best option for high-quality publications.

## 7.4.3 Using keyboards to enter non-visual data

Just as keyboards can choose which variant of a diacritic to use in a specific context, it can also enter additional data into the encoded text. In Southeast Asia, many of the scripts can be written either with or without word breaks. Text without such breaks can be difficult to process, as the application needs to know when to break a line so as to keep words intact. Ideally, as part of the rendering process, an application would perform linguistic analysis on the text in order to recognize where valid line breaking possibilities exist. It may not be feasible to implement such capability in every situation, however. As an alternative, non-spacing word break characters could be used. But some typists find it hard to remember to enter in those non-visual breaks when they are not displayed on the screen.

Here some analysis could be built into the keyboard. It could watch the stream of key presses, analyze the syllable structure, and automatically enter in the breaks. This is not always foolproof, but can be a great help to the entry of text.

## 7.5 Tools for data entry and editing

- [Keyman](#)<sup>85</sup> — “is a program which enables you to use your existing keyboard to type in other languages by remapping the character keys and substituting your chosen script. Keyman particularly provides a solution for languages which require a more sophisticated keyboard layout than letter-for-letter substitutions.”
- [Some tools and resources for character input](#)<sup>86</sup>
- [BabelMap](#)<sup>87</sup> — “BabelMap is a Windows character map utility that allows the user to select and copy any Unicode 3.2 character.”

### 7.5.1 Keyman keyboards

- [Known Unicode Keyman Keyboards](#)<sup>88</sup>
- [Bible Translation & Literacy keyboard \(BTL in Kenya\)](#)<sup>89</sup>
- [SIL Keyboarding Chart for Africa](#)<sup>90</sup>

### 7.5.2 Macintosh Unicode keyboard layouts

- [Collection of Unicode keyboard layouts for Mac OS X](#)<sup>91</sup>.
- [IPAkeys](#)<sup>92</sup> — IPA Unicode keyboard layout for Mac OS X.
- [Unicode Keyboards for Mac OS](#)<sup>93</sup> — Web-based tool for creating new Unicode keyboard layout files from a list of keystrokes and desired output characters.
- [Installable Keyboard Layouts](#)<sup>94</sup> — Technical documentation on the installable keyboard layouts supported by Mac OS X 10.2 and later.

### 7.5.3 Reference info on OS-supplied layouts

- [Windows Keyboard Layouts](#)<sup>95</sup> (From Microsoft Global Development site)

### 7.5.4 Copyleft, FLOSS and open source tools

A number of copyleft, FLOSS and open source tools are listed in [Resources for Writing Systems Implementation using Copyleft and FLOSS \(Free/Libre and Open Source Software\)](#)<sup>96</sup>.

<sup>85</sup> <http://www.tavultesoft.com>

<sup>86</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=inputtoolinks](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=inputtoolinks)

<sup>87</sup> <http://uk.geocities.com/BabelStone1357/Software/BabelMap.html>

<sup>88</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=KeymanKeyboardLinks](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=KeymanKeyboardLinks)

<sup>89</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=BTLKeyboarding](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=BTLKeyboarding)

<sup>90</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=AfricanKeyboard1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=AfricanKeyboard1)

<sup>91</sup> <http://quinon.com/files/keylayouts/>

<sup>92</sup> [http://www.apple.com/downloads/macosx/productivity\\_tools/ipakeys.html](http://www.apple.com/downloads/macosx/productivity_tools/ipakeys.html)

<sup>93</sup> <http://wordherd.com/keyboards/>

<sup>94</sup> <http://developer.apple.com/technotes/tn2002/tn2056.html>

<sup>95</sup> <http://www.microsoft.com/globaldev/reference/keyboards.aspx>

<sup>96</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=FLOSS](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=FLOSS)

## Section 8

### Technical Details: Glyph Design

---

*Author: Victor Gaultney*

The most basic computing need for a given writing system is a font. Without one, a WSI cannot exist, and the quality and accuracy of the letters is commonly how WSIs are judged. This section addresses the need for quality fonts and how they can be obtained, developed and improved. Specifics about “smart” fonts and their behavior are covered in [Section 9](#)<sup>97</sup>. A general glossary of font terminology can be found on the [Microsoft Typography web site](#)<sup>98</sup>.

#### 8.1 Basic font requirements

The purpose of a font is to correctly display text in a language using the appropriate writing system. At a minimum, this requires the font to be complete and accurate for the language(s) covered by the font. Technically, this means that a font must:

- *Contain all necessary glyphs for the writing system.* For example, many African writing systems use the letter **ŋ** to signify the sound ‘ng’ as in the English word ‘sing’. It is not enough to have the letters ‘n’ and ‘g’ in the font and display them in place of the **ŋ**, the special symbol needs to be present in the font.
- *Display glyphs accurately.* Each glyph needs to be the right shape according to the requirements of the writing system. Although two languages may share the same basic system, each may have particular preferences over how certain glyphs are shaped. The font needs to accommodate these cultural preferences, which can include different preferences in spacing. For example, Thai is normally written without word breaks, whereas some minority languages that use the Thai script prefer that word spaces be used. There can also be differences in how much space is preferred between words and even letters. Finally, diacritics and other marks must be oriented correctly relative to the base glyphs.
- *Support a useful encoding.* The font needs to be encoded according to some agreed standard, whether that be a proprietary encoding for a particular application, or an international standard such as Unicode. If a glyph is not accessible to the application, then it is as if the glyph did not exist at all. The choice of which standard to support is closely related to the individual WSI, although Unicode should be the default choice in most situations.

#### 8.2 The need for quality

WSIs are typically created by programmers or linguists, not typeface designers. Even though the computational complexity required to make a WSI work may be impressive, most WSIs suffer from fonts that are poor in quality, and may result in documents that are difficult or unpleasant to read. This is because most of the typeface design is done by those untrained in the task.

---

<sup>97</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_9\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_9_1)

<sup>98</sup> <https://www.microsoft.com/typography/glossary/content.htm>

Typeface design is a subtle art, and the details of how lettershapes are shaped and interact are complex. So it is very valuable to consult experienced designers in the development of a WSI, and integrate their recommendations into the implementation.

It is a mistake to assume that language communities with little experience in technology will be satisfied with anything that only somewhat resembles their script. On the contrary, these readers are often more sophisticated and exacting than their counterparts who use the Latin script. For example, in Asia, there is a long history of writing, literacy and literature in many beautiful, indigenous scripts. Because of this strong manuscript tradition, the expectation is that fonts ought to closely follow traditional shapes. Yes, there is openness to modern interpretations, and a recognition of technical limitations, but satisfaction is usually linked with how faithfully the design echoes the tradition.

The history of Bengali typefaces illustrates this phenomenon. Bengali type has been available since 1778, and the printed history of the script is full of attempts to recreate the characteristic shapes with new technology. By the mid-20th century, the major font foundries, Monotype and Linotype, had systems that could typeset books and newspapers successfully. But both, and the Linotype in particular, had to make significant compromises in the rendering of the script due to technical limitations. Nevertheless, the general Bengali population seemed satisfied with the result. When new technology became available in the late 1970s, Linotype developed a new typeface that recreated the traditional style and forms that hearkened back to the manuscript tradition. Despite predictions that no one would accept it—they seemed satisfied with what they had—it became very successful and is now considered to be the definitive design for Bengali fonts. Nothing else is generally accepted for text material.<sup>99</sup>

WSI development needs to take into account the need for typefaces that accurately reflect the writing style and acknowledge the written history of the script. Because of the complex tools and subtle design sense required to create fonts, it is a specialized craft. Rarely do those who develop keyboards, encoding systems and other programmatic WSI components have the skills and training necessary, and so they must look elsewhere for fonts.

## 8.3 Sources for fonts

*So where can quality typefaces be found?* Traditional foundries as well as individual developers have designed good fonts for non-Latin writing systems. Generally, those companies with a long tradition of quality design offer the best fonts, but charge accordingly for them. Free fonts can be found all over the internet, but most lack in completeness and overall quality. There are exceptions to both of these, so it is worth investigating a variety of sources.

### 8.3.1 Commercial companies

Most commercial font vendors are not multi-faceted corporations, but are smaller companies focused solely on font development. As such, they rely on sales of their typefaces as their primary income source. Hence, their products can be expensive, and incorporation of their fonts in other WSIs requires payment of significant licensing fees. Their fonts are usually of good quality, though, and have high acceptance within language communities. Some of these companies are:

- [Monotype](http://www.monotype.com/)<sup>100</sup>, [Linotype](http://www.linotype.com/)<sup>101</sup>, [Adobe](http://www.adobe.com/)<sup>102</sup>. These traditional font foundries have a long history of design excellence. Monotype, in particular, has a library of non-Latin fonts that is second to none in design quality, and often serves as the standard by which others are judged. Adobe is the youngest of this group, and has relatively few non-Latin fonts, but is a quality foundry. All three are open to licensing their fonts to others for inclusion in WSIs.
- [Apple](http://developer.apple.com/fonts/)<sup>103</sup>, [Microsoft](http://www.microsoft.com/)<sup>104</sup>. Though not recognized as font foundries in their own right, they both do significant font development. Most of their designs are licensed from Linotype or Monotype, but

<sup>99</sup> See [ROS1999]

<sup>100</sup> <http://www.monotype.com/>

<sup>101</sup> <http://www.linotype.com/>

<sup>102</sup> <http://www.adobe.com/>

<sup>103</sup> <http://developer.apple.com/fonts/>

they do use local font providers, and work with the foundries to add extra features, such as additional glyphs or smart font programming to make the fonts work in their operating systems. Although these companies will not license their fonts to others, many of the fonts are shipped as standard components of their software. This means that WSI developers may be able to create solutions that assume that certain fonts will be present. Although not ideal, this is one option for developers<sup>105</sup>.

- [Linguist's Software](#)<sup>106</sup>, [Xenotype](#)<sup>107</sup>. These companies, and others like them, specialize in fonts and WSIs for less-common languages. They tend to fill in the gaps around traditional foundries, and provide an alternative to other commercial fonts. Linguist's Software has an impressive list of available fonts, and their products tend to be very complete and tailored to academic users. Like Xenotype, though, the quality of their designs vary widely. They may, however, be the only commercially available font for a particular script.

### 8.3.2 Public and not-for-profit

An alternative to commercial fonts are those developed by public and non-for-profit agencies. The goal of these groups is to enable technological advancement and literature production in both majority and minority writing systems. Their fonts are generally free for individual use, but may require modest licensing fees for inclusion in third-party WSIs. Font quality among these groups varies widely, but is often good. Because of their close association with language communities, these agencies tend to develop products that accurately reflect linguistic needs and preferences, and meet practical publishing needs.

There are only a few instances where governments have specifically developed and distributed fonts, but government-sponsored agencies can have an active role. [The National Electronics and Computer Technology Center \(NECTEC\)](#)<sup>108</sup> in Thailand has led many of the computing efforts in that country, and has developed fonts that match the preferred style for text publishing.

More work is done by NGOs. These organizations range in size from small to large, and those differences are reflected in how widely their fonts are known. Many NGO fonts are developed and deployed locally, usually by untrained designers, and are not known outside of the language communities. Larger groups, such as SIL International<sup>109</sup>, have a broader scope, have greater resources (including trained designers), and are able to take on major font development projects.

### 8.3.3 Freeware/shareware

Although most public and not-for-profit fonts are inexpensive or free, there is also a large and growing body of freeware and shareware fonts developed by individuals and distributed over the internet. Most of these are incomplete, inaccurate, of poor design quality, or do not function correctly, and are poor choices for inclusion in WSIs.

A small subset of these fonts, however, may be suitable. Internet searches can be successful in finding freeware fonts, and may even bring up reviews or comments on specific fonts. There are also many guides to non-Western fonts; three of these are:

- [SIL International Fonts in Cyberspace](#)<sup>110</sup>
- [Yamada Language Center](#)<sup>111</sup>

<sup>104</sup> <http://www.microsoft.com/typography/>

<sup>105</sup> It should be noted, though, that these companies do not allow their fonts to be modified in any way, or distributed outside the operating system.

<sup>106</sup> <http://www.linguistsoftware.com/>

<sup>107</sup> <http://www.xenotypetech.com/>

<sup>108</sup> <http://www.nectec.or.th/home/>

<sup>109</sup> <http://www.sil.org/> and <http://scripts.sil.org/>

<sup>110</sup> <http://www.sil.org/computing/fonts/>

<sup>111</sup> <http://babel.uoregon.edu/yamada/guides.html>

- [Alan Wood's Unicode Resources](#)<sup>112</sup>

### 8.3.4 Intellectual property and copyright issues

One of the challenges of incorporating third-party fonts in WSIs relates to intellectual property and copyright issues. It is not always easy to determine the true source of a font design. Many fonts on the internet are simply illegal versions of commercial fonts. So it is important to look at the copyright notice within the fonts, and even ask the developer where the design originated.

Fonts often need modification to work with a specific WSI or to meet the needs of a specific language community. It is important to know the licensing restrictions of any fonts that are modified. Many commercial and some public groups prohibit any modification, however minor. Others give their fonts very broad, open-source-style licensing.

The related paper "Intellectual Property Concerns in the Development of Complex Script and Language Resources" gives much more detail on these issues and discusses the unique nature of fonts as both works of art and pieces of computer software.

## 8.4 Process of font development

Despite the wide variety of fonts available from commercial and non-commercial sources, there remain situations where a new font design is needed. No font may exist, or those that are available may be inappropriate because of style or quality. There also may be licensing restrictions that hinder the use or modification of a particular font.

If significant funding is available, fonts can be commissioned from professional designers. Traditional foundries, such as Monotype, have a long history of this, but most recent commissions go to individuals or small foundries such as [Tiro Typeworks](#)<sup>113</sup>, who are known for a specific area of expertise. Internet mailing lists, such as [TYPO-L](#)<sup>114</sup> and [Type-Design](#)<sup>115</sup> can also be a means to find designers interested in work, and professional associations such as [ATypI](#)<sup>116</sup> can help.

The font development process, whether done by a foundry or by an individual, is similar to the WSI development process outlined in [Section 2](#)<sup>117</sup>.

### 8.4.1 Initial research and planning

Careful planning is important to a font project's success. Because the designer is often someone without an intimate knowledge of the writing system, they need reliable information from others about letter shapes, style and behavior. The basic steps are to:

*Determine detailed needs.* A font project needs a clear specification of all the glyphs needed and how they are to be encoded. This description will likely change as more information is discovered, but some initial listing is useful. The designer will also need to know how the glyphs are to interact with one another: how diacritics relate to base characters, common and uncommon letter combinations, etc. An orthography description, if available, can be very helpful.

*Identify key script experts.* The most important source for script information is the language community itself and the script experts within. These should be people whom the community recognizes as authoritative with regard to the written language—scribes, teachers, leaders, artists. Ongoing interaction

---

<sup>112</sup> <http://www.alanwood.net/unicode/fonts.html>

<sup>113</sup> <http://www.tiro.com/>

<sup>114</sup> <http://gmunch.home.pipeline.com/typo-L/>

<sup>115</sup> <http://type-design.p90.net/>

<sup>116</sup> <http://www.atypi.org/>

<sup>117</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_2](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_2)



with these experts can ensure that the font accurately reflects the writing system and meets the needs of the community.

*Gather other sources and references.* It is not enough to simply talk with script experts. Samples of the writing system are needed. There should be a variety of these—handwritten samples, school primers, existing books—and the script experts should give their opinion on which are the most ‘correct’. The internet can also be a source of information, but is often misleading or wrong. Published books and academic research are often more reliable and can provide useful historical background. Finally, electronic versions of authentic text material (folktales, etc.), if they exist, can be useful in testing.

*Make style decisions.* Most language communities would love to have more than a single style of font for their use. This raises priority questions and requires some additional investigation. *What will be the primary use of the font—long document publishing, literacy primers, signage, on-screen reading (such as on the internet)?* The use should be reflected in the style of the font. Tradition is also important. *Is an ‘italic’ version needed for a script with no history of slanted forms?* The answer is sometimes yes and sometimes no. For example, the Ethiopic script traditionally has a single style of letterform, a bold letter with calligraphic elements. *Is it appropriate to create an italic version? And is an even bolder form even conceivable?* There is no tradition of this, but the availability of the Bold and Italic buttons in word processing applications encourages users to desire these modern variants.



## 8.4.2 Glyph design procedures

Once planning is complete, and key style decisions are made, the design of glyphs can begin. Designers vary in their preferred design process. Some draw their glyphs directly on screen, while others prefer to draw shapes on paper and then bring them into the computer. In order to give a complete picture of what could be involved in the design process, the latter situation is described here. This is, again, only an example of the process and is not a definitive method.

*Drawing.* After study of the writing system, and the intended use for the font, the designer draws tentative lettershapes on paper, with a height of typically 2-5 cm. The lettershapes may be drawn as complete words, as an alphabet sample, or as seemingly random combinations of letters. The main goal is not to create final, perfect forms, but to experiment with stroke weights, terminal designs, initial spacing and general proportions.

*Scanning.* Once these initial shapes have been drawn, they are scanned into the computer, preferably at a resolution of approximately 300-400 pixels in height. Any scanning software would work for this, as long as the scanned image can be cropped and imported into the font design package. Detailed instructions on how prepare, scan and import images into the Macromedia Fontographer font design program can be found in [Scanning Tips](#)<sup>118</sup>.

*Initial digital outlines.* Digital fonts are typically described not in terms of pixels, but as mathematical curves that define the outline of the glyph. So the scanned images are used only as a guideline for the preparation of the outlines. Once the scanned images are imported into the font design application, the designer can begin to draw initial outlines of a few key glyphs, based generally on the drawn and scanned shapes. These are tentative, and will likely be changed throughout the next step.

*Refining parameters.* After the basic shapes have been drawn, the designer can create draft fonts in order to test glyph size, weight, etc., and then revise the draft shapes repeatedly until they look generally correct. At the end of this process, there should be specific parameters determined for the font—letter height, stroke widths, length of serifs, amount of contrast, and other numeric guidelines.

*Completing the glyph set.* The next step is to design the remaining glyphs, based upon the style and parameters of the initial key glyphs.

*Adjusting spacing.* Good spacing can improve a mediocre typeface, but bad spacing can ruin a beautiful one. Great care must be taken to ensure that letters are spaced properly. This includes kerning for pairs

<sup>118</sup> <http://www.sil.org/~gaultney/Hints/scan.html>

of letters that, because of their design, would run into one another or leave large gaps of whitespace between them. The goal is to create a flowing, uninterrupted line of text with even texture. This is important for readability, especially for beginning readers.

*Hinting.* Legibility of letters on computer screens, or on low-resolution printing devices, is heavily dependent on 'hints' contained in the font<sup>119</sup>. These are special computer instructions that specify how the mathematical outlines are turned into pixels. Hinting is a highly technical task, and most independent designers avoid it due to the time and expertise it requires. Modern font development applications include auto-hinting routines that can improve a font's appearance on screen, but the results are not as good as manual hinting. For more discussion on hinting strategies, see [Notes on Hinting with FontLab](#)<sup>120</sup>.

*Packaging and distribution.* This involves packaging the font and any related documentation and instructions for the user or for those who are building the font into a larger WSI or software application.

### 8.4.3 Testing

Although testing is usually done throughout the whole font design process, it also needs to be done once the font is generally completed, with all glyphs designed and spaced. There are four kinds of testing:

- *technical*—to see if it works with other WSI components and in the appropriate software environments
- *internal designer*—the kind done by the designer during the design process
- *external designer*—review by another designer or colleague to inspect the glyphs for design problems
- *script expert*—review done by experts in the writing system in order to ensure accuracy and general satisfaction throughout the language community

## 8.5 Conclusion

Fonts are critical WSI components. Font design, however, is a complicated task, and requires artistic as well as analytical skills. Most WSI developers do not have the expertise to design their own fonts, and need to look to other sources. There are many companies and individuals who will license fonts and even prepare new designs on commission. Despite this, it is sometimes necessary for a WSI developer to take on the task.

## 8.6 Sources for information

The following are sources for information on fonts and typeface design.

### 8.6.1 Tools

Many of these tools come from FontLab Ltd., a company that has become the main font tool provider for the font design industry.

[DTL Fontmaster](#)<sup>121</sup>—specialized, but expensive tools for design and font production

[Font Creator](#)<sup>122</sup>—a little known, and limited, TrueType font editor

---

<sup>119</sup> A general description of hinting can be found at <http://www.microsoft.com/typography/hinting/hinting.htm>

<sup>120</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=NotesOnHintingWithFL](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=NotesOnHintingWithFL)

<sup>121</sup> <http://www.fontmaster.nl/>

<sup>122</sup> <http://www.high-logic.com/>

[FontLab](#)<sup>123</sup>—the font design tool that has become the industry standard

[Fontographer](#)<sup>124</sup>—this was the main design tool, but has not been updated in many years and is rapidly becoming obsolete

[Scanfont](#)<sup>125</sup>—a tool for scanning drawn glyphs and importing them into FontLab

[TransType](#)<sup>126</sup>—a font converter

[TypeTool](#)<sup>127</sup>—a lightweight, and cheaper, version of FontLab

Microsoft and Apple also provide font development tools on their Web sites. See [section 8.6.2](#).

A large number of copyleft, FLOSS and open source tools are listed in [Resources for Writing Systems Implementation using Copyleft and FLOSS \(Free/Libre and Open Source Software\)](#)<sup>128</sup>.

## 8.6.2 Web sites

[Adobe Type Topics](#)<sup>129</sup>—a variety of information on Adobe fonts and general type issues

[Apple Fonts/Tools](#)<sup>130</sup>—technical information and tools from Apple Computer

[Microsoft Typography](#)<sup>131</sup>—excellent site about the technical aspects of type design and Microsoft's tools

[TrueType Typography](#)<sup>132</sup>—general information about the TrueType font format

## 8.6.3 Publications

Brigham, Robert, *The elements of typographic style*. (2nd ed), Point Roberts: WA, Hartley & Marks, 1996. This is a general reference to use of fonts and type, but can be useful to anyone studying design.

Moye, Stephen, *Fontographer: type by design*. MIS Press, 1995. Though out of print, this book is a useful guide to type design processes. It is specifically for Fontographer, but can be applicable to other tools.

Tracy, Walter, *Letters of credit: a view of type design*. London: Gordon Fraser, 1986. This is the best book on the foundational concepts behind type design, and gives particularly good advice on spacing Latin fonts.

The [TypeBooks](#)<sup>133</sup> web site is a good source for books and reviews.

## 8.7 References

[ROS1999] Fiona Ross, [The Printed Bengali Character and its Evolution](#). London: Curzon Press, 1999.

---

<sup>123</sup> <http://www.fontlab.com>

<sup>124</sup> <http://www.macromedia.com/software/fontographer/>

<sup>125</sup> <http://www.fontlab.com/html/scanfont.html>

<sup>126</sup> <http://www.fontlab.com/html/transtype.html>

<sup>127</sup> <http://www.fontlab.com/html/typetool.html>

<sup>128</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=FLOSS](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=FLOSS)

<sup>129</sup> <http://www.adobe.com/type/topics/main.html>

<sup>130</sup> <http://developer.apple.com/fonts/>

<sup>131</sup> <http://www.microsoft.com/typography/>

<sup>132</sup> <http://www.truefont.demon.co.uk/>

<sup>133</sup> <http://www.typebooks.org/>



## Section 9

### Technical Details: Smart Rendering

*Authors: Martin Hosken, Victor Gaultney*

For most of the history of computing the relationship between encoding and rendering has been relatively simple. Each codepoint has a glyph associated with it, and using whichever rendering technology is available, that glyph will be displayed on the output media. But with the advent of Unicode, and the need to be able to use an encoding which is linguistically motivated rather than visually motivated, the rendering process has become more complicated. In this section we will examine the rendering process from the viewpoint of modern rendering requirements.

#### Note

[Technical Details: Characters, Codepoints, Glyphs](#)<sup>134</sup> should be read before this section.

### 9.1 The Rendering Process

The basic process is that an encoded string of characters, stored in the computer, is converted to a glyph string. This glyph string is then processed in some way to arrive at a sequence of positioned glyphs which are then rendered to the output media.

Rendering is an ambiguous word, and we will use the term **glyph rendering** to describe the process of making the electronic equivalent of marks on a page, and **rendering** to describe the whole process as shown below.

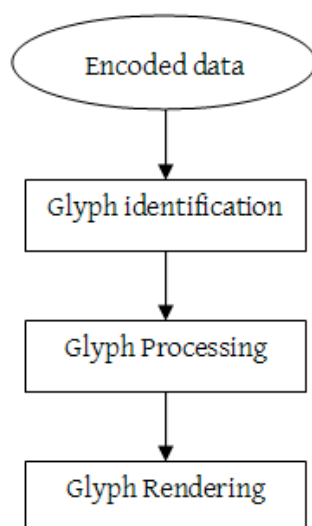


Figure1: Rendering process overview

<sup>134</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_5\\_1](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_5_1)

This section will concentrate entirely on TrueType based fonts. But the concepts presented here can be easily applied to other technologies.

### 9.1.1 Glyph identification

The first step in the rendering process is to convert an encoded string to a string of glyph references. This conversion is a simple one-to-one correspondence. For every codepoint there is one corresponding glyph. Thus the number of glyphs in the glyph string will be the same as the number of codepoints in the encoded string. Each font contains a table (called the **cmap**) that specifies the exact correspondence.

### 9.1.2 Glyph processing

In the glyph processing phase the string of glyph references (also called glyph IDs) is converted to another string of glyph IDs. In the case of simple rendering (with 'dumb' fonts) this transformation is simple—there is no conversion. In more complex systems, sophisticated routines can transform this string. This can be useful for diacritic positioning, forming ligatures, and other linguistic transformations. It is also where glyph positioning is begun. This step contains the most flexibility for WSI developers, and will be explained in more detail after mentioning glyph rendering.

### 9.1.3 Glyph rendering

At the end of the process is the actual action of putting marks on paper or displaying shapes on the screen. This involves taking the glyph outlines for each glyph and displaying them at the position indicated for that glyph. Therefore the renderer requires not only the glyph ID, but also the position of the glyph on the screen. The renderer also needs to know such things as how large to render the glyph, using what color and other stylistic information.

### 9.1.4 User interaction

Although not technically part of the rendering process, user interaction must be considered. In an editing environment, after the glyphs have been rendered, users need to be able to interact with the text as represented by the visual glyphs. This includes actions such as text selection and cursor movement.

## 9.2 Glyph processing—Dumb Fonts

Glyph processing is where the WSI developer can intervene. The traditional approach taken to glyph processing has been very simple:

- For each glyph in the string, set its vertical position to be the same as the previous glyph, and set the horizontal position to be directly after the previous glyph in the string.

Notice that this does not imply any particular directionality. Thus if we are rendering left to right, the new glyph will be placed to the right of the previous one, while if we were rendering a right to left script, then it would be to the left. For dumb fonts, this directionality is fixed for each string.

Dumb fonts, therefore, can be said to have the following characteristics:

- One-to-one codepoint to glyph association
- Simple sequential positioning

As a result of these limitations, WSI developers have devised new encodings which work within the limitations and allow for the rendering of more complex scripts. For example, there may be different codepoints associated with different positions of the same glyph so that diacritics are positioned approximately correctly. Likewise, variant forms of glyphs will each have their own codepoint. The effect of creating such encodings has been a proliferation of encodings, often associated with a particular font or family of fonts. Thus the interpretation of data requires constant reference to its special encoding.

Another problem with such a scheme is that there can be no more glyphs than there are codepoints in the encoding. Most of these systems used older, 8-bit encodings which have a maximum of 256 codepoints, and were limited to 256 glyphs. This is particularly problematic for writing systems which require more than 256 glyphs. Ethiopic requires 600-1200 separate symbols. Devanagari has numerous conjunct forms that cannot fit within a 256 glyph limit. Both of these writing systems can be accurately encoded using less than 256 codes, but cannot be displayed correctly without the additional ligatures and separate forms needed for letter combinations.

You might think that 16-bit Unicode encoding would solve this problem. The reverse is true. Unicode does not follow the principle of one codepoint per glyph, but instead expects the rendering system to be able to do the necessary processing to render the text. This requires smart rendering systems.

Finally, changes to computer operating systems over the last few years have sometimes disabled these dumb font WSIs. Such WSIs heavily use tricks to work around operating system limitations, and when the OS changes, the open doors used by developers have been shut. The result is that a WSI may, at first, seem to work in a new version of an OS or an application, but have disabling problems, such as a range of letters that no longer display correctly, etc.

### 9.3 Glyph Processing—Smart Fonts

Although many good WSIs have used the dumb fonts approach, it has limited usefulness in modern systems. The best opportunities for solid WSI development rest in the use of smart font technologies.

Smart fonts are those with a more complex glyph processing model. From this simple definition a whole plethora of approaches is possible. The skill in creating a glyph processing model is to focus on converting the original string of glyphs into a new string of glyphs, where each glyph is appropriately positioned. There are a number of basic principles common to all the different technologies, though they differ a little in each particular implementation. The three main TrueType-based smart font rendering technologies are

- **OpenType**—Microsoft's smart font technology. This is a combination of a new font format (based on TrueType) and operating system routines that can be used to transform glyph strings. This layer is rather complex and difficult for applications to support directly, and so Microsoft adds another layer on top to do much of the higher level processing. This other layer is called **Uniscribe**. An application need not use Uniscribe, and can interact with OpenType directly, but will need to do the processing that is done by Uniscribe, itself.
- **AAT**—Apple's smart font technology (Apple Advanced Typography). This is also a new, TrueType-based font format with rendering routines. The technology is based almost exclusively on the older GX technology, with identical concepts. The only real difference is that AAT is Unicode-based and has a different application programming interface than GX. But from the glyph processing perspective the two technologies are nearly identical. The currently used application interface is ATSUI (Apple Type Services for Unicode Imaging), which is a bit like Uniscribe in that it looks after such high level actions as line breaking. But since the AAT tables provide greater control over glyph processing, the line between what is handled in code and by the font gives more control to the font.
- **Graphite**—SIL's cross-platform (Windows/Linux), open source offering. This was developed due to limitations in OpenType and patent issues with AAT. Coming last, it is hoped that it brings together the strengths of the other two technologies with few of their weaknesses.

This discussion will follow the Graphite processing model most closely while interacting briefly with the other two technologies.

In smart glyph processing, the glyph stream goes through two key phases with the possibility of a third. The first phase is called **substitution** and is concerned with ensuring that the glyph string consists of the right glyphs in the right order in the string. Thus it involves such concepts as glyph replacement and re-ordering. The second phase—**positioning**—is concerned with ensuring that the glyphs are correctly positioned. It involves such concepts as kerning and shifting. The last phase—**justification**—may or may not be used. It involves working with the application to increase or decrease the space taken up by the string when it is rendered. It involves such concepts as kashidas and maximum allowable space at various points through the string.

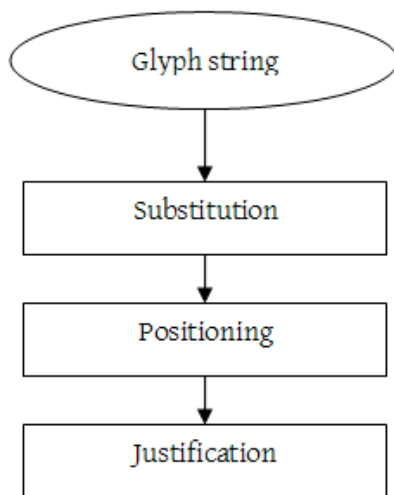


Figure 2: Smart glyph processing overview

### 9.3.1 Substitution

The substitution phase involves transforming the input glyph string to ensure that it contains the right glyphs in the right order. Thus all the approaches of normal string transformation are needed. The following capabilities are particularly required:

- *Ligature replacement* involves, in its most general form, taking a sequence of glyphs and replacing them with another sequence of different length. This requirement crops up in nearly every script.
- *Contextual substitution* means that it must be possible to constrain any substitution action to be done only in a particular context. For example, replacing a dotted “i” with a dotless one should only occur if there is an upper diacritic following the “i”.
- *Re-ordering* involves moving glyphs relative to each other in the glyph string. Note that this is not about positioning, as such, but is more to do with which order characters are displayed as opposed to how they are stored. The classic example of this is that in Devanagari, the *ikar* vowel is stored after its preceding consonant, but is rendered before it.
- *Glyph insertion* is sometimes needed, particularly contextually. In Burmese, it is needed in order to insert glyphs before the initial consonant, for example. Glyph insertion is more problematic than glyph deletion since it may involve the glyph string changing length, which involves memory allocation issues and the possibility of pointers moving, etc.
- *Glyph deletion* is simply the ability to remove a glyph from the string. This is useful for glyphs which are mapped from functional codes, like the zero-width joiner or zero-width non-joiner which may be used as part of the context for some other substitution, but ultimately do not display anything and may get in the way of positioning.

### 9.3.2 Bidirectional Ordering

The whole issue of rendering right to left scripts is fraught with difficulty. Not only do we have to consider which direction a string is to be rendered in, but also the wider directional context in which the string is to be rendered. Thus rendering a left to right string in a right to left paragraph is a very different proposition from rendering it in a left to right context. In addition, in Arabic, for example, numbers are written left to right. Thus, not only might the string need to be rendered in a different order, but parts of the string may need to be rendered in a different direction to other parts.

The real complexities in the issue appear when it comes to line breaking. Consider some Hebrew text embedded in an English paragraph and a line break occurring in the middle of the Hebrew. The Hebrew



character nearest the last English character before the Hebrew will be very different if there is a line break as compared to if there is no line break.

Genesis begins : בְּרֵאשִׁית בָּרָא אֱלֹהִים אֶת הַשָּׁמַיִם וְאֶת הָאָרֶץ meaning “In the...”

Left to right paragraph order. No line break

Genesis begins בְּרֵאשִׁית בָּרָא אֱלֹהִים אֶת  
: הַשָּׁמַיִם וְאֶת הָאָרֶץ meaning “In the...”

Left to right paragraph order. Line break in Hebrew text

Genesis begins בְּרֵאשִׁית בָּרָא אֱלֹהִים אֶת הַשָּׁמַיִם וְאֶת הָאָרֶץ meaning “In the...”

Right to left paragraph order. No line break

Genesis begins בְּרֵאשִׁית בָּרָא אֱלֹהִים אֶת  
הַשָּׁמַיִם וְאֶת הָאָרֶץ meaning “In the...”

Right to left paragraph order. Line break in Hebrew text

Figure 3: Bidirectional line-breaking examples

To help in understanding which order we are talking about at any time, we talk of *logical* and *surface* order. Logical order is the same as the order in which the encoded string is stored, that is, in reading order. Thus the first letter of a sentence precedes the second regardless of where they are positioned relative to each other.

Surface order is the order in which glyphs are rendered and is system dependent. For a system that always renders left to right, then surface order may be the opposite order to logical order. Most systems use the overall writing system direction for the surface order. This, for the most part, means that glyphs are rendered in logical order. But there may be glyphs which are rendered in the opposite order to logical order, for example, Arabic numbers.

AAT and OpenType process everything in surface order and address the line breaking issue outside the rendering issue. Graphite processes the substitution in logical order and then processes the positioning in surface order, which for Graphite, is the underlying writing system order. Thus in Arabic, the numbers would be processed for substitution in reading order and then positioned in reverse reading order.

### 9.3.3 Positioning

The input to the positioning phase is a glyph string in the order in which the glyphs are to be rendered. The default activity is to render the glyphs as though this were a dumb renderer with the glyphs lined up in boxes each with the width of the advance width. The task of the positioning phase is to move various glyphs from their default position and perhaps to close up any gaps left behind.

#### 9.3.3.1 Kerning

More sophisticated applications, even with dumb rendering systems, allowed for kerning. Kerning adjusts the relative position of two glyphs and also adjusts the positions of all following glyphs. Thus the classic example of WAVE.

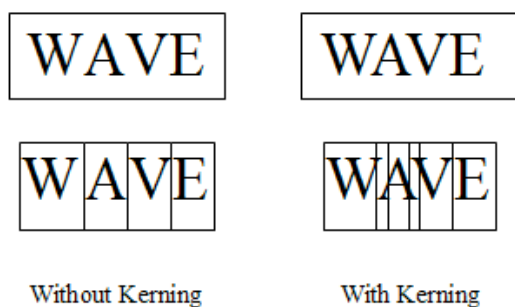


Figure 4: Kerning

Kerning is a key concept in positioning, but it may be extended to cross-stream kerning whereby there is movement in the vertical direction as well as the horizontal. Using this concept and the ability to reset the vertical direction to the baseline again, it is possible to do some fairly sophisticated positioning. AAT provides very sophisticated contextual kerning as the basis of its positioning.

### 9.3.3.2 Shifting

In addition to kerning, there is the concept of shifting, which is like kerning, but does not involve any other glyphs moving. Thus if we were to shift the “A” in “WAVE” rather than kern it (and leave the “V” unshifted and unshifted) we would see the following:

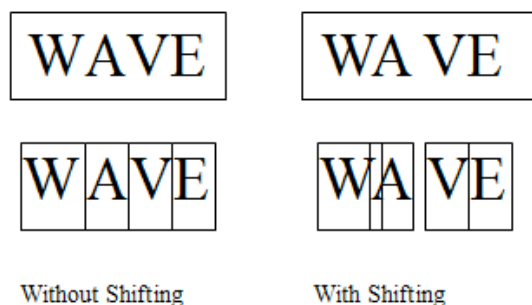


Figure 5: Shifting

This capability is particularly useful for positioning zero-width diacritics.

### 9.3.3.3 Attachment Points

The most powerful way of positioning things like diacritics is the concept of attachment points. Consider the problem of trying to attach an acute accent over an “A”. There are a number of ways of achieving this.

- We could substitute an “Á” glyph, where the two pieces are already positioned. This is fine for this simple example, but in scripts which use diacritics for vowels, you would end up with so many glyphs that the problem would become unmanageable.
- We could use shifting on a zero-width diacritic for the acute accent. This would work, but we would have to know precisely how much to shift the accent by.
- Alternatively, we could insert a special point in the “A” glyph and a special point in the accent glyph and then position the accent such that its special point coincides with the special point in the “A” glyph. Then the two would be lined up exactly.

The latter approach is very powerful and can be used even when there are no attachment points actually designed into the glyph. All the positioning phase needs to know is where the attachment points ought to be, even if they do not actually exist in the font file. The advantage of actually drawing the attachment points into the glyph is that then they can be hinted and the diacritic attachment will then be correctly positioned according to hinting<sup>135</sup>.

The concept of attachment points is so powerful that it forms a core concept in Graphite where it is used to help address the issue of allowable cursor points and the ability to measure existing positioning to be used in positioning other glyphs. OpenType allows positioning by attachment. AAT does not have this capability.

The positioning phase needs to address such issues as:

- Kerning, both simple and contextual (apply this kerning in this context)
- Diacritic attachment
- Cursive attachment where one glyph attaches to the next glyph in a chain
- Handling collisions where diacritics overstrike each other
- Repositioning base characters when diacritics are wider than the base character
- Double diacritics and ensuring the appropriate position given two base characters to worry about.

Having positioned the glyphs, the primary role in glyph processing is completed, and the data can be handed over to the renderer for final rendering.

But there are other issues that need to be addressed. What happens if the text over-fills the space allocated for it? How should line breaking be handled, particularly in scripts with no inter-word spaces? What about justification in scripts that use kashidas, or do not have much whitespace to widen? What about features?

### 9.3.4 Justification

Justification is done in much closer conjunction with the application than simple rendering. The first two phases can be achieved by passing an appropriate string to the smart renderer and leaving it to do all the work. Justification requires the application to make some decisions about where extra width is to be generated or where width must be reduced.

In order to achieve this, the smart font needs to both provide information to the application about good places to vary width and by how much, and then to make the necessary changes when the application specifies width changes at certain places. Width changes may be achieved by a number of approaches

- Inserting space between letters or words
- Inserting joining marks (kashidas) between characters in a cursive script
- Expanding ligatures to separate letters or contracting from sequences to ligatures

Each of these techniques provide ways of changing the width which have greater or lesser impact on the reader. Clearly, it is best to make adjustments which minimize the impact on the reader where possible. But which ones are those? For this reason, smart fonts will return some kind of weighting as to how welcome width adjustment is at each location in the glyph string.

---

<sup>135</sup> The Microsoft TrueType rendering engine does not make the locations of points following hinting available to calling applications, therefore hinted attachment points are not possible on the Windows platform. OpenType gets around this by allowing for resolution specific modifications to attachment point positions.

### 9.3.5 Line Breaking

Line breaking can be very simple (after any space) or very complicated (using neural networks). The normal approaches used by applications for handling line breaks are very limited. For example, breaking a line at a particular position may cause the line before to grow and so the new line may still be too long. Not only this, but creating a line break at a position may change the rendering of the lines before and after.

This brings out an important change for applications between dumb rendering and smart rendering. It is not enough to pass text to the rendering system character by character because the rendering system may need to change how a character is rendered dependent upon the characters around it. Therefore it is necessary for the rendering system to pass appropriate lengths of text to the renderer (whatever they might be!) This also means that an application can't just add characters to a line until it fills up and then move on to the next line. Instead it is almost easier to have a paragraph layout subsystem that interacts with the renderer in a generic way. This then leaves the question of where line breaks might occur to the writing system and so, ultimately, to the font.

So, one of the early phases of processing is to decide on valid line-break positions. Uniscribe does this in the Microsoft context, ATSUI does line breaking in the AAT context. Both of these use knowledge of Unicode character attributes to decide on line break points. Graphite, on the other hand, does line breaking using glyph attributes and allows them to be changed contextually. This has the disadvantage that much of the Unicode character attributes need to be reflected into the font as glyph attributes, but does allow for more sophisticated language based contextual control.

### 9.3.6 Features

The features mechanism allows a particular run of text to pass information to the glyph processor about how the string should be rendered. For example, a feature may be used to indicate whether certain ligatures should be made or not. Each feature has a name and also a number. The name allows users to interact with the feature and the number allows software to interact with the feature. Features can take various values, each with their own name. A user may set a particular feature to a particular value for a run of text, and the glyph processor can use the knowledge of the value of a feature to change how it processes the glyphs.

OpenType, on the other hand, uses the feature concept in a very different way. Firstly, features are a binary concept, a feature is either set or not set. Secondly, they are used as a way for the higher level processor (Uniscribe) to pass extra information along with a glyph ID to the font for processing. For example, Uniscribe converts Arabic characters to the nominal form and then marks whether a glyph is word initial, medial, final or isolate using a feature for each. Thus the string passed to the font for processing has different features set for different glyphs.

## 9.4 User Interaction

In an editing environment, once the glyphs are on the screen, a user may well want to interact with what they see. This section looks at some of the rendering issues involved in editing.

### 9.4.1 Hit Testing

When the user clicks the mouse somewhere in a section of text, the editing process needs to know where to place the cursor in the underlying text. For simple linear rendering, this is not difficult. But once re-ordering, bi-directionality, split glyphs and even diacritics are involved, hit testing can become difficult. A good rendering system will handle this for an application.

### 9.4.2 Selection

When the user clicks the mouse and drags, they expect the normal behavior of character selection to occur, where the glyphs involved become highlighted in some way. The question an application developer has to decide is whether the selection is *logical* or *visual*.

*Logical selection* is the process of selecting a contiguous sequence of characters in the underlying text. Due to re-ordering, etc. during rendering, this may result in a non-contiguous sequence of glyphs being highlighted. While the visual process of selection may look rather odd, having a contiguous sequence of characters selected is usually much easier for application developers.

On the other hand, users may strongly prefer *visual selection*, where a contiguous sequence of glyphs is highlighted, regardless of what is being selected in the underlying stored text. This may seem to be the most visually appealing approach to use, but the complexity increase on the part of application developers, as they try to grapple with non-contiguous sequences of characters in their selections, may mean that logical selection is more appropriate.

### 9.4.3 Arrow Keys

What happens when the user presses an arrow key? Should the cursor move logically or visually? This issue is akin to the previous one regarding selection. Logical cursor movement moves the cursor linearly through the underlying text. The result can mean the cursor jumps around on the screen, which may confuse users. Visual cursor movement ensures a linear movement on the screen, but may involve jumping around in the underlying text. Visual cursor movement, however, is not as hard to implement as visual selection.

There is, however, not a single appropriate method of cursor handling. An application may provide logical cursor movement for text that includes re-ordered glyphs, split graphs and other rearrangement, but may provide visual movement if the text is bidirectional.

## 9.5 References

[CON2000] Constable, Peter. Understanding multilingual software on MS Windows: the answer to the ultimate question of fonts, keyboards and everything. ms. Dallas, TX: SIL International, 2000. Available in CTC Resource Collection 2000 CD-ROM, by SIL International.

## Section 10

### Glossary

---

*Authors: Melinda Lyons, et al.*

**abstract character** — a unit of information used for the organization, control or representation of textual data. Abstract characters may be non-graphic characters used in textual information systems to control the organization of textual data (e.g. U+FFFF INTERLINEAR ANNOTATION ANCHOR), or to control the presentation of textual data (e.g. U+200D ZERO WIDTH JOINER).

**abstract character repertoire** — a collection of abstract characters compiled for the purposes of encoding. See also [charset](#).

**advance height** — the amount by which the current display position is adjusted vertically after rendering a given [glyph](#). This number is generally only meaningful for vertical [writing systems](#), and is usually zero within [fonts](#) used for horizontal writing systems.

**advance width** — the amount by which the current display position is adjusted horizontally after rendering a given [glyph](#).

**ASCII** — a standard that defines the 7-bit numbers ([codepoints](#)) needed for most of the U.S. English [writing system](#). The initials stand for American Standard Code for Information Interchange. Also specified as ISO 646-IRV.

**ascent** — the distance between the top of the line of text and the baseline, or the distance from the baseline to the top of the highest [glyph](#) in a [font](#).

**attachment point** — a point defined relative to a [glyph](#) outline such that if two attachment points on two glyphs are positioned on top of each other, the glyphs are positioned correctly relative to each other. For example, a base character may have an attachment point used to position a [diacritic](#), which would also have an attachment point.

**baseline** — the vertical point of origin for all the [glyphs](#) rendered on a single line. [Roman scripts](#) have a baseline on which the glyphs appear to “sit,” with occasional descenders below. Many Indic scripts have a “hanging” baseline, in which the bulk of the letters are placed below the baseline, with occasional ascenders above the line. Some scripts, such as Chinese, use a centered baseline, where the glyphs are all positioned with their centers on the baseline.

**Basic Multilingual Plane (BMP)** — the portion of [Unicode’s codespace](#) in which all of the most commonly used [characters](#) are [encoded](#), corresponding to [codepoints](#) U+0000 to U+FFFF, abbreviated as BMP. Also known as [Plane 0](#)<sup>136</sup>. See also [Supplementary Planes](#).

**bidirectionality** — the characteristic of some [writing systems](#) to contain ranges of text that are written left-to-right as well as ranges that are written right-to-left. Specifically, in Arabic and Hebrew scripts, most text is written right-to-left, but numbers are written left-to-right. This can also be used to refer to text containing runs in multiple writing systems, some RTL and some LTR.

**BMP** — See [Basic Multilingual Plane](#).

**bom** — See [byte order mark](#).

---

<sup>136</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=3plane](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=3plane)

**bounding box** — the rectangular area containing the entire visual portion of a [glyph](#) but excluding the side-bearings and advance width (or height).

**byte order mark (bom)** — the [Unicode](#) character U+FEFF ZERO WIDTH NO-BREAK SPACE when used as the first character in a [UTF-16](#) or [UTF-32](#) plain text file to indicate the byte serialization order, i.e. whether the least significant byte comes first (little-endian) or the most significant byte comes first (big-endian). Byte order is not an issue for [UTF-8](#), though the byte order mark is sometimes added to the beginning of UTF-8 encoded files as an encoding signature that applications can look for to detect that the file is encoded in UTF-8. See [http://www.unicode.org/unicode/faq/utf\\_bom.html](http://www.unicode.org/unicode/faq/utf_bom.html).

**cascading style sheets (CSS)** — one of two stylesheet languages used in Web-based protocols (the other is [XSL](#)). CSS is mainly used for rendering HTML, but can also be used for rendering [XML](#). It is much less complex than XSL, i.e., it can only be used when the structure of the source document is already very close to what is desired in the final form.

**character** — (1) a symbol used in writing, distinguished from others by its meaning, not its specific shape; similar to [grapheme](#). It relates to the domain of orthographies and writing. See [orthographic character](#).

(2) specific to the implementation of computers and other information systems. See also [abstract character](#) and [encoded character](#).

**character encoding form** — a system for representing the [codepoints](#) associated with a particular coded character set in terms of code values of a particular datatype or size. For many situations, this is a trivial mapping: codepoints are represented by bytes with the same integer value as the codepoint. Some encoding forms may represent codepoints in terms of 16- or 32-bit values, though, and some 8-bit encoding forms may be able to represent a [codespace](#) that has more than 256 codepoints by using multiple-byte sequences. Most [encoding forms](#) are designed specifically for use in connection with a particular [coded character set](#); e.g. [UTF-8](#) is used specifically for encoded representation of the [Universal Character Set](#) defined by [Unicode](#) and ISO/IEC 10646. Some encoding forms may be designed for use with multiple repertoires, however. For example, the ISO 2022 encoding form supports an open collection of coded character sets and specifies changes between character sets in a data stream using escape sequences.

**character encoding scheme** — a [character encoding form](#) with a specific byte order serialization (relevant mainly for 16- or 32-bit encoding forms).

**character set encoding** — a system for encoded representation of textual data that specifies the following: (1) a [coded character set](#), (2) one or more [character encoding forms](#) and (3) one or more [character encoding schemes](#).

**charset** — an identifier used to specify a set of characters. Used particularly in Microsoft Windows and [TrueType fonts](#), and in HTML and other Internet or Web protocols to refer to identifiers for particular subsets of the [Universal Character Set](#).

**CJKV (Chinese, Japanese, Korean and Vietnamese)** — the significance of this grouping of languages is that all have [writing systems](#) that use Han ideographic [characters](#).

**cmap** — character-glyph map: the table within a font containing a mapping of [codepoints](#) (characters) to [glyph ID](#) numbers. In an [Unicode](#)-based font the codepoints are Unicode values; in other fonts they correspond to other [encodings](#).

**coded character set** — an abstract character repertoire together with an assignment of numeric [codepoints](#) for each character; a collection of [encoded characters](#). Also called a [codepage](#).

**codepage** — (1) synonym for [coded character set](#).

(2) synonym for [character set encoding](#); i.e. In some contexts, codepage is used to refer to a specification of a character repertoire and an encoding form for representing that repertoire.

(3) In some systems, a mapping between encoded characters in [Unicode](#) and a non-Unicode encoding form; e.g. Microsoft Windows codepage 1252.

**codepoint** — a numeric value used as an encoded representation of some [abstract character](#) within a computer or information system. Codepoints are integer values used to represent particular [characters](#) within a particular [encoding](#).

**codespace** — the full range of numeric [codepoint](#) values allowed in a [coded character set](#).

**complex script** — a script characterized by one or more of the following: a very large set of characters, right-to-left or vertical rendering, [bidirectionality](#), contextual [glyph](#) selection (shaping), use of [ligatures](#), complex glyph positioning, glyph reordering, and splitting characters into multiple glyphs.

**conjunct** — a [ligature](#), in particular, a ligature representing a consonant cluster in an Indic script.

**CSS** — See [cascading style sheets](#).

**dead key** — a key in a particular keyboard layout that does not generate a [character](#), but rather changes the character generated by a following keystroke. Dead keys are commonly used to enter [accented](#) forms of letters in writing systems based on [Roman script](#).

**deep encoding** — See [semantic encoding](#).

**descent** — the distance between the bottom of the line of text and the baseline, or the distance from the baseline to the bottom of the lowest [glyph](#) in a [font](#).

**diacritic** — a written symbol which is structurally dependent upon another symbol; that is, a symbol that does not occur independently, but always occurs with and is visually positioned in relation to another [character](#), usually above or below. Diacritics are also sometimes referred to as accents. For example, acute, grave, circumflex, etc.

**digraph** — a [multigraph](#) composed of two components.

**display encoding** — See [presentation-form encoding](#).

**document** — a collection of information. This includes the common sense of the word, i.e. an organisation of primarily textual information that can be produced by a word processing or data processing application. It goes beyond this, however, to include structured information held within an [XML](#) file. Each XML file is considered to contain one document, whatever the structure and type of that information.

**Document Type Definition (DTD)** — a markup declaration used by [SGML](#) and [XML](#) that contains the formal specifications, or grammar, of an SGML or XML document. One use of the DTD is to run a validation process over an XML file, which indicates if it matches the DTD, or if not, provides a listing of each line at which the file fails some part of the required structure.

**DTD** — See [Document Type Definition](#).

**em square** — the square grid which is the basis for the design of all [glyphs](#) within a given [font](#); so called because it historically corresponded to the size of the letter M. When rendering, the requested point size specifies the size of the font's em square to which all glyphs are scaled.

**em units** — the coordinates in which points in a [glyph](#) are defined. An important number is the number of em units in the [em square](#).

**encoded character** — an [abstract character](#) in some repertoire together with a [codepoint](#) to which it is assigned within a [coded character set](#)<sup>137</sup>. Encoded characters do not necessarily correspond to [graphemes](#).

**encoding** — (1) synonym for a [character encoding form](#).

(2) synonym for a [character set encoding](#). This usage is common, especially in cases in which distinctions between a coded character set and a character encoding form is not important (i.e. 8-bit, single-byte implementations). Someone might think of an encoding as simply a mapping between byte sequences and the [abstract characters](#) they represent, though this model is not adequate to describe some implementations, particularly [CJKV](#) standards, or [Unicode](#) and ISO/IEC 10646.

---

<sup>137</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=codedcharset](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=codedcharset)



**Extensible Markup Language (XML)** — a standard for marking up data so as to clearly indicate its structure, generally in a way that indicates the meaning of different parts of it rather than how they will be displayed. See <http://www.w3.org/XML/> for details.

**Extensible Stylesheet Language (XSL)** — a language for expressing stylesheets. It consists of two parts: [XSL transformations](http://www.w3.org/Style/XSL/) (XSLT) and an [XML](http://www.w3.org/Style/XSL/) vocabulary for specifying formatting semantics. See <http://www.w3.org/Style/XSL/> for full details.

**Extensible Stylesheet Language Transformations (XSLT)** — a language used to convert one [XML](http://www.w3.org/TR/xslt) document into another. See <http://www.w3.org/TR/xslt> for full specifications.

**feature** — a way of indicating variant [renderings](#) for a particular string using the same [font](#); for example, enabling some [ligature](#) replacements or not.

**font** — a file containing a collection of [glyphs](#) and related supporting information used to [render](#) text.

**GDL** — See [Graphite](#).

**glyph** — a shape that is the visual representation of a [character](#). It is a graphic object stored within a [font](#). Glyphs are objects that are recognizably related to particular characters and which are dependent on particular design (i.e. g, **g** and *g* are all distinct glyphs). Glyphs may or may not correspond to characters in a one-to-one manner. For example, a single character may correspond to multiple glyphs that have complementary distributions based upon context (e.g. final and non-final sigma in Greek), or several characters may correspond to a single glyph known as a [ligature](#) (e.g. conjuncts in Devanagari script). (For more information on glyphs and their relationship to characters, see ISO/IEC TR 15285.)

**glyph ID** — the unique number within a [font](#) identifying a single [glyph](#).

**glyph outline** — a series of curves describing the shape of a [glyph](#). The [renderer](#) will fill in this outline to make a solid glyph appear.

**grapheme** — anything that functions as a distinct unit within an orthography. A grapheme may be a single [character](#), a [multigraph](#), or a [diacritic](#), but in all cases graphemes are defined in relation to the particular orthography.

**Graphite** — a package developed by [SIL](#)<sup>138</sup> to provide “[smart rendering](#)” for [complex](#) writing systems in an extensible way. It is programmable using a language called Graphite Description Language (GDL). Because it is extensible, it can be used to provide rendering for minority languages not supported by [Uniscribe](#).

**IME** — See [input method editor](#)<sup>139</sup>.

**input method** — any mechanism used to enter textual data, such as keyboards, speech recognition or handwriting recognition. The most common form of input method is the keyboard. Input method is intended to include all forms of keyboard handling, including but not limited to input methods that are available for Chinese and other very-large-character-set languages and that are commonly known as [input method editors](#) (IMEs). An IME is taken to be a specific type of the more general class of input methods.

**input method editor (IME)** — a special form of keyboard input method that makes use of additional windows for character editing or selection in order to facilitate keyboard entry of [writing systems](#) with very large character sets.

**internationalization** — a process for producing software that can easily be adapted for use in (almost) any cultural environment; i.e. a methodology for producing software that can be [script-enabled](#) and is [localisable](#). Sometimes abbreviated as “I18N”.

**kern** — to adjust the display position whilst [rendering](#) in order to visually improve the spacing between two [glyphs](#). For instance, kerning might be used on the word WAVE to reduce the illusion of white space between the diagonal strokes of the W, A, and V.

---

<sup>138</sup> <http://www.sil.org>

<sup>139</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=ime](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=ime)

**Keyman** — an input method program which changes and rearranges incoming [characters](#) to allow easy ways of typing data in [writing systems](#) that would otherwise be difficult or inconvenient to type. See [www.tavultesoft.com/keyman](http://www.tavultesoft.com/keyman)<sup>140</sup>.

**LANGID** — in the Microsoft Win32 API, a 16-bit integer used to identify a language or locale. A LANGID is composed of a 10-bit primary language identifier together with a 6-bit sub-language identifier (the latter being used to indicate regional distinctions for locales that use the same language).

**language ID** — a constant value within some system used for metadata identification of the language in which information is expressed. May be numeric or character based, depending on the system.

**Latin script** — See [Roman script](#).

**left side-bearing** — the white space at the left edge of a [glyph's](#) visual representation, or more specifically, the distance between the current horizontal display position and the left edge of the glyph's [bounding box](#)<sup>141</sup>. A positive left side-bearing indicates white space between the glyph and the previous one; a negative left side-bearing indicates overlap or overhang between them.

**ligature** — a single shape or [glyph](#) that represents two or more underlying [characters](#). See also [conjunct](#).

**locale** — a collection of parameters that affect how information is expressed or presented within a particular group of users, generally distinguished from one another on the basis of language or location (usually country). Locale settings affect things such as number formats, calendrical systems and date and time formats, as well as language and [writing system](#).

**localisability** — the extent to which the design and implementation of a software product allows potential for [localisation](#) of the software.

**localisation** — The process of adapting software for use by users of different languages or in different geographic regions. For purposes of this document, localisation has to do with the language and [script](#) of users, and is distinct from [script enabling](#), which has to do with the script in which language data is written. The localisation process may include such modifications as translating user-interface text, translating help files and documentation, changing icons, modifying the visual design of dialog boxes, etc. Sometimes abbreviated "L10N".

**mnemonic keyboard** — a keyboard layout based on the characters appearing on the keytops of the keyboard. See also [positional keyboard](#).

**multigraph** — a combination of two or more written symbols or [orthographic characters](#) (e.g. letters) that are used together within an orthography to represent a single sound. (Combinations consisting of two characters are also known as [digraphs](#).)

**multi-language enabling** — See [script enabling](#).

**multi-script encoding** — an encoding implementation for some particular language that is designed to enable [input](#) to and [rendering](#) from that [encoding](#) using more than one [writing system](#). When such an implementation is used, the different writing systems are normally based on different [scripts](#).

**multi-script enabling** — See [script enabling](#).

**non-Roman script** — A script using a set of characters other than those used by the ancient [Romans](#). Non-Roman scripts include relatively simple ones such as Cyrillic, Georgian, and Vai, and [complex scripts](#) such as Arabic, Tamil, and Khmer.

**normalization** — transformation of data to a normal form. For historical reasons, the [Unicode](#) standard allows some [characters](#) to have more than one encoded representation. For example, a may be represented as a single [codepoint](#), U+00E1 LATIN SMALL LETTER A WITH ACUTE, or two codepoints, U+0061 LATIN SMALL LETTER A and U+0301 COMBINING ACUTE ACCENT. A normalization scheme is used to standardize the codepoints so that every character is always represented by the same sequence of codepoints. Normalization is described in the Unicode Standard Section 5.7, Normalization.

---

<sup>140</sup> <http://www.tavultesoft.com/keyman/>

<sup>141</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Boundbox](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Boundbox)

**OpenType** — Microsoft's [smart font](#) rendering technology. An extension to the [TrueType font](#) specification. See also [Uniscribe](#).

**orthographic character** — A written symbol that is conventionally perceived as a distinct unit of writing in some [writing system](#) or orthography.

**PDF** — See [Portable Document Format](#).

**PERL** — See [Practical Extraction and Reporting Language](#).

**plain text** — textual data that contains no document-structure or format markup, or any tagging devices that are controlled by a higher-level protocol. The meaning of plain text data is determined solely by the character encoding convention used for the data.

**plane** — one technical use of this term is for 64K blocks of codepoints in [Unicode](#). Plane zero is the original 64K [codepoints](#) that can be represented in a single 16-bit character. See also [Basic Multilingual Plane](#), [supplementary planes](#), and [surrogate pair](#).

**Portable Document Format (PDF)** — a particular file format for the storage of electronic documents in a paged form. Created by [Adobe](#)<sup>142</sup> around their Adobe Acrobat product. Usually created from a [Postscript](#) page description.

**positional keyboard** — a keyboard layout defined in terms of the relative positions of keys rather than what they have printed on them. See also [mnemonic keyboard](#).

**Postscript** — a page description language defined by [Adobe](#)<sup>143</sup>. Originally implemented in laser printers so pages were described in terms of line drawing commands rather than as a bitmap.

**Postscript font** — a font in a format suitable for use within a [Postscript](#) document. There are many types. Type 1 is the most common and is what is meant most commonly when people refer to Postscript fonts. There are also ways of embedding other font formats into a Postscript document. For example a Type 42 font is a [TrueType font](#) formatted for use within a Postscript document. Type 1 fonts differ in the way their outlines are described from TrueType fonts.

**Postscript name** — a name associated with a [glyph](#) by the font's designer. Originally a name assigned by [Adobe](#)<sup>144</sup> to certain standard glyphs.

**Practical Extraction and Reporting Language (PERL)** — an interpreted programming language particularly strong for text processing.

**presentation-form encoding** — a character encoding system in which the [abstract characters](#) that are encoded match one-for-one with the [glyphs](#) required for text display. Such encodings allow correct rendering of [writing systems](#) on "dumb" [rendering](#) systems by having distinct [codepoints](#)<sup>145</sup> for contextual forms, positional variants, etc. and are designed on the basis of rendering needs rather than on the basis of character semantics (the linguistically relevant information). Also known as glyph encoding, display encoding or surface encoding; distinguished from [semantic encoding](#).

**Private Use Area (PUA)** — a range of [Unicode codepoints](#) (E000 - F8FF and [planes](#) 15 and 16) that are reserved for private definition and use within an organisation or corporation for creating proprietary, non-standard character definitions. For more information see The Unicode Consortium, 1996, pp. 619 ff.

**PUA** — See [Private Use Area](#).

**rasterising** — converting a graphical image described in terms of lines and fills into a bitmap for display on an imaging device.

---

<sup>142</sup> <http://www.adobe.com>

<sup>143</sup> <http://www.adobe.com>

<sup>144</sup> <http://www.adobe.com>

<sup>145</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=codepoint](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=codepoint)

**regression test** — a test (usually a whole set of tests, often automated) designed to check that a program has not “regressed”, that is, that previous capabilities have not been compromised by introducing new ones.

**render** — to display or draw text on an output device (usually the computer screen or paper). This usually consists of two processes: transforming a sequence of [characters](#) to a set of positioned [glyphs](#) and [rasterising](#) those glyphs into a bitmap for display on the output device.

**right side-bearing** — the white space at the right edge of a [glyph's](#) visual representation, or more specifically, the distance between the display position after a glyph is rendered and the right edge of the glyph's [bounding box](#). A positive right side-bearing indicates white space between the glyph and the following one; a negative right side-bearing indicates overlap or overhang between them.

**Roman script** — The script based on the alphabet developed by the ancient Romans ("A B C D E F G ..."), and used by most of the languages of Europe, including English, French, German, Czech, Polish, Swedish, Estonian, etc. Also called Latin script.

**schema** — in markup, a set of rules for document structure and content.

**script** — a maximal collection of [characters](#) used for writing languages or for transcribing linguistic data that have graphic representations, share common characteristics of appearance, share a common set of typical behaviours, have a common history of development, and that would be identified as being related by some community of users.

**Script Description File (SDF)** — a file describing certain kinds of complex script behaviour, used to control a rendering engine to which it has given its name. Created by Tim Erickson and used in [Shoobox](#)<sup>146</sup>, [LinguaLinks](#)<sup>147</sup>, and ScriptPad.

**script enabling** — providing the capability in software to allow documents to include text in multiple languages or scripts, and to handle input, display, editing and other text-related operations of text data in multiple languages and scripts. Script enabling has to do with the script in which language data is written, as opposed to [localisation](#), which has to do with the language and script of the user interface.

**SDF** — See [Script Description File](#).

**semantic encoding** — an encoding that has the property of one [codepoint](#) for every semantically distinct character (the linguistically relevant units). In general, such encodings require the use of “smart” rendering systems for correct appearance to be achieved, but are more appropriate for all other operations performed on the text, especially for any form of analysis. Also known as deep encoding; distinguished from [presentation-form encoding](#).

**SFM** — See [Standard Format Marker](#).

**SGML** — See [Standard Generalized Markup Language](#).

**side bearing** — the white space at the edge of a [glyph](#); see [left side-bearing](#), [right side-bearing](#). There can also be top and bottom side bearings, of use when [rendering](#) text vertically.

**sort key** — a sequence of numbers that when appropriately processed using a particular standard algorithm will position the corresponding string in the correct sort position in relation to other strings. The sort key need not correspond one number to one [codepoint](#) in the input string.

**Standard Format Marker (SFM)** — SIL has a proprietary format called “standard format markers” (SFM). It is possible (and even probable) that SFMs in a single document have different character encodings. When converting to one encoding (Unicode) these must be converted with different mapping files. A standard format marker begins with a backslash (\). For example, \p would represent a paragraph tag.

**Standard Generalized Markup Language (SGML)** — a notation for generalized markup developed by the [International Organization for Standardization](#)<sup>148</sup> (ISO). It separates textual information from the

---

<sup>146</sup> <http://www.sil.org/computing/catalog/shoobox.html>

<sup>147</sup> <http://www.sil.org/computing/catalog/linglink.html>

<sup>148</sup> <http://www.iso.org/>

processing function used for formatting. It was found difficult to parse, due to the many variants possible, and so [XML](#) was developed as a subset to resolve the ambiguities and to make parsing easier.

**smart font** — a [font](#) capable of performing transformations on complex patterns of [glyphs](#), above and beyond the simple character-to-glyph mapping that is a basic function of font rendering (see [cmap](#)). The information specifying the smart behavior is typically in the form of extra tables embedded in the font, and will generally allow layered transformations involving one-to-many, many-to-one, and many-to-many mappings of glyphs.

**smart rendering** — a [rendering](#) process that uses a [smart font](#).

**supplementary planes** — [Unicode](#) Planes 1 through 16, consisting of the supplementary code points, corresponding to [codepoints](#) U+10000 to U+10FFFF. In The Unicode Standard 3.1, characters were assigned in the supplementary planes for the first time, in Planes 1, 2 and 14. See also [Basic Multilingual Plane](#).

**surface encoding** — See [presentation form encoding](#).

**surrogate pair** — a mechanism in the [UTF-16](#) encoding form of [Unicode](#) in which two 16-bit code units from the range 0xD800 to 0xDFFF are used to encode Unicode [supplementary plane](#) characters, i.e. with Unicode scalar values in the range U+10000 to U+10FFFF.

**tokenisation** — the process of analysing a string into a contiguous sequence of smaller units: for example, word breaking or syllable breaking or the creation of a [sort key](#).

**TrueType font** — font format used primarily in Windows and on the Mac, allows for [glyph](#) scaling and hinting.

**Unicode** — An industry-wide [character set encoding](#) standard that aims eventually to provide a single standard that supports all the [scripts](#) of the world. Unicode is closely related to ISO/IEC 10646.

**Uniscribe (Unicode Script Processor)** — due to technical limitations in [OpenType](#), it is necessary to pre-process strings before applying OpenType smart behaviour. Microsoft uses a particular DLL (Dynamic Link Library) called Uniscribe to do this pre-processing. Uniscribe does all of the script specific, [font](#) generic processing of a string (such as reordering) leaving the font specific processing (such as contextual forms) to the OpenType lookups of a font.

**Universal Character Set (UCS)** — the coded character set defined by [Unicode](#) and ISO/IEC 10646, intended to support all commonly used characters from all [writing systems](#), current and past.

**UTF-8** — an encoding form for storing [Unicode codepoints](#) in terms of 8-bit bytes. Characters are encoding listing sequences of 1-4 bytes. Characters in the [ASCII](#) character set are all represented using a single byte. See [http://www.unicode.org/unicode/faq/utf\\_bom.html](http://www.unicode.org/unicode/faq/utf_bom.html).

**UTF-16** — an encoding form for storing [Unicode codepoints](#) in 16-bit words. It includes the concept of [surrogate pairs](#) to encode values from U+10000 - U+10FFFF as two 16-bit words.

**UTF-32** — an encoding form for storing [Unicode codepoints](#) in 32-bit words. Since 32 bits encompasses the entire range of Unicode, every codepoint is encoded as a single 32-bit word. See Unicode Technical Report #19.

**Visual OpenType Layout Tool (VOLT)** — a tool to build [OpenType](#) tables and add them to a [font](#).

**VOLT** — See [Visual OpenType Layout Tool](#).

**writing system** — an implementation of one or more [scripts](#) to form a complete system for writing a particular language. Most writing systems are based primarily upon a single script; writing systems for Japanese and Korean are notable exceptions. Many languages have multiple writing systems, however, each based on different scripts; e.g. the Mongolian language can be written using Mongolian or Cyrillic scripts. A writing system uses some subset of the characters of the script or scripts on which it is based with most or all of the behaviours typical to that script and possibly certain behaviours that are peculiar to that particular writing system.

**XML** — See [Extensible Markup Language](#).

**XSL** — See [Extensible Stylesheet Language](#).

**XSLT** — See [Extensible Stylesheet Language Transformations](#).

<sup>149</sup> <http://scripts.sil.org/>

<sup>150</sup> <http://www.unicode.org/>

<sup>151</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guidelines\\_Sec\\_6\\_2](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guidelines_Sec_6_2)

<sup>152</sup> [http://scripts.sil.org/cms/scripts/page.php?site\\_id=nrsi&item\\_id=Guideines\\_Sec\\_6\\_3](http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&item_id=Guideines_Sec_6_3)

<sup>153</sup> <http://www.evertype.com/sc2wg2.html>

---

# Index

- academics, **22**
- accent(s), 5, 11, 12, **28**, 66
- accessibility, 6
- Acrobat, 39
- actor(s), 16, **17**, 23
- Adobe Systems, 32, 54, 59
- African writing systems, 53
- Alan Wood's Unicode Resources, 56
- alphabet, 4
- Amharic, 19
- analysis, 6, **8**, 9, 14, 51
- Apple Advanced Typography (AAT), 32, 63, 65, 66, 67, 68
- Apple Computer, 8, 32, 38, 54, 59, 63
- application(s), 6, 11, 13, 24, 25, 26, 51, 63, 67, 68
- Arabic, 4, 26, 27, 34, 64, 65, 68
- ASCII, 30, 38
- Association Typographique Internationale (ATypI), 22, 56
- ATSUI (Apple Type Services for Unicode Imaging), **63**, 68
- attachment points, **66**, 67
- BabelMap, 52
- Bantu, 29
- Batak, 18
- behavior(s), 4, 12, 14, 24, 26, 27, 56
- Bengali, 31, 33, 54
- Burmese, 64
- character(s), 7, 27, **28**, 29, 30, 31, 33, 34, 36, 38, 42, 48, 61, 68
  - assignments, 41
  - attributes, 68
  - encoding, 38
- China, 21
- Chinese, 5, 34, 36, 50
- codepages, 38, 39
- codepoint(s), **30**, 32, 34, 47, 61, 62
- collisions, 67
- combining order, 51
- commercial distribution, 4, 13, 39, 54, 56
- communication, 4, 5, 6, 10, 15, 22
- community, 4, 11, 12, 13
- component(s), 4, **6**, 7, 8, 9, 10, 11, 12, 13, 14, 58
- conjunct(s), 25, 26, 42, 63
- conversion, 6, **9**, 12, 14, 15, 46, 61, 62, 63
- copyleft, 52, 59
- copyright, 56
- culture, 4, 11, 16, 53
- cursor, 47, 51, 62, 69
- Cyrillic, 4
- Danish, 28
- database, 12
- designer(s), 31, 32, 53, 55, 56, 57, 58
- Devanagari, 42, 63, 64
- developer(s), 12, 13, 15, 18, 26, 33, 38, 55, 56, 62, 69
- development, 6, 12, 24, 56
- diacritic(s), 5, 10, 24, **28**, 31, 42, 50, 51, 53, 56, 62, 66, 67, 68
- digraph, 28, 42
- directionality, 6, 26, 47, 62, 64, 65, 68, 69
- distribution, 12, 58
- documentation, **15**, 58
- DTL Fontmaster, 58
- Dutch, 29
- dynamic composition, 42
- editing, 5, 8, 24, **47**, 52, 68
- encoding(s), 5, **7**, 8, 9, 10, 12, 13, 34, 36, **38**, 40, 53, 61, 62
- encourager, **17**, 20, 22
- English, 4, 6, 7, 8, 28, 32, 33, 36, 38, 42, 48, 64
- Ethiopic, 22, 57, 63
- Everson Typography, 44
- export, 26, 27
- Farsi, 4, 19
- features, 68
- FLOSS, 52, 59
- font, 5, 6, 12, 53
- Font Creator, 58
- font(s), 8, 13
  - commercial, 56
  - design, 54
  - development, 56
  - dumb, 62, 63
  - modification, 56
  - non-Latin/non-Roman, 54
  - simple, 4
  - smart, 63, 67
  - specification, 56
  - style, 56, 57, 62
- FontLab, 58, 59
- Fontographer, 57, 59
- Fonts in Cyberspace, 55
- freeware, 13, 54, 55
- French, 28, 29
- Gentium, 14
- globalization, 18
- glyph(s), **31**, 32, 33, 34, 53, 56, 61, 62
  - composite, 31
  - deletion, 64
  - design, 57
  - ID, 62, 68
  - identification, 62
  - insertion, 64
  - positioning, 62, 65
  - processing, 62, 63
  - references, 62
  - rendering, 61, 62
  - replacement, 63
  - shape, 53
  - split, 68, 69
  - substitution, 64, 66
- government, 4, 7, 18, **20**, 21, 55
- grapheme(s), 28, **29**, 31, 38, 42
- Graphite, 19, 32, **63**, 65, 67, 68
- Greek, 14, 31, 34, 35
- Gujarati, 11
- GX, 63
- handwriting, 32, 34
- Hebrew, 6, 64
- hexadecimal notation, 30
- Hindi, 26
- hinting, **58**, 67
- hit testing, 68
- horizontal, 66
- hyphenation, 6, 8
- IBM, 38
- implementer, **16**, 20
- import, 26, 27
- India, 11
- Indic scripts, 24, 31, 42
- industry, **18**, 22, 38
- input, 5, 7, 34, 36, 39
- input method editors (IMEs), 34, **50**
- Integration, 15
- intellectual property (IP), 12, 13, 56
- International Electrotechnical Commission (IEC), 40
- International Font Technology Association (IFTA), 22
- International Organization for Standardization (ISO), 18, **19**, 21, 40
- International Phonetic Alphabet (IPA), 28, 34, 49, 52
- Internet Engineering Task Force (IETF), 20
- ISO/IEC, 41
- ISO/IEC 10646, 40, 41, 44, 45

- ISO/IEC JTC 1/SC 2/WG 2 (WG2), 44
- Japanese, 34, 36
- Joint Technical Committee 1 (JTC1), 41
- justification, 63, 67
- kashidas, 63, 67
- Kerning, 57, 63, 65, 66, 67  
contextual, 66, 67  
cross-stream, 66
- keyboard(s), 34, 39, 48, 50, 51, 52  
definition, 8  
layout, 10, 15, 17, 22, 47, 48, 52
- keyboarding, 5, 7, 8, 10, 12, 14, 24
- Keyman, 8, 15, 19, 49, 52
- keys  
arrow, 69  
dead, 35, 49, 50  
modifier, 49  
operator, 49, 50, 51
- keystroke(s), 8, 34, 36, 47
- Khmer, 25
- Korean, 34, 36
- language communities, 4, 5, 6, 12, 16, 17, 20, 21, 22, 42, 54, 56, 57, 58
- language expert, 16
- language(s), 4, 5, 10, 16, 20, 53
- Lanna, 31, 33
- Lao, 14, 19
- LaoScript for Windows, 14
- Latin, 50
- legibility, 58
- legislation, 21
- lettershapes, 5, 54, 57
- licensing, 54, 55, 56
- ligature(s), 5, 8, 24, 31, 42, 62, 63, 64, 67
- line breaking, 51, 64, 65, 67, 68
- Linguist's Software, 55
- linguists, 22, 39, 53
- Linotype, 54
- Linux, 63
- Logical Framework Analysis, 11
- Macintosh, 6, 8, 14, 27, 30, 38, 49, 52
- mailing lists, 56
- mapping(s), 33, 34, 35, 36
- Microsoft, 8, 9, 10, 18, 21, 32, 36, 38, 52, 54, 59, 63, 67, 68
- Ministry for Capacity Building of Ethiopia and the Economic Commission for Africa (ECA), 22
- minority languages, 13, 18, 19, 20, 22, 53
- Mithi, 18
- Mongolia, 21
- Monotype, 54, 56
- Mozilla, 19
- multigraphs, 28
- Multilingual Computing & Technology magazine, 23
- Myanmar, 17
- National Electronics and Computer Technology Center of the Royal Thai Government (NECTEC), 21, 55
- national standards bodies, 18, 21, 38, 44
- needs, 11, 24, 56
- needs statement, 25
- Nepalese, 19
- New Tai Lue, 14
- non-governmental organizations (NGOs), 4, 13, 20, 55
- non-Latin/non-Roman fonts, 8, 54
- non-Latin/non-Roman writing systems, 54
- Non-Roman Script Initiative (NRSI), 7, 25, 45
- not-for-profit agencies, 40, 55
- open source, 19, 21, 52, 56, 59, 63
- OpenOffice, 21
- OpenType, 8, 25, 26, 32, 63, 65, 67, 68
- operating system(s), 6, 9, 24, 25, 26, 55, 63
- orthography, 4, 16, 17, 18, 20, 21, 22, 28, 30, 42  
description, 56  
statement, 25
- output, 7
- People's Republic of China, 21
- phonemes, 9, 42
- phonetic representation, 50
- planning, 4, 6, 11, 12, 13, 56
- polygraphs, 28
- positioning, 8, 62, 63, 65, 67
- press, 22
- professional groups, 22, 56
- public agencies, 55, 56
- publishing, 6, 11, 12, 39
- quality, 18, 53, 54, 55
- Quality and Standards Authority of Ethiopia (QSAE), 21
- readability, 58
- rearrangement, 26, 69
- regulation, 17
- rendering, 5, 8, 10, 12, 34, 51, 61, 62, 65  
dumb, 33, 51, 68  
glyph, 61, 62  
linear, 68  
simple, 62  
smart, 8, 13, 14, 32, 33, 51, 63, 68
- re-ordering, 63, 64, 68, 69
- repositioning, 67
- research, 20, 56
- researcher, 16, 20, 22
- roles, 16, 23
- Russian, 4
- Scanfont, 59
- scanning, 57
- Script Encoding Initiative, 45
- script expert(s), 56, 57, 58
- selection, 68
- sequence checking, 50, 51
- sequential positioning, 62
- shareware, 14, 55
- shifting, 63, 66
- Shoebox, 10
- SIL International, 7, 13, 14, 19, 20, 45, 55, 63
- Sindhi, 4
- sorting, 5, 6, 8, 9, 10, 12
- Southeast Asia, 11, 51
- spacing, 53, 57
- Spanish, 9
- speech synthesis, 8, 9
- spell-checking, 5, 6, 8, 9
- stakeholders, 26
- standard(s), 7, 9, 13, 15, 17, 18, 21, 22, 42, 43, 53
- standardization, 17, 18, 19, 20, 21, 22
- standardizer, 17
- standards bodies, 19, 38
- storage, 7
- substitution, 63, 64, 65
- syllabary, 4
- Tai Dam, 11
- Tai Lue, 14, 28
- technical working groups, 21
- testing, 15, 58
- Text Encoding Model, 7
- text selection, 47, 62
- Thai, 9, 10, 14, 21, 34, 48, 53
- Thailand, 55
- Tiro Typeworks, 56
- tools  
data entry and editing, 52  
design, 58
- tradition, 54, 57
- TransType, 59
- trigraph, 42
- TrueType, 59, 62, 63, 67
- Type-Design mailing list, 56
- TypeTool, 59
- TYPO-L mailing list, 56
- Ukrainian, 4
- UNESCO, 21
- Unicode, 7, 9, 13, 14, 15, 17, 19, 27, 30, 36, 38, 39, 40, 41, 42, 43, 44, 45, 46, 51, 52, 53, 56, 61, 63, 68
- Unicode Consortium, 19, 39, 40, 41, 42
- Unicode email discussion list, 43
- Unicode Standard, 27, 40, 41, 42, 43, 44
- Unicode Standard Annex (UAX), 41
- Unicode Technical Committee (UTC), 40, 41, 44, 45



**Guidelines for Writing System Support**

2003-10-31

Page 81 of 81

---

|   |   |  |
|---|---|--|
| Unicode Technical Reports (UTRs),<br>36, 41 | WG2, 44, 45   | writing system(s), 4, 5, 10, 24, 26,<br>38, 42, 44, 46, 53 |
| Uniscribe, 26, <b>63</b> , 68               | Windows, 6, 8, 14, 27, 30, 36, 38,<br>52, 63, 67          | Xenotype, 55   |
| Universal Character Set (UCS), 40           | windows, candidate and<br>composition, 50                 | Xerox, 39  |
| user interaction, <b>62</b> , 68            | word breaking, 9, 51                                      | Xishuangbanna Dai, 14                                      |
| user(s), 11, 15, 20, 24, 68                 | word demarcation, 8, 9                                    | Yamada Language Center, 55                                 |
| vendors, <b>18</b> , 38                     | World Wide Web (W3C), 20                                  | zero-width diacritic, 66                                   |
| vertical, 66                                | writing system implementations<br>(WSIs), 4, 5, 9, 10, 11 | zero-width joiner, 64                                      |
| Vietnamese, 36                              |   | zero-width non-joiner, 64                                  |
| visual representation, 51                   |   |  |