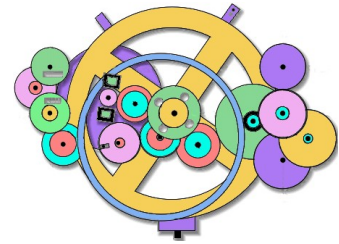


# Antikythera Publications



## DATABASE DESIGN NOTE SERIES

Relational Database Design  
<http://www.AntikytheraPubs.com>  
sweiss @ AntikytheraPubs.com

### Exploring Complex Text Layout (CTL) Multi-script Database Series #2 – Version 2

Prepared by: S. L. Weiss and F. Oberle (แฟรงค์ โอเบลล์ – ฟร็องก์ ओबरली)

As we saw in “Exploring Alphabets” – the first in this series of Design Notes, understanding and handling the storage and display of many non-Latin Scripts can be tricky concepts that many database designers and DBAs are unfamiliar with.

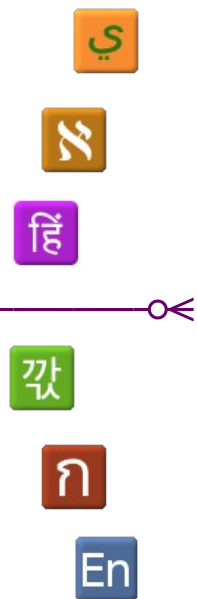
That earlier document introduced the idea of a lack of one-to-one correspondence between the characters we see (or think we see) on the screen, and the characters stored in our databases.

This document is meant to be an informal – and therefore sometimes less than precise – introduction to the factors that led some early product developers to categorize many Scripts as “Complex.” So, depending on the particular Scripts you need to support in your database, you will need to become familiar with many of these factors.

Because the names of your company’s new customers أمينة, แฟรงค์, Jennifer, りく and आदित्या could very well end up on printed reports and correspondence, a familiarity with the various characteristics the Scripts used to write their names have can be very helpful in analyzing any issues that arise.

Revised for public distribution: 20 December 2016 and January 2018

See page 42 for information on other material from Antikythera Publications.



Copyright © 2016 & 2018 by the Authors and Antikythera Publications (updated 2019)

Permission is granted to distribute unaltered copies of this document, so long as this is not done for commercial purposes.



[www.AntikytheraPubs.com](http://www.AntikytheraPubs.com)

## Database Design Notes about Multi-Language/Multi-Script Database Considerations

1. Exploring Alphabets
2. **Exploring Complex Text Layout**
3. Exploring UTF-8
4. Evaluating Fonts for use in Multi-Lingual Documents
5. Exploring Bidirectional Text Entry
6. Exploring Arabic Script Behavior
7. Exploring Han Script Behavior in Chinese

# Database Design Note Series – Exploring Complex Text Layout

## PREFACE

This document is meant to be an informal – and therefore sometimes less than precise – introduction to what are commonly known as Complex Text attributes. In spite of the CTL acronym’s appearance in many pieces of software – the most notable examples being word processors such as LibreOffice Writer and Microsoft Word – that purport to handle such layouts, many of them are rather vague about what exactly is being supported and use the term rather indiscriminately. Published standards occasionally allude to some of these “complex” attributes in one guise or another, but provide no formal definitions.<sup>1</sup>

## Objectives

The objectives of this document therefore are to provide:

- a broad overview of the subject area referred to with the unfortunate name Complex Text Layout<sup>2</sup>, usually given the acronym CTL;
- specific examples of various CTL characteristics in at least one appropriate script and language;
- specific instructions on how each example can be duplicated by a user/writer/developer who may have no familiarity with the script or language used;
- an implicit justification that ‘complex’ has, over time, become an arbitrary and obsolete categorization that should be abandoned as quickly (but in as non-disruptive a manner) as possible. What’s now considered ‘complex’ – ‘oddities’ or ‘exceptions’ – should be viewed simply as equally legitimate alternatives, some of which long predate what we consider “normal.”. With generalized design made possible by advances in software technology and continued progress toward globalization, these distinctions can and should be abandoned.

## Requirements

To experiment with these CTL behaviors, rather than simply reading about them, you will need:

- Access to a decent source of reasonably identical text samples in many languages. The opening section of the United Nations Universal Declaration of Human Rights<sup>3</sup> is a good choice, since its content – in over 360 different Languages – comes from the same source.
- Knowledge of at least one technique for entering Unicode characters that don’t appear as keys on whatever keyboard is in use. An overview of such techniques is provided in Character Entry Methods on page 35. We recommend that developers install an Input Method Editor (IME) for any Language or Script that will be used in any databases being updated/enhanced to support multi-lingual, multi-script data.
- An installed font containing the Unicode Basic Latin, Latin-1, Thai, Devanagari, Basic Hebrew, Basic Arabic, and Korean Code Blocks; the FreeSerif font<sup>4</sup> (the NanumMyeongjo font is used for Korean) is recommended unless you know how to confirm the presence of the scripts listed above on your own. Methods for identifying which of your installed fonts support which particular Languages and Scripts are out of scope for this paper, but another Design Note in this series covers this in detail (see footnote 4 below).

---

1 Tellingly, the phrases ‘complex text’ and ‘language’ are seldom found in these formal discussions!

2 We do enough ranting about this on our own, but the site <https://aharoni.wordpress.com/2011/10/23/western-asian-and-complex/> has an interesting take on this subject as well.

3 “Human beings are born free in dignity and equal rights.” The site [www.omniglot.com/udhr/index.htm](http://www.omniglot.com/udhr/index.htm) contains links to the text of Article 1 of this declaration translated into over 300 languages which can easily be copied into any test documents.

4 The FreeFont family can be downloaded from <http://www.gnu.org/software/freefont/> and likely other sites. Although there is no such thing as a “pan-unicode” font, everyone who deals with multiple scripts and/or languages should have a decent single fallback font that can be used to cover all the languages you will need. Also see the fourth Design Note in this series *Evaluating Fonts for use in Multi-Lingual Documents*, also available from [www.AntikytheraPubs.com/i18n.htm](http://www.AntikytheraPubs.com/i18n.htm).

# TABLE OF CONTENTS

Preface.....	3
Objectives.....	3
Requirements.....	3
Defining Complex Text Layout – What it is and what it isn’t!.....	7
FIGURE 1 – LANGUAGES DIALOG IN LIBREOFFICE WRITER.....	7
Role of System Components in Managing text Layouts.....	7
Characteristics of so-called Complex Text Layout (CTL).....	9
Characters and Character Cells.....	9
Alphabetic Characters.....	9
A IS FOR APPLE.....	9
Consonants and Vowels.....	9
Vowel Varieties.....	9
Characters as Diacritics.....	10
Accents, Tones, and Breathing Marks as Diacritics.....	10
Character Cells.....	10
Contextual Character Forms (Alternative Characters).....	10
Contextual Character Shaping (Shape Changing).....	11
Character Reordering and Placement.....	11
Illegal Character Combinations.....	12
Composite Characters.....	12
Ligatures (Composite Glyphs).....	12
Dead Keys.....	12
FIGURE 2 – DEAD KEYS ON A MANUAL THAI TYPEWRITER (CIRCA 1970).....	12
Perceived Cultural Issues.....	13
Text Layout Direction (Writing Mode).....	13
FIGURE 3 – BOUSTROPHEDON TEXT LAYOUT.....	14
FIGURE 4 – REVERSE BOUSTROPHEDON TEXT LAYOUT.....	14
FIGURE 5 – THE PHAESTOS DISK.....	14
Mixed Text Directions – Bidirectional Text.....	15
Default Paragraph Directionality (Primary Text Direction) in Text with Mixed Directions.....	15
Cursor Movement when Entering or Editing Text in Bidirectional Paragraphs.....	15
Paired Symbols in Text with Mixed Directions.....	17
FIGURE 6 – ENGLISH (US) KEYBOARD LAYOUT SEGMENT.....	17
FIGURE 7 – HEBREW KEYBOARD LAYOUT SEGMENT.....	17
Rulers, Guides, and Tabs in Text with Mixed Directions.....	18
Left, Right, and Center Tab Stops.....	18
FIGURE 8 – LEFT-TO-RIGHT RULER WITH TAB SETTINGS.....	18
FIGURE 9 – RIGHT-TO-LEFT RULER WITH MIRRORED TAB SETTINGS.....	19
Decimal Tab Stops.....	19
FIGURE 10 – LEFT-TO-RIGHT RULER WITH A DECIMAL TAB SETTING.....	19
FIGURE 11 – MIRRORED RTL RULER WITH A DECIMAL TAB SETTING.....	19
Detecting Primary Text Direction in Paragraphs.....	20
Text Alignment in Documents with Mixed Directions.....	20

FIGURE 12 – LEFT ALIGN.....	20
FIGURE 13 – BIDIRECTIONAL.....	20
Transitioning between Text Directions within Paragraphs.....	20
Justification.....	20
Ragged Justification.....	20
Full Justification.....	21
Kashideh.....	21
Word Breaks, Line Breaks, and Hyphenation.....	21
Collation and Sorting.....	22
A Final Reminder.....	22
CTL Examples in Practice.....	23
Disclaimer.....	23
Common Examples of Mixing Numeric Scripts.....	23
Thai Script examples using ภาษาไทย ( the Thai Language ).....	24
Brief Comments about Thai.....	24
Examples for Experimentation (No knowledge of Thai needed).....	24
FIGURE 14 – THAI, WITHOUT AND WITH FULL JUSTIFICATION.....	25
Devanagari Script examples using हिंदी भाषा ( the Hindi Language ).....	26
Brief Comments about Hindi.....	26
Examples for Experimentation (No knowledge of Hindi needed).....	26
FIGURE 15 – SAMPLE HINDI TEXT BLOCK.....	27
Arabic Script examples using اللغة العربية ( the Arabic Language ).....	27
Brief Comments about Arabic.....	27
Examples for Experimentation (No knowledge of Arabic needed).....	27
Kashideh Justification and emphasis.....	28
FIGURE 16 – ARABIC, BEFORE AND AFTER KASHIDEH JUSTIFICATION.....	28
Hebrew Script examples using שפת עברית ( the Hebrew Language ).....	29
Brief Comments about Hebrew.....	29
Examples for Experimentation (No knowledge of Hebrew needed).....	29
FIGURE 17 – SAMPLE HEBREW TEXT BLOCK SHOWN WITH AND WITHOUT VOWEL INDICATORS.....	29
Korean Script examples using 한국어 ( the Korean Language ).....	30
Brief Comments about Korean.....	30
FIGURE 18 – KOREAN TEXT EXAMPLE.....	30
Examples for Experimentation (No knowledge of Korean needed).....	32
Korean Numeric Characters.....	33
Converting Jamo Combinations to Hangeul Syllables (the basic Math).....	33
Converting Hangeul Syllables to Jamo Components (the basic Math).....	34
Character Entry Methods.....	35
FIGURE 19 – “INSERT > SPECIAL CHARACTER...” DIALOG BOX IN LIBREOFFICE WRITER.....	35
FIGURE 20 – “ONBOARD” ON-SCREEN KEYBOARD – TYPICAL OF MANY AVAILABLE.....	36
Keyboard Maps used for this Document.....	37
Thai Script.....	37

Thai Typing Demonstration/Practice (no knowledge of Thai required).....	37
FIGURE 14A – ABBREVIATED THAI TEXT SAMPLE.....	37
Devanagari Script.....	38
Hindi Typing Demonstration/Practice (no knowledge of Hindi or Devanagari required).....	38
FIGURE 15A – ABBREVIATED HINDI TEXT SAMPLE.....	38
Arabic Script.....	39
Modern Standard Arabic Typing Demonstration/Practice (no knowledge of Arabic required).....	39
FIGURE 16A – ABBREVIATED MODERN ARABIC TEXT SAMPLE.....	39
Hebrew Script.....	40
Hebrew Typing Demonstration/Practice (no knowledge of Hebrew required).....	40
FIGURE 17A – ABBREVIATED HEBREW TEXT SAMPLE.....	40
Korean Script.....	41
Korean Typing Demonstration/Practice (no knowledge of Korean required).....	41
FIGURE 18A – ABBREVIATED KOREAN TEXT SAMPLE.....	41
Other Publications.....	42

## DEFINING COMPLEX TEXT LAYOUT – WHAT IT IS AND WHAT IT ISN'T!

A minimal definition of “Text Layout” for this document would be: “the placement of single or composite characters and/or symbols into virtual character cells, and arranging those cells in the sequence and direction in which they are intended to be displayed or printed.” It is important to stress that this sequence does not always match the order in which these “characters” are stored or transmitted. The definition of a character cell and how it differs from its contents is likewise important to understanding the significance of many of the attributes of so-called “complex text.” At this point, it is only necessary to be aware that such cells exist; the use and significance of character cells will become clear as the document proceeds. The previous paper in this series, *Exploring Alphabets*,<sup>5</sup> has more information.

So what makes text layout “complex?” On a Microsoft FAQ page,<sup>5</sup> the author says: “A complex script is one that requires special processing to display and process.” The page continues with some examples, but nothing that could be considered a definition, and the term “special” is a bit vague.

The LibreOffice Writer Guide<sup>6</sup> says that Writer offers “support for Asian languages (Chinese, Japanese, Korean) and support for CTL (complex text layout) languages such as Hindi, Thai, Hebrew, and Arabic.” On the surface, these distinctions seem reasonable, but that is questionable.

Its *Getting Started Guide* offers a somewhat different interpretation of CTL: “LibreOffice ... provides support for both Complex Text Layout (CTL) and Right to Left (RTL) layout languages (such as Urdu, Hebrew, and Arabic).” Writer’s configuration dialog layouts imply moreover that these categories are related to Language or Locale which, while not entirely untrue, is misleading.

Wikipedia says that complex text layout is “the typesetting of writing systems in which the shape or positioning of a grapheme depends on its relation to other graphemes. The term is used in the field of software internationalization, where each grapheme is a character.”<sup>7</sup> The first sentence in this quote is correct, but the second is rather questionable. You can research the precise meaning of words such as character, symbol, glyph and grapheme if you wish, but with no obvious consensus on the distinctions, this paper will simply discuss the concepts using words that seem to us appropriate for the context.

The common thread in all these comments is that “complex” is a relative and often quite arbitrary term, and one could be forgiven for suspecting it is mostly used to refer to layout issues the software developers were unfamiliar with, had never encountered, or had never even heard of.

### Role of System Components in Managing text Layouts

There are often many elements of a system that contribute to what is considered the layout of text on a screen or on paper, some of which work together well, and some where conflicts occur. Such components include the Keyboard used to enter the text and the low level BIOS and boot managers of the machine that initially recognize the keyboard and determine what to do with its transmissions. Layered above those are the Operating System, which refines the interpretation of inputs passed to it from the hardware, and optional input method editors (IMEs) and utilities, which

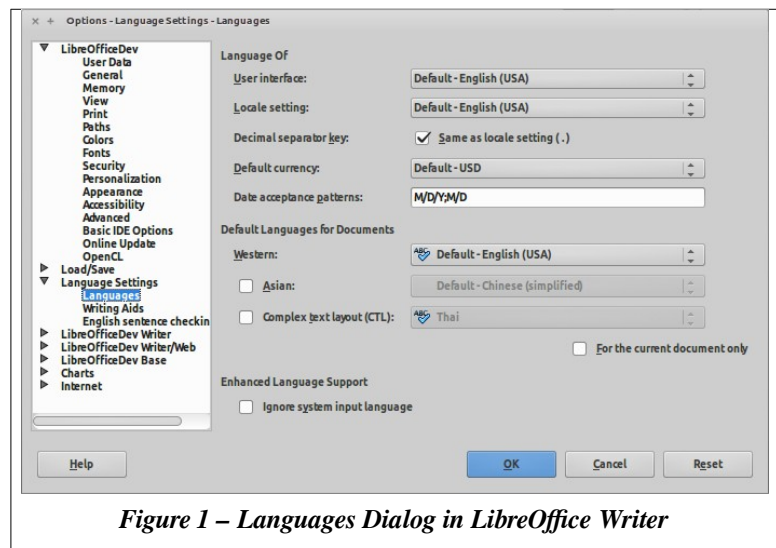


Figure 1 – Languages Dialog in LibreOffice Writer

5 See <https://msdn.microsoft.com/en-us/goglobal/bb688172.aspx>; this is a bit dated, but still seems to be their view.

6 Page 67; see <https://wiki.documentfoundation.org/images/e/e6/WG42-WriterGuideLO.pdf> to view or download.

7 See [http://en.wikipedia.org/wiki/Complex\\_text\\_layout](http://en.wikipedia.org/wiki/Complex_text_layout)

serve to translate (map) any input key sequences from native button pushes<sup>8</sup> to characters normally not relevant to a particular operating system.

Finally, there are individual applications, ranging from command line terminals, shells and desktop managers to much larger and more sophisticated<sup>9</sup> applications and suites for a wide variety of uses – many not obviously affected by text layout, although all are to some degree or another. Word processors and publishing suites come to mind immediately, but even graphics programs make use of text layout capabilities, both for display and printing. Music players should have no difficulty displaying the native-language names of recordings that are available from any location in the world. Genealogy software should be able to quote and discuss texts from the many languages that have been spoken by a user’s ancestors. Enough said.

Early computing systems were quite limited in their text layout capabilities, and generalization and internationalization of software designs was – to put it simply – technically prohibitive and too expensive. As technology matured, handling of many characteristics discussed here came to be added in a totally uncoordinated and ad hoc basis.

Over the past decades, however, as various desired capabilities became technically feasible and appeared more often, the most efficient and appropriate locations for many of these functions in the hierarchy of system components have become settled. Printer support, for example, once handled by individual applications based on their own specific needs, has long since migrated into the operating system, which presents such support as a service to any application that might need it.

The distribution of significant portions of text layout capabilities now seems to be in the late stages of such a migration. The proliferation of incompatible character encodings has now been settled with the adoption of Unicode standards, although many obsolete or primitive mappings linger on. Storage layouts of character encodings in UTF-8 format has provided even further generalization and standardization. In most modern operating systems, the alphabets of any languages – given the appropriate fonts to render them – can be freely intermixed, with their correct ordering and display transformations kept intact. Mostly. Strange anomalies will pop up in the course of any transition to a multi-lingual, multi-script system, though, and data custodians need to be aware of the impacts on their schemas.

A key component for the correct interpretation and display of “complex” text<sup>10</sup>, which isn’t covered in any detail here, is the “rendering engine.” Although not of immediate interest here, many references are available concerning these.<sup>11</sup>

So far, the recognition of Language (as opposed to merely the writing system or script used to represent it on screen or paper) remains the province of applications<sup>12</sup> rather than operating systems or even input methods. Capabilities related specifically to language, such as Spelling, Style and Grammar Checking and the like are examples of such functions, although even these now appear to some extent as operating system services. The appropriate home for other functions, such as Collation and Sorting still hasn’t been settled and, particularly where these encompass multiple languages,<sup>13</sup> may likely always remain the province of specialized software.

---

8 Regardless of the characters printed on the key tops, keyboards have no concept at all of languages or scripts; they’re simply a collection of switches arranged and grouped into what is hopefully a useful layout for a particular user or group.

9 Well, most attempt to be “sophisticated,” but the results can of course vary wildly.

10 For example, the layout adaptations shown in Examples for Experimentation (No knowledge of Thai needed) on page 24.

11 A dated, but still mostly accurate and informative link is: <http://behdad.org/text/> A link aimed primarily at text rendering on the web is: <http://blog.typekit.com/2010/10/05/type-rendering-on-the-web/> (first of an Adobe series with continuation links)

12 And don’t forget, any DBMS or RDBMS is not itself actually a “database” but an *Application* used to help create and manage the databases you build with its facilities.

13 This does occur, although I’ve never seen a convincing case that there is any useful purpose to doing so. Even in publications with side-by-side translations, indexes that intermingle words of different languages can often be more difficult to use than if the languages are kept separate.



## CHARACTERISTICS OF SO-CALLED COMPLEX TEXT LAYOUT (CTL)

Attempting to cover all the CTL characteristics that go into supporting a useful multilingual text editor or word processor would be impractical, but this section will introduce and at least informally define selected characteristics related to text layout. Following this overview, examples of the more interesting ones will be presented using appropriate languages in *CTL Examples in Practice* beginning on page 23.

### Characters and Character Cells

Much of this section is a review and more detailed commentary on subjects introduced in the first Design Note of this series, *Exploring Alphabets*, which we assume you have at least skimmed. In languages like English, we tend to think of a character as a single entity – a letter of the alphabet, a symbol used for punctuation (comma, period and such), a number or a mathematical symbol (0-9, plus and minus signs, etc.), and various symbols such as @, #, & and !. Thus, we don't tend to distinguish between characters and character cells, but for any text layout discussion, this becomes necessary. First, however, we'll look at some further distinctions among the variety of characters and symbols we use.

### Alphabetic Characters

An alphabetic character can be defined most simply as one that is part of a language's alphabet. The alphabet song taught to school children in the U.S., for instance, does not include the comma, semicolon, period, the tab, or even the space. Obvious, perhaps, but this series of Design Notes will show how fuzzy interpretation of this term can result in user interface oddities, particularly in paragraphs that mix script directions. A simple summary might be:

- All writing is done with symbols of some sort.
- All characters are symbols but not all symbols are characters;
- Not all characters are part of an alphabet;
- Some alphabetic characters may not “look like” what we call “letters.”

Detection of alphabetic characters is key to handling multi- or bidirectional paragraph layouts in multilingual documents; an illustration of this is included in the fifth Design Note in this series, *Exploring Bidirectional Text Entry*.



*A is for Apple*

### Consonants and Vowels

In English, we don't consider there to be any distinction between the representation of consonants and vowels in our text layouts; a vowel like 'e' is just as much of an alphabetic character as a consonant like 'f.' But this is not universally true in the world's alphabets. In order to be pronounced out loud, every syllable (every phoneme, if you will) must have a vowel sound but need not have any consonant, and in spite of the fact that no consonant can be spoken without adding some implicit vowel sound, consonants are considered more important in some writing systems. Vowels may even be considered optional in some of those.

### Vowel Varieties

In some writing systems, vowels aren't recorded at all. In others, vowel sounds in a particular syllable are indicated with techniques as varied as adding certain modifications to a syllable's consonant,<sup>14</sup> placing them as diacritics above or below the consonant, or – as in English – recording the vowel as a “real” independent character. To make things more interesting, even vowels that have full character status may not always be placed in the order in which they are pronounced in a particular syllable.<sup>15</sup>

<sup>14</sup> This is known as Contextual Shaping; see the eponymous section on page 11.

<sup>15</sup> See the Thai and Hindi *Examples for Experimentation* beginning on pages 24 and 26 respectively, but if you're thinking such bizarre anomalies never occur in English, consider the spelling of the last syllable in the word “syllable.” Other considerations related to vowel varieties in particular are presented in *Character Reordering and Placement* on page 11.

Recognition of this variety of vowel placements will help when considering text layout in multiple scripts and languages. Examples of each type of vowel placement will be given in *CTL Examples in Practice*.

## Characters as Diacritics

Vowel markings that are placed above or below a consonant belong to a symbol class called diacritics; other examples of diacritic “characters”<sup>16</sup> include the cedilla, the umlaut, and various tones, accents and breathing marks, any of which help indicate a difference in pronunciation of the base syllable.

## Accents, Tones, and Breathing Marks as Diacritics

These symbols are also common examples of diacritics. One fallacy in some CTL discussions is that complex text layout is more commonly required in so-called “tonal languages.” Languages often mentioned in this context include Chinese and Japanese but, except when spoken by politicians during their corruption trials, *all* spoken languages use both accents and tones. The phrase “tonal languages” simply refers to writing systems that explicitly represent these – even in cases where these representations are optional.

To clarify this distinction: Say the following sentence out loud: “But you know that left is the opposite of right, right?” – first with an accent on the word “you” and then again with the accent on the word “know.” Each suggests an entirely different interpretation, but the difference in placement of the vocal accent supplies enough information that a listener would get a sense of which was meaning was intended.

The differences heard between the last two words (“right” and “right”) is clearly a tonal one. No matter how hard you try, it is difficult to pronounce the last two words the same; you will use different tones and inflections. In English, we’re just expected to know which tones and accents to use based on the context. From childhood, we learn to recognize and use tones and accents in the same way we learn to memorize the secret rules for pronouncing words with ‘ough’ in them.<sup>17</sup> We are just deprived of a way to write our tones!

## Character Cells

As introduced in the first note of this series, *Exploring Alphabets*, many languages utilize “characters” that, in fact, are actually composed of multiple symbols intended to be displayed as if they were a single entity. Thus, the importance of a character cell – a position on a display or page that is treated as if it were one character, but may in fact “contain” more than one stored symbol – must be recognized when dealing with Multi-Lingual, Multi-Script data.

## Contextual Character Forms (Alternative Characters)

Character *Forms* should not be confused with a simple difference in the *shape* of a character such as those encountered when using different fonts. The “A” character might appear as A in one font and as A in another, but these differences are due to the font designs, and are irrelevant to text layout considerations.<sup>18</sup>

When discussing text layout, Character Forms are differences in a character based, not on a style, but on its position in a word or what its neighboring characters are. Positions are generally characterized as:

- isolated: the form used when a character is displayed by itself, and not part of a larger word.
- initial: the form used when a character is the first character in a word.
- median: the form used when a character is somewhere in the middle of the word.
- final: the form used when a character is the last character in a word.

---

16 Distinguishing between “diacritic characters” (those considered actual characters in Latin scripts) and simple “diacritics” is another case of inconsistent usage among the “experts.” For this paper, the word “diacritic” will generally be used, even for diacritics that are full alphabetic characters; for text layout purposes – complex or otherwise – the distinction isn’t relevant.

17 You’re certainly familiar with the rules for determining the correct pronunciations of *Through*, *Though*, *Tough*, *Cough*, *Hiccough*, *Bough*, *Bought*, and *Lough*. Right? Luckily, most folks use the modern spellings of hiccup and Loch.

18 ... except for glyph widths of course. The fonts are Monotype Corsiva in the first instance and Carbon Block in the second.

Not all scripts have contextual character forms, and those that do might have few or many examples. The lowercase Greek letter Sigma, for instance, generally is given the form  $\sigma$  (U+03C3), but has a final form  $\varsigma$  (U+03C2) that is (and must be) used at the end of a word.<sup>19</sup> While these are both the same character in the Greek alphabet,<sup>20</sup> each is given a separate code point, and stored in memory and on disk as a separate character. Existing technology could automatically perform such a substitution either on disk or just for display if, for instance,  $\sigma$  was followed by a space, but no purpose would be served, as the current usage is firmly entrenched. The lowercase Sigma is the only instance of a contextual character form in Greek, although Arabic Scripts make extensive use of these.

### Contextual Character Shaping (Shape Changing)

Contextual character shaping seems similar to contextual character forms, and the shaping is determined by the same character position categories (isolated, initial, median, and final). The difference is that, rather than being different characters, contextual shaping alters the character's displayed form to suit its position. The changes in shape do not, however, represent different alphabetic characters, although a computer system may display – but not store – the different shapes by using substitute symbols located in a suitable font.

In Arabic, for instance, the character that is more or less equivalent to the Latin “B” is ب (U+0628).

That “isolated form” is changed to several other forms when required: the ب character is written as ب (U+FB54) in its initial form, ب (U+FB55) in its medial form, and ب (U+FB53)<sup>21</sup> in its final form. Twenty-two of the twenty-eight Arabic letters use all four contextual character forms, but only their isolated forms are stored in memory or on disk – none of the variants are stored and all of them are used only for display.<sup>22</sup>

Another form of character alteration that looks similar to contextual character shaping is the deliberate “stretching” of some characters in scalable fonts in order to assist with justification, but this shouldn't be confused with Contextual Character Shaping, which is language-related rather than layout-related. Details of such alterations are mentioned in *Full Justification* on page 21 and in the reference given in footnote 1.

Symbols in Boustrophedon writing systems (see discussion on page 14) also make use of different forms.

### Character Reordering and Placement

Despite the previously mentioned discrimination against the lowly vowel classes in some cultures,<sup>23</sup> even vowels that are *stored* in consonant-vowel order are *displayed* in vowel-consonant order in certain scripts. The phrase हिन्दी भाषा in the *Examples for Experimentation (No knowledge of Hindi needed)* section on page 26 gives an example of this in which the second character entered, a vowel, is displayed before the first character, a consonant. In this example, the character reordering will be transparently handled by a properly designed system.

The term Character Reordering, however, is appropriately used only in cases where the entry order and storage sequence differs from the written/printed presentation. Although swapping of consonant and vowel sounds when a word is spelled correctly occurs often – as in the word ไทย illustrated and described toward the end of the section *Examples for Experimentation (No knowledge of Thai needed)* or any English word ending with “le,” (e.g. “double,” “triple,” and similar examples), these apparent exchanges are simply part of the language, and their spelling is the writer's responsibility.

In many cases, Character Reordering behavior affects how words are sorted in a given language or script; this is addressed in more detail in *Collation and Sorting* on page 22.

---

19 With Latin keyboard remapping provided by Input Methods, these are typed using the s and w keys respectively.

20 The upper case Sigma  $\Sigma$  is used for both but, given that upper case letters aren't used to end words, this isn't surprising. Many (but not all) Latin keyboards that are remapped will produce the capital  $\Sigma$  whether the capital S or W keys are used.

21 Look closely; there are two dots at the bottom rather than one.

22 The reasons for this relate to sorting, spell checking, and so forth. Still, some applications inexplicably ignore these issues.

23 See *Vowel Varieties* on page 9.

## Illegal Character Combinations

Microsoft considers handling of “illegal character combinations” to be an element of complex text layout, noting that “Since Thai syllables consist of a consonant optionally followed by one vowel and/or one tone mark, some character combinations (e.g. two vowel marks in succession) are nonsensical. Thus, one of the tasks of complex script enabling is to filter out or disallow illegal character combinations.”<sup>24</sup>

Such “help” is questionable, and seems to straddle whatever line exists between text layout implementation and auto-correct capabilities. My experience is that Thais generally consider such things to be in the same class as an English writer who uses a word processor’s auto-correct facility to alter “teh” to “the.”

## Composite Characters

The contents of character cells containing more than one symbol are generally known as composite characters, but not all composite characters are the same. The difference is that while some combinations are assembled “on the fly” for display from symbols that are stored separately on disk, others are formed by some combination of symbols for which a single preassembled replacement has been defined.

In most Latin scripts, for example, the letter ‘a’ with an acute accent (a diacritic) is both stored on disk and displayed as the single character ‘á’ regardless of whether it was entered directly on a keyboard or via some “compose key” sequence (e.g. **Compose** + **a** + **'**) or similar feature of the operating system or application.

In other scripts, characters may be stored and transmitted separately, but still displayed in one character cell.<sup>25</sup> In Thai, for example, while the consonant **๒** and vowel **ะ** are considered individual letters, each of which is stored separately on disk, they are displayed in one character cell as a composite character **ั๒**.

Some implications of these differences will be discussed in “Cursor Movement and Editing Keys” below.

## Ligatures (Composite Glyphs)

In the world outside computers, a Ligature is something used to bind or wrap two or more things together. In typography, however, it refers to two or more characters that are displayed as if they were one; the combination, which is treated as a single character cell, is not a composite character but a composite glyph. The word aesthetic could be displayed with a common a+e ligature as æsthetic, but if stored in that manner would fail a spell check. The difference, once again, is that æ is not a letter of the English alphabet, while an n+~ ligature, such as ñ, is an actual alphabetic character in the Spanish alphabet. The differences affect collation sequences as well as spell checking. Once again, some applications support stored ligatures with convoluted work-arounds.

## Dead Keys

This term originated with typewriters, and is often encountered in discussions about the entry of composite characters. Normally, when a key was typed, the platen carrying the paper moved to place the next printing position where a hammer would strike. Keys intended to place diacritics above or below another were offset so this could happen, and designed so that the platen roller and paper would not move when the key was struck. Hence, the name “Dead Key.” In the Thai keyboard segment illustrated in Fig-

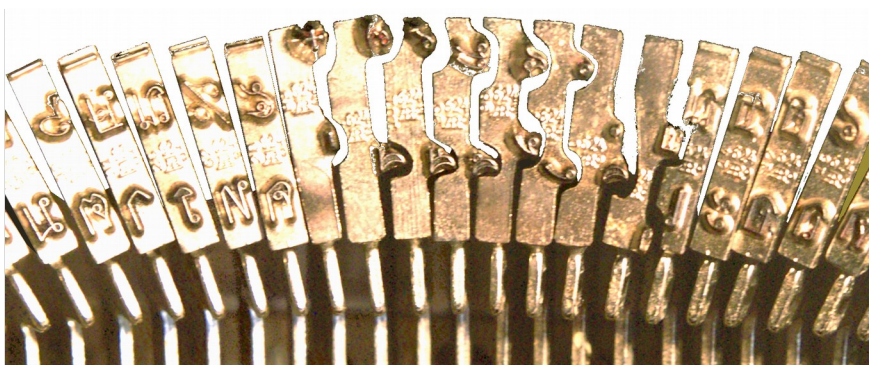


Figure 2 – Dead Keys on a manual Thai Typewriter (circa 1970)

<sup>24</sup> See the reference in footnote 5.

<sup>25</sup> In the reference cited in footnote 5, the Microsoft FAQ refers to such character cells containing multiple elements as “piles.”

ure 2, the symbols on the eight “dead key” hammers in the center are offset both horizontally and vertically to permit them to be printed above or below the character previously typed.

While in Thai and Greek, “post-fix” diacritics are typed *after* the base character as shown in Figure 2, the dead keys on European typewriters had no offset on their hammers, requiring that dead keys for these “pre-fix” diacritics need to be entered *before* the base character. Thus, software developers must accommodate yet another set of conflicting conventions to insure the resulting applications work according to local habits.

Symbols entered as Dead Keys are stored in memory either as separate individual characters and only displayed together in a cell, or stored as composite characters, replacing the individual symbols in storage as well as the display. Most Latin composite characters such as À, à, Ë, ë and the like are given discrete Unicode values for legacy reasons that are beyond the scope of this paper. The Unicode Consortium has publicly stated, however, that no more of these will be added in order to minimize redundancy,<sup>26</sup> since current technology can compose such characters for display quite flexibly and efficiently.

### Text Layout Direction (Writing Mode)

Text direction refers to the direction the symbols in a line of text are laid out, but has nothing at all to do with how the data representing that line is transmitted or stored on disk. Text direction is a characteristic of the writing system/alphabet rather than the Language, and should be determined by the Unicode block of whatever character has been entered. Text direction is also called by the vague term ‘writing mode’ and is usually classified – and not very well – as either ‘normal’ or ‘RTL.’

“Normal” typically means that a line of text is displayed or printed horizontally from left to right; RTL means that the text runs from right to left. The expected acronym LTR is seldom encountered and text direction is generally never mentioned or considered unless it is RTL. See the note about such inconsistent perceptions to the right.

Of course, some languages can also be written vertically, so we would be forgiven for expecting some acronym like TTB (top to bottom), but that doesn’t seem to be the case. Instead, such languages/scripts are referred to as ‘Asian.’ Vertical text layout is indeed usually Asian, but any suggestion that ‘Asian’ languages in general are written vertically reflects at least some level of ignorance. Scripts written from top to bottom are apparently even less “normal” than those written from right to left.

### Perceived Cultural Issues

Why is left-to-right text considered “normal” and not even given its own acronym?

And why aren’t top-to-bottom scripts given their own acronyms? Neither of these two questions has a good answer.

And why, as some wonder, do writers using English and other western scripts get the prime real estate (the seven-bit codes) at the very beginning of the Unicode block tables?

The answer to this is more a matter of technical reality than cultural insensitivity. The modern computer era began and was primarily driven by English speakers during and immediately after World War II. The result is that the world’s operating systems, kernels, command lines, programming languages, APIs and so forth continue to require what we affectionately remember as “lower ASCII” from the not-so-distant past.

This is why even the nationally sanctioned “official” fonts of most countries include Latin scripts as well as their country’s own.

Menu options and error messages can now be routinely presented in host languages without a great deal of pain using Locale definitions, but attempts over the years to translate programming language key words generally offer far too few advantages to justify the effort involved. Sharing source code across borders and dealing with differences in paired symbol usage are just a few difficulties.

Well designed internationalized applications, along with UTF-8, really insures that actual users of computers *can be* provided with all that these devices have to offer.

Most apparent ‘cultural’ issues are differences in perspective that can be and are being accommodated through better generalization, although many such issues still remain.

<sup>26</sup> One simple reason is that if one has five base characters that can accept any of five diacritics, a set of additional composite characters would total 5\*5, or 25, whereas if the characters can be assembled for display as needed, no additional characters would be needed but the basic 5+5, or 10. Obviously, this calculation can get more interesting, but you get the idea.

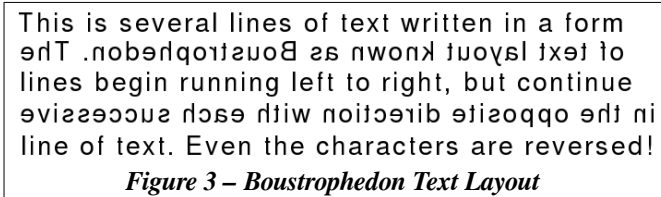
Given the current state of the art, there should be no need whatever for a user to explicitly select right-to-left or any other direction; directionality is a characteristic of a particular script and can easily be determined from the Unicode block of any alphabetic character entered. If desirable, a user should of course be able to alter the direction.

In order to present a more rational approach to discussing text direction in the examples that follow, several new acronyms will be introduced.<sup>27</sup> These are simply:

- **LRTB:** Left-to-Right; Top-to-Bottom. Examples include almost all European and Southeast Asian scripts. This layout is becoming a common alternative used with many vertical scripts as well.
- **RLTB:** Right-to-Left; Top-to-Bottom. Examples are middle-eastern scripts such as Hebrew and Arabic. RTL layouts, by the way, preceded LRTB layouts in history, giving them a prior claim to ‘Normal!’
- **TBRL:** Top-to-Bottom; Right-to-Left. This acronym reflects a change in precedence from horizontal to vertical; examples of traditional TBRL scripts include many Chinese, Japanese, and Korean scripts, although not all scripts used by even those languages are laid out vertically. Horizontal left-to-right layouts are becoming ever more common with globalization.
- **TBLR:** Top-to-Bottom; Left-to-Right. An example of a script using a TBLR layout is Mongolian.

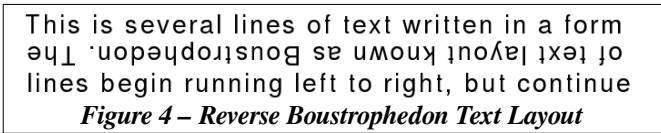
Other interesting text layout directions include Boustrophedon, Reverse Boustrophedon, and Spiral, none of which are used in any contemporary language unless intended as decorative elements.

Boustrophedon writing examples are horizontal, but change direction as each line is presented, reversing the character shapes as well. Actual Boustrophedon writing isn’t normally used with the Latin alphabet as shown on the right, but that seems the best way to illustrate it for non-archeologists who combine the Boustrophedon layout with often obscure character sets to write scholarly articles about – what else? – ancient Boustrophedon writings in Safaitic, Sabaean, early Greek & Latin. The undeciphered Rongorongo inscriptions found on Easter Island are another, but unrelated, example of a Boustrophedon layout.



*Figure 3 – Boustrophedon Text Layout*

Reverse Boustrophedon, simulated on the right, is quite similar, but the characters in the alternating lines of text are inverted instead of reversed. The etymology of the term Boustrophedon derives from the alternating path taken by an ox while plowing a field. No acronym is used for Boustrophedon, since it isn’t now, and will not likely ever be, supported by anything other than graphics software.



*Figure 4 – Reverse Boustrophedon Text Layout*

Spiral layouts are also used in the Linear-A syllabic script used in at least one dead language – believed most likely to be Hittite, or an early form of one of the Semitic or Greek languages. The Phaestos disk, illustrated on the right, is the primary example of such writing although, since it has never been convincingly translated, some scholars dispute whether it is read from the inside out – the obvious assumption – or from the outside in. Spiral layouts also aren’t given their own acronym, although it’s tempting to use SPRL, since the final RL will introduce the same sort of confusion promoted by the common pairing of RTL with CTL, rather than LTR.



Interestingly, none of the extant examples of spiral layout seem to have any remaining space. Many assume the ancients simply stopped writing when they ran out of space, but I prefer to think of these as very early examples of full justification.

<sup>27</sup> New obscure acronyms, after all, are a traditional means by which technologists simulate progress and pretend to innovate!

## Mixed Text Directions – Bidirectional Text

Although many documents are written with just one language and, therefore, one script and text direction, any modern system should be able to transparently support the use of multiple languages without requiring user intervention. When mixing text segments having different directionality in the same document however, a number of interesting issues arise. These issues – and their typical solutions – depend on a variety of factors which are discussed (again informally) in this section.

### Default Paragraph Directionality (Primary Text Direction) in Text with Mixed Directions

Even in the case of side-by-side or interleaved translations of one language to another, or commentary in one language regarding text in another, it is usually the case that a document will be written with one language being considered as the primary. This language is known as the Document Default Language, and the direction of the script used for that language is then considered the Paragraph Default. At least conceptually, the primary, or document default, language is not at all related to the default language of the operating system, the application, or any other user interface in use, although these distinctions are commonly blurred in applications designed without multilingual use in mind.

It is more practical, however, to ignore the document and consider each paragraph to have its own primary language and, therefore, default direction, since that offers the most granular control of text layout. The default Paragraph Direction will of course depend on the document's default language.

Commonly used applications vary considerably in their handling of mixed Directionality, and we'll discuss that subject in later Notes in this series but, for now, it is sufficient that you are aware of that.

The most interesting circumstance is the pairing of horizontal and vertical text segments, shown in the band on the right side of the page, where the primary paragraph language is Chinese. Sections of the normally horizontal English text that are included will also be laid out vertically, usually with the characters rotated as shown here, although this can vary depending on the actual content. Thus, LRTB segments are transformed into simply TB segments. If the non-Chinese text happened to be a normally RLTB language such as Arabic, it would by the same convention, be arranged as simply BT. (The Chinese text is from the opening of the United Nations Universal Declaration of Human Rights. See Footnote 3.) Chinese and Han Script will be discussed in a separate Design Note 7.

To say that there are technical difficulties with mixing horizontal and vertical text, however, is a bit simplistic and misleading. The primary difficulty is determining what any such technical solution ought to accomplish – what a resulting mixture should look like on paper in order to make sense for an average reader. For this and various other reasons, the use of vertically oriented text layouts is gradually disappearing, with the Chinese government itself driving that change as early as 1956.

Page margins generally remain the same where scripts having multiple directions are mixed, unless there is some specific design need, which is usually unrelated to simply LRTB-RLTB considerations.

### Cursor Movement when Entering or Editing Text in Bidirectional Paragraphs

If handled incorrectly by an application, cursor movement or character selection with a mouse during bidirectional text entry can be quite disconcerting to a user. Confusing cursor behavior during text entry is most often the result of incorrectly identifying transitions from one script to another. An example of this is provided in the section titled *Typing “This is English and שפה עברית is Hebrew” – a step-by-step illustration* in Design Note #5, *Exploring Bidirectional Text Entry*. Transitions between different text layout directions can also occur in paragraphs in which only a single language is used; many right-to-left languages display numeric characters from left-to-right. An example of how this is handled is provided in Figure 9 – Right-to-Left Ruler with Mirrored Tab Settings and its accompanying explanation on page 19.

The table below summarizes the actions of various keys during cursor movement, character entry, and editing in paragraphs with different default text layout directions:

人人生而自由，在尊严和权利上一律平等。LATIN IS ROTATED IN PREDOMINANTLY VERTICAL TEXT BLOCKS.



### Cursor Movement Keys





Home Key	in LRTB Layouts	The Home key will move the cursor to the beginning of the line, which is the left-most position.
	in RLTB Layouts	The Home key will also move the cursor to the beginning of the line, which is the right-most position.
End Key	in LRTB Layouts	The End key will move the cursor to the end of the line, which is the right-most position.
	in RLTB Layouts	The End key will also move the cursor to the end of the line, which is the left-most position.
→ Key (Forward Key)	in LRTB Layouts	The → key will move the cursor right (in the direction indicated by the arrow) to the next character cell.
	in RLTB Layouts	The → key will also move the cursor to the next character cell, but in the case of right-to-left scripts, that character cell is on the left.
← Key (Reverse Key)	in LRTB Layouts	The ← key will move the cursor left (in the direction indicated by the arrow) to the previous character cell.
	in RLTB Layouts	The ← key will also move the cursor to the previous character cell, but in the case of right-to-left scripts, that character cell is on the right.

### Character Entry and Editing Keys

← Backspace Key	in LRTB Layouts	The ← Backspace key will delete the character to the left of the cursor, the direction indicated by the arrow.
	in RLTB Layouts	In spite of the left-facing arrow on the Backspace key of many keyboards, this will delete the character to the right of the cursor.
Delete Key	in LRTB Layouts	The Delete key will delete the character to the right of the cursor.
	in RLTB Layouts	The Delete key will delete the character to the left of the cursor.
Tab→ Key (Forward)	in LRTB Layouts	During text entry or editing, the Tab→ key will insert a usually invisible character that will move the cursor to a character cell beginning at the next tab stop to the right. The Tab→ key is also used in some applications to move the cursor to the “left” or “next” cell in layout structures such as tables.
	in RLTB Layouts	During text entry or editing, the Tab→ key will usually do nothing, but it is used in some applications to move the cursor to the “right” or “previous” cell in layout structures such as tables.
← Tab Key (Back)	in LRTB Layouts	During text entry or editing, the ← Tab key will usually do nothing, but it is used in some applications to move the cursor to the “previous” cell in layout structures such as tables.




 Key (Back) ... continued	in RLTB Layouts	The  key will insert a usually invisible character that will move the cursor to a character cell beginning at the next tab stop to the right. See the comments below regarding tab stops in documents having bidirectional layouts.
---	-----------------	--

**Comments:** The motion of the  and  cursor keys seems particularly contrary unless they are referred to by names similar to the  and  keys seen on media players; such symbols can only be considered “intuitive” in the Microsoft sense of the word – which is to say it’s been done that way for so long users’ habits have become stratified. That’s just how it’s done. Implementation of Tab behavior in right-to-left text, like that of the cursor behavior described above, varies considerably across applications. See *Rulers, Guides, and Tabs in Text with Mixed Directions* on page 18 for illustrations.

### Paired Symbols in Text with Mixed Directions

Most written languages use a variety of paired delimiter symbols, such as parentheses, braces, brackets, and quotation marks. With parentheses – to use one such pair as an example – the first of the symbol in left-to-right scripts is typically referred to as either an ‘opening parenthesis’ or a ‘left parenthesis.’ The second of the pair is correspondingly known as a ‘right parenthesis’ or a ‘closing parenthesis.’<sup>28</sup> Other paired symbols are similarly named. Most scripts, whether left-to-right or right-to-left, use identical Unicode characters for the ‘opening’ parenthesis (U+0028) and ‘closing’ parenthesis (U+0029) regardless of their position on the keyboard.

Such seemingly minor variations in naming reflect the sorts of perspective differences that a user needs to be aware of when mixing scripts within a single document, particularly when Input Method Editors are used to dynamically switch keyboard layouts. Three common pairs of such delimiters – the Parentheses, Curly Brackets, and Square Brackets – are shown in Figure 6 as they are laid out on the upper right of a typical “western” keyboard layout.<sup>29</sup> In the case of a left-to-right layout such as this, whether the phrase “left parenthesis” or “opening parenthesis” is used makes no difference when describing the  key.

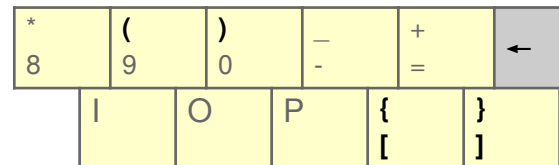






Figure 6 – English (US) Keyboard Layout Segment

So what about right-to-left scripts such as Hebrew or Arabic? As can be seen in the equivalent keyboard segment on the right, the Hebrew perspective is that the key combination of  +  is likewise used as an opening parenthesis, and the  +  as the closing parenthesis, regardless of what the symbol looks like. Thus, when mixing Latin and Hebrew scripts by switching keyboards, an identical perspective applies. The same perspective carries through to the other paired characters on the Hebrew layout, including the ‘<’ and ‘>’ symbols that are not shown in the illustration.

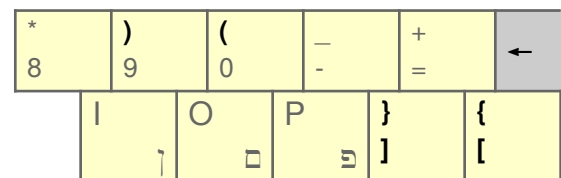


Figure 7 – Hebrew Keyboard Layout Segment

This difference in perspective is not reflected in all right-to-left language keyboards, however; many, though not all, Arabic keyboards have the paired delimiters in the same locations as on Latin keyboards.<sup>30</sup> This perhaps reflects the wide variety of languages that use the Arabic script. Choose your keyboard layouts wisely! More details on using bidirectional data with paired delimiters will be presented in Design Note #5, *Exploring Bidirectional Text Entry*.

<sup>28</sup> The Unicode Consortium originally used the names ‘opening parenthesis’ for the ‘(’ character U+0028 and ‘closing parenthesis’ for the ‘)’ character U+0029, but later adopted the more neutral ‘left’ and ‘right’ terms as the official names, leaving ‘opening’ and ‘closing’ as alternate names.

<sup>29</sup> Many languages also share these and other paired symbols, but place them in different keyboard locations.

<sup>30</sup> Both forms can be found on Amazon, for instance, but there is no mention in their descriptions that this difference exists.

Quotation marks, another common symbol pair, appear in many interesting varieties. What the French call *guillemets* (« and »), are used in many languages.<sup>31</sup> Danish and Hungarian writing use the same symbols, but in reverse; the » marks the beginning of a quotation and the « indicates the ending. Similar, though not identical symbols (《 and 》) are used in Korean and simplified Chinese, but are different code points.<sup>32</sup>

In order to avoid headaches, we’ll ignore any discussion of how paired character matching algorithms<sup>33</sup> are affected by the different open-close-left-right perspectives. And we certainly don’t want to consider what implications there might be of exchanging the / and \ keys, which is left for the reader to ponder.

### Rulers, Guides, and Tabs in Text with Mixed Directions

Perspective in the treatment of ruler displays, guides, tab stops, line indents, and the like is somewhat analogous to that of paired parentheses, where the L symbol may be viewed as representing a “left tab stop” or a “forward tab stop,” i.e. one continuing the left-to-right motion of the text being displayed. In a right-to-left paragraph, however, it is the J symbol or a “right tab stop” that represents forward motion.

Common conventions for tab stop symbols should help: a user can view the vertical bar on the tab marker as the location where placement of character cells will resume after the Tab key is pressed. The bottom right angle “hook” should always point in the direction that character cells will be laid out from that point.

Unfortunately, many applications treat these various guides inconsistently. Some consider them as Page-centric rather than Paragraph-centric or, even worse, as being dependent on the installation’s user-interface language; many applications require elaborate configurations to handle bi-directional text on a paragraph basis, and some simply ignore the issues involved with handling multiple text directions. Documentation for many products will often provide clues to the level of support for mixing text directions.<sup>34</sup>

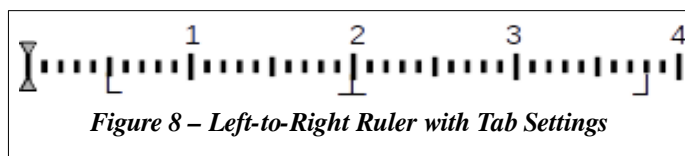
Before discussing decimal tabs, which have their own unique issues, we should begin with the more common left, right, and center tabs, since proper handling of these seems to be rather straightforward.

#### Left, Right, and Center Tab Stops

Assume that the Default Language of a document uses a left-to-right Script; this implies that, unless explicitly changed by a user, the default paragraph direction is also left-to-right. For this example, assume also that the default paragraph style uses a customized set of tab stops. The following illustrates how these tab stops would be mirrored as defaults for any independent right-to-left paragraphs in that document:

The user’s defined tab settings are shown in Figure 8, and listed in tab setting dialogs as follows:

- 0.50" Left      Text continues *Forward* (right) from this point.
- 2.00" Centered      Text spreads in both directions around this point.
- 3.80" Right      Text continues in *Reverse* (left) from this point.

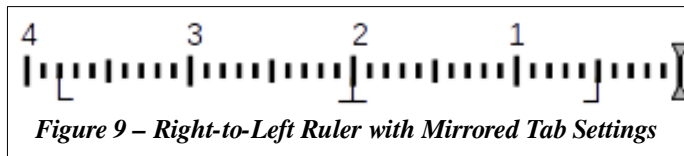


These tab settings are, of course, measured from the left margin – the LRTB point of reference.

31 Including Armenian, Azerbaijani, Basque, Belarusian, Catalan, Swiss, German, Greek, Italian, Latvian, Norwegian, Persian, Portuguese, Russian, Spanish and Ukrainian. In Finnish and Swedish, the » is sometimes used to open and close a phrase.  
 32 The Unicode values for these are « (U+00AB), » (U+00BB), 《 (U+300A), 》 (U+300B). The latter two are ‘full width’ symbols to match the fixed width of vertical columns of symbols; when primarily written horizontally, symbol use is more flexible.  
 33 e.g. where placing the cursor on one symbol of a pair in many programming editors causes its twin to be indicated in some fashion.  
 34 Although LibreOffice, for example, supports multiple interface languages, it has only minimal support for mixing text directions. The help in its Writer component says, for instance, “Initially the default tabs are shown on the horizontal ruler. Once you set a tab, only the default tabs to the right of the tab that you have set are available.” Use of the phrase “to the right of the tab” is a clear indication that Writer assumes a predominantly left-to-right world as many applications do.

Where the Default Paragraph Direction is right-to-left in a document that is primarily left-to-right, the Tab Settings would be mirrored, as seen here:

- 0.50" Right Text continues *Forward* (left) from this point.
- 2.00" Centered Text spreads in both directions around this point.
- 3.80" Left Text continues in *Reverse* (right) from this point.



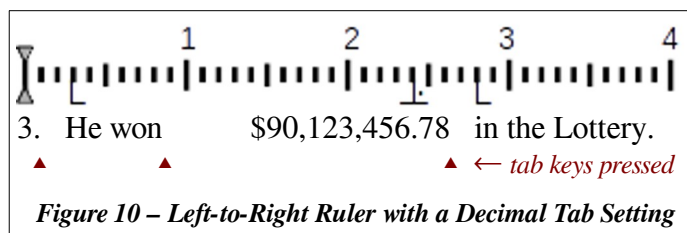
With right-to-left “exception” paragraphs in this example document, the default tab direction settings should be reversed, and measured from the right margin. The tab at 0.5" remains a “forward tab,” but in this case needs to be transformed to a “right tab” in order to correctly mirror the left-to-right settings.<sup>35</sup>

It must be noted that some mainstream applications consider Tabs within text to be white space (which is correct), but also interpret such Tabs as Latin characters (which they are not) due to their 0x09 code point.

### Decimal Tab Stops

Because numbers are generally displayed from left-to-right even in languages that are otherwise laid out in the opposite direction, mirroring of tab settings presents more interesting challenges when those settings include decimal tab stops. Figure 10 illustrates a left-to-right layout with three tab stops:

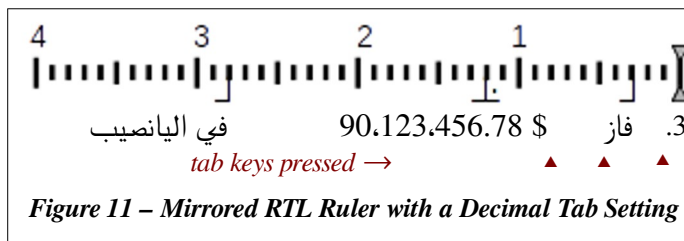
- 0.30" Left Text continues *Forward* (right) from this point.
- 2.40" Decimal Digits continue in *Reverse* (left) from this point until the decimal point is entered, after which the digits continue *Forward* (right) until the number entry has been completed.
- 2.80" Left Text continues *Forward* (right) from this point.



The small red triangles indicate the points at which the user pressed the **Tab** key while typing the phrase. After “3.” was typed, the tab key moved the cursor to the 2.40" position and the numbers were entered right-to-left until the decimal key restored its normal left-to-right direction. When the tab after the final “8” was typed, the cursor moved to the 2.80" position, at which point the user typed “in ...,” etc.

Transformation of decimal tab settings for use in mirrored right-to-left paragraphs is a bit more complex than the simple swap and direction reversal described above. As with the earlier example, the 0.3" and 2.8" tabs are simply converted from Left Tabs to Right Tabs but, in this example, the decimal tab value for the RLTB Arabic text would be converted from 2.40" to perhaps 1.20" as shown in Figure 11 below.

- 0.30" Right Text continues *Forward* (left) from this point.
- 1.20" Decimal Digits continue in *Forward* (left) from this point until the decimal point is entered, after which the digits continue *Reversed* (right) until the number entry has been completed with the “\$” entry.
- 2.80" Right Text continues *Forward* (left) from this point.



Assuming that the values expected to be displayed in the mirrored layouts are similar in terms of the quantity of digits expected before and after the decimal point,<sup>36</sup> the relative distance between the tab stop preceding the decimal location and that following it need to be swapped, but things aren’t that simple. In the left-to-right example, the larger distance we wish to swap from one side to the other is not the distance from the 0.3" tab to the 2.4" tab, but rather the average distance between the endings of whatever text begins at the 0.3" tab to the 2.4" tab. In Figure 10, this would be from about 0.85" to the 2.4" tab stop.

<sup>35</sup> The user may need to alter these settings for various reasons, or in specific paragraphs, but in many cases, a well-designed interface will preclude such a need.

<sup>36</sup> Currency, as shown here, assumes a greater possible number of integer units than decimal units, but the same reasoning would apply for different value types. Remember too that there are a variety of locale-dependent delimiters used in numbers (not applicable in this example), and a variety of currency indicator placements (seen here with the \$ being placed after the value).

Without delving into how this is calculated (because, after all, such an arbitrary result can only serve as a useful starting point that will likely need to be fine tuned by a user depending on a particular combination of languages, font sizes, and other factors), we can refer to the original 0.3", 2.4" and 2.8" tabs respectively as  $\alpha$ ,  $\delta$ , and  $\omega$ . The mirrored decimal tab would then be set at  $2.25 \times (\omega - \delta) + \alpha$ , or, in this case 1.20".

More precise layouts would likely be handled by tables or grids that have been available in most word processors for decades. Note that similar mirroring considerations also apply to bullets, numbering, etc.

### Detecting Primary Text Direction in Paragraphs

In dedicated multilingual applications, a convention evolved at least twenty-five years ago that the Primary Text Direction for a given paragraph should be determined by the initial character of the paragraph, but the meaning of “the initial character,” which meant quite narrowly the first *alphabetic* character, was often lost when that convention was eventually adopted in more general purpose editors and word processors.

The term “alphabetic character” does *not* include shared values (e.g. common punctuation as well as the space and tab characters) in what is still called the lower ASCII range. A good example of this can be seen in Figure 11, where the first “character” (the numeric character “3”) is shared across many scripts. The paragraph, clearly intended as right-to-left Arabic, is mistakenly set as left-to-right by several applications when such a decision should be deferred until an actual alphabetic character is encountered.

### Text Alignment in Documents with Mixed Directions

A common default for the beginning of text lines in left-to-right layouts such as those shown in Figures 8 and 10 is the left margin. This is reflected in, and often controlled by, icons such as those shown in Figure 12 on the right.

Unlike the settings for tab stops, such choices are not subject to differences in perspective; left and right, after all, have the same interpretation regardless of language or culture. Furthermore, two of the alignment choices shown in Figure 13 would be interpreted the same regardless of perspective.

What is often overlooked in some application interfaces is how paragraph alignment *defaults* should be handled if the primary text direction of a paragraph differs from the document’s primary text direction. When such a change is detected, a user should reasonably expect the text alignment in use to be mirrored as well. If the default paragraph direction is left-to-right, for instance, and the default alignment for that paragraph is left, the alignment should be mirrored (i.e. set to be right-aligned) when the paragraph direction is changed.



Figure 12 – Left Align

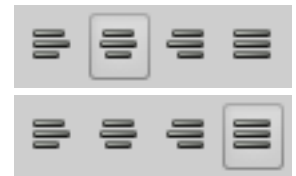


Figure 13 – Bidirectional

### Transitioning between Text Directions within Paragraphs

Smooth transitions for users entering text in bidirectional paragraphs can be disrupted when the default Script (and therefore directionality) of the current paragraph is detected improperly. A detailed example of how this commonly occurs is presented in Design Note #5, *Exploring Bidirectional Text Entry*.

### Justification

Justification describes not the direction in which the text is laid out, but how it relates to the page margins. The choices given for justification in most contemporary applications are Left, Center, Right, and Full, as shown in Figure 12, but these terms actually represent two general classes: Ragged and Full.

### Ragged Justification

The term Justification is often used rather loosely to indicate which margin any lines of displayed text are aligned against. Most lines of text in this document, for instance, begin at the left margin and, while such a layout is colloquially known as “left justification,” it is more correctly known as “flush left” and/or “ragged right” – ragged because the

lines of text can be and often are of different lengths. Text may be and is often set either “ragged left” or “ragged right” regardless of a paragraph’s default layout direction.

### Full Justification

Justification, in its more formal use, is altering a line of text in order to extend it so that the lines in a paragraph are laid out to align evenly (or appear to)<sup>37</sup> against both margins. The exception to this is that the last line in a paragraph will not be altered in cases where it is short enough to produce humorous results if so extended.

There are several techniques used by software to justify text. The simplest is to stretch the spaces between words to extend the line length sufficiently to meet the ending margin. Slightly more pleasing<sup>38</sup> results can be obtained by also adding slight increases to the space between the displayed character *cells* – but not between *characters* as some sources incorrectly suggest. This is one example why distinguishing between characters and character cells is important. Spaces that would otherwise appear at the beginning or ending of lines then need to be suppressed in any justified output.

Justification of lines in Thai sentences, incidentally, must always be accomplished by adjusting the spacing between letters, since Thai doesn’t separate words with spaces. Spaces are only used to separate sentences from each other.<sup>39</sup>

Another technique involves subtle<sup>40</sup> ‘stretching’ or ‘shrinking’ the widths of individual character cell contents (selective contextual shaping) to minimize the need for extending space widths excessively.

The best results when applying full justification are obtained when the paragraph is considered as a whole, rather than simply on a line by line basis. Further discussion is out of scope for a document like this, but searches for the following justification algorithms are recommended for those with an interest:

- Knuth and Plass: 1981; This is the same Donald Knuth any good programmer should already know.
- Hochuli and Kinross: 1996;
- Hàn Thé Thành: 1999; See footnote 37 for one interesting link.
- Haralambous and Bella: 2006;
- Elyaakoubi and Lazrek: 2010;<sup>41</sup>

### Kashideh

An interesting alternate technique for obtaining full justification is known by the term Kashideh (کشیده) which means “extended” or “stretched” in Persian. Kashideh layout can be used for justification in a variety of languages that use Arabic scripts, including Persian, Urdu, Pashto, and Jawi, as well as with Devanagari scripts in languages like Hindi, Sanskrit, Bengali, Nepali, etc. See Figure 16 – Arabic, before and after Kashideh Justification on page 28 for a detailed comparison of ragged right and Kashideh justification.

Perhaps this form of justification does qualify as “complex,” although “elegant” would seem to be a better term.

### Word Breaks, Line Breaks, and Hyphenation

Although the majority of contemporary scripts use spaces between words, and at least some form of punctuation to delimit sentences, some do not. As mentioned earlier, for example, Thai uses no spaces between words. This can be seen in Figure 14 – Thai, without and with full Justification on page 25. Observe that the first space is only seen between the words ๓๓๓ and ๓๓ about two-thirds of the way into the first line, which is the break between the first two

---

37 There is actually a difference between having character edges aligned against the margin and merely “appearing” to be aligned against the margin edges; see <http://www.tug.org/TUGboat/Articles/tb25-1/thanh.pdf> for a discussion.

38 This is of course a subjective term, but is based on centuries of typography traditions across many cultures and technologies.

39 See Figure 14 – Thai, without and with full Justification on page 25 for an example.

40 There are, of course, often some “not-so-subtle” differences in various designers’ interpretations of “subtle.”

41 See <http://quod.lib.umich.edu/j/jep/3336451.0013.105?view=text;rgn=main> to download their article in the Journal of Electronic Publishing (v13#1).

sentences.<sup>42</sup> The detection of the line break causing the large space at the end of the first line occurs after the whole word မှီ, since the following word ကာမ won't fit on the line.

The mechanisms by which this occurs, while considered by some to be a Complex Text Layout function, are beyond the scope of this document, but in most modern systems are provided by lower level services and not by higher level applications such as word processors.

## Collation and Sorting

Collation and Sorting is determined for the primary user interface by the Locale setting but, as noted earlier, well-designed applications make no assumption that such choices have any inherent relation to the choice of primary language in any particular document other than perhaps setting a default starting point for new documents.

Collation and Sorting is only tangentially related to what applications refer to as Complex Text Layout, and is more properly in the Language or possibly the Script domain, but the sorting orders in many languages don't always follow what is commonly called "alphabetical order."

Depending on the language, Composite Characters may or may not be considered when sorting. In German, the displayed character ä, although stored as an independent character, is considered simply a variant of the base character a (i.e. an a with a diacritic umlaut) for purposes of sorting. The ä in Swedish, however, is actually the second last letter in the Swedish alphabet between å and ö (none of which therefore are composite characters) and sorted accordingly.

In English, vowels are distributed (scattered?) among the consonants in the alphabet, with a, e, i, o and u being the 1st, 5th, 9th, 15th, and 21st characters respectively. In other languages the alphabetic vowels – whether actual characters or dead key diacritics – are considered a separate sequence. In Hindi, the Vowel order (अ आ इ ई उ ऊ ऋ ए ऐ ओ औ) is distinct from and comes before the Consonant order (क ख ग घ ङ च छ ज झ ञ ट ठ ड ढ ण त थ द ध न प फ ब भ म य र ल व श ष स and ह) and the sorting reflects this.<sup>43</sup> In Thai, the situation is reversed; the vowels (๑ ๒ ๓ ๔ ๕ ๖ ๗ ๘ ๙ ๐ ๑ ๒ ๓ ๔ ๕ ๖ ๗ ๘ ๙ ๐) are grouped together at the end of the alphabet, but the sort order considers each vowel after every individual consonant.<sup>44</sup>

In Korean, the sort order is based on the Jamo components (more or less equivalent to what we call "letters")<sup>45</sup> within each displayed "syllable." The exact order for these, though, differs between North and South Korea.

## A FINAL REMINDER

At the beginning, we stated "This document is meant to be an informal – and therefore sometimes less than precise – introduction ..." etc. A much more detailed discussion of all the subjects, Scripts, and Languages presented here is provided in the Unicode Standard (version 10, dated June 2017). The 1,044 page pdf version of this standard can be downloaded from:

<http://www.unicode.org/versions/Unicode12.0.0/UnicodeStandard-12.0.pdf>.

With the background and examples provided in this introduction, the Unicode Standard and its various other related publications should provide any additional detail you may need for whatever specific Languages and Scripts you may need to support in order to handle the data for which you are responsible.

The next section will present examples of CTL attributes in the context of actual scripts and languages in order to offer a better "feel" for the variety of behavior you will encounter when dealing with different Scripts.

<sup>42</sup> For the curious, the word ကာမ that appears at the beginning of all three sentences in this selection translates to "we."

<sup>43</sup> It's not quite that simple in practice, but this document is intended only as an introduction.

<sup>44</sup> Thai sorting is even more interesting in practice, but you get the idea. Note that the intermingling of both full character and the upper and lower diacritic vowels indicates that the only difference between them in Thai is display placement.

<sup>45</sup> See the section titled "Korean Script examples using 한국어 ( the Korean Language )" beginning on page 30 for a more complete explanation.

## CTL EXAMPLES IN PRACTICE

### Disclaimer

The examples provided in this document are not intended as a comprehensive means of testing of what are inexplicably known as Complex Text Layout capabilities, but are merely intended as a means to determine if such capabilities exist in a particular environment and how well they are supported. The primary intent is to permit those with little or no knowledge of the languages requiring such capabilities to explore how and under what conditions those capabilities function when working properly.

Of course, it is necessary to insure that a font containing the required character glyphs is present on whatever system is being tested. Since both contemporary operating systems and applications often perform both font substitution as well as glyph substitution – often without notification, warning or any apparent rationale, care must be taken to select an appropriate font for these examples so that CTL capabilities rather than font or glyph substitution capabilities are being tested.<sup>46</sup> If these tests are run within an application, any CTL processing should be turned OFF to begin with.

It should also be noted that all comments refer to Unicode used with UTF-8<sup>47</sup> operating systems and applications; except for very arcane usage requirements, this combination should be viewed as “standard” for modern systems.

There are a number of ways to enter these characters. If there is an “input method” active (such as iBus), and configured for Thai using the keyboard layout specified, the character combinations can be typed on a Latin keyboard using the keys shown immediately below each sample. The “ñ” in column 1, for instance, is typed as “[bj]”. This, of course, requires that the active font not only contains the Thai script in the correct Unicode block, but reports that properly via its internal tables (see footnote 46). The two lines below the Latin equivalent keys indicates the Unicode values and their decimal equivalents, and can be used as described in the *Character Entry Methods* section that begins on page 35.

### Common Examples of Mixing Numeric Scripts



**Thai Script**, discussed on page 24, has its own numeric characters. In the Thai 50 Baht note on the left, the denomination is shown with the Thai numerals ๕๐ on the lower left and Latin/Arabic numerals 50 on the top left.

The serial number in Thai Script [๓ ๗ ๑๑๑๕๐๕๖] is on the upper left, while the Latin version [3 A 7178086] is on the lower right.

Note that the Thai ñ and Latin A are the first letters of their respective alphabets. They are not, however, equivalent characters, the ñ being a consonant with a “g” sound.

**Devanagari Script** discussed on page 26 has its own numeric characters as well. In the Nepalese 500 Rupee note on the right, the denomination is shown with the numerals ५०० on the lower left and 500 on the lower right. The serial number, ३२६३३२, is given only in Devanagari numerals.

The Hindi Language of India also uses Devanagari Script but, since 1949 the Indian Rupee has not used Devanagari numerals on its currency.<sup>48</sup>



46 See the document “Evaluating Fonts for use in Multi-Lingual Documents” for more detailed information on choosing an appropriate font based on the particular Scripts to be intermingled in any particular document.

47 See the document “Exploring UTF-8” (available at [www.AntikytheraPubs.com/i18n.htm](http://www.AntikytheraPubs.com/i18n.htm)) for details of UTF representations.

48 Well, not officially. See <https://thewire.in/81737/madras-hc-questions-legality-of-devanagari-script-on-rs-2000-notes/> et.al.

# THAI SCRIPT EXAMPLES USING ภาษาไทย ( THE THAI LANGUAGE )

## Brief Comments about Thai

In addition to the Thai language itself, Thai script is also used for Pali, some versions of Sanskrit, and other minority languages.

The Thai language alphabetic characters, as well as its diacritic vowels and tone markings, are defined in Unicode Block U+0E00-0E7F under Southeast Asian Scripts.<sup>49</sup> Thai is written from left-to-right, and top-to-bottom (LRTB), and there are no “capital” or small” letters; the Shift key is used in Thai to type different characters.

Thai uses no spaces between individual words, but its syllables are constructed in a precise enough fashion that identification of word and syllable breaks is fairly straightforward, making hyphenation and line breaking equally straightforward. Spaces delimit sentences in Thai, and Thai characters are not generally connected to each other in print unless the intent is decorative, such as with posters, general advertising, or similar uses.

Thai has its own set of digits – the 0 to 9 equivalents are: ๐ ๑ ๒ ๓ ๔ ๕ ๖ ๗ ๘ ๙ – but generally uses the shared numerals defined in the Latin code block (see Thai currency example on previous page). On computer keyboards with separate numeric keypads, the Thai convention is that the keys on the main area of the keyboard produce the Thai digits ๐ to ๙, while those on the numeric keypad produce the “western/European/Indian/Arabic” characters 0 to 9.

## Examples for Experimentation (No knowledge of Thai needed)

The recommended keyboard mapping for these examples is that defined by TIS-820.2538,<sup>50</sup> although the Kedmanee keyboard layout is very similar and can be used if TIS-820 is not available. Latin key presses used with the TIS-820 layout are shown directly below the Thai characters; the following lines provide the Unicode hexadecimal values and their decimal equivalents for use with single character entry methods.

In the center two columns (4 and 5) of this table are two similar-looking Thai consonants (๒ and ๓) that school children call Baw-Baimai (Baimai means leaf in English) and Bpaw Bplah (fish); the only difference in appearance between these characters is that the right side extends higher in the ๒ than in the ๓.

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8
Vowel + Tone <sup>51</sup>	Mai-ek Tone	“i” vowel	Baw (leaf)	Bpaw (fish)	“i” vowel	Mai-ek Tone	Vowel + Tone
บ๋	บ่	บิ	บ	ป	ปิ	ป่	ป๋
[ b j	[ j	[ b	[	x	x b	x j	x b j
U+0E1A U+0E34 U+0E48	U+0E1A U+0E48	U+0E1A U+0E34	U+0E1A	U+0E1B	U+0E1B U+0E34	U+0E1B U+0E48	U+0E1B U+0E34 U+0E48
3610d, 3636d, 3656d	3610d, 3656d	3610d, 3636d	3610d	3611d	3611d, 3636d	3611d, 3656d	3611d, 3636d, 3656d

Progressing outward from the center, the ๒ and ๓ characters appear again in columns 3 and 6 with a diacritic vowel <sup>๑</sup>, one of several that are positioned above their associated consonant. Note that in column 3, the vowel is in its normal position. In column 6, however, it is shifted to the left to accommodate the ๓ character’s right side ascender.<sup>52</sup>

49 See <http://www.unicode.org/charts/PDF/U0E00.pdf>; a full index of Unicode charts is at <http://www.unicode.org/charts/>.

50 2538 is the Thai year corresponding to 1995 in most calendars. See this keyboard layout on page 37.

51 This is an example of diacritic ordering or ranking. In Thai, a diacritic vowel “outranks” a tone mark, and is thus placed closer to the base character. Very few applications correct such entry errors.

52 As a matter of interest, the word ปิ in column 3 happens to mean “business” in Thai, but the other examples in this table are not actual Thai words; they are simply used to illustrate flexible diacritic positioning – considered a CTL characteristic.

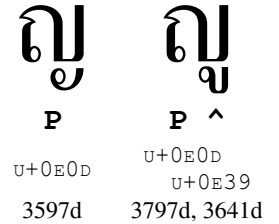
Examples in the Thai Language - ภาษาไทย



In columns 2 and 7 respectively, the Mai-ek <sup>๑</sup>, one of four diacritic tone indicators used in Thai, has been placed above each letter. Again, because of the right side ascender of the ๗ character, the Mai-ek tone mark in column 7 needs to be shifted to the left relative to the position shown in column 2.

Columns 1 and 8 illustrate the correct placement of the vowel and tone mark diacritics when they are used together. Diacritic vowels in Thai “outrank” tone marks and so remain where they were shown in columns 3 and 6, but the tone mark needs to be placed higher than its normal position. Thus, tone marks in Thai can appear in any of four distinct positions depending on the circumstances.

Thai also has some other diacritic vowels that are placed below their related consonants, such as the vowel ๑ in the second word (๗๓ – the 4<sup>th</sup> and 5<sup>th</sup> character cells) highlighted in Figure 14 below. In the illustration to the right, a similar vowel ๑ is shown paired with the Yaw consonant ๗ to illustrate another interesting layout convention. The ๗๑ character itself appears to be composite, since its lower element is separate,<sup>53</sup> but it isn’t. When combined with the lower diacritic vowel ๑ however, the lower element is simply replaced in the cell.



In modern operating systems, the Thai contextual diacritic placements and shape changing described in this section are provided by the operating system or input method based on data contained in the font itself, and are not dependent on any features of higher level applications, including so-called Complex Text Layout (CTL) functions.

In Thai, Dead Keys for diacritics are “post-fix,” i.e. typed *after* the consonants<sup>54</sup> with which they are associated; diacritic vowels are expected to be entered before any tone marks they are paired with. The expanded grep-like techniques of the m17n library’s font layout tables<sup>55</sup> used by many operating systems to perform all of the arrangements and modifications described above are also able to correct paired diacritics that are entered in the wrong order, but don’t generally do this – considering these to be typing rather than layout errors.

Thai also has a class of vowels that are full characters (as opposed to diacritics), but some, such as the ไ in the word ไทย (“Thai”) are placed before the consonant – in this case, the ๓ (≈ the Latin t) – with which they are paired, while others like ๑ are laid out more traditionally and follow the consonant. Entry order of these, while also “correctable,” is considered a user’s responsibility. Although word processors that have auto-correct functionality could reasonably support such instances, this isn’t considered extensible.

Snippets from the United Nations Universal Declaration of Human Rights (see footnote 3) have been used earlier; on the right is the entire Article 1 in Thai which, like all these translations, is more idiomatic than literal. The English version is “All people are born free and equal in dignity and rights. They are endowed with reason and conscience and should act towards one another in a spirit of brotherhood.”

เราทุกคนเกิดมาอย่างอิสระ เราทุกคนมีความ  
คิดและความเข้าใจเป็นของเราเอง เราทุกคน  
ควรได้รับการปฏิบัติในทางเดียวกัน.  
เราทุกคนเกิดมาอย่างอิสระ เราทุกคนมีความ  
คิดและความเข้าใจเป็นของเราเอง เราทุกคน  
ควรได้รับการปฏิบัติในทางเดียวกัน!

Cursor keys in Thai (left and right arrows) operate on character cells rather than individual characters, but editing keys (e.g. backspace, delete, etc.) generally act on the individual characters within the cells. Many applications permit modifier keys to modify the behavior of these keys when desired.

*Figure 14 – Thai, without and with full Justification*

Figure 14 compares ragged right and full justification in Thai (see page 21). There are no spaces between words, such as the highlighted ๗๓ in the first sentence. The second line of the fully justified example illustrates how spacing between character cells is altered to “stretch” the line to the right margin. Spaces, which delineate sentences, are highlighted, as is the “period” which, in Thai, is typically found only at the end of a paragraph.

53 At some time in history it seems to have been, but has been considered a single letter for centuries.

54 See Figure 2 – Dead Keys on a manual Thai Typewriter (circa 1970) on page 12 for an illustration of how dead keys function on manual Thai typewriters.

55 In Ubuntu, such \*.flt files are located in /usr/share/m17n; their location in other distributions varies.

## DEVANAGARI SCRIPT EXAMPLES USING हिंदी भाषा ( THE HINDI LANGUAGE )

### Brief Comments about Hindi

The South Asian Devanagari script used for Hindi is likely used in more languages than any other, being the primary script used for Bodo, Konkani, Maithili, Marathi, Nepali, Pali, classical Sanskrit and Sindi as well as in more than one hundred other languages in the areas around India, Nepal (see Nepalese currency example on page 23), and Bhutan.

Hindi itself is one of India’s official languages and the official language for several of its twenty-nine states including Bihar, Haryana, Himachal Pradesh, Jharkhand, Madhya Pradesh, and Uttar Pradesh.

Devanagari Script is defined in Unicode blocks `U+0900-097F` and `U+A8E0-A8FF`,<sup>56</sup> and is written from left-to-right, and top-to-bottom (LRTB). Unlike many scripts, the “baseline” for Devanagari scripts is more pronounced and is closer to the top of its character cells than the bottom, making this a recognizable script even for those unfamiliar with any of the languages that use it. Devanagari characters are typically connected to one another within any individual word.

Devanagari script has its own set of numeric digits – the 0 to 9 equivalents are: ० १ २ ३ ४ ५ ६ ७ ८ ९ – but generally uses the shared numerals defined in the Latin code block. The computer keyboard used for these examples can only produce the Devanagari digits with the `AltGr` modifier key, but others, like the “Hindi (Wx)” keyboard mentioned below, place these on the top row, with the Latin 0-9 digits on the numeric keypad.

### Examples for Experimentation (No knowledge of Hindi needed)

The keyboard mapping used to create these examples is named “Inscript (m17n),” one of many keyboards used to type Devanagari script using Latin letters. The Latin key presses when using this layout are shown directly below the Hindi characters, and the following lines provide the Unicode hexadecimal values and their decimal equivalents as was done in the earlier Thai example.

This example illustrates a two word, ten character sequence हिंदी भाषा meaning “(the) Hindi language.” The word “the” is parenthesized because Hindi doesn’t use articles (e.g. “a,” “the,” and similar words).

Char 1	Char 2	Char 3	Char 4	Char 5	Char 6	Char 7	Char 8	Char 9	Char 10
ह	ि	ं	द	ी		भ	ा	ष	ा
u	f <sup>57</sup>	x	o	r	space	y	e	<	e
0x939 2361d	0x93f 2367d	0x902 2306d	0x926 2342d	0x940 2368d	0x20 32d	0x92d 2349d	0x93e 2366d	0x937 2359d	0x93e 2366d
ह	हि	हिं	हिंद	हिंदी		भ	भा	भाष	भाषा

If the phonetic “Hindi (Bolnagri)” keyboard is used, the Latin letters should be `h i n u d I space B a S a`. The phonetic “Hindi (Wx)” can also be used with the Latin letters `h i n x I space B A R A`. In either case, take care to use the Shift key to create the Latin upper case letters shown. As with Thai, Devanagari script has no concept of “capital” and “small” letters and, when shifted, the same keys produce entirely different and unrelated characters.

Devanagari fonts without all the appropriate ligatures may still produce readable text, but for extended use of any particular language in these families, the presence of any desired ligatures should be confirmed.<sup>58</sup>

The first two characters are an example of Character Reordering, introduced on page 11. When the second character, the vowel ि, is entered after the initial consonant ह, it is placed to the left of it in a manner similar to what happens

56 See <http://www.unicode.org/charts/PDF/U0900.pdf> & <http://www.unicode.org/charts/PDF/UA8E0.pdf> respectively.

57 The dotted circle is not part of the Devanagari characters; it is a display convention often used to indicate the relative placement for vowels or diacritics that cannot or do not stand on their own. In some languages, such as Thai, a stand-alone diacritic or vowel such as the <sup>ˆ</sup> shown on page 24, may appear alone, but will more commonly be displayed over a silent “holder” consonant – in Thai, a “ə” – so they will appear as <sup>ˆ</sup>ै. The dotted circle convention, therefore, isn’t typically seen in such Scripts.

58 How this can be done is described in Design Note #4, *Evaluating Fonts for use in Multi-Lingual Documents*.

with sequential entry of characters in a right-to-left (RTL) script. Unlike the Thai example above, Hindi illustrates the use of the CTL characteristic known as character reordering, which appears not only with Devanagari script, but many others. Support for character reordering such as illustrated is needed because of the wide variety of keyboard layouts in use. With phonetic keyboards, the sounds produced by the initial characters in this example need a reversal as part of their correct display placement.

On the right is the entire Article 1 of the United Nations Universal Declaration of Human Rights (see footnote 3) in Hindi for comparison to other examples used within this document.

Etymology buffs might note the similarity between the pronunciation of the Hindi (भाषा “basa/basha”) and Thai (ภาษา “pasa”) words for “language.”

सभी मनुष्यों को गौरव और अधिकारों के मामले में जन्मजात स्वतन्त्रता और समानता प्राप्त है। उन्हें बुद्धि और अन्तरात्मा की देन प्राप्त है और परस्पर उन्हें भाईचारे के भाव से बर्ताव करना चाहिए।

Figure 15 – Sample Hindi Text Block

## ARABIC SCRIPT EXAMPLES USING اللغة العربية ( THE ARABIC LANGUAGE )

### Brief Comments about Arabic

Not surprisingly, the Arabic language uses characters from the Arabic script in Unicode’s Middle Eastern group,<sup>59</sup> defined in Block U+0600–06FF as well as several supplemental blocks.<sup>60</sup> Arabic is written from right-to-left and top-to-bottom (RLTB). Arabic script is said to appear “cursive,” although that is subjective.

Arabic script is used to write what is known as “Modern Standard Arabic,” as well as Arabic dialects<sup>61</sup> or variants (e.g. Algerian, Egyptian, Lebanese, Moroccan, and Syrian), and less related languages such as Āynu, Azeri, Baluchi, Beja, Bosnian, Brahui, Chechen, Crimean Tatar, Dari, Gilaki, Hausa, Kabyle, Karakalpak, Konkani, Kashmiri, older Kazakh and Kyrgyz, Khowar, Kurdish, Malay, Marwari, Mandekan, Mazandarani, Morisco, Pashto, Persian/Farsi, Punjabi, Rajasthani, Salar, Saraiki, Shabaki, Shughni, Sindhi, Somali, Tatar, Tausūg, Turkish, Urdu, Uyghur, Uzbek, Wakhi and a number of other languages. Several of these, such as Kazakh and Kyrgyz, are written in several Scripts.

Arabic script includes a set of digits – the 0 to 9 equivalents<sup>62</sup> are: ٠ ١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩, although the shared numerals defined in the Latin code block are more often used depending on the specific language and context.

Arabic characters within a word are very often connected, and Arabic script makes extensive use of Contextual Shaping, with many characters having isolated, initial, median and final versions. Additionally, some median versions may vary depending on the specific characters surrounding them.

### Examples for Experimentation (No knowledge of Arabic needed)

Here we show how to enter the thirteen character sequence for the Arabic phrase اللغة العربية (meaning “the Arabic Language”) using the same three methods described earlier.

The top row, labeled “Char 1” through “Char 13” shows the sequence in which the characters are entered and stored in memory and on disk, while the second row shows the individual Arabic characters.

The keyboard mapping used here is the “Arabic (qwerty/digits)” and the third row shows the letters to be typed on a standard Latin keyboard if that keyboard is selected as the input method.

The fourth and fifth rows show the Unicode values in hexadecimal and decimal respectively. Finally, the fifth row shows the resulting progression of the modifications made to the characters as each word is typed.

59 See <http://www.unicode.org/charts/PDF/U0600.pdf>

60 These include the Arabic Supplement (U+0750–077F), Arabic Extended-A (U+08A0–08FF), Arabic Presentation Forms-A (U+FB50–FDFF) and Arabic Presentation Forms-B (U+FE70–FEFF). Presentation Forms are discussed later.

61 And recall that there is no widely accepted definition of “dialect.” Its usage is often based as much on politics as linguistics.

62 Note the similarity of the digits 1, 2, 3, and 9 to ASCII numerals. Persian and Urdu flavors of Arabic Script have variants of these.

Char 1	Char 2	Char 3	Char 4	Char 5	Char 6	Char 7	Char 8	Char 9	Char 10	Char 11	Char 12	Char 13
ا	ل	ل	غ	ة		ا	ل	ع	ر	ب	ي	ه
h	g	g	y	m	space	h	g	u	v	f	d	i
0x627	0x644	0x644	0x63a	0x629	0x20	0x627	0x644	0x639	0x631	0x628	0x64a	0x647
1575d	1604d	1604d	1594d	1577d	32d	1575d	1604d	1593d	1585d	1576d	1610d	1607d
ا	ال	الل	اللغ	اللغة		ا	ال	الع	العرب	العربي	العربية	

When the first character is typed, the computer should immediately recognize that a right-to-left script is being used, even if the default paragraph is set as left-to-right; the new character (ا) will appear as usual with the exception that the cursor should now be located to the left of the character instead of its right. If this entry is in a bidirectional paragraph, some applications will alter the cursor to indicate the direction the text will flow when the next character is entered. More details will be presented in the Hebrew Script section on page 29 and in Design Note #5.

Examples in the Arabic Language, cont.

Then, as expected, when the second character ل is entered, it will be placed to the left of the first. When the third character, which notably is the same character as the second, is entered,<sup>63</sup> the variation in character display becomes evident. The new ل is placed, as expected to the left of the previous one, but the previous one has changed form. This alteration is generally handled transparently by modern Input Method utilities using Open Type font technologies,<sup>64</sup> regardless of whether a specific script has been activated as the current keyboard or not; these utilities will generally recognize the script that has been entered and act accordingly.

The appearance of the fourth character entered (غ) is likewise altered for display with the previous ل character. Also note how the eleventh character was altered when the twelfth (ي) was entered.

The sixth character entered is a space. As can be seen from its hexadecimal and decimal values, the space character is not part of the Arabic script block, but is the same character shared with and used by a variety of scripts. The difficulties in handling such shared characters in paragraphs with bidirectional text will be deferred for the moment, but are discussed in the Hebrew Script examples section that begins on page 29.

### *Kashideh Justification and emphasis*

An interesting use of Contextual Shaping in certain Arabic and other Devanagari scripts is a method of full justification known by the Arabic term *Kashideh*; with handwritten manuscripts, this was accomplished by stretching certain connecting lines between characters rather than simply inserting spaces. Most scripts are fully justified by expanding the spacing between either words, letters or both, but when *Kashideh* is implemented with technology, this is typically accomplished using character variations available in the Arabic Presentation Forms (see footnote 60).

To the right are two Arabic renditions from Article 1 of the Universal Declaration of Human Rights used throughout this document. The top example shows the text in its basic flush right, ragged left layout, while the lower example shows full justification using the *Kashideh* layout method.

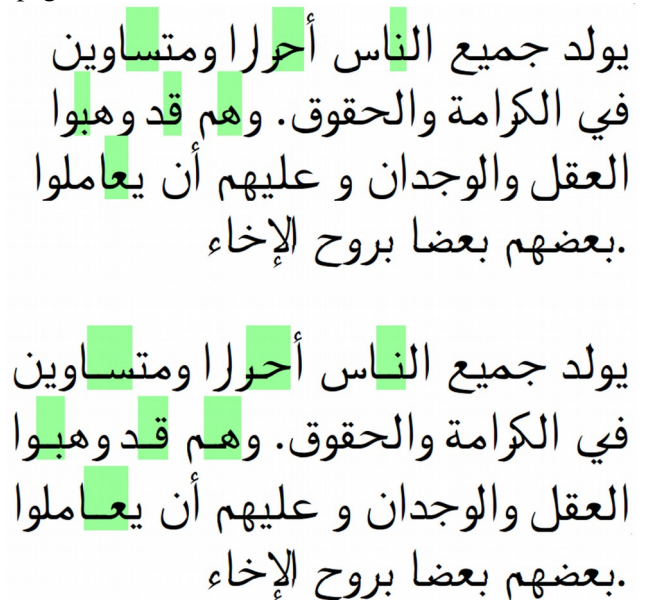


Figure 16 – Arabic, before and after *Kashideh Justification*

63 And, importantly, placed in memory or on disk using the same character code (u+0644).

64 N.B. Fonts with Open Type capabilities include not only those with .otf extensions, but modern TrueType fonts as well.

Specific examples of the differences are highlighted for clarity. Note that, in this example, none of the spacing between words or characters was altered to achieve full justification, although the use of Kashideh doesn't prohibit this.

Kashideh is often used automatically when full justification of an all-Arabic paragraph is requested in many applications, but it can also be applied, usually at an application user's request, to emphasize an important word or to correspond to phonetic inflection – similar in principle to how Bold and Underline are sometimes utilized in Latin scripts.

## HEBREW SCRIPT EXAMPLES USING שפת עברית ( THE HEBREW LANGUAGE )

### Brief Comments about Hebrew

The Middle Eastern Hebrew Script, defined in Unicode block U+0590-05FF,<sup>65</sup> is used in several unrelated languages, primarily the Hebrew language of course, but also Judeo-Arabic, Ladino, Yiddish and others. Hebrew script is written from right-to-left and top-to-bottom (RLTB) similar to the Arabic script above. Unlike Arabic, however, the Hebrew perspective of paired symbols<sup>66</sup> is that characters such as parentheses, brackets, and braces are the reverse of what they are in Arabic or in most western scripts.

Hebrew script includes five characters with different final forms. The character כ, for instance is written as ך at the end of a word. Although there is no distinction between “capital” and “small” letters, Hebrew text, unlike many non-Latin scripts, does have both serif and sans-serif as well as some purely decorative typeface designs.

### Examples for Experimentation (No knowledge of Hebrew needed)

The keyboard mapping used for these examples is a very basic “Hebrew”<sup>67</sup> rather than any of the biblical or phonetic variants that may also be available. The first example illustrates the two word, nine character sequence שפת עברית, meaning “(the) Hebrew language.” The bottom row shows the cumulative results as each character is entered.

שפת עברית – Examples in the Hebrew Language

Char 1	Char 2	Char 3	Char 4	Char 5	Char 6	Char 7	Char 8	Char 9
ש	פ	ת		ע	כ	ר	י	ת
a	p	,	space	g	f	r	h	,
0x5e9	0x5e4	0x5ea	0x20	0x5e2	0x5d1	0x5e8	0x5d9	0x5ea
1513d	1508d	1514d	32d	1506d	1489d	1512d	1497d	1514d
ש	שפ	שפת		ע	עכ	עכר	עכרי	עברית

Display of the characters is straightforward, since only the five characters with final forms will change “on the fly” when a word ending is reached, and Hebrew requires no character reordering.

The Hebrew version of the United Nations Universal Declaration of Human Rights Article 1 (see footnote 3) is shown below in versions with and without the inclusion of Hebrew's optional diacritics:

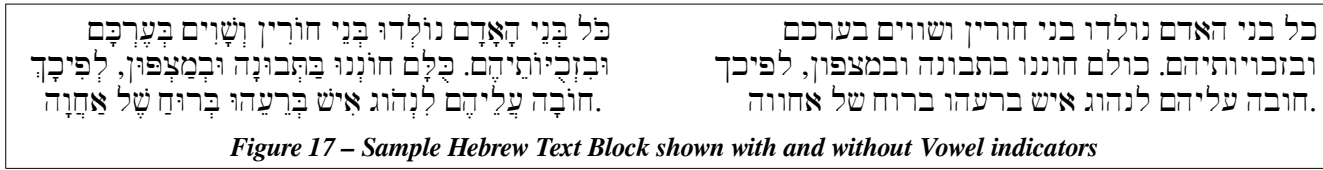


Figure 17 – Sample Hebrew Text Block shown with and without Vowel indicators

65 See <http://www.unicode.org/charts/PDF/U0590.pdf> and <http://www.unicode.org/charts/PDF/UFB00.pdf>, the latter containing the Hebrew “Alphabetic Presentation Forms” Unicode block U+FB00-FB4F.  
 66 See “Paired Symbols in Text with Mixed Directions” on page 17.  
 67 The Hebrew Lyx keyboard layout used in this example is shown on page 40.

## KOREAN SCRIPT EXAMPLES USING 한국어 ( THE KOREAN LANGUAGE )

### Brief Comments about Korean

Omniglot categorizes the Korean language, along with Basque, Cofán, Japanese, Páez, Ticuna, and Urarina, as an Isolate – one “with no known connections to any other languages.” Korea’s similarly unique left-to-right Script, used only by the Korean language, did not evolve over time as others did, but was deliberately designed by a group of scholars under the direction of an early Korean linguist<sup>68</sup> in the year 1443.

In the opening paragraphs of “Exploring Alphabets<sup>69</sup>,” we glossed over primitive logographic writing systems in which each symbol represented a specific sound, word or idea, because these are not relevant to most database design efforts. An example of Hieroglyphs, used in several early writing systems, appears in Figure 3 on page 7 of that document. An example of Chinese appears in the right margin on page 14 of this document, showing a few of its logograms.

To the right is the Korean translation of the now familiar Article 1 of the U.N.’s Universal Declaration of Human Rights. At first glance, this block of text may appear to the uninitiated as yet another Asian logographic writing system like Chinese or Japanese because:

- It appears to have a wide variety of complex symbols, all of which seem to be of a similar if not identical size.
- An examination of the code points<sup>70</sup> in this segment indicates that they all come from the Unicode Plane 0xAC00-0xD7AF (“Hangul Syllables” 음절 한글) which, upon examination, can be seen to contain over 11,000 code points.

모든 인간은 태어날 때부터 자유로  
우며 그 존엄과 권리에 있어 동등  
하다. 인간은 천부적으로 이성과  
양심을 부여받았으며 서로 형제애  
의 정신으로 행동하여야 한다.

Figure 18 – Korean text example

This impression is quite misleading, however. A brief glance at a Korean keyboard (whether real or virtual) shows that Korean text can be typed easily using far fewer keys than most other languages require. So what’s going on?

Korean Script is, in fact, quite orderly and logical, and simply requires a slight change in perspective to grasp. From a Korean perspective, each of the above symbols is viewed as a “character.” From a western perspective, however, we would consider each of these symbols to be “syllables,” with the individual components within the symbol being the actual “characters.” These “component” characters, known as Jamo, part of the much smaller Unicode Plane 0x1100-0x11FF (“Hangul Jamo” 자모 한글),<sup>71</sup> are the “letters” that appear on a Korean keyboard.<sup>72</sup>

Like alphabetic characters in other Scripts, each individual Jamo is characterized as a vowel or consonant, although several of these represent what are essentially ligatures.<sup>73</sup> The consonant ㄴ, for instance, is a double ㄴ, while ㅃ is a double ㅃ. The reason why doubled consonant sounds are given their own code point will become apparent shortly.

A Korean syllable must always be composed of a leading consonant followed by a vowel; the first syllable in Figure 18 above (모) is composed of the leading consonant ㅁ and the vowel ㅜ, which are merged to form the single glyph 모. For syllables that begin with a vowel sound (e.g. 인), a silent consonant ㅇ is provided to keep the rules unequivocal.

A Korean syllable may also incorporate an optional ending consonant: the second syllable above (든) is composed of the leading consonant ㄷ, the medial vowel ㅡ, and the final consonant ㄴ, which are merged into the syllable 든. The table below will help as we present some key points to understand about Korean writing and data storage.

68 This Linguist’s day job, interestingly, was “King of Korea” – he is remembered for many reasons as King Sejong the Great.

69 The first Design Note in this series, also available from [www.AntikytheraPubs.com/i18n.htm](http://www.AntikytheraPubs.com/i18n.htm).

70 What you see will be affected by how your operating system stores data; in UTF-8 (strongly recommended for systems supporting multiple Scripts), the AC00 code point value of the 가 syllable is stored as EAB080. See Database Design Note 3 “Exploring UTF-8” for more information. On-line conversion tools such as <https://r12a.github.io/apps/conversion/> are also helpful.

71 There are supplemental Jamo in Unicode Planes 0xA960-0x097F (“Hangul Jamo Extended-A”) and 0xD7B0-0xD7FF (“Hangul Jamo Extended-B”), but these are not considered here, nor is the 0x3130-0x318F “Hangul Compatibility Jamo”) plane.

72 The standard Korean keyboard layout used in this example is given on page 41.

73 See Database Design Note #1: “Character Codes & Character Cells” beginning on page 8. The difference is that Korean Ligatures are not simply combinations of displayed glyphs, but actual characters.

Table of Initial Consonants, Medial Vowels, and Ending Consonants for Koean Jamo Characters (components) Showing standard order/index number, key press on Latin <sup>74</sup> keyboard, and Hexadecimal and Decimal values of Unicode Code Points																											
Order/Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
Lead/Initial Consonant	ㄱ	ㅋ	ㄴ	ㄷ	ㅌ	ㄹ	ㅁ	ㅂ	ㅃ	ㅅ	ㅆ	ㅇ	ㅈ	ㅊ	ㅌ	ㅍ	ㅑ	ㅓ	ㅕ	ㅗ	ㅛ	ㅜ	ㅠ	ㅡ			
Key Press	r	R	s	e	E	f	a	q	Q	t	T	d	w	W	c	z	x	v	g								
Jamo U+11+	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12								
Jamo d4300+	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70								
(Medial) Vowel	ㅏ	ㅑ	ㅓ	ㅕ	ㅗ	ㅛ	ㅜ	ㅠ	ㅡ	ㅣ																	
Key Press(es)	k	o	i	O	j	p	u	P	h	hk	ho	hl	y	n	nj	np	nl	b	m	ml	l						
Jamo U+11+	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73	74	75						
Jamo d4400+	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69						
Ending/Tail Consonant	ㄱ	ㅋ	ㄴ	ㄷ	ㅌ	ㄹ	ㅁ	ㅂ	ㅃ	ㅅ	ㅆ	ㅇ	ㅈ	ㅊ	ㅌ	ㅍ	ㅑ	ㅓ	ㅕ	ㅗ	ㅛ	ㅜ	ㅠ	ㅡ			
Key Press(es)	r	R	rt	s	sw	sg	e	f	fr	fa	fq	ft	fx	fv	fg	a	q	qt	t	T	d	w	W	z	x	v	g
Jamo U+11+	A8	A9	AA	AB	AC	AD	AE	AF	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF	C0	C1	C2
Jamo d4500+	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

There are several things to notice in the table above:

- ↘ There are no “capital” or “small” letters in Korean, but the key presses used to produce these Korean Jamo *are* case-sensitive. A small **t** and a capital **T** produce different results (ㅌ and ㅆ respectively).
- ↘ The Jamo Unicode Code Point ranges in the above chart are restricted to 0x1100–0x1112 (initial consonants), 0x1161–0x1175 (medial vowels), and 0x11A8–0x11C2 (ending consonants). The “missing” code points are defined to support obsolete Jamo/characters, and are not relevant to this discussion.
- ↘ The decimal values corresponding to the Jamo Unicode Code Points are provided for reference in a subsequent discussion of how the final stored Hangul syllable values are calculated from the Jamo components.
- ↘ Five of the Lead/Initial Consonants are simply “shortcuts” for double characters (e.g. typing a small **e** produces the ㅓ character; typing either a double **e** or a single capital **E** will produce the ㅕ character.)<sup>75</sup>
- ↘ The only two capital letters on a Latin keyboard assigned to produce vowels are independent from the combinations they represent. Based on the common behavior of capital letter “shortcut” assignments used for consonants, one might expect that the ㅑ vowel produced by typing the capital **O** might also be produced by typing **i** **l**, but this isn’t the case with vowels. Neither will typing **u** **l** produce the ㅓ character that the **P** key produces. The remaining characters representing what we call diphthongs are all formed by typing their component vowels.
- ↘ Sixteen of the glyphs in Ending/Tail Consonants row appear to be identical to (and actually represent the same “character”) as entries in the Initial Consonants area, but it is important to note that each “final” version of the character is given its own Code Point in the Unicode Hangul Jamo Plane that is distinct from its “leading” version.

74 Key presses required on a U.S. English keyboard with the iBus “han2(m17n)” mapping for Korean. See this layout on page 41.  
75 Not all input methods support the use of both methods. Only three of these pairs appear in the Ending Consonants block.

### Examples for Experimentation (No knowledge of Korean needed)

To best illustrate how Jamo characters are “assembled” into Hangeul syllables, the following table will follow how the system responds as each character of the Korean words 사람 (“person”) and 물 (“water”) are typed.

Dynamic assembly of typed Jamo “characters” into displayed Hangeul “syllables”								
	Example 1: Typing 사람 (person)					Example 2: Typing 물 (water)		
Character Count	1	2	3	4	5	6	7	8
Key Press	t	k	f	k	a	a	n	f
Consonant or Medial Vowel	leading consonant U1109	medial? vowel U1161	consonant U11AF ? U1105 ?	medial? vowel U1161	ending consonant U11B7	leading consonant U1106	medial vowel U116E	ending consonant U11AF
Jamo “character”	ㅏ	ㅑ	ㄹ	ㅑ	ㅓ	ㅓ	ㅓ	ㄹ
Screen Display	<u>ㅏ</u>	<u>ㅑ</u>	<u>ㅑ</u>	<u>ㅑ</u> ㅓ	<u>ㅑ</u> ㅓㅓ	ㅓ	ㅓ	ㅓ
Stored Hangeul				ㅓ	ㅓㅓ			ㅓ

- Key press 1: Starting with a blank document, press the small **t** key and an underlined Jamo ㅏ<sup>76</sup> will appear. Because any valid Korean syllable *must* contain at least an initial consonant and a vowel, it cannot be a completed syllable, and the system will wait for it to be completed. Observe the underlines throughout this exercise.
- Key press 2: The **ㅑ** Jamo produced by typing **k** is recognized as a valid vowel, and is therefore combined with the previous character to display ㅑㅓ. Although this is a valid single syllable word meaning “four,” the possibility still remains that a final consonant may appear, so the word “four” is not assumed. Note that, because of the strict rules for syllable construction in Korean, there is no need for dictionary access to verify that ㅑㅓ is a valid syllable.
- Key press 3: The **ㄹ** Jamo, typed with the **f**, is a valid consonant. Since **ㄹ** can function as both a leading or final consonant (resulting in a Jamo value of either 0x1105 or 0x11AF respectively), the input system remains in a tentative state until that can be determined, and the **ㄹ** is temporarily attached to the ㅑㅓ to display the ㅑㅓㄹ syllable. This is actually a single syllable word meaning “flesh.”
- Key press 4: The **ㅑ** Jamo is once again recognized as a valid vowel. Since a vowel cannot begin a syllable, though, it is now clear that the previous **ㄹ** must be the leading consonant of a new (second) syllable – not an optional final consonant of the previous syllable. The previous syllable ㅑㅓ (“flesh”) is now recognized as complete without the **ㄹ**, “corrected” to read ㅑㅓ (“four”), and converted to the Hangeul syllable ㅓ. It is now stored, not as two Jamo elements 0x1109 and 0x1161, but a single Hangeul syllable 0xc0ac. Additionally, the screen display shows the **ㄹ** and **ㅑ** tentatively combined into a ㅑㅓㄹ syllable in the same manner as above – waiting to determine if the syllable is complete or if a final consonant will follow.
- Key press 5: Once the final **ㅓ** is typed, and the resulting **ㅓ** is recognized as a valid consonant, there can be no valid additions to the syllable, so the ㅑㅓㄹ is stored as the Hangeul syllable 0xb77c once another key is pressed.
- Key press 6: When the **ㅓ** key is typed again, another **ㅓ** appears, as might be expected. This time, however, the Jamo value in temporary memory is 0x1106 rather than 0x11b7, since this **ㅓ** *must be* a leading consonant.
- Key press 7: Typing **n** adds the vowel **ㅓ** to form a possible one-syllable word “radish,” but it remains unclear whether that is the end of the syllable, or if there is a final consonant still to be entered.
- Key press 8: The final key press **f**, the same valid consonant **ㄹ** that we saw in key press 3, completes and “commits” the syllable. ㅓㅓㄹ is stored as the Hangeul character/syllable 0xbb3c and the one syllable word is complete. Note that any non-Korean character (e.g. a space) will “commit” a valid Jamo sequence, but erase an incomplete one.

<sup>76</sup> The underlining indicates that the character is not yet “committed” as we would say in the world of database management.



## Korean Numeric Characters

Korean generally uses the shared numerals defined in the Latin code block, but has two distinct sets of “digits” (character blocks) of its own. The first set of 0 to 9 equivalents, used for time, dates, money, addresses, and any number equal to 100 or above, is:

Digit:	0	1	2	3	4	5	6	7	8	9
Hangul:	공	일	이	삼	사	오	육	칠	팔	구
Key Presses:	rhd <sup>77</sup>	dlf	dl	tka	tk	dh	dbr	wlf	vkf	rn

A separate set of number representations, limited to numbers below 100, is used for stating a person’s age, but is not shown here.

## Converting Jamo Combinations to Hangul Syllables (the basic Math)

Luckily, the Unicode Consortium complemented King Sejong’s logic and order by making the conversion between Jamo and completed Hangul syllables equally straightforward. Knowing at least one method (and there are several) for performing such conversions isn’t always necessary, but can help in analyzing data migration issues that may arise.

The formula<sup>78</sup> inputs and results are all shown as decimal values for convenience using any available calculator.

---

Formula: $HPO + ((JamoL-JOV) * LCO) + ((JamoV-V2C) * JVO) + (if\ JamoF\ Not\ Null\ then\ (JamoF-FCO)\ else\ 0)$ , where	
<b>JamoL</b> is the lead consonant; <b>JamoV</b> is the vowel; <b>JamoF</b> is the optional Final consonant. Constants used are shown below:	
<b>HPS</b> = 44032	0xAC00 Hangul Plane Start: Location of first Hangul Syllable ㄱ in the Unicode “Hangul Syllables” plane
<b>HPO</b> = 43416	0XA998 Hangul Plane Offset: Offset from beginning of Unicode’s “Hangul Jamo” plane to HPS above
<b>JOV</b> = 4351	0x10FF Jamo Ordinal Value: Offset from “Hangul Jamo” plane to Leading Consonant Ordinal Values 1 to 19
<b>LCO</b> = 588	0x024c Lead Consonant Offset: Offset from Lead Consonant to Consonant-Vowel pairs in “Hangul Syllable” plane
<b>V2C</b> = 4448	0x1160 Vowel to Combination: Offset from Vowel Jamo to its consonant+vowel group in “Hangul Syllable” plane
<b>JVO</b> = 28	0x001c Jamo Vowel Offset: Distance between consonant-vowel combination pairs in any “Hangul Syllable” block
<b>FCO</b> = 4519	0x11A7 Final Consonant Offset: Offset added to Ordinal Value of Jamo Final Consonant: 1 to 27

---

Convert Jamo ㅁ (0x1106, 4358) and ㅡ (0x1169, 4457) to the Hangul syllable ㅁ (0xBAA8, 47784)

HPO	+	((JamoL-JOV) * LCO)	+	((JamoV-V2C) * JVO)	+	if JamoF Not Null then JamoF-FCO else 0	=	Hangul Code Point (dec) (hex)
43416	+	((4358-4351)*588)	+	((4457-4448)*28)	+	0		
43416	+	(7*588)	+	(9*28)	+	0		
43416	+	4116	+	252	+	0	=	47784 0xBAA8

Convert Jamo ㅓ (0x1101, 4353), ㅕ (0x1161, 4449), and ㅗ (0x11AA, 4522) to the Hangul syllable ㅓ (0xAE4F, 44623)

HPO	+	((JamoL-JOV) * LCO)	+	((JamoV-V2C) * JVO)	+	if JamoF Not Null then JamoF-FCO else 0	=	Hangul Code Point (dec) (hex)
43416	+	((4353-4351)*588)	+	((4449-4448)*28)	+	(4522-4519)		
43416	+	(2*588)	+	(1*28)	+	3		
43416	+	1176	+	28	+	3	=	44623 0xAE4F

<sup>77</sup> These “number representations” are not available directly from the Korean keyboard layout, but are typed using the Latin key presses shown in this row.

<sup>78</sup> All Hangul ⇄ Jamo conversion formulas given in this section are based on section 3.12 “Conjoining Jamo Behavior” in Chapter 3, “Conformance” of The Unicode® Standard Version 9.0 – Core Specification. As shown here, they are made to be more suitable for hand calculation than implementation as code: the form  $A - (int(A/B) * B)$  shown in the procedure for extracting JamoV from Hangul, for instance, is merely  $A \bmod B$  done in steps. Libraries or example Code for performing these conversions in a variety of programming languages can easily be found with a web search.

## Converting Hangul Syllables to Jamo Components (the basic Math)

During normal operations, extracting the Jamo values from a syllable isn't normally useful, but the following algorithm/sequence/formula will accomplish the task if needed:

The first step is to calculate a value (here called Temp) commonly used in the extraction of each of the Jamo components from the Hangul syllable.

Extraction of each portion of the Hangul syllable is then accomplished using the formulas shown.

Finally, the result of the secondary calculation for the final character will indicate whether or not such a character exists. If it does, the final value is calculated; if it doesn't, none was present in the syllable.

Examples (based on the two example Hangul syllables 모 and 꺾 composed in the previous section):

### Extract Jamo Component Values from Hangul 47784

Temp = Hangul-HPS = 47784-44032 = 3752

Calculate JamoL from Hangul 47784

```
1 + JOV + int(Temp/LCO)
1 + 4351 + int(3752/588)
1 + 4351 + int(6.38095)
1 + 4351 + 6 = 4358 (ㅁ)
```

Calculate JamoV from Hangul 47784

```
1 + int((Temp - (int(Temp/LCO) * LCO)) / JVO) + V2C
1 + int((3752 - (int(3752/588) * 588)) / 28) + 4448
1 + int((3752 - (int(6.38095) * 588)) / 28) + 4448
1 + int((3752 - (6 * 588)) / 28) + 4448
1 + int((3752 - (3528)) / 28) + 4448
1 + int((3752 - 3528) / 28) + 4448
1 + int((224) / 28) + 4448
1 + int(8) + 4448
1 + 8 + 4448 = 4457 (ㅓ)
```

Calculate JamoF from Hangul 44784

```
JFTmp = Temp - (int(Temp/JVO) * JVO)
JFTmp = 3752 - (int(3752/28) * 28)
JFTmp = 3752 - (int(134) * 28)
JFTmp = 3752 - (134 * 28)
JFTmp = 3752 - 3752 = 0
IF JFTmp > 0
THEN JamoF = JFTmp + FCO
ELSE JamoF is null;
JamoF is null
```

### Extract Jamo Component Values from Hangul 44623

Temp= Hangul-HPS = 44623-44032 = 591

Calculate JamoL from Hangul 44623

```
1 + JOV + int(Temp/LCO)
1 + 4351 + int(591/588)
1 + 4351 + int(1.00510)
1 + 4351 + 1 = 4353 (ㅍ)
```

Calculate JamoV from Hangul 44623

```
1 + int((Temp - (int(Temp/LCO) * LCO)) / JVO) + V2C
1 + int((591 - (int(591/588) * 588)) / 28) + 4448
1 + int((591 - (int(1.0051) * 588)) / 28) + 4448
1 + int((591 - (1 * 588)) / 28) + 4448
1 + int((591 - (588)) / 28) + 4448
1 + int((591 - 588) / 28) + 4448
1 + int((3) / 28) + 4448
1 + int(0.10714) + 4448
1 + 0 + 4448 = 4449 (ㅑ)
```

Calculate JamoF from Hangul 44623 (4522)

```
JFTmp = Temp - (int(Temp/JVO) * JVO)
JFTmp = 591 - (int(591/28) * 28)
JFTmp = 591 - (int(21.107) * 28)
JFTmp = 591 - (21 * 28)
JFTmp = 591 - 588 = 3
IF JFTmp > 0
THEN JamoF = JFTmp + FCO
ELSE JamoF is null;
JamoF = 3 + 4519 = 4522 (ㅓ)
```

## CHARACTER ENTRY METHODS

There are many techniques for entering non-Latin characters on a Latin keyboard, and a discussion of these is beyond the scope of this paper, but generally speaking, they fall into the following broad categories:

- Activating some form of **Input Method Editor (IME)** and selecting the particular script desired.  
This is the most efficient method available for entering more than a few characters, and any operating system offers *at least* one IME. In Linux, for instance, iBus seems to be the most common such method currently offered. For each of the examples in this document, the Latin keys are listed that will result in the desired output when an appropriate Input Method is active. With iBus installed, using the **Shift**+**b** keys will produce the Devanagari character ऋ using the “Hindi Bolnagri” keyboard<sup>79</sup>, or the ऌ character using a “Hindi Inscript” keyboard.
- Utilizing a character insertion map provided by either an operating system, an Input Method Editor (see above), or by the application in use. The one from LibreOffice Writer shown below is typical.

If there is only an occasional need to enter a non-standard character for the language in use, most operating systems provide character map utilities, as do many office applications such as Microsoft Word and Excel, LibreOffice Writer, and so forth. Many of these permit directly choosing a Unicode Block as well as a font to make locating a particular character easier. Then, if that particular block isn't present, different fonts can be selected until the desired character is found.

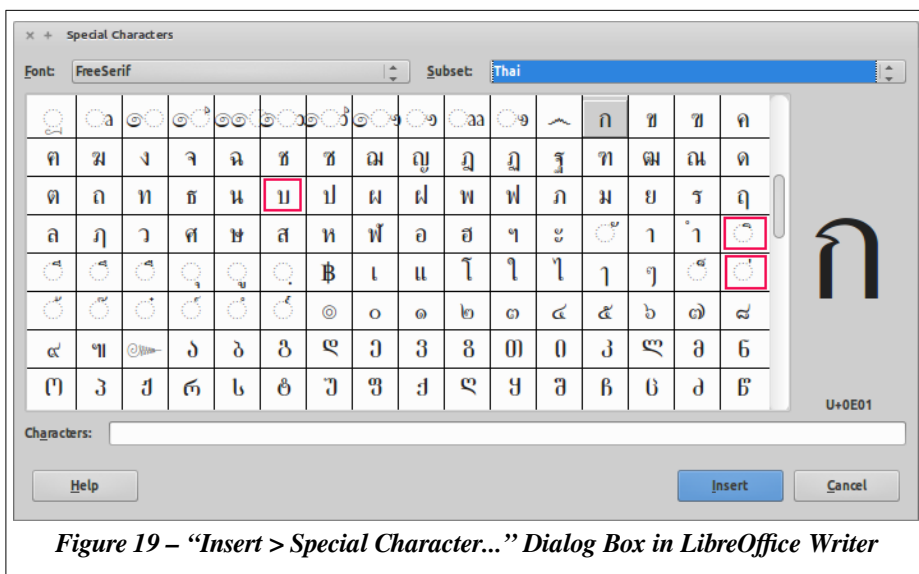


Figure 19 – “Insert > Special Character...” Dialog Box in LibreOffice Writer

In *Examples for Experimentation (No knowledge of Thai needed)*

on page 24, the composite character ก็ was shown as part of the first example. The dialog box shown above can be used to enter this series by selecting the Thai Subset, then clicking the cells outlined in top to bottom order, followed by the Insert button. Most character insertion maps operate similarly, regardless of the operating system.

- Directly entering the Unicode **Hexadecimal value** for the desired character.  
This is typically accomplished with some key sequence indicating that the next characters are intended as hexadecimal Unicode values. In Ubuntu and several other Linux flavors, for instance, the key sequence **Ctrl**+**Shift**+**u** can be typed in most applications, including the command line, resulting in the display of an underlined u when the keys are released. If the hexadecimal sequence **e 2 a** is typed followed by either the **Space** or **Enter** key, the Unicode character ก (a Thai character representing an “S” sound) should be displayed; neither a space or carriage return should be added as the result of using the **Space** or **Enter** keys in this process. Unicode hexadecimal values are given in all the examples in this paper, and others can be found from the references provided as footnotes in each script example section.

<sup>79</sup> What this means is that keyboard input is treated as if it were coming from a Bolnagri keyboard layout, one of several key arrangements that can be used to type the Devanagari characters used in the Hindi language. See the *Examples for Experimentation (No knowledge of Hindi needed)* section on page 26.

- Directly entering the Unicode **Decimal value** for the desired character. In Windows, this is usually accomplished by holding the **Alt** key and entering the decimal value – note that this must be done with the numeric keypad – for a character. Holding the **Alt** key while entering 65 on the numeric keypad will enter the Latin character “A” and holding the **Alt** key while entering 233 on the numeric keypad will enter the accented e character é. For Unicode values above 255, there may be additional steps required to activate this capability but these are well documented by Microsoft. Unicode decimal values are given in all the examples in this paper, and others can be found from various references provided on-line.
- Some applications define default key combinations to facilitate entry of Unicode character glyphs. In LibreOffice, for instance, this sequence is **Alt+x**. To type the symbol for a musical quarter note in Writer, the sequence **l d 1 5 f Alt+x** will produce the quarter note character ♩. The **Alt+x** combination is particularly useful for examining character sequences that have been pasted into a document, since it can be used to convert characters into hexadecimal sequences as well. Place the cursor immediately after the pasted character “\$” and press **Alt+x**. The “\$” character will be replaced by “U+0586.” This can be helpful in identifying that the script in use is Armenian<sup>80</sup>, and that “\$” is the small “feh” character so that, if necessary, you can select the appropriate keyboard mapping for further editing. The capital “feh” character is ₴ (U+0556) if you care.
- Using an on-screen keyboard for a particular script and clicking on the desired character. There are two types commonly available. The first is merely a display that shows key mappings. A second more useful and popular type permits a user to click on its keys as if it were a second keyboard.

In Linux, the on-screen Onboard keyboard<sup>81</sup> is laid out to resemble a keyboard, and can be a useful adjunct for those who have selected an Input Method, but aren’t very familiar with the keyboard layout. Onboard follows whatever Input Method alphabet is currently active and, like other similarly sophisticated on-screen keyboards<sup>82</sup>, can interpret long clicks (holding down the mouse button) on certain keys will display a variety of optional characters that are related to that key; long-clicking the **a** key, for instance, will display all the permissible combinations of “a” with various diacritics, such as á, à, â, ä, å, ã, ã, ã, ã, ã, ã, and æ. Some of these maps are provided by Desktop Managers rather than their underlying operating systems.



*Figure 20 – “Onboard” on-screen keyboard – typical of many available*

- Another method for entering composite characters is to use the operating system’s “Compose Key” functionality (often assigned to the right **Alt** key, but can usually be user-defined). Pressing and releasing the Compose Key, and then sequentially typing **"** then **e** results in an ë. Using a capital E, of course, results in an Ë. Because the combinations defined for the Compose key – which may or may not be edited by a user – are typically amenable to “guessing,” this method is the first method to try if you have need for entering a single unusual character. Try guessing the combinations for all the various diacritics in the previous paragraph and you’ll see. Make a guess how you might enter the ç character.
- Linux systems using KDE-based applications, which don’t recognize most of the options described above, provide their own, command-based option. In the Kate text editor, for instance, pressing [F7] and entering `char 0xe27` results in the Thai ๓ character, while entering `char 488` produces the Ā character. While this means it supports either hexadecimal or decimal Unicode character identifiers, it may be the least convenient method for entering multiple characters from non-Latin Scripts in practice.

80 This is done, of course, by entering the hex code on the page <http://unicode.org/charts/>

81 This illustration shows the Onboard utility with the Model-M Theme running on Ubuntu. Like many such utilities, the Onboard keyboard can be extensively configured, so it is worthwhile exploring the documentation for your selected choice.

82 Better supported on mobile devices using Operating Systems like Android than on some more mature desktop systems.

## KEYBOARD MAPS USED FOR THIS DOCUMENT

Selected keyboard layouts referenced in this document are illustrated below for convenience. Because there are many alternate layouts in use for various Scripts and Languages, it is advisable to insure that keyboard maps are standardized as much as possible within your organization, and appropriate key mapping charts made available to users.

### Thai Script

The Thai TIS-820(2538) layout shown here is used for the section “Thai Script examples using ภาษาไทย ( the Thai Language )” beginning on page 24.



Thai numeric characters (digits) for 1 through 9 (๑-๙) are shown in green on the top row of the keyboard; the Thai ๐ (zero) key is entered with (shift+Q). Arabic numbers (0-9) are typed using the numeric keypad (not shown here). Do not confuse the Latin “@” character (shift+2) with the Thai digit “๑” (“1”) (shift+/), which are both entered using the same key press combination! Typing the English letters shown below when the Thai Input Method is active will provide an additional example of how Thai text is entered into a multi-lingual system.

### Thai Typing Demonstration/Practice (no knowledge of Thai required)

To the right is the first sentence (and more) of the Thai version of the United Nations Universal Declaration of Human Rights.<sup>83</sup> Unlike many languages, there are generally no spaces between words. Spaces are used to indicate breaks between sentences. The first sentence, therefore, which you are encouraged to type using the proper Input Method Editor, is the following:

เราทุกคนเกิดมาอย่างอิสระ เราทุกคนมีความ  
คิดและความเข้าใจเป็นของตนเอง ...

Figure 14a – Abbreviated Thai Text Sample

เราทุกคนเกิดมาอย่างอิสระ (literally: “we” “every” “person” “birth+come” “at” “independent”)

This brief segment consists of twenty-four typed characters that occupy twenty displayed character cells<sup>84</sup>:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
เ	ร	า	ทุ	ก	ค	น	เ	ก	ิ	ด	ม	า	อ	ย	า	ง	อิ	ส	ร	ะ
g	i	k	m6	d	8	o	g	db	f	,	k	v	pj	k	'	vb	l	i	t	

Recall that Thai Dead Keys (used in cells 4, 9, 14 and 17) are post-fix, i.e. they must be typed after the base character.

83 See footnote 3 on page 3 for a link to this text in more than three hundred different languages.

84 See Database Design Note #1: “Character Codes & Character Cells” beginning on page 8. The difference is important.

## Devanagari Script

The Devanagari-Inscript layout shown here is used for the section “Devanagari Script examples using हिंदी भाषा ( the Hindi Language )” beginning on page 26.



In this particular keyboard layout, Dead Keys<sup>85</sup> for all Diacritics (e.g. those obtained with the **s**, **x**, **X** and other keys) are Post-Fix Keys – i.e. they are to be typed after the character over or next to which they are to be placed.

### Hindi Typing Demonstration/Practice (no knowledge of Hindi or Devanagari required)

On the right is a short segment from the beginning of Article 1 of the United Nations Universal Declaration of Human Rights<sup>86</sup>. Instructions for typing the first five words will help illustrate a few behavioral characteristics not covered earlier.

सभी मनुष्यों को गौरव और अधिकारों के मामले में  
जन्मजात स्वतन्त्रता और समानता ...

Figure 15a – Abbreviated Hindi Text Sample

Our sample text, therefore, which you are encouraged to type using the proper Input Method Editor, is the following:

सभी मनुष्यों को गौरव और अ...

This segment consists of twenty-four typed characters, including spaces, occupying seventeen character cells<sup>87</sup>:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
स	भी		म	नु	ष्	भों		को		गौ	र	व		औ	र	
m	Yr	space	c	vg	<d	Yax	space	ka	space	iq	j	b	space	Q	j	space
सभी			मनुष्भों				को			गौरव				और		

Using the left or right cursor keys over character cell 2 will show that this combination is treated as a single cell, and if the cursor is placed between cells 1 and 2, pressing the **Delete** key will remove cell 2 entirely. But placing the cursor between cells 2 and 3 and using the **Backspace** key will simply remove the **ी** component from cell 2.

This is likewise the case with several of the other combined characters; place the cursor between cells 9 and 10, or between 11 and 12, for example to observe the same behavior.

85 For the meaning (and use) of Dead Keys, see the discussion on page 12.

86 See footnote 3 on page 3 for a link to this text in more than three hundred different languages.

87 See Database Design Note #1: “Character Codes & Character Cells” beginning on page 8. The difference is important.

## Arabic Script

The Arabic layout shown here is used for the section “Arabic Script examples using اللغة العربية ( the Arabic Language )” beginning on page 27.



Note that the paired delimiters, e.g. (, ), {, }, and [], (on the **(**, **)**, **D**, **F**, **C**, and **V** keys) are reversed on the layout to accommodate the right-to-left layout of the many languages that use Arabic Script. Arabic versions of the paired delimiters < and >, although they appear on the Latin **{** and **}** keys, are not reversed.

Regardless of the normal right-to-left text flow in languages using the Arabic Script<sup>88</sup>, Numeric Characters (0-9) are always laid out in left-to-right order.

### Modern Standard Arabic Typing Demonstration/Practice (no knowledge of Arabic required)

The box to the right contains the Arabic translation of Article 1 of the United Nations Universal Declaration of Human Rights<sup>89</sup>. Instructions for typing the first five words will help illustrate a few behavioral characteristics not covered earlier.

يولد جميع الناس أحراراً متساوين في الكرامة  
والحقوق. وقد وهبوا عقلاً وضميراً وعليهم أن يعامل  
بعضهم بعضاً بروح الإخاء

Our sample text, therefore, which you are encouraged to type using the proper Input Method Editor, is the following:

Figure 16a – Abbreviated Modern Arabic Text Sample

يولد جميع الناس أحراراً (literally “generates” (as in “birth”) “all” “people” “free”)

This segment consists of twenty-one typed characters, including spaces, occupying twenty-one character cells<sup>90</sup>:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
ي	و	ل	د		ج	م	ي	ع		ا	ل	ن	ا	س		أ	ح	ر	ا	ر
d	,	g	]	space	[	l	d	u	space	h	g	k	h	s	space	H	p	v	h	v
يولد					جميع					الناس					أحراراً					

88 As is the case with all commonly used right-to-left scripts.

89 See footnote 3 on page 3 for a link to this text in more than three hundred different languages.

90 See Database Design Note #1: “Character Codes & Character Cells” beginning on page 8. The difference is important.

## Hebrew Script

The Hebrew (lyx) keyboard layout shown here is used for the section “Hebrew Script examples using שפת עברית ( the Hebrew Language )” beginning on page 29.



In the Hebrew Lyx keyboard layout, Dead Keys<sup>91</sup> for all Diacritics are post-fix – i.e. they are typed after the character over which they are to be placed. Note that paired delimiters, e.g. (), {}, [], and <>, are reversed on Hebrew keyboard layouts to accommodate the right-to-left layout of Hebrew and other languages that use the Hebrew Script.<sup>92</sup> Regardless of Hebrew Script’s normal right-to-left text flow, numeric characters (0-9) are laid out in left-to-right order.

### Hebrew Typing Demonstration/Practice (no knowledge of Hebrew required)

The Hebrew translation of the first sentence in Article 1 of the United Nations Universal Declaration of Human Rights<sup>93</sup> is shown on the right. Instructions for typing the first five words will help illustrate a few of Hebrew’s behavioral characteristics.

כל בני האדם נולדו בני חורין ושווים בערכם  
ובזכויותיהם.

Figure 17a – Abbreviated Hebrew Text Sample

Our sample text, therefore, which you are encouraged to type using the proper Input Method Editor, is the following:

כל בני האדם נולדו בני (literally “all” “my son” “man” “were born” “my son”)

This segment consists of twenty-one typed characters, including spaces, occupying twenty-one character cells<sup>94</sup>:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
כ	ל		ב	נ	י		ה	א	ד	ם		נ	ו	ל	ד	ו		ב	נ	י
f	k	space	c	b	h	space	v	t	s	o	space	b	u	k	s	u	space	c	b	h
כל		בני				האדם					נולדו					בני				
כל בני האדם נולדו בני																				

Beware: the importance of correctly distinguishing between similar characters can easily be shown by substituting “x” (ס) for “o” (ם) in the third word, which alters the meaning from “All men were born...” to “All the sons of Hades were born...” – a somewhat embarrassing error!

91 For the meaning (and use) of Dead Keys, see the discussion on page 12.

92 See “Paired Symbols in Text with Mixed Directions” on page 17 for a more detailed discussion of Hebrew’s paired delimiters.

93 See footnote 3 on page 3 for a link to this text in more than three hundred different languages.

94 See Database Design Note #1: “Character Codes & Character Cells” beginning on page 8. The difference is important.



## Korean Script

The Korean Jamo layout shown here, also known as “han2(m17n)” layout, is used for the section “Korean Script examples using 한국어 ( the Korean Language )” beginning on page 30.



The keys on a Korean keyboard generate Jamo, “building block characters” that are then combined by the system into Hangeul syllables. These syllables are what is displayed as Korean text. Typing the English letters shown below when a Korean Input Method is active will provide an additional example of how Korean Hangeul text can be entered into a multi-lingual system by successively combining Jamo “characters” into Hangeul “syllables” and then forming words.

### Korean Typing Demonstration/Practice (no knowledge of Korean required)

The box to the right contains the beginning of the Korean version of the United Nations Universal Declaration of Human Rights.<sup>95</sup> Unlike many languages, there are generally no spaces between words. Spaces are used to indicate breaks between sentences. The four words therefore are the following:

모든 인간은 태어날 때부터 자유로  
우며 그 존엄과 권리에 ... continued

Figure 18a – Abbreviated Korean Text Sample

모든 인간은 태어날 때해터 (literally “All” “Human” “birth” “when”)

This segment consists of thirty-one typed Latin keys (including spaces) that occupy fifteen displayed character cells<sup>96</sup>:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
ㅏ	ㅑ	ㅓ	ㅕ	ㅗ		ㅇ	ㅣ	ㄹ	ㄷ	ㅌ	ㄱ	ㅋ	ㅇ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ	ㅍ
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15																
모	든		인	간	은		태	어	날		때	해	터																	
a	h	e	m	s	space	d	l	s	r	k	s	d	m	s	space	x	o	d	j	s	k	f	space	E	o	g	o	x	j	space
모든		인간은					태어날					때해터																		

Comparing the Jamo ㄱ (typed *character* 10) with its appearance in the combined Hangeul syllable 간 (upper left of *character cell* 5) also illustrates how the shapes of Jamo component characters are sometimes altered stylistically when combined with others to improve the aesthetics of the syllable and text as a whole. - 플

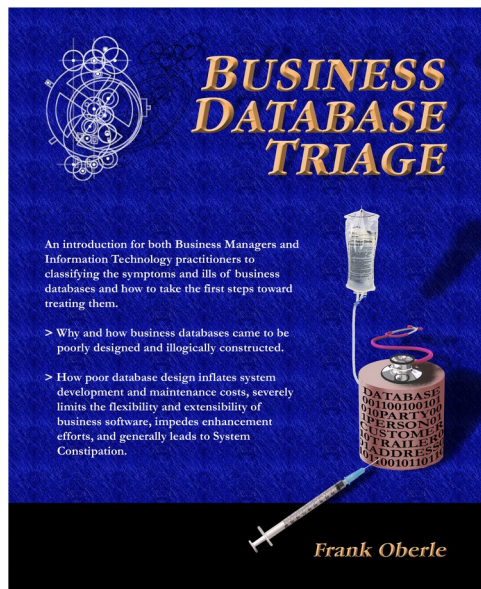
95 See footnote 3 on page 3 for a link to this text in more than three hundred different languages.

96 See Database Design Note #1: “Character Codes & Character Cells” beginning on page 8. The difference is important.

---

## Other Publications

# Antikythera Publications



[www.AntikytheraPubs.com](http://www.AntikytheraPubs.com)

In addition to an ongoing series of Database Design Notes, Antikythera Publications recently released the book “*Business Database Triage*” (ISBN-10: 0615916937) that demonstrates how commonly encountered business database designs often cause significant, although largely unrecognized, difficulties with the development and maintenance of application software. Examples in the book illustrate how some typical database designs impede the ability of software developers to respond to new business opportunities – a key requirement of most businesses.

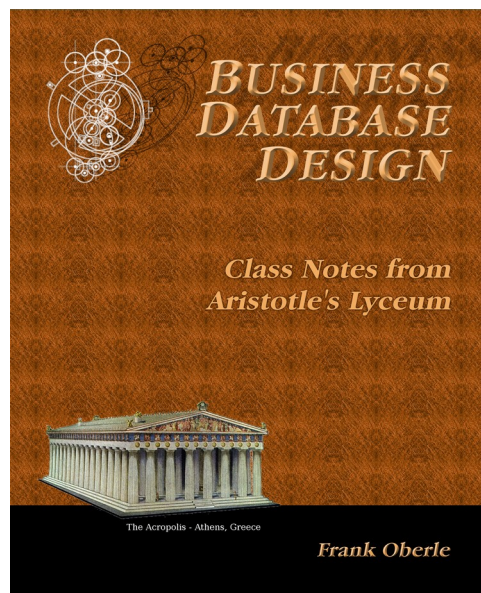
A number of examples of solutions to curing business system constipation are presented. Urban legends, such as the so-called object-relational impedance mismatch, are debunked – shown to be based mostly on illogical database (and sometimes object) designs.

“*Business Database Triage*” is available through major book retailers in most countries, or from the following on-line vendors, each of which has a full description of the book on their site:

CreateSpace: <https://www.createspace.com/4513537>

Amazon:

[www.amazon.com/Business-Database-Triage-Frank-Oberle/dp/0615916937](http://www.amazon.com/Business-Database-Triage-Frank-Oberle/dp/0615916937)



A follow-up book, “*Business Database Design – Class Notes from Aristotle’s Lyceum*” is due to be available for classroom use in late 2014.

“*Business Database Design*” leads the reader through the logical design and analysis techniques of data organization in more detail than the earlier work – which concentrated more on understanding and identifying problems caused by illogical database design rather than their solutions.

These logical approaches to data organization, espoused by Aristotle and an “A-List” of his successors, have formed the basis for scientific discovery over more than 2,400 years, and directly led to the technology we deal with today, notably including both relational and object theory.

“*Business Database Triage*” explained the reasons why these principles were virtually impossible to apply during the early years of our transition to the use of computers in business, but since the technology is now sufficiently mature that such compromises can no longer be justified, the time has come to relearn logical data organization techniques and apply them to our businesses.