

# Exploring Complex Text Layout

Frank Oberle; July, 2015

## PREFACE

This document is meant to be an informal – and therefore sometimes less than precise – introduction to what are commonly considered Complex Text attributes. In spite of the CTL acronym’s appearance in many pieces of software 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 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, 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. With generalized design made possible by advances in software technology and continued progress toward globalization, these distinctions can and should be abandoned.
- supporting background material for a document related to LibreOffice titled ‘Bugs-and-a-Horse.’

## REQUIREMENTS

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

- 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 31.
- An installed font containing the Unicode Basic Latin, Latin-1, Thai, Devanagari, Basic Hebrew, and Basic Arabic Code Blocks; the FreeSerif font<sup>2</sup> is recommended unless you know how to confirm the presence of the scripts listed above on your own.
- 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 source.

---

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

2 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.

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.

# TABLE OF CONTENTS

Exploring Complex Text Layout.....	1
Preface.....	1
Objectives.....	1
Requirements.....	1
Defining Complex Text Layout – What it is and what it isn’t!.....	4
FIGURE 1 – LANGUAGES DIALOG IN LIBREOFFICE WRITER.....	4
Role of System Components in Managing text Layouts.....	4
Characteristics of so-called Complex Text Layout (CTL).....	6
Characters and Character Cells.....	6
Alphabetic Characters.....	6
A IS FOR APPLE.....	6
Consonants and Vowels.....	6
Vowel Varieties.....	6
Characters as Diacritics.....	7
Tones, Accents, and Breathing Marks as Diacritics.....	7
Character Cells.....	7
Contextual Character Forms (Alternative Characters).....	7
Contextual Character Shaping (Shape Changing).....	8
Character Reordering and Placement.....	8
Illegal Character Combinations.....	9
Composite Characters.....	9
Ligatures (Composite Glyphs).....	9
Dead Keys.....	9
FIGURE 2 – DEAD KEYS ON A MANUAL THAI TYPEWRITER (CIRCA 1970).....	9
Text Layout Direction (Writing Mode).....	10
Perceived Cultural Issues.....	10
FIGURE 3 – BOUSTROPHEDON TEXT LAYOUT.....	11
FIGURE 4 – REVERSE BOUSTROPHEDON TEXT LAYOUT.....	11
FIGURE 5 – THE PHAESTOS DISK.....	11
Mixed Text Directions – Bidirectional Text.....	12
Default Paragraph Direction (Primary Text Direction) in Text with Mixed Directions.....	12
Cursor Movement when Entering or Editing Text in Bidirectional Paragraphs.....	12
Paired Symbols in Text with Mixed Directions.....	14
FIGURE 6 – ENGLISH (US) KEYBOARD LAYOUT SEGMENT.....	14
FIGURE 7 – HEBREW KEYBOARD LAYOUT SEGMENT.....	14
Rulers, Guides, and Tabs in Text with Mixed Directions.....	15
Left, Right, and Center Tab Stops.....	15
FIGURE 8 – LEFT-TO-RIGHT RULER WITH TAB SETTINGS.....	16
FIGURE 9 – RIGHT-TO-LEFT RULER WITH MIRRORED TAB SETTINGS.....	16

Decimal Tab Stops.....	16
FIGURE 10 – LEFT-TO-RIGHT RULER WITH A DECIMAL TAB SETTING.....	16
FIGURE 11 – MIRRORED RTL RULER WITH A DECIMAL TAB SETTING.....	16
Detecting Primary Text Direction in Paragraphs.....	17
Text Alignment in Documents with Mixed Directions.....	17
FIGURE 12 – LEFT ALIGN.....	17
FIGURE 13 – BIDIRECTIONAL.....	17
Transitioning between Text Directions within Paragraphs.....	17
Justification.....	18
Ragged Justification.....	18
Full Justification.....	18
Kashideh.....	18
Word Breaks, Line Breaks, and Hyphenation.....	19
Collation and Sorting.....	19
CTL Examples in Practice.....	20
Disclaimer.....	20
Thai Script examples using ภาษาไทย ( the Thai Language ).....	21
Brief Comments about Thai.....	21
Examples for Experimentation (No knowledge of Thai needed).....	21
FIGURE 14 – SAMPLE THAI TEXT BLOCK.....	22
Devanagari Script examples using हिन्दी भाषा ( the Hindi Language ).....	23
Brief Comments about Hindi.....	23
Examples for Experimentation (No knowledge of Hindi needed).....	23
FIGURE 15 – SAMPLE HINDI TEXT BLOCK.....	24
Arabic Script examples using اللغة العربية ( the Arabic Language ).....	24
Brief Comments about Arabic.....	24
Examples for Experimentation (No knowledge of Arabic needed).....	24
Kashideh Justification and emphasis.....	25
FIGURE 16 – SAMPLE ARABIC TEXT, INCLUDING KASHIDEH JUSTIFICATION.....	26
Hebrew Script examples using שפה עברית ( the Hebrew Language ).....	26
Brief Comments about Hebrew.....	26
Examples for Experimentation (No knowledge of Hebrew needed).....	26
FIGURE 17 – SAMPLE HEBREW TEXT BLOCKS WITH AND WITHOUT VOWEL INDICATORS.....	27
Transitioning between English and Hebrew in a Bidirectional Paragraph – an Example.....	27
Character Entry Methods.....	31
FIGURE 18 – “INSERT > SPECIAL CHARACTER...” DIALOG BOX IN LIBREOFFICE WRITER.....	31
FIGURE 19 – “ONBOARD” ON-SCREEN KEYBOARD – TYPICAL OF MANY AVAILABLE.....	32
UTF-8 (ISO-10646) Code Point Representations.....	33

## 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.

So what makes text layout “complex?” On a Microsoft FAQ page,<sup>4</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.

The LibreOffice Writer Guide<sup>5</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.

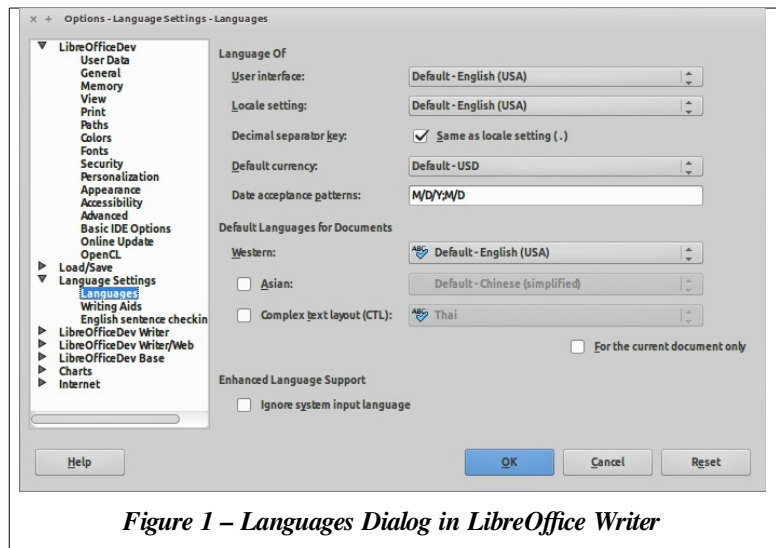


Figure 1 – Languages Dialog in LibreOffice Writer

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>6</sup> The first sentence in this quote is correct, but the second is 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 me appropriate.

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

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

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

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

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 and utilities, which serve to translate (map) any input key sequences from native button pushes<sup>7</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>8</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 discuss texts from the many languages that have been spoken by a user's ancestors. Enough said.

Early systems were quite limited in their text layout capabilities and generalization and internationalization of software designs was simply technically prohibitive and too expensive. As technology matured, handling of many characteristics discussed here were added in a totally 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.

So far, the recognition of language (as opposed to merely the writing system or script<sup>9</sup> used to represent it on screen or paper) remains the province of applications 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 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>10</sup> may likely always remain the province of specialized software.

---

7 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.

8 Well, most attempt to be, but the results can of course vary wildly.

9 See the accompanying document *Bugs-and-a-Horse.pdf* for a detailed discussion of the terms Language, Dialect, Country, Writing System, Script, Abjad, Abugida, Alphabet, Key Code, Scan Code, Character Code, Character Cell, Locale, Typeface, Glyph, and Font. Some of these terms will be used in this document; if any of their meanings is unclear, refer to that document.

10 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 they 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 20.

### CHARACTERS AND CHARACTER CELLS

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 other 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 paper 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.”



*A is for Apple*

Detection of alphabetic characters is key to handling multi- or bidirectional paragraph layouts in multilingual documents; an illustration of this is included in the section titled *Transitioning between English and Hebrew in a Bidirectional Paragraph – an Example* that begins on page 27.

#### 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>11</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>12</sup>

11 This is known as Contextual Shaping; see the eponymous section on page 8.

12 See the Thai and Hindi *Examples for Experimentation* beginning on pages 21 and 23 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 8.

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>13</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.

### Tones, Accents, 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>14</sup> We are just deprived of a way to write them!

### Character Cells

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.

### Contextual Character Forms (Alternative Characters)

Character Form 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>15</sup>

When discussing text layout, Character Forms are actual differences in a character based 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.

Not all scripts have contextual character forms, and those that do might have few or many examples. The

---

13 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.

14 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.

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

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>16</sup> While these are both the same character in the Greek alphabet,<sup>17</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.

### 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 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  $\b{}$  (U+FB54) in its initial form,  $\b{}$  (U+FB55) in its medial form, and  $\b{}$  (U+FB53)<sup>18</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>19</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 18 and in the reference given in footnote 35.

Symbols in Boustrophedon writing systems (see discussion on page 11) also make use of different forms.

### Character Reordering and Placement

Despite the previously mentioned discrimination against the lowly vowel classes in some cultures,<sup>20</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 23 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 is done transparently by the 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 ไท้บ 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 19.

---

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

17 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.

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

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

20 See *Vowel Varieties* on page 6.



## 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>21</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 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>22</sup> In Thai, for example, while the consonant `๒` and vowel `๑` are considered individual letters, each of which is stored separately on disk, they are displayed in one character cell as a composite character `๒๑`.

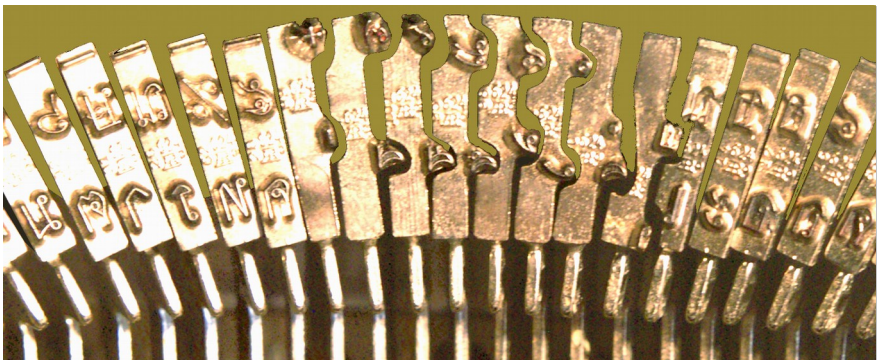
Some implications of these differences will be discussed in “Cursor Movement and Editing Keys” below.

## 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 alphabet, while an n+~ ligature, such as ñ, is an actual alphabetic character. 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 older manual 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



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

<sup>21</sup> See the reference in footnote 4.

<sup>22</sup> In the reference cited in footnote 4, the Microsoft FAQ refers to such character cells containing multiple elements as “piles.”

the platen roller and paper would not move when the key was struck. Hence, the name “Dead Key.” In the segment of a Thai keyboard shown, 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, 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 diacritics needed 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>23</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.

Given the current state of the art, there should be no need whatever for a user to explicitly select right-to-left or any

### 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? These two questions have no good answer.

And why, as some wonder, do writers using English and other western scripts get the prime real estate (the seven bit codes) in 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 is affectionately known as lower ASCII.

This is why even nationally sanctioned “official” fonts 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 insure 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.

23 The 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.

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>24</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. RLTB layouts 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 alternate 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

This is several lines of text written in a form of text layout known as Boustrophedon. The lines begin running left to right, but continue reversing direction with each successive line of text. Even the characters are reversed!

*Figure 3 – Boustrophedon Text Layout*

Boustrophedon layout with often obscure character sets to write scholarly articles about – what else? – ancient Boustrophedon writings in Safaitic, Sabaeen, early Greek & Latin. The undeciphered Rongorongo inscriptions found on Easter Island are another, but unrelated, example of a Boustrophedon layout.

Reverse Boustrophedon, simulated on the right, is quite similar, but the characters in the alternating lines of text are inverted as well as reversed. The etymology of the term Boustrophedon derives from

This is several lines of text written in a form of text layout known as Reverse Boustrophedon. The lines begin running left to right, but continue reversing direction with each successive line of text. Even the characters are reversed!

*Figure 4 – Reverse Boustrophedon Text Layout*

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 be, supported by anything other than graphics software.

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 its 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.



*Figure 5 – The Phaestos Disk*

24 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 Direction (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 default to the document's default language.

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.<sup>25</sup> 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.

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 *Transitioning between English and Hebrew in a Bidirectional Paragraph – an Example* on page 27, which includes a step by step review of transitions in either direction to illustrate both proper and improper cursor behavior.

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 16.

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.

25 The Chinese text is from the opening of the United Nations Universal Declaration of Human Rights. See Footnote 3.


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





**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.
	in RLTB Layouts	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. 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, that users’ habits have become stratified. That’s just how it’s done and, like the pronunciations given in footnote 14 (see page 7), these likely won’t ever change. Implementation of Tab behavior in right-to-left text is seldom seen in any but very specialized applications, however. If such support is implemented in more mainstream applications, it should likely be accompanied by an option to reverse the behavior of the left and right cursor keys as well for consistency. See *Rulers, Guides, and Tabs in Text with Mixed Directions* on page 15 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>26</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>27</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.

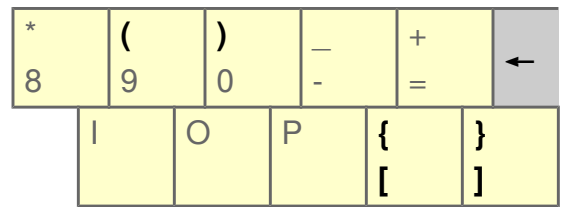






Figure 6 – English (US) Keyboard Layout Segment

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

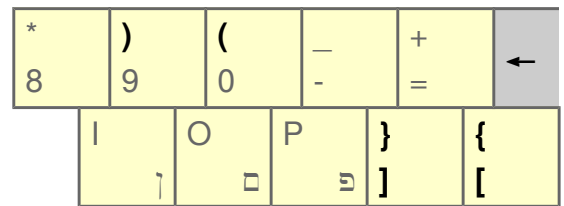


Figure 7 – Hebrew Keyboard Layout Segment



26 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.

27 Many languages also share these and other paired symbols, but place them in different keyboard locations.

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.


This difference in perspective is not reflected in all right-to-left languages keyboards, however; some Arabic keyboards have the paired delimiters in the same locations as on Latin keyboards.<sup>28</sup> This perhaps reflects the wide variety of languages that use the Arabic script. Choose keyboard layouts wisely!

Quotation marks, another common symbol pair, appear in many interesting varieties. What the French call *guillemets* (« and »), are used in many languages.<sup>29</sup> Danish and Hungarian writing uses 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>30</sup>

In order to avoid headaches, we’ll ignore any discussion of how paired character matching algorithms<sup>31</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  $\perp$  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  $\lrcorner$  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  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>32</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:

- 
- 28 Both forms can be found on Amazon, for instance, but there is no mention in their descriptions that this difference exists.
  - 29 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.
  - 30 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.
  - 31 e.g. where placing the cursor on one symbol of a pair causes its twin to be indicated in some fashion.
  - 32 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.

- 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.

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.

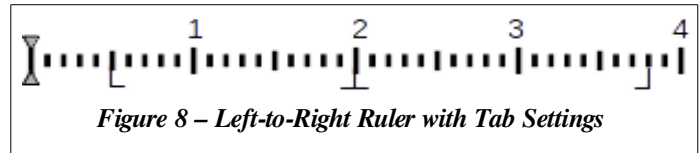


Figure 8 – Left-to-Right Ruler with Tab Settings

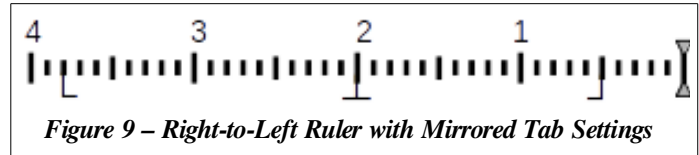


Figure 9 – Right-to-Left Ruler with Mirrored Tab Settings

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>33</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.

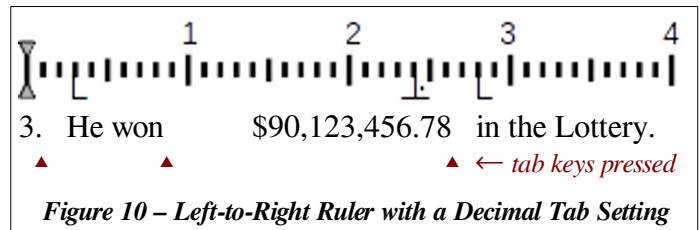


Figure 10 – Left-to-Right Ruler with a Decimal Tab Setting

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.

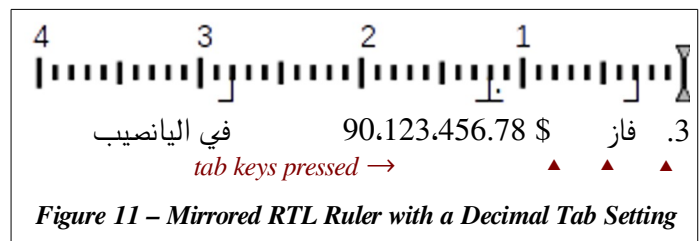


Figure 11 – Mirrored RTL Ruler with a Decimal Tab Setting

Assuming that the values expected to be displayed in the mirrored layouts are similar in terms of the

<sup>33</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.



quantity of digits expected before and after the decimal point,<sup>34</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.

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.



Figure 12 – Left Align

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.



Figure 13 – Bidirectional

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.

### Transitioning between Text Directions within Paragraphs

Smooth transitions for users entering text in bidirectional paragraphs can be disrupted when the current paragraph script is detected improperly. A detailed example of how this occurs is presented beginning on page 27 in *Transitioning between English and Hebrew in a Bidirectional Paragraph – an Example*.

---

34 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 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).

## 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 are usually 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 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<sup>35</sup> the ending margin. This can become problematic with languages that don’t delimit words with spaces, although there are techniques for overcoming this.

Slightly more pleasing<sup>36</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 of why there is a distinction between characters and character cells. Of course, spaces that would otherwise appear at the beginning or ending of lines need to be suppressed in any justified output.

Another technique involves subtle<sup>37</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 35 for one interesting link.
- Haralambous and Bella: 2006;
- Elyaakoubi and Lazrek: 2010;<sup>38</sup>

### Kashideh

An interesting alternate technique for obtaining full justification is known by the Arabic term Kashideh. 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 – Sample Arabic Text, including Kashideh Justification* on page 26 for a

---

35 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.

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

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

38 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).

detailed comparison of ragged right and Kashideh justification.

Perhaps this form of justification actually 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 – Sample Thai Text Block* on page 22. Observe that the first space is only seen between the words อีสระ and เรา about two-thirds of the way into the first line, which is the break between the first two sentences.<sup>39</sup> The detection of the line break causing the large space at the end of the first line occurs after the whole word ี่, since the following word ความ 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 call 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 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 and 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>40</sup> In Thai, the situation is reversed; the vowels (ะ ั ิ ี ึ ุ ู ุ่ ุ๊ ุ๋ ุ่ ุ๊ ุ๋ and ฤษ) are grouped together at the end of the alphabet, but the sort order considers each vowel after every consonant.<sup>41</sup>

The next section will present examples of CTL attributes in the context of actual scripts and languages.

---

39 For the curious, the word เรา that appears at the beginning of all three sentences in this selection translates to “we.”

40 It's not quite that simple in practice, but this document is intended only as an introduction.

41 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.

## 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 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. 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>42</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 “i̋” in column 1, for instance, is typed as “[bj]”. This, of course, requires that the active font contains the Thai script. See footnote 2.

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 31.

---

42 See the section *UTF-8 (ISO-10646) Code Point Representations* on page 33 for details of UTF representations.

# Thai Script examples using ภาษาไทย ( the Thai Language )

## BRIEF COMMENTS ABOUT THAI

Examples in the Thai Language – ภาษาไทย

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>43</sup> Thai is written from left-to-right, and top-to-bottom (LRTB), and has no case differences; 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 and advertising.

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. 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>44</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 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>45</sup>	Mai-ek Tone	“i” vowel	Baw (leaf)	Bpaw (fish)	“i” vowel	Mai-ek Tone	Vowel + Tone
[ 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	<b>3610d</b>	<b>3611d</b>	3611d, 3636d	3611d, 3656d	3611d, 3636d, 3656d

The ๒ and ๓ characters appear again in columns 3 and 6 with the diacritic vowel ิ, one of several that are positioned above their associated consonant. Note that in column 3, the vowel is in its normal position. In

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

44 2538 is the Thai year corresponding to 1995 in most calendars.

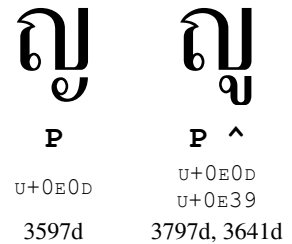
45 This is an example of diacritic ordering or ranking. In Thai, a diacritic vowel “outranks” a tone mark, and is thus placed closer to the base character. Very few applications correct such entry errors.

column 6, however, it is shifted to the left to accommodate the ๗ character’s right side ascender.<sup>46</sup>

In columns 2 and 7 respectively, the Mai-ek, 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 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 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 ุ in the second word (ทุก – the 4<sup>th</sup> and 5<sup>th</sup> character cells) highlighted in Figure 14. In the illustration to the right, a similar vowel ู is shown paired with the Yaw consonant ญ to illustrate another interesting layout convention. The ญ character itself appears to be composite, since its lower element is separated,<sup>47</sup> but isn’t. When combined with the lower diacritic vowel ู however, the lower element is simply replaced in the cell.



At least in Linux operating systems, the Thai contextual diacritic placements and shape changing described in this section are provided by the operating system or input method 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 typed *after* the consonants<sup>48</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>49</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 ใ in the word ไทย (“Thai”) are placed before the consonant – in this case, the ๓ ( $\approx$  the Latin t) – with which they are paired, while others like ะ 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 เราทุกคนมี  
ความคิดและความเข้าใจเป็นของเราเอง เรา  
ทุกคนควรได้รับการปฏิบัติในทางเดียวกัน.

Figure 14 – Sample Thai Text Block

Cursor and editing keys in Thai (arrows, backspace, delete, etc.) operate on character cells rather than individual characters, but many applications permit modifier keys to cause them to work on characters.

46 As a matter of interest, the word ฝ 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.

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

48 See Figure 2 on page 9 for an illustration of how dead keys function on manual Thai typewriters.

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

# Devanagari Script examples using हिन्दी भाषा ( the Hindi Language )

## BRIEF COMMENTS ABOUT HINDI

Examples in the Hindi Language – हिन्दी भाषा

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, 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.

The Devanagari script is defined in Unicode blocks U+0900-097F and U+A8E0-A8FF,<sup>50</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 languages with which it is used. 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 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 “Hindi (Bolnagri),” one of several phonetic keyboards used to type Devanagari script using Latin letters that produce similar sounds. 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 in the earlier Thai example.

This example illustrates a two word, ten character sequence हिन्दी 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	ि	न	द	ी		भ	ा	ष	ा
h	i <sup>51</sup>	n	d	I	space	B	a	S	a
0x939	0x93f	0x928	0x922	0x940	0x20	0x92d	0x93e	0x937	0x93e
2361d	2367d	2344d	2338d	2368d	32d	2349d	2366d	2359d	2366d
ह	हि	हिन	हिनद	हिन्दी		भ	भा	भाष	भाषा

If the “Hindi (Bolnagri)” keyboard is not available, “Hindi (Wx)” can be used, but the Latin letters above should be replaced by **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 we saw with Thai, there is no concept of upper and lower case with Devanagari script and, when shifted, the same keys produce entirely different and unrelated characters.

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

51 The dotted circle is not part of the Devanagari characters; it is a convention used to indicate their relative placement for vowels that cannot or do not stand on their own. Some languages, such as Thai, whose vowels are often used in syllables without consonants, explicitly pair such vowels with a “holder” consonant (the ँ), so the circle convention isn’t seen as often.

Devanagari fonts without all the appropriate ligatures will still produce readable text,<sup>52</sup> but for extended use of any particular language in these families, the presence of any desired ligatures should be confirmed.

The first two characters are an example of Character Reordering, introduced on page 8. When the vowel ि is entered after the consonant ह, it is placed to the left of it in a manner similar to what happens 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 the other examples used throughout this document.

Etymology buffs might note the similarity between the Hindi (हिन्दी) and Thai (ภาษา) 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>53</sup> defined in Block U+0600-06FF as well as several supplemental blocks.<sup>54</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>55</sup> or variants (e.g. Algerian, Egyptian, Lebanese, Moroccan, and Syrian), and less related languages such as Äynu, Azeri, Baluchi, Beja, Bosnian, Brahui, Chechen, Crimean Tatar, Dari, Gilaki, Hausa, Kabyle, Karakalpak, Konkani, Kashmiri, Kazakh, Khowar, Kurdish, Kyrgyz, 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

Arabic script includes a set of digits – the 0 to 9 equivalents<sup>56</sup> are: ٠ ١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩, although the shared numerals defined in the Latin code block are 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.

52 The 3<sup>rd</sup> and 4<sup>th</sup> characters above combine to display the word as हिन्दी, but the uncombined form हिनदी is still understood.

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

54 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.

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

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



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.

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	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 right of the character instead of its left. 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 section.

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>57</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, 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 26.

### **KASHIDEH JUSTIFICATION AND EMPHASIS**

An interesting use of Contextual Shaping in certain Arabic and 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.<sup>58</sup>

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

58 See footnote 54 for further information.

To the right are two Arabic renditions of Article 1 from the Universal Declaration of Human Rights used throughout this document.

The top example shows the text in its basic right-to-left, flush right, ragged left layout.

The lower sample displays the same text fully justified using the Kashideh layout technique.

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.

يولد جميع الناس أحراراً وملتساوين  
في الكرامة والحقوق. وهم قد وهبوا  
العقل والوجدان و عليهم أن يعاملوا  
بعضهم بعضا بروح الإخاء.

يولد جميع الناس أحراراً وملتساوين  
في الكرامة والحقوق. وهم قد وهبوا  
العقل والوجدان و عليهم أن يعاملوا  
بعضهم بعضا بروح الإخاء.

Figure 16 – Sample Arabic Text, including Kashideh Justification

## 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>59</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>60</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 upper and lower case, Hebrew text, unlike most non-Latin scripts, does have both serif and sans-serif typeface designs.

### EXAMPLES FOR EXPERIMENTATION (NO KNOWLEDGE OF HEBREW NEEDED)

The keyboard mapping used for these examples is a very basic “Hebrew” 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.

59 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.

60 See “Paired Symbols in Text with Mixed Directions” on page 14.

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	עכרית

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 only issue is the treatment of the space character which, as with the Arabic script discussed earlier, is not a Hebrew-specific character, but one that also appears in many scripts, both LRTB and RLTB. The implications of this symbol sharing are discussed after presenting a sample paragraph of Hebrew.

Figure 14 below presents the United Nations Universal Declaration of Human Rights Article 1 (see footnote 3) in Hebrew for comparison to other examples used earlier. In this example, however, the text is shown in versions with and without the inclusion of Hebrew’s optional diacritic vowels:

כל בני האדם נולדו חורין ושוים בערךם ובזכויותיהם. כלם חוננו בתבונה ובמצפון, לפיכך חובה עליהם לנהוג איש ברעהו ברוח של אחווה.	כל בני האדם נולדו בני חורין ושווים בערכם ובזכויותיהם. כולם חוננו בתבונה ובמצפון, לפיכך חובה עליהם לנהוג איש ברעהו ברוח של אחווה.
--	--

Figure 17 – Sample Hebrew Text Blocks with and without Vowel indicators

### TRANSITIONING BETWEEN ENGLISH AND HEBREW IN A BIDIRECTIONAL PARAGRAPH – AN EXAMPLE

A number of behaviors change when bidirectional text is included in a single paragraph. These include text selection, the behavior of cursor movement (e.g. ← and →) and editing keys (e.g. ←Backspace and Delete), as well as how a particular application reacts during the transition process itself.

Although the differences in these behaviors may be relatively obvious when the cursor is within a block of text having the same directionality, they can be mystifying at the transition points. The following line will illustrate the transition between left-to-right and right-to-left scripts that appear in bidirectional paragraphs:

This is English text and שפת עברית is Hebrew text.

The highlighted text contains sixteen characters in sixteen character cells, stored in memory and on disk as twenty-four bytes. Examining just this segment can illustrate how many issues involving bidirectional text are handled, although many applications leave something to be desired in their implementations.

A summary of these sixteen character cells is shown below with their entry and disk storage order indicated in the top row and the display order in the row below:

Character Entry and Disk Storage Order:	1	2	3	4	13	12	11	10	9	8	7	6	5	14	15	16
Character Display Order:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Actual Character Display:	a	n	d		ת	י	ר	כ	ע		ת	פ	ש		i	s

The Space character is one of several symbols that are shared across multiple scripts, and so might seem to be an easy character to handle properly in bidirectional text such as this example. Using some applications, however, suggests that this apparently not the case.

Intelligent handling of cursor movement and various selection and editing actions, for instance, requires the computer to know whether each space is logically part of a left-to-right segment or right-to-left segment. To show in detail how this occurs, we'll examine how applications should react to each key stroke as a user enters this short sample.

Assuming that the user is typing in a paragraph having a primary text direction of left-to-right and is using an Input Method for Hebrew, the text sample may be entered using any of the following key sequences:

... with the keyboard switching done after each space is entered, the keystrokes might be similar to:

a n d space #1 keyboard switch a p , space #2 g f r h , space #3 keyboard switch i s

... with the keyboard switching initiated before each space is entered, the sequence would be:

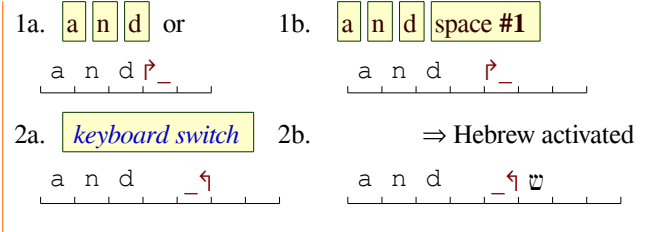
a n d keyboard switch space #1 a p , space #2 g f r h , keyboard switch space #3 i s

Or, if the text is entered on a character by character basis, using either single character insertion utilities or direct Unicode character techniques,<sup>61</sup> the actual Hebrew characters could also be entered directly:

a n d space #1 ש ת פ ט space #2 ת י ר כ ע space #3 i s

The keyboard switch symbol indicates whatever action the user takes to change from the Latin to a Hebrew<sup>62</sup> keyboard layout. Although shown here as a key press, this may involve no keystrokes at all if, for instance, a mouse was used; conversely, this keyboard switching may involve multiple keystrokes.

In the illustration to the right, the state of the logical displayed character cells is shown below the key strokes for each step. The  $\overset{\curvearrowright}{\_}$  and  $\overset{\curvearrowleft}{\_}$  symbols mimic the cursors used by many applications to indicate how the next typed character will be treated – i.e. as part of either an LRTB ( $\overset{\curvearrowright}{\_}$ ) or RLTB ( $\overset{\curvearrowleft}{\_}$ ) sequence.



Because of their role in predicting behavior of cursor movement and editing keys, consistent behavior of these cursors is important for usability.<sup>63</sup>

Step 1 shows the key strokes used for entry of the first word, which may or may not include the space character. In either case, the initial Latin “a” sets the default paragraph direction to left-to-right (LRTB) if it wasn’t already set that way. If the space was included (as in 1b), it is considered Latin, since there was nothing to change the default.

The user changes the paragraph direction from its default LTR layout to RTL by either selecting a new keyboard layout (2a), or by entering a Hebrew character (2b). Selection of a new keyboard layout is a change to another writing system and, by implication, a different language. It should therefore always insert a space between the two sections; if the user didn’t explicitly enter a space (as in example 1b) one will be added. The user should, of course, be able to override this behavior.

61 See “Character Entry Methods” on page 31 for more information.

62 Although Hebrew is used in this example, the same issues are relevant regardless of the non-default paragraph script in use.

63 In paragraphs containing bidirectional scripts, most applications indicate the direction of the current text flow by using such modified cursors – usually with a flag or arrow at the top to show where the next character typed will be placed.

In either case, the result will be that shown in step 2a or, if the user caused the script change by entering the Hebrew װ character, the result will be as shown in 2b.

Once the new script (Hebrew in this example) becomes active, the cursor will be placed at the entry to the fifth character cell regardless. The only difference is that, in 2a, it is ready to receive the first Hebrew character, while in 2b, it is ready to receive a new character (which doesn't necessarily need to be Hebrew of course). In either case, the cursor shows the text direction is leftwards, as expected from an RTL script.

Steps 3a and 3b show the entry of the remaining letters of the Hebrew word תפח, which should leave the cursor in the same static position, and awaiting what is assumed to be another Hebrew character.

When the second space is entered, the application should still consider it to be laid out as if it were right-to-left Hebrew text since nothing was done to cause a return to the paragraph's default LTR layout. The expected and only reasonable behavior is shown in Step 4a to the right.

One of the more annoying user interface errors that appears in some applications purporting to support bidirectional text entry is illustrated as 4.x on the right. This is almost certainly the result of considering *any* symbol from the Latin Unicode block to indicate a return to the default left-to-right layout.

Step 5 illustrates the state of the cursor after the entry of ן, the first letter in the second Hebrew word. Once the ן character has been entered, the aberrant cursor from step 4x resumes its correct behavior after confusing the user with its psychotic hopping around.

The second Hebrew word is shown completed in step 6; the cursor remains in the same location, with the system prepared to continue with entry of additional characters from the Hebrew script. As in step 4, the entry of the following space should be handled as shown in step 7, but the same interface errors seen in step 4x appear in applications that assume Latin code points are always equivalent to Latin characters.

Returning the paragraph to its default left-to-right direction should result in the display shown in step 8.

The transition from the non-default layout (in this case, the RTL Hebrew) back to the default (the LTR

English) can be accomplished either explicitly, by switching the active keyboard or implicitly, by entering an alphabetical character from the default paragraph script – in this case English/Latin.

Any transition from a non-default layout to the default layout (regardless of the direction of either Script) must, like its opposite counterpart above, insure that at least one (but only one) space separates different scripts. When going from the non-default to the default, however, it is more often the case that spaces need to be removed or relocated.

Note that the next potential cell (the fifth cell) in step 7 as well as the space in the sixth cell no longer appear in step 8, having been rearranged so that the potential cell was removed and the space following the Hebrew text was relocated to the fourteenth cell. The fifteenth cell then becomes the next potential cell.

3a. [a] [n] [d] [ ] (also 3b.) 3c. [ ] [ח] [פ] [ת] ⇒ Hebrew active  
 a n d ת פ ח

4a. [space #2] – correct user interface behavior.  
 a n d ת פ ח

4x. [space #2] – illustrating a common user interface error!  
 a n d ת פ ח

5. [g]  
 a n d ת פ ח

6. [f] [r] [h] [ ]  
 a n d ת ר כ ע ת פ ח

7. [Space #3]  
 a n d ת ר כ ע ת פ ח

8. [keyboard switch] ⇒ English activated  
 a n d ת ר כ ע ת פ ח

The remaining characters from our short sample of bidirectional text are shown sequentially in steps 9 and 10 to the right. In most applications that have even minimal support for bidirectional text and use alternate cursors as usability aids, these cursors appear everywhere within a bidirectional paragraph, but aren't used otherwise.



Be aware that some sources refer to what I call “shared characters” as “neutral characters” and consider alphabetic to be inherently left-to-right or right-to-left.<sup>64</sup>

This completes the CTL Examples in Practice section. The remainder is a brief overview of methods used to enter characters that don't appear on a keyboard or are cumbersome to enter from the keyboard currently being used or emulated.

---

64 See an alternative description of Hebrew right-to-left text layout at [http://dotancohen.com/howto/rtl\\_right\\_to\\_left.html](http://dotancohen.com/howto/rtl_right_to_left.html) that uses this terminology. It also discusses the various Unicode non-printing symbols that can be used to explicitly set text direction (U+200E and U+200F) as well as other related characters, all of which I believe should only be used for backwards compatibility with older applications.

## 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 categories:

- Utilizing a character insertion map provided by either an operating system, an Input Method, or by the application in use. The one from LibreOffice Writer below is typical.

If there is only an occasional need to enter a non-standard character for the language in use, Windows and most flavors of Linux, for instance, have operating system level character map utilities, as do many office applications such as Microsoft Word and Excel, LibreOffice Writer, and so forth.

The best of these permit directly choosing a font

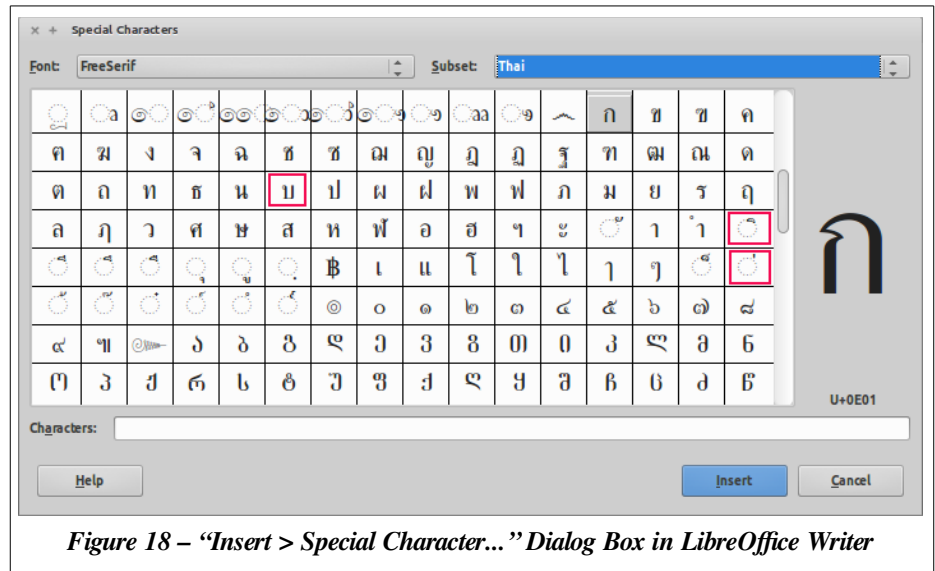


Figure 18 – “Insert > Special Character...” Dialog Box in LibreOffice Writer

as well as a Unicode Block 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.

In *Examples for Experimentation (No knowledge of Thai needed)* on page 21, the character combination ศ็ was shown as part of the first example. The dialog box shown in Figure 17 can be used to enter this combination by selecting the Thai script and then clicking the cells outlined in top to bottom order, followed by the Insert button. Most character insertion maps operate almost identically.

- 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, 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 ศ (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.

- 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.

- Activating some form of **Input Method** and selecting the particular script selected.

For entry of anything beyond a few characters, this is the most efficient method available, and any operating system offers at least one form of Input Method. In Linux, for instance, iBus seems to be the most common input method currently in use. 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, and the “Hindi Bolnagri” keyboard<sup>65</sup> selected, pressing the **Shift** + **b** keys will type the Devanagari character ऋ.

- 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 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>66</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>67</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 various permissible combinations of “a” with various diacritics, such as á, à, â, ä, å, ã, ã, ã, ã, ã, ã, ã, ã, ã, and æ. Some maps are provided by Desktop Managers rather than their underlying operating systems. Like many such utilities, the Onboard keyboard can be extensively configured.

65 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 23.

66 This illustration shows the Onboard utility with the Model-M Theme running on Ubuntu 14.04. The optional language selector has not been activated, so its behavior is to reflect whatever Input Method keyboard mapping is active.

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



## UTF-8 (ISO-10646) Code Point Representations

Unicode provides individual code points, or values, to represent up to 1,114,112 unique symbols used to write all of the world's present and past languages. Not all of these have yet been assigned of course, but for any language likely to be used in today's computer applications, this is quite sufficient.

The number 1,114,112 can be represented by the binary sequence 10001000000000000000, a series of twenty-one bits. Because computers generally store numbers in bytes – sequences of eight bits – this series of bits can be represented as bytes by adding three leading *zero* bits to give twenty-four bits, or an even three bytes as follows: 00010001 00000000 00000000. In hexadecimal notation that would be 0x110000.

If all of our data storage and transmission were simply converted from the one-byte-per-character many of the early adopters of computer technology used to three-bytes-per-character representations, text in all the world's languages could be reliably exchanged, right?

A little thought will show that data reliability easily disappears in most circumstances. If transmissions are interrupted, which bytes in the sequence are the first of the three used for each symbol? It might appear that, since we've arbitrarily added some leading *zeroes*, we could perhaps use those positions to indicate the leading byte – perhaps by making that byte begin with a 1 rather than a 0. We won't go into detail here, but some further thought will indicate that there is no guarantee at all that some of the second and third bytes might not also legitimately begin with a 1. Byte streams could easily be completely misinterpreted!

And then there is the issue of file size. Since the dawn of the computer age, most data files were produced by countries who concerned themselves with only Latin symbols and their own character sets, for which one-byte-per-character was often sufficient. A conversion to three-byte representations would result therefore in a significant portion of the world's data files immediately tripling in size; coupled with the higher error rates and resulting repeat transmissions due to the drop in reliable identification of the characters simply wouldn't do, even if supporting the Unicode set was becoming more of a necessity.

Covering the rocky path to a reasonable solution is beyond the scope of this document but, in brief, a number of attempts were made to settle on some ways to “transform” the Unicode bit sequences into patterns that could be reliably distinguished; these eventually became Unicode Transformation Formats, the three most prominent of which were:

- UTF-32: a fixed width format in which each symbol is stored as a four byte (32 bit) unit; UTF-32 is essentially the natural twenty-two bit sequence described above extended to cover standard storage sizes. Thus, all the issues discussed earlier are relevant. This format is also known as UCS-4, although UTF-32 is actually a subset of UCS-4.<sup>68</sup>
- UTF-16: a variable length format where each symbol is stored as either one or two double-byte (16 bit) units, and is therefore either 16 or 32 bits wide; although derived from the earlier UCS-2, it is not the same. Many programming language libraries continue to use UTF-16 for internal storage, although that is slowly changing. And, then, finally came ...
- UTF-8: a variable length format where each symbol is stored as anywhere from one to four bytes (8 to 32 bits); although this might initially seem unwieldy from programming or data transmission standpoints, it is becoming recognized as the most sensible compromise between efficiency and size.

Essentially, the reduction from -32 to -8 represented an improvement in granularity. The remainder of this section will be devoted solely to examples of how UTF-8 layouts correspond to Unicode point values.

---

<sup>68</sup> The acronym UCS means Universal Character Set, and refers to the initial attempts to represent all the Unicode code points.

The “8” in the UTF-8 designation does not indicate that each character consists of eight bits (one byte), but rather that each character consists of exact multiples of eight bits, i.e. complete bytes. A UTF-8 character, as mentioned earlier, may consist of one to four bytes.

A single byte UTF-8 code point (character or symbol identifier) representation always begins with a binary *zero*. The corollary is that any byte beginning with a *zero* bit represents a single Unicode symbol. These single byte characters are direct descendants of what used to be termed “lower ASCII,” and several of the non-alphabetic characters in this range are shared among many of the world’s scripts.

The first byte of each multi-byte UTF-8 code point representation begins with one, two or three *one* bits in sequence followed by a *zero*. The number of *one* bits indicates the total number of bytes that form the character – thus a byte beginning with three *one* characters followed by a *zero* indicates the beginning of a three byte sequence, and that two more bytes will follow to complete the representation of the code point.

Any byte beginning with a *one-zero* sequence is a continuation byte, and only has meaning when considered with an initial byte, i.e. a byte beginning with two or more *one* bits.

The table below illustrates the construction and range of Unicode values that can be represented by the one, two, three, and four byte Unicode transformation layouts:

Minimum Value		Maximum Value	
<b>Single Byte UTF-8 Representations (7 content bits): Decimal values from 32 through 127 (96 code points)</b>			
Unicode Binary:	0010.0000	Unicode Binary:	0111.1111
Unicode Hexadecimal:	0x20	Unicode Hexadecimal:	0x7f
Unicode Decimal:	32	Unicode Decimal:	127
UTF-8 Binary:	0010.0000	UTF-8 Binary:	0111.1111
UTF-8 Hexadecimal:	0x20	UTF-8 Hexadecimal:	0x7f
UTF-8 Decimal:	32	UTF-8 Decimal:	127
<b>Two Byte UTF-8 Representations (11 content bits): Decimal values from 128 through 2,047 (1,920 code points)</b>			
Unicode Binary:	0000.0000 1000.0000	Unicode Binary:	0000.0111.1111.1111
Unicode Hexadecimal:	0x0080	Unicode Hexadecimal:	0x07ff
Unicode Decimal:	128	Unicode Decimal:	2,047
UTF-8 Binary:	1100.0010 1000.0000	UTF-8 Binary:	1101.1111 1011.1111
UTF-8 Hexadecimal:	0xc280	UTF-8 Hexadecimal:	0xdfbf
UTF-8 Decimal:	49,792	UTF-8 Decimal:	57,279
<b>Three Byte UTF-8 Representations (16 content bits): Decimal values from 2,048 through 65,535 (63,488 code points)</b>			
Unicode Binary:	0000.1000 0000.0000	Unicode Binary:	1111.1111 1111.1111
Unicode Hexadecimal:	0x0800	Unicode Hexadecimal:	0xffff
Unicode Decimal:	2,048	Unicode Decimal:	65,535
UTF-8 Binary:	1110.0000 1010.0000 1000.0000	UTF-8 Binary:	1110.1111 1011.1111 1011.1111
UTF-8 Hexadecimal:	0xe0a080	UTF-8 Hexadecimal:	0xefbfbf
UTF-8 Decimal:	14,721,152	UTF-8 Decimal:	15,712,191
<b>Four Byte UTF-8 Representations (21 content bits): Decimal values from 65,536 through 1,114,112 (1,048,577 code points)</b>			
Unicode Binary:	0000.0001 0000.0000 0000.0000	Unicode Binary:	0001.0001 0000.0000 0000.0000
Unicode Hexadecimal:	0x010000	Unicode Hexadecimal:	0x110000
Unicode Decimal:	65,536	Unicode Decimal:	1,114,112
UTF-8 Bin:	1111.0000 1001.0000 1000.0000 1000.0000	UTF-8 Bin:	1111.0100 1001.0000 1000.0000 1000.0000
UTF-8 Hexadecimal:	0xf0000000	UTF-8 Hexadecimal:	0xf4908080
UTF-8 Decimal:	4,026,531,840	UTF-8 Decimal:	4,103,110,784

Values below 32 in the single byte representations remain reserved – as they have always been – for control codes such as STX and ETX, the line feed and carriage return, although the code to ring the teletype’s bell (0x07) to signify an incoming message hasn’t been used for many generations.<sup>69</sup> The number of possible

69 Sort of a precursor to AOL’s famous “You’ve got mail” message. The bell and clatter of an old Kleinschmidt teletype machine engendered a greater sense of connection with the world than anything Facebook or Twitter provide!

code points listed for each section does not imply or suggest that all such positions (e.g.  $0 \times 7F$ ) are valid.

In order to clarify the Unicode transformations and illustrate one drawback of the UTF-8 format, the following charts will provide specific examples of UTF-8 representations for one, two, three, and four byte formats using several scripts, two of which have already been introduced in this document.

The mandatory bit values for UTF-8 encoding are highlighted like this: **0** – the remaining bits form what is known as the “payload” for the byte, i.e. the bits available for storing the actual Unicode values.

**One byte UTF-8 Representation:** used for Unicode code points 32 (the space) through 126 (~) as well as the standard control characters mentioned earlier. This range is usually referred to as “Basic Latin,” but as discussed earlier in this paper, only values from  $0 \times 41$  to  $0 \times 60$  (decimal 65-90: A-Z), and  $0 \times 61$  to  $0 \times 80$  (decimal 97-122: a-z) should be considered as Latin Characters, since the others are shared by many scripts.

When pressing the **A** and **a** keys, the UTF-8 transformation is rather straightforward.

Decimal	Hex	Character	
65	$0 \times 41$	A	<b>0</b> 1 0 0 0 0 1
			Character Bit Sequence: - 1 2 3 4 5 6 7 <b>A</b> = *1000001 - Note that bit 2 (32 weight) is OFF
			Decimal Value of each Bit: - 64 32 16 8 4 2 1 <b>a</b> = *1100001 - Note that bit 2 (32 weight) is ON
97	$0 \times 61$	a	<b>0</b> 1 1 0 0 0 1

In UTF-8 transformations, the leading 0 indicates this is a one byte character, and the final seven bits are the binary value of the character. Note that the 32-value bit (bit 2) value determines the case difference between capital and small Latin letters.

**Two byte UTF-8 Representation:** used for Unicode code points  $0 \times 0080$  to  $0 \times 07FF$  (decimal 128-2047), and is illustrated here with the Unicode Basic Greek script block  $0 \times 0370$ - $03FF$ , specifically the letters that result from using a Greek keyboard mapping with a Latin keyboard and typing the same **A** and **a** keys as before.

Decimal	Hex	Character			
913	$0 \times 0391$	Α	<b>110</b> 0 1 1 1 0	<b>10</b> 0 1 0 0 0 1	This is the Greek “Alpha,” not the Latin “A”
			Character Bit Sequence: - - - 1 2 3 4 5	- - 6 7 8 9 10 11	A = *****011-10010001 (Capital Α) α = *****011-10110001 (small α)
945	$0 \times 03b1$	α	<b>110</b> 0 1 1 1 0	<b>10</b> 1 1 0 0 0 1	This is the Greek “alpha,” not the Latin “a”

In a UTF-8 layout, two consecutive leading 1 bits followed by a 0 indicates the character is two bytes long. Thus, the two-byte Unicode hexadecimal value  $U+0391$  is transformed to the two byte UTF-8  $0 \times CE91$  value.

Basic Greek, it should be noted, also uses capital and small letters and, like the Latin, these are identified by bit 2 of the payload. The sixth through eleventh content bits are carried in the single continuation byte.

As noted in the table on the previous page, two byte UTF-8 representations can contain Unicode code point values from 128 through 2,047. Thus, scripts such as Hebrew and Arabic,<sup>70</sup> which are assigned code values from 1,424 to 1,535 ( $U+0590$ - $05FF$ ) and 1,536 to 1,791 ( $U+0600$ - $06FF$ ) respectively, can each be represented in two UTF-8 bytes and, thus are no longer than the two bytes they would occupy in a straight binary representation of their code point values.

Characters in Devanagari and Thai scripts<sup>71</sup> on the other hand, with assigned code values from 2,304 to

70 Examples of which are shown on pages 26 and 24 respectively of this document.

71 Examples of which are shown on pages 23 and 21 respectively of this document.

2,431 ( $U+0900-097F$ ) and 3,584 to 3,711 ( $U+0E00-0E7F$ ) respectively, each of which requires only two bytes in a raw binary representation, require three bytes each in their UTF-8 transformations.

This illustrates the major compromise of the UTF-8 transformation format. Scripts residing at the range boundaries approach a size<sup>72</sup> that is double what it was without the transformation.

As a matter of interest, the N’Ko script used to write the Maninka, Bambara, and Dyula languages and variants used in Guinea, Côte d’Ivoire and Mali, with an assigned Unicode block that ranges from 1,984 to 2,047 ( $U+07C0-07FF$ ), contains the highest value characters in two byte UTF-8 transformations.

**Three byte UTF-8 Representation:** The lowest values requiring three byte representations in UTF-8 belong to Samaritan, a right-to-left script used in ancient Hebrew and Aramaic and defined in the Unicode block ( $U+0800-083F$ ).<sup>73</sup>

The three byte UTF-8 representation example illustrated here uses Thai script, which is located in the Unicode Block  $U+0E00-0E7F$ , and uses the letters that result from using a Thai TIS-820 keyboard mapping on a Latin keyboard, and typing the same **A** and **a** keys as before.

Decimal	Hex	Character	e	0	-	b	8	-	a	4	Becomes	0e-24	
3620	0x0e24	ຸ	1110	0000		10	111000		10	100100			
		Character Bit Sequence:	- - - -	1 2 3 4		5 6 7 8 9 10			11 12 13 14 15 16				
3615	0x0e1f	ຸ	1110	0000		10	111000		10	011111		Becomes	0e-1f
			e	0	-	b	8	-	9	f			

Three consecutive leading 1 bits followed by a 0 indicates the character is three bytes long. There will therefore need to be two continuation bytes, each of which must begin with 10. In this case, the two-byte Unicode hexadecimal value  $U+0E24$  is transformed to the three byte UTF-8  $0xE0B8A4$  value.

**Four byte UTF-8 Representation:** For an example of a four byte UTF-8 representation, the table below uses symbols from the Unicode Musical Symbols Block  $U+1D100-1D1FF$ . In this case, the equivalent decimal values for that block range from 119,040 to 119,295, each requiring seventeen bits (three bytes if rounded) to represent in raw binary, but four bytes when transformed into UTF-8, another example of the increasing in size resulting from transformations near range boundaries.

Decimal	Hex	Character	11110	0000		10	011101		10	000100		10	011110
119070	0x1d11e	♫	11110	0000		10	011101		10	000100		10	011110
		Character Bit Sequence:	- - - -	1 2 3		4 5 6 7 8 9			10 11 12 13 14 15			16 17 18 19 20 21	
119136	0x1d160	♪	11110	0000		10	011101		10	000101		10	100000

Four consecutive leading 1 bits followed by a 0 indicates the character is four bytes long. There will therefore need to be three continuation bytes, each of which must begin with 10. As with the other examples, the seventeen content bits are placed in the UTF-8 structure, transforming the three byte  $U+1D11E$  to the four byte  $0XF09D849E$  transformation.

72 These are generally not actually doubled, because of the single byte shared characters (e.g. the space) used by these scripts.  
 73 A form of this script is still in use by a very small group of people in the modern day Palestinian West Bank.