Introduction to ICT for Development
Course code: 2I4360, 2I1160, IB7000
23 October 2005, Creative Commons Deed. Attribution-NonCommercial-ShareAlike 2.0

Primer to Localization of Software

Escudero Pascual Alberto <aep@it46.se>
Berthilson Louise <louise@it46.se>
2005/10/27 – Version 1.5

Table of Contents

Introduction to software localization

We are living in a world where currently 6,900 living languages exists and approximately 350 of them (5%) are spoken by at least 1 million people. According to Unesco Global Studies one language disappears on average every two weeks. 80% of African tongues has no writing system at all.

English is just one of these 6900 languages, but due to the military, economic, scientific, political and cultural influence of the British Empire in the 18th and 19th centuries and the strong influence of the United Estates during this century, English has been imposed as "the language of the world" and by many people regarded as "the language" of the educated and developed world. Although English is currently only the third greatest native language with approximately 380 million speakers, a third of the world's population speak at least some English since many countries include English as a mandatory second language to learn. Still, two thirds of the world's population, some 4 billion people, does not speak English. As a result, a great part of the world does not have access to a computer software in their local language. Some of them does not even have the technical possibility to type the characters of their language as it might exist neither a standardized way nor an available implementation.

For many people in the world, localization of software is not an issue. In many cases decision makers already speak at least one of the worlds most "powerful" languages. In general terms, a language is regarded as "powerful" when enables business opportunities with independence of the total number of speakers.

As a matter of fact, major proprietary softwares are never localized to a language if there is not enough business opportunities for that specific language. This might explain why a proprietary software Microsoft Windows is available in Islandic (spoken by 239 000 people) while is not available in Hindi (180-480 million people) or Bengali (200 million people).

Why software localization is needed

Since so many people speak English after all, should we not just teach the rest of the people English, so that we do not have to bother about software localization? The dilemma that face is: Either we teach billions of people English or we teach computers to speak other languages than English.

Looking at the problem from a pure cost efficient angle, teaching computers to speak other languages than English is definitely less expensive. But, there are also many more reasons to software localization than the pure financial ones.

I common misconception is that computers speak English by nature. They do not. They speak zeros and ones, bites and bytes that can be translated to any language.

Benefits of localization

Localization of computer software brings benefits to its end users independent of what language they speak and where in the world they live. Some of the benefits include:

1. It reduces the amount of training needed to make "new "users use computers
2. Reduces the costs for licences of proprietary software
3. Gives users their right to communicate in their native language.

4. Helps to preserve minority or nearly extinguished languages.
5. Supports higher education in native languages.
6. Provides a fair access to knowledge.
7. Enables to preserve and spread local culture

What is software localization?

The Localization Industry Standards Association[1] (LISA) defines localization (l10n) as:

“Localization involves taking a product and making it linguistically and culturally appropriate to the target locale (country/region and language) where it will be used and sold.”

Localization of software facilitates the use of local languages in computers and make computers accessible to non-English speaking communities.

Software localization is generally speaking the adaptation and customization of an internationalized product to a specific market. Localization of computer software is a topic that does not only refer to pure translation of software strings, but it also includes adoption of the computer software to the very local way of “thinking” and understanding computers in the target language.

The software localization can be divided into text localization and cultural localization.

Text localization

The translation part of the localization includes translation of software strings, manuals, documentation, help text, error messages etc. to the specific language.

During the translation activity, a set of software changes (programming skills) might be needed, consider for example that the size of dialog boxes or a combo-box might need to be altered as the translated string can differ a great deal in length depending on the language. In average, translation of English string into any non-English language results in longer strings that the original. For example, a very common word in computer software is “No”, in same languages that word can be as long as the Swahili equivalent “Hapana”.

When dealing with text localization, the writing direction of the language is of great importance and will always imply major design changes. Not all languages are read from left-to-right (LTR), like most of the western languages written with the Latin script. For example, text in Arabic and Hebrew are read from right-to-left (RTL), while numbers are written from left-to-right (with the most significant digit to the left). As RTL languages are often mixed with LTF scripts, support for bidirectional (bidi) writing is generally needed.

Furthermore, since languages can be written in different directions, also the user interface of a software requires a certain layout depending on the language. In a left-to-right language, we are used to search the information of a page from the upper left corner down to the bottom right corner. That applies to all sort of data, text, tables, images etc. For a right-to-left language, the most important information or the starting point of a sequence of data is always placed in the upper right corner. A step-by-step instruction must therefore be mirrored for a right-to-left language so that the user does not read the instructions backwards.

Many Asian scripts, such as Chinese and Japanese are written from top-to-bottom (from right to left). Mongolian (using the traditional alphabet) is the only vertical script that is progressing from left-to right . And yes, there also exist scripts that are written from bottom to top (Western Pacific Islands).

Some languages even change the direction depending on the content context. Imagine that in software localization!

[1] http://www.lisa.org

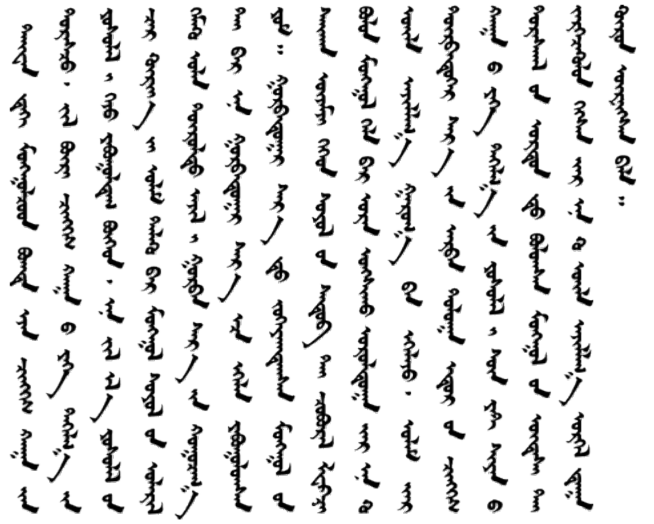


Image 1. Sample of Mongolian written in the traditional alphabet

Cultural localization

Even though pure text localization is good enough for adoption and understanding of a computer software in a new language, cultural localization can be needed to create that special feeling of a totally localized and user adopted software.

The software adaptation to a local language through cultural localization includes formats (date, time, currency etc), icons, images, colors, shapes of objects, multimedia files as sounds and videos, etc.

Cultural localization can become the most relevant task if the software is multimedia oriented and includes large amount of images, icons and colored text.

Colors, for example, can have different meaning depending of the culture. While the red color denotes danger in many western countries, it is a sign of happiness and good luck in Asian countries. Also, the white color is a sign or purity and cleanness in many countries while it represents death in Japan.

Internationalization (i18n) and localizatio*n (l10n)*

Internationalization is defined by LISA as :
“The process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for redesign. Internationalization takes place at the level of program design and document development.”

Internationalization is often called ”i18n”, where the number 18 refers to the number of letters omitted. In the same way, localization is often abbreviated as “l10n”.
Internationalization of a computer software is a prerequisite for localization to take place.

Internationalization and localization of software is nothing new. It has been going on since the first commercial software was released. The problem with proprietary software, is that the source code is not open and hence, the software can not be localized by anyone else. Only the owner of the software, that has access to the program code, can localize the software. That implies, that if the market for a software in a specific language in not beneficial enough, there will be a localized version of the software to that language. Even though there is a demand from such kind of software from many user communities and might be people willing to do the work, it can not be done as the code is not available.

And, this is where Free Software or Free Open Source Software (FOSS) comes into the picture.

What is Free Software?

The basic idea behind Free Software or Free Open Source Software (FOSS) is the availability of open (public available) source code. Anyone has the right to make modifications and redistribute the code (full definition: http://www.opensource.org/docs/definition.php). This means that open source code constantly is evolving and it is accessible to anyone.

"Free software" does not mean "free of charge", but is a matter of the users' freedom to run, copy, distribute, study, change and improve the software. It refers to four kinds of freedom for the users:

Freedom 0: To run the program, for any purpose

Freedom 1: To study how the program works, and adapt it to your needs. Access to the source code is a precondition for this.

Freedom 2: To redistribute copies so that you can help your neighbours.

Freedom 3: To improve the program, and release your improvements to the public, so that the whole community benefits. Access to the source code is a precondition for this.

These freedoms implies that you have the right, with or without modifications, free or charge or not, distribute copies a free software to anyone anywhere without permission from anyone.[2]

For further definition of software categories, see http://www.gnu.org/philosophy/categories.html.

Members of the (free) open source community claims that this way of working leads to better software compared to the traditional closed model with proprietary software. Of course, representatives of proprietary software will claim the opposite. I will leave that question for yourself to evaluate, but there is a fact that can not be discussed: Open source software promotes localization of computer software in a way that proprietary software never can do.

Why Free Open Source Software?

The greatest benefits of FOSS for any user community is the affordability (in terms of costs) and accessibility (in terms of usability).

Affordability

FOSS does not need to be free of charge, but the fee for distribution is far away from proprietary software licenses. In developing countries, where the license fee for a proprietary operating system or office suite exceeds an annual salary, using proprietary software with legal licenses is not an option. As a result of this, an "illegal" market of proprietary software expands. The use of FOSS also allows adaptation of the software to fit your private needs, which can lead to savings in many in several ways.

Accessibility

As the software can be localized to any local language, the accessibility of computers in the non-English

[2] http://www.gnu.org/philosophy/free-sw.html

speaking user groups, whether they speak an ancient tribal language of Ethiopia or a minority language of northern Sweden, is significant increased.

Some of the opportunities that FOSS is already given in developing countries in the long term are:

1. Reduction of ICT investment in both public and private sector
2. Building internal capacity and skills in ICT
3. Improving advocacy and education though the use of ICT
4. Creating business opportunities in ICT (national and international)
5. Implementing e-government

Localization components

Before the actual localization process of a software can take place there are a set of requirement that needs to be fulfilled for the target language. For example, can all, if any, characters of the target language be represented in a computer? Are there available fonts for the script that the target language? Is there a standardized way of expressing common computer related words in the target language?

Before any "translation" can take place, these are issues that need to be solved before.

Some of other additional localization components might not fall into the category of strict requirement before before the translation of the software strings. For example, a spell checker is very useful for text editors and email clients, but it does not necessarily need to exists before the software strings can be translated.

The following 7 chapters will describe various localization components that are closely related to any software localization and in many cases are dependent on each other.

Locale, a very local file

To fully localize a software to a local language requires more then just string translation. As mentioned in "Cultural Localization", the look and feel of a software needs to be adopted to the local conventions as well. A great deal of those local conventions are defined in a file called Locale. The Locale identifies a set of user preferences for a given user community. A locale commonly includes information regarding:

1. Formating and parsing of numbers, dates, times, currencies
2. Measurement units
3. Translated names for languages, countries, scripts, regions, time-zones
4. Collation ordering for sorting, searching and matching text
5. Text boundaries (character, word, line, and sentence)
6. Text transformations (including transliterations)

Common Locale Data Repository[3]

Most operating systems and major applications have their own repositories of locales. The fact that there are several locales for a single languages makes things even more difficult as the way that user preferences are expressed can differ from one software to another.

The avoid that situation, the Common Locale Data Repository (CLDR) project was created under the Unicode Consortium. The purpose of CLDR was to provide a general XML format for locales to facilitate the exchange of local information in application and system development. The CLDR project aims to have a common repository by making this information openly available.

When localizing a major software that requires some level of local formats, you should investigate whether a locale for your language exists or not. The latest version (1.3) of CLDR can be found here:

3 http://www.unicode.org/cldr/

http://www.unicode.org/cldr/version/1.3.html

If no locale exists for your language, you might consider creating one. A locale is not needed before the actual string translation of the software can begin. But it is a good idea to start early to define the locale, since many applications will use the locale to represent local values as currencies symbols or dates.

Locale naming

The first step of creating a locale is to name the locale correctly according to the naming convention for locales specified by Open i18n[4] guidelines.

The locale should accordingly to Open i18n guidelines be identified by the language, country and the character set in use. The syntax is the following:

{language}_{territory}.{codeset}[@{modifiers}]

language: is a string representing the language in the locale consisting of letters only. The character string should be chosen from a set of ISO standards with the following priority:

1. Pick a two lowercase letters ISO 639-1 code if exists: en, fr, fi, se etc.
2. Pick a three lowercase letters ISO 639-2 code if exists: eng, fra, fin, swe etc.
3. No ISO 639 codes are available. A string consisting of more than three (uppercase) letters should be chosen (not to conflict with future extensions of the ISO 639 series for existing language codes)

The ISO 639-1 codes can be found at Ethnologue's website[5] . The ISO 639-2 codes are handled by the USA The Library of Congress[6].

territory: is a character string (two letters) that represents the geographical territory (country or region) of the language. If a two-letter region/country code exists in ISO 3166-1, that value should be used.

If ISO 3166-1 does not specify a value for the territory, no standard value exist and a lowercase string consisting of more than two letters should be chosen.

The codes can be found within the International Organization for Standardization (ISO)[7]

codeset: describes the character set used in the locale.

Examples of codeset for locales are:
UTF-8, ISO-8859-1, ISO-8859-2, ISO-8859-5, GB-2312 etc.

modifiers: An optional field that adds additional information to the locales. The modifier consists of a keyword and an option value. Several options are separated by commas.

Accordingly to this naming convention of locales, the name of the locale for Swahili spoken in Tanzania is: sw_TZ.iso-8859-1

[4] http://www.openi18n.org/
[5] http://www.ethnologue.com/
[6] http://www.loc.gov/
[7] http://www.iso.org/

Locale Data Fields

The locale definition file includes a set of locale categories that needs to be filled in. Below, each category is briefly described.

LC_CTYPE: Specifies which characters are considered as alphabetic, numeric, punctuation, hexadecimal, blank, control characters etc. It also defines case conversions. *Note: yes, a "." is just one way of "punctuation".*

LC_NUMERIC: Defines rules for non-monetary numeric information.

LC_MESSAGES: Defines the format of affirmative and negative system responses.

LC_TIME: Defines rules for formatting the date and time information.

LC_COLLATE: Defines the collating information for characters defined for your language.

LC_MONETARY: Defines the rules and symbols for formatting monetary information.

If you are using Linux, run the command 'locale' followed by any of the above categories. Your current locale settings will be displayed.

For example:

```
root@it46:/home/it46 # locale LC_TIME
Sun;Mon;Tue;Wed;Thu;Fri;Sat
Sunday;Monday;Tuesday;Wednesday;Thursday;Friday;Saturday
Jan;Feb;Mar;Apr;May;Jun;Jul;Aug;Sep;Oct;Nov;Dec
January;February;March;April;May;June;July;August;September;October;November;December
AM;PM
```

XML Format

This section will give a brief description of the structure of the lXML format for the locale data and give some examples of elements in the XML file.

The following elements are included in the XML file:

1. identity
2. localeDisplayNames
3. layout
4. characters
5. delimiters
6. dates
7. numbers
8. collations
9. posix

Delimiters

The delimiter element specifies the delimiters user for bracketing quotations. While you in English write:

She said, “I'm not stupid!”

In Spanish you would write:

Ella dice: <¡Yo no soy tonta!>

Alternate marks are used when quotations are embedded. An example in English,

She said, “Remember what I told you: ‘After rain comes sunshine!’”

The English way of defining quotation marks is in the Locale coded as follows:

```
<quotationStart>“</quotationStart>
<quotationEnd>”</quotationEnd>
<alternateQuotationStart>‘</alternateQuotationStart>
<alternateQuotationEnd>’</alternateQuotationEnd>
```

Calendars

The default value for the calendar is Gregorian[8] but there are a number of another calendars that are used in computer software (Islamic, Chinese, Islamic-civil, Hebrew, Japanese, Buddhist, Persian and Coptic calendar).

The Calendar element also specifies the translations of the name of the months, the name of the days of the week (full name and abbreviations), first day of the week, first day of the weekend, local convention for AM/PM, quarters of the year and AC/DC.

Consider how you translate AM/PM for a language that uses sunrise and sunset as reference times for start and finish of time measures. For example, in greater parts of Ethiopia, a 24 hour time system is unknown. Traditionally there are no less than 7 named divisions to a day in Amharic (one of Ethiopias three official languages among its 84[9] living languages). Read the article 'Day Period Use Cases' (http://yacob.org/notes/DayPeriods/) by Daniel Yacob (localizer of Amharic software, Geez Foundation) that discusses the issue.

In Swahili, the name of the days of the week are called the First Day (jumamosi), the Second Day (jumapili), the Third Day (jumanne) etc until the Fifth Day, starting with Sunday as the first day. No standardized way for abbreviations of the names of the days exists until 2004 when the first Swahili locale was created. Normally, you would choose the first 3 letters of the day, but that is not feasible since five our of seven start by 'juma'. In the example below, you will see how the problem was solved.

```
<DaysOfWeek>
        <Day>
          <DayID>sun</DayID>
          <DefaultAbbrvName>Jpl</DefaultAbbrvName>
          <DefaultFullName>Jumapili</DefaultFullName>
        </Day>
```

8 http://en.wikipedia.org/wiki/Gregorian_Calendar

9 http://www.ethnologue.com/show_country.asp?name=ET

```
        <Day>
          <DayID>mon</DayID>
         <DefaultAbbrvName>Jtt</DefaultAbbrvName>
          <DefaultFullName>Jumatatu</DefaultFullName>
        </Day>
        <Day>
          <DayID>tue</DayID>
          <DefaultAbbrvName>Jnn</DefaultAbbrvName>
          <DefaultFullName>Jumanne</DefaultFullName>
        </Day>
        <Day>
          <DayID>wed</DayID>
          <DefaultAbbrvName>Jtn</DefaultAbbrvName>
          <DefaultFullName>Jumatano</DefaultFullName>
        </Day>
        <Day>
          <DayID>thu</DayID>
          <DefaultAbbrvName>Alh</DefaultAbbrvName>
          <DefaultFullName>Alhamisi</DefaultFullName>
        </Day>
        <Day>
          <DayID>fri</DayID>
          <DefaultAbbrvName>Ijm</DefaultAbbrvName>
          <DefaultFullName>Ijumaa</DefaultFullName>
        </Day>
        <Day>
          <DayID>sat</DayID>
          <DefaultAbbrvName>Jms</DefaultAbbrvName>
          <DefaultFullName>Jumamosi</DefaultFullName>
        </Day>
      </DaysOfWeek>
```

Below follows an example of how eras can be described in a Gregorian calendar and in a Buddhist calendar.

```
//Gregorian

   <eras>
     <eraAbbr>
      <era type="O">BC</era>
      <era type="1">AD</era>
     </eraAbbr>
     <eraName>
      <era type="0">Before Christ</era>
      <era type="1">Anno Domini</era>
     </eraName>
   </eras>

//Buddhist

   <eras>
      <era type="0">BE</era>
   </eras>
```

Here is an example of the calendar element for English and German describing relative types of days. As you can see, there exists only three relative types for English while German have five different relative types to express days. How many relative types does your native language have?

```
<calendar>
  <fields>
...
   <field type='day'>
    <displayName>Day</displayName>
    <relative type='-1'>Yesterday</relative>
    <relative type='0'>Today</relative>
    <relative type='1'>Tomorrow</relative>
   </field>
...
  </fields>
</calendars>
```

```
<calendar>
  <fields>
...
   <field type='day'>
    <displayName>Tag</displayName>
    <relative type='-2'>Vorgestern</relative>
    <relative type='-1'>Gestern</relative>
    <relative type='0'>Heute</relative>
    <relative type='1'>Morgen</relative>
    <relative type='2'>Übermorgen</relative>
   </field>
...
  </fields>
</calendars>
```

Glossary

"Short list of words related to a specific topic, with brief definitions, that are arranged alphabetically"

When working with translation, consistency of translated terms is extremely important for the overall quality of the final work. Since many strings associated to the same concept do normally appear multiple times in a software, they need to be translated in the same way. Sometimes, an English term can mean two different things depending on the context, so automatic translation is generally not a good approach. Also translation of related concepts as "open file" or "close file" must ensure that the word "file" is written in the same way not to cause confusion.

The existence of a glossary is a necessary step to ensure consistency in the use of terminology during the translation phase. It is highly recommended to make sure that you have a good glossary for your software in place before you start with the translation of strings.

For many minority languages, that are not official languages, no glossaries exists. If that is the case, a glossary has to be created before the translation phase of the project start, and common guidelines needs to be provided to the team of translators.

So how do you create a glossary suitable for a localization effort? The first thing you need is a list of word suitable for the software you are going to localize. The word list should consists of common single words (sometimes concepts with multiple words) from the source code. (See: section "extraction of strings" for more information on how to extract those word lists).

The process of creating new terms involves joint work of linguistic and computer science experts. The development of new IT terms requires that the linguistic expert fully understands the meaning of the IT term and its context before trying to match or extend the meaning of a word in the destination language. No single computer scientist will fully understand the meaning of each computer term that he/she comes across, so a way to identify the correct definition of a word or expression is needed during the glossary development. That can be done easily using popular tools as Google[10] (define: <word>) or the online encyclopedia Wikipedia[11].

In a language where no formal education exist, many words that belong to the industrial world, does not even exist. It is not strange that many African tribal languages spoken in rural areas lack expressions as "Configure Printer" or "Default gateway". So how do you translate those words that seem not have an exact match in the local language?

How are new words created?

This topic could be a standalone course since it is an area within linguistics with a great research scope. However, a brief introduction to some of the techniques used in creation of new terminology will be given.

[10] http://www.google.com
[11] http://en.wikipedia.org

Loanword

The approach of simply "loan" a word from another language, that does not exist in the target language, is a common method in creating "new" words. The word can be "loaned" in is original form or slightly changed to better fit the target language. For example, Internet and CD-ROM are English "loanwords" that many languages have adopted straight away, while "chatta" , "maila", "programmera" and "installera" are English loadwords that have been slightly modified to the Swedish language.

Transliteration

Sometimes, words are loaned and then translated to the target language but keeping its original meaning. For example the Swedish word "mjukvara" comes from "software" {mjuk/soft, vara/ware} and hårddisk is a translated loanword of "hard drive" {hård/hard, drive/disk}. The word "mouse" for a hand held pointing device used with computers is a common loanword in many languages (mus in Swedish, puku in Swahili, ratón in Spanish or Maus in German).

Semantic expansion

Semantics is a subfield within linguistics that studies the actual meaning of words, phrases, sentences, and texts. With semantics, old words can get new meanings as the true meaning of those words fits into new situations. The word "virus" has traditionally had the meaning of "an microorganism that can infect cells and cause disease", but nowadays the meaning of a computer virus is widely accepted.

Metaphors

Sometimes, metaphor techniques are suitable to create new words when other techniques does not fit. One example where metaphoric thinking often have been used, is with the character "@" which in computers are used to indicate the location or institution of the e-mail recipient.

The funny shape of the character has contributed to a list of strange translations in various languages. In Swedish it is called "snabel-a" (trunk-a, like in elephant's trunk) or "kanelbulle (cinnamon bun). The @-sign is commonly called "*sobatjka"* (small dog) in Russinan, "*kissanhäntä"* (cat tail) in Finnish, "*kukac" (worm) in Hungarian*, "*apestaart" (monkey tail) in Dutch and Afrikaans, "Klammeraffe" (climbing monkey)* in German, "*ensaimada" (spiral shaped pastery) in Catalan or "chiocciola" (snail) in Italian.*

The Swahili word for "password" is another example of metaphoric creation of new words. The Swahili word "*-nywila" (password)* has been derived from the historical word "*nywinywila"* which was used to refer to the concept password (code word to pass inside) during the 'Majimaji war' against the Germans in the early 20th century (1905-1907), which the Tanzanian eventually won. During the process of defining a new glossary the word "*nywinywila" was adopted in its* reduced form *'nywila'* just for simplification purposes.

Adoption of new words

The process of adopting new words takes time and it is not unusual that many "new" words will feel strange (some might even say funny) and unfamiliar the first time(s) that they are used. But an unfamiliar word does not need to be bad. It just takes some time for us, human beings, to adopt to new things. Many of you did probably react when you heard the word "mouse" in your native language for the first time, since rodents are not or should not very frequently present in computer environments. After some years, you probably use nowadays the word without even thinking about its very original meaning.

It is common to believe that a new term sounds better in a foreign language, specially for those that can master another language as English. In many cases, a sign of social status is to be able to use foreign words rather than local ones.

Remember that English is not that magic language that “contains” all the words in the world. It just happens to be the language that most software are created for and has become the reference language for many areas due to its high status in the world. I am pretty confident that early computer users also found the words “laptop” and “download” uncomfortable to use the first time they heard them. English words are also created, think in words like book-mark, gate-way, key-board, head-phones etc. there is nothing magic in those words.

Writing Systems / Scripts

“A writing system, also called a script, is a type of symbolic communication system used to represent elements or statements expressible in some spoken language, for the purpose of communication. [12]“

A writing system can be shared by many languages, like the Latin alphabet is for most of the west European languages.

There exist a handful different types of writing systems that are classified accordingly to what the symbols of that script represents.

Type of writing system	**What each symbol represents**	**Example**
Logographic	morpheme	Chinese *hanzi*
Syllabic	syllable	Japanese *kana*
Alphabetic	phoneme	Latin
Abugida	consonant+vowel, vowel	Indian *devanagari*
Abjad	consonant	Arabic
Featural	phonetic feature	Korean *hangul*

Table 1. Types of writing systems

Logographic

In a logographic writing system, in theory, each symbol represents one idea, in difference to other writing systems where each symbol primarily represents a sound or a combination of sounds.

Each symbol in a logographic writing system represent parts of words or whole words (called morpheme). If the logograph resemble the thing(s) they represent, they are sometimes known as pictograms (or pictographs). If the logograph represent an abstract idea, they are known as Ideogram.
As each word requires an own symbol, the number of symbols in such a language, is of course high. Chinese, that is a typical logographic writing system, use around 3-4 000 symbols for normal communication (to be “literate”) but a good word word processor should support at least 10 000 characters. Unicode support more than 70000 Han characters that is a .....
The huge number of symbols in a logographic writing system, does not only cause trouble for people to learn and memorize, but challenge also the computer industry to tackle problem with large keyboards.

The oldest known writings systems, that dates back to zzz, were primarily logographic and were based on pictographic and ideographic symbols.

An advantage of logographic writing systems is each symbol can be used in different languages but with another meaning. That means that closely related languages to chinese (mandarin) like dialiects of Chinese , Korean and Japanese can use logographs from the Chinese writing system.

Many languages that are not classified as logographic, uses some logograms. For example, the Arabic numbers (1,2,3) used by most western languages are actually logograms as well as the characters & (empersand) and @ (at).

Ideographic scripts and pictographic scripts are not thought to be able to express all that can be communicated

[12] http://en.wikipedia.org/wiki/Writing_system

by language. That is, no *full* writing system can be completely pictographic or ideographic; it must be able to refer directly to a language in order to faithfully represent that language. Hieroglyphs were commonly thought to be ideographic before they were translated, and to this day Chinese is often erroneously said to be ideographic.
Although a few pictographic or ideographic scripts exist today, there is no single way to read them, because there is no one-to-one correspondance between symbol and language. In some cases, only the author of a text can read it with any certainty, and it may be said that they are *interpreted* rather than read. Such scripts often work best as mnemonic aids for oral texts, or as outlines that will be fleshed out in speech.

Syllabic writing systems

A syllabary writng systems consists of a set of written symbols that represent syllables which together (or alone) make up words. A symbol consists of consonant and vowel sounds or a single vowel sound.

Alphabetic writing systems

An alphabet (from Greek) is a small set of basic symbols which each represent a phoneme of a language. In a perfect phonological alphabet, there would be a one-to-one matching between a letter and a phonetic sound, i.e person could know the spelling of a word by given its pronouncation, and opposite, given the spelling a person could pronounce the word correcly.
Pure phonological alphabets are easy to learn (for example Finnish) that languages with complex and irregular spelling systems.

Abjads

Abjads is an alphabetic writing system that has only symbols for consonants and not for vowels. However, some scripts based on Abjads, like Arabic and Hebrew have markings (notations) for vowels while other Abjads solemnly represents consonants.

Abugidas

Abugida is an alphabetic writing system that has symbols for consonants with an inherent vowel and where modifications of the basic symbol indicate other following vowels than the inherent one.
For example, in Devanagari, there is no basic sign to represent the consonant *k.* Instead, the k is represented in combination with other characters . In this way, the unmodified character क represents the syllable *ka* (the *a* is the "inherent" vowel). By changing vovel to the character k, the character k- can be represtned as कि (ki), कु (ku) , के (ke) and को (ko).
In most abugidas the vowel modifications appears as letters or diacrets (accent marks) above, below, to the left or to the right of the consonantal character (like the example above). But there are also cases where the form of the consonant symbol changes or rotates as in.
The Ethiopic script as well as the Canadian Aboriginal Syllabics can we considered abugidas, however, both with some exceptions. The largest group of abugias can be found in the Brahmic family of scripts that includes almost all scripts used in India and Southeast Asia.

Featural writing systems

In a **featural** writing system, each part of each symbol corresponds to a phonetic feature. Sounds that are phonetically related have symbols that are related.
The greatest featural writing system is Hangul that is used in Korean, even though Hangul also has logographic and alphabet aspects in addition to features.

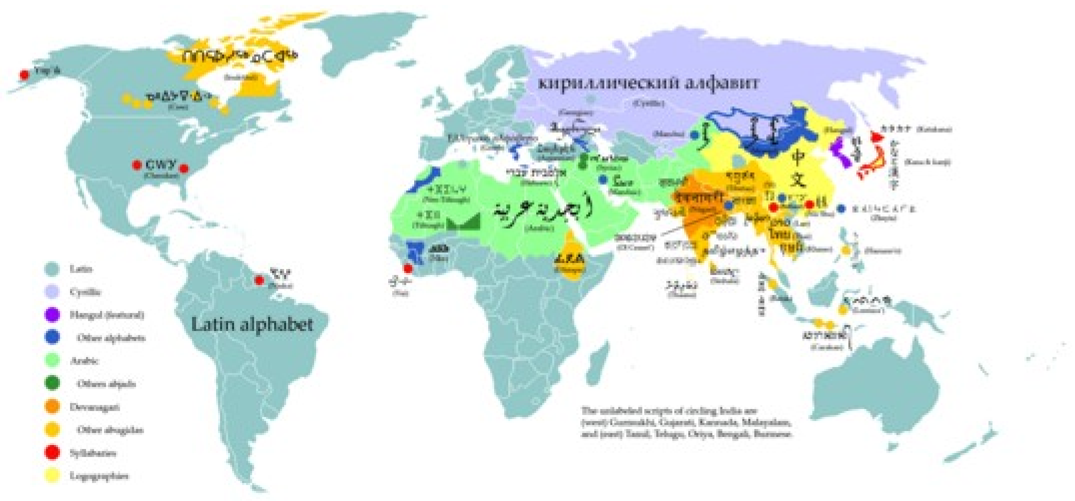


Image 2. A map of the writing systems of the world.

Character Encodings

Character Repertoire and Character Encodings

Every language is characterized by a set of characters that are defined in a writing system. A character is the basic unit of a text and forms other textual units as words when they are put together. The set of characters that constitutes a language is called its “Character Repertoire”. A character repertoire normally consists of (1) a set of commonly used characters, (2) characters that are not so commonly used, and (3) characters with distinct meanings. Such characters could for example serve as intonation modifiers which can indicate that a vowel sound has to be prolonged. Other characters can act as m*odifiers and act as operators*, meaning that they do not occur by themselves, but only in conjunction with another (for example, ` and ˆ).

To be able to represent human languages in computers that just speak with zeros and ones, we need to represent each character in a way that the computer understands. That means that we need a way to map each “human” character to a “computer” character. This kind of character mapping is called *character encoding*.

When localizing a software into a specific language, we need to make sure that the encoding scheme that includes your character repertoire is supported. If not, the character of your language will not be properly displayed.

If your character repertoire is not supported in any existing character encoding, you must focus on standardizing a character encoding scheme for your language. Or even worse, if your language does not even have a standardized character repertoire, you must start by defining one.

Furthermore, a script (writing system) can have more than one character encoding scheme. For example, the Chinese language has multiple encodings in widespread use (for example *Big5* and EUC-CN*)* .

A given script may have multiple encodings of various reasons:

1. Historical reasons: different manufacturers may have designed and implemented their own encodings before standardization took place. For example, IBM's EBCDIC character set encoding was implemented before the ASCII encoding was formalized.

2. Different goals: encodings could have been designed with different goals in mind: for space efficiency (for example, the Shift-JIS encoding for Japanese, is more space efficient than the EUC-JP encoding), or perhaps to facilitate sorting and collating text or ensure compatibility with an existing encoding.

3. Incorrect existing encoding: *errors in existing standard encodings could require a new encoding to be defined. For example, the ISCII (Indian Standard Code for Information Interchange) standard's support for Tamil and Kannada was sufficiently inferior that these respective language groups formulated their own character encoding standards (TSCII and KSCLP respectively).*

Some languages can be written using different writing systems, for example in Serbian, the message “Insert Movie and Sound” can be written as “Umetni film i zvuk” using the Latin writing system or as “Уметни филм и звук” using Cyrillic.

Properties of a character encoding

Stateful and stateless encodings

Encodings can be stateful or stateless. A stateful encoding means that the encoding has the characteristics that a character is dependent both on its own value and on the characters already encountered. An example of stateful encoding is Shift-JIS that is used to encode the Japanese language. Shift-JIS is grouped as multiple sets of symbols with each set containing less than 256 symbols. Special "escape symbols" embedded in the data stream are used to notify the application to switch between the appropriate sets.

Stateless encodings implies that each numeric value return the same meaning irrespective of the symbols that precede it in a data stream.

Single-Byte and Multi-Byte encodings

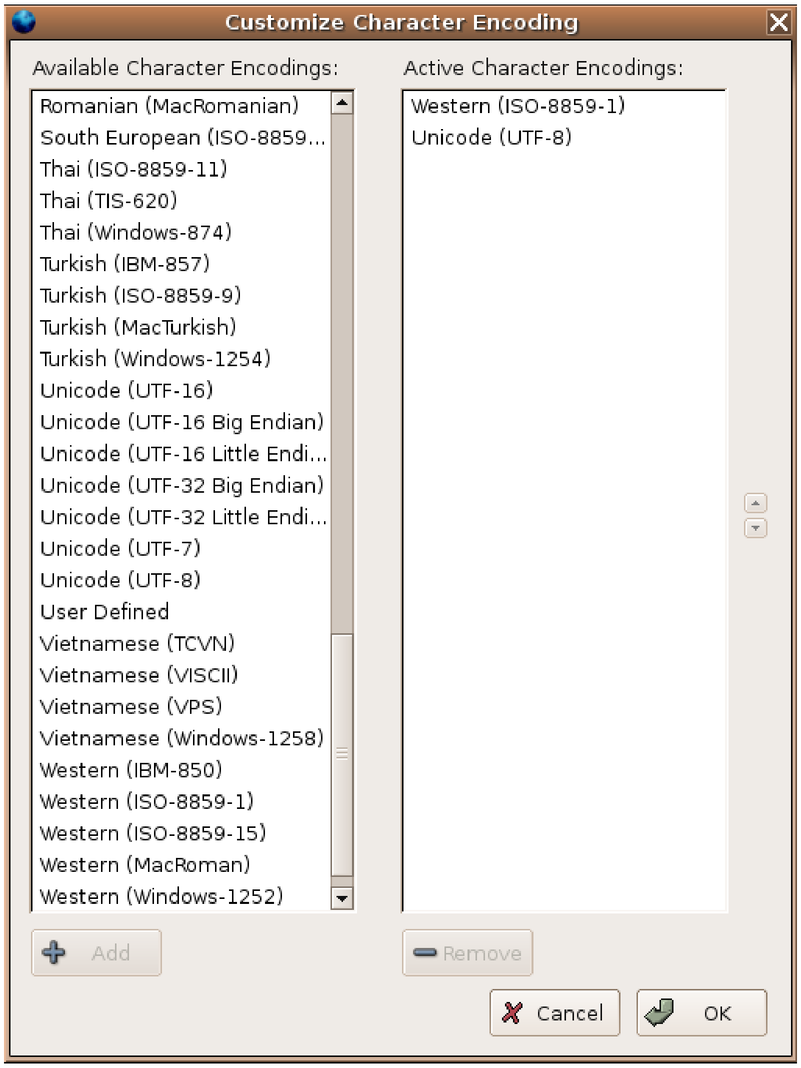
Encodings are also classified as Single-byte or Multi-byte encodings.

Single-Byte encoding schemes use a single byte (a maximum of 8 bits) to represent each character. This is the most efficient way to encode text since they they take least amount of space and is very fast to process as one character is always represented by one single byte. In a single-byte encoding, a maximum of 256 characters can be mapped. Scripts with a larger character repertoire must be encoded with a multi-Byte encoding scheme, which applies to most languages.

In a multi-Byte encoding characters can *either have* a fixed *or* a variable number of bytes to represent a character.

Examples of character encodings

There are many different character encodings in use today. Just take a quick look in your browser: (normally under View->Character Encoding) and see how many are supported. If you take for example Firefox 1.0.6, no less than 92 character encodings are supported!



Available Character Encodings:
Romanian (MacRomanian)
South European (ISO-8859...
Thai (ISO-8859-11)
Thai (TIS-620)
Thai (Windows-874)
Turkish (IBM-857)
Turkish (ISO-8859-9)
Turkish (MacTurkish)
Turkish (Windows-1254)
Unicode (UTF-16)
Unicode (UTF-16 Big Endian)
Unicode (UTF-16 Little Endi...
Unicode (UTF-32 Big Endian)
Unicode (UTF-32 Little Endi...
Unicode (UTF-7)
Unicode (UTF-8)
User Defined
Vietnamese (TCVN)
Vietnamese (VISCII)
Vietnamese (VPS)
Vietnamese (Windows-1258)
Western (IBM-850)
Western (ISO-8859-1)
Western (ISO-8859-15)
Western (MacRoman)
Western (Windows-1252)

Active Character Encodings:
Western (ISO-8859-1)
Unicode (UTF-8)

Image 3: Supported encoding systems in Firefox 1.0.6

Some of the major encoding schemes in use today will be presented in the following section.

ASCII

ASCII (***A**merican **S**tandard **C**ode for **I**nformation **I**nterchange*) is a 7-bit (stateless, single-byte) encoding system that was invented in the early days of Unix and C. ASCII is based on the roman alphabet used in modern English. With its 95 printable characters (out of 127) it can represent 26 upper-case letters, 26 lower-case letters, 10 digits and about 30 punctuation marks.

Base64

Base64 is an encoding system used to convert binary-encoded data into printable ASCII characters using a base of 64. In Base64, the 6 lower bits of a byte are used to encode binary data, as a result 64 printable ASCII characters are used to encode binary files.

The only characters used in this encoding are the upper and lower-case Roman alphabet characters A-Z, a-z and 0-9. The remaining 2 characters are normally used for plus and minus sign depending on which version of base64 you use.

This property has made Base64 a popular encoding to transfer email among other things.

ISO 8859

The 96 printable ASCII characters are insufficient for most modern languages (except English) based on the Roman alphabet. Characters as å,ä å (Swedish), ñ, ¿, ¡ (Spanish), ü,ß (German) ø,æ (Danish) are needed to fully encode some character repertoires.

The standardization bodies ISO and IEC tried to solve that problem by using all the 8 bit in a byte. ASCII was using the 8 bit for parity check. By using the 8th bit IEC/ISO was able to add another 128 characters (256 in total). However, not even with the 8th bit, all character repertoires based on the Roman alphabet could not fit into a single byte (they exceeded 256 characters together) so several sub-mappings were defined.

ISO 8859 (AKA ISO/IEC 8859), is a joint ISO and IEC standard for a number of sets of 8-bit character encodings. The standard ISO 8859 is divided into 15 different separate groups (such as ISO 8859-1, ISO 8859-2, etc) where each and one of them is regarded as a standard of its own.

ISO 8859-1 is a 8-bit character encoding for the West European languages. ISO 8859-1 supports Afrikaans, Basque, Catalan, Danish, Dutch, English, Faeroese, Finnish, French, Galician, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish and Swedish.

The table below gives you an overview of what languages are covetered in the 15 ISO 8859 standards.

Standard	**Type of language**	**Comment**
ISO 8859-1	*Latin-1* *Western European*	Covering most modern European languages: Danish, Dutch, English, Faeroese, Finnish*, French*, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, Swedish, Albanian and the African languages Afrikaans and Swahili.
ISO 8859-2	*Latin-2* *Central European*	Supports Central and Eastern European languages that use the Roman alphabet, including Polish, Croatian, Czech, Slovak, Slovenian and Hungarian.
ISO 8859-3	*Latin-3* *South European*	*Supports South European languages such as Turkish, Maltese and Esperanto.*
ISO 8859-4	*Latin-4* *North European*	*Supports the North European languages Estonian*, Latvian, Lithuanian, Greenlandic and Sami.*
ISO 8859-5	*Latin/Cyrillic*	Covers most Slavic languages that use the Cyrillic alphabet such as Belarusian, Bulgarian, Macedonian, Russian, Serbian and Ukrainian.
ISO 8859-6	*Latin/Arabic*	Covers the most common Arabic characters. Doesn't support

		other languages using the Arabic script since they need support for bidi and cursive joining.
ISO 8859-7	*Latin/Greek*	Covers the modern Greek languages but can also be used for Ancient Greek written without accents or in monotonic orthography. Lacks the diacritics for polytonic orthography which is introduced in Unicode.
ISO 8859-8	*Latin/Hebrew*	Covers the modern Hebrew alphabet as it is used in Israel.
ISO 8859-9	*Latin-5* *Turkish*	Almost the same as ISO 8859-1, but replaces the rarely used Icelandic characters with Turkish ones. Also used for Kurdish.
ISO 8859-10	*Latin-6* *Nordic*	A rearrangement of Latin-4 that is considered to be more useful for Nordic languages. Baltic languages use Latin-4 more.
ISO 8859-11	*Latin/Thai*	Contains most glyphs needed for the Thai language (same as TIS 620)
ISO 8859-12 **NOT EXISTING**	*Latin/Devanagari*	The work to support Devanagari in ISO 8859 was officially abandoned in 1997. For example, Unicode is covering Devanagari. The standard does currently not exist.
ISO 8859-13	*Latin-7* *Baltic Rim*	Support for some additional characters used in Baltic languages which were missing in Latin-4 and Latin-6.
ISO 8859-14	*Latin-8* *Celtic*	Covers Celtic languages such as Gaelic and Breton.
ISO 8859-15	*Latin-9*	A revision of ISO 8859-1 that removes some little-used symbols, replacing them with the Euro symbol (€) and the letters *Š*, *š*, *Ž*, *ž*, *Œ*, *œ*, and *Ÿ*, which completes the coverage of French, Finnish and Estonian.
ISO 8859-16	*Latin-10* *South-Eastern European*	Covers Albanian, Croatian, Hungarian, Italian, Polish, Romanian Slovenian, Finnish, French, German and Irish Gaelic with focus on letters rather than than symbols. The Euro symbol is included.

- * Partially supported. Missing characters are supported in ISO 8859-15.

Table 2. Description of all ISO 8859 encoding standards.

UCS

The first draft of UCS (Universal Character Set) was defined by the international standardization bodies ISO/IEC in 1990. The purpose was to compose a universal character set of all existing languages.

UCS (ISO 10646) constitutes of a 31-bit character set that allows a maximum of 2,147,483,648 characters . UCS maps each character to integers called numeric **code points** and assigns each character an official name. The most commonly used characters, including all characters supported in earlier defined encodings can be found in the first 65 534 positions. This subset of UCS is called the Basic Multilingual Plane (BMP). Characters that has been added outside of this subset, are normally characters that are used for specific application such as historical scripts or scientific notation.

Two version of UCS exists, UCS-2 and UCS-4. UCS-2 uses 2 bytes per character (65 000 characters) while UCS-4 uses 4 bytes per character. While UCS-2 only can represent the characters of the Basic Multilingual

Plane, UCS-4 can also represent all the “Unicode standard” characters.

Unicode

Unicode is an effort of the Unicode Consortium to create a single Character Repertoire covering all the possible characters across the world.

Just like UCS, Unicode uses unique code points for each character. The code points are then mapped into a sequence of code units that can be 8, 16 or 32 bits. This kind of code point mapping is called the Character Encoding Form (CEF).

Unicode has defined three CEF, UTF-8, UTF-16 and UTF-32.

UTF-32 is the simplest possible form of Unicode encoding. 32-bits are used to store the code points of all the possible characters and hence the name.

The greatest problem with this encoding is that documents encoded with UTF-32 that contain ACSII-like encoding, will be very inefficient as only 7 bits are actually needed to represent one character. If UTF-32 is used, a four times the space of ASCII encoding will be required. Due to that reason, UTF-32 is rarely used.

UTF-16 is a variable width encoding which uses either one or two 16-bit words for each character. Since the order of the bytes are dependent on the hardware in use, the first byte of the data stream is allocated to indicate the order of the bytes.

UTF-8 is also a variable width encoding since it uses one or more 8-bit bytes for each character. The first 128 code-points are represented by one single byte and are similar to the ASCII encoding. The remaining code-points use from two up to six bytes. This is the most frequently used Unicode encoding and it has several advantages:

1. Files and strings containing only 7-bit ASCII characters have the same encoding in both UTF-8 and ASCII.
2. Good compatibility with most functions of the standard library of the C programming language due to its use of 8-bit values
3. For text that uses few (but some) non-ASCII characters, UTF-8 is very efficient since it will only use more than one byte for those characters and not for the ASCII characters. In average, one character will be represented with a little bit more than 1 byte.

UTF-7 is another Unicode standard that was originally designed to become a *Mail-Safe Transformation Format of Unicode (i.e. a Unicode encoding for e-mail without the need of MIME encapsulation). UTF7* is similar to UTF-8, it is a variable-length character encoding but it uses only 7 bits of a byte and leaves the most significant byte to be zero. UTF-7 is rarely used due to the complexity to process.

It is highly recommended that you use Unicode (specially UTF-8) as your encoding if your target language is supported by Unicode. Unicode is the only universal Character repertoire that currently exist and it is being updated continuously by the Unicode Consortium to support more encodings. Unicode has been widely adopted and is used by industry leaders as Apple, HP, IBM, JustSystem, Microsoft, Oracle, SAP, Sun, Sybase and many others. Unicode is supported by many operating systems and all modern browsers.

If you are curious about what characters that are included in Unicode, try the “Unicode character pickers” (by Richard Ishida, Internationalization Activity Lead in the World Wide Web Consortium) at http://people.w3.org/rishida/scripts/pickers/ . The Unicode Character Pickers allows you to input and display Thai, Bengali and other non Latin based scripts.

Try also the UniView (by Richard Ishida, W3C), where you can get the Unicode mapping for all encodings included in Unicode 4.1.0 (http://people.w3.org/rishida/scripts/uniview/help.html).

Normalisation

As mentioned before, characters can be encoded in one or several different encoding systems. In addition, some characters can have multiple representations in the same encoding systems, which can cause problems for many fundamental operations in text processing.

For instance, in ISO 8859-1 the letter 'ç' can only be represented as the single character E7 'ç'. In Unicode the same character can be represented as the single character U+00E7 'ç' *OR* as a combination of 'c' and '¸'. Additionally, in HTML it could be represented in three different ways: as &ccedil; OR &#231;(decimal) OR \xE7 (hexadecimal).

Some operations that are vulnerable to such multiple representations of a single character is string matching, indexing, searching, sorting, regular expression matching, selection, etc. The operation of string matching is normally done by comparing two string byte per byte (binary comparison). In the case that some of the characters in a string are represented in different ways, even though they refer to the same original character, such string matching is not possible. Also, if the two string are encoded in different encoding systems, the string matching will not work properly.

The solution to this problem lies in normalization which means that strings are converted to a common canonical encoding before the binary matching takes place, in order to compare them correctly.

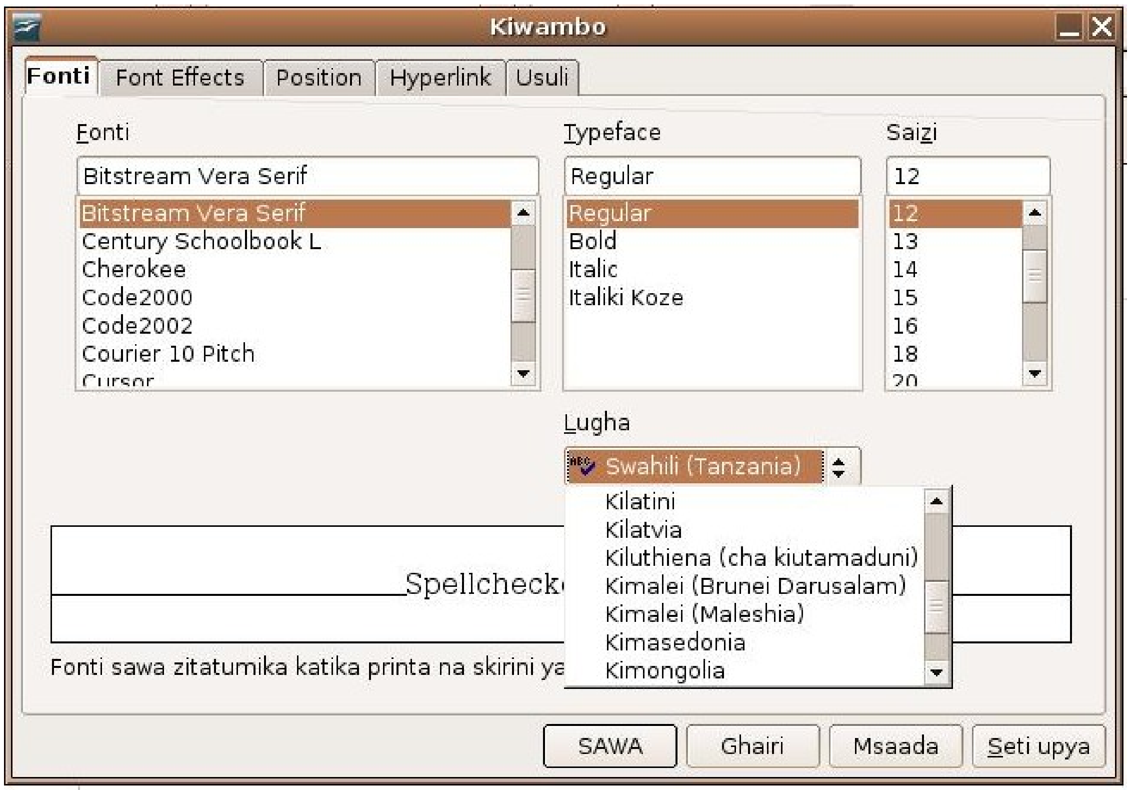


Image 4: Swahili Spellchecker

Collation

Collation is a general term for the process of ordering (sorting and arranging) strings of characters in a certain predefined order. The operation of sorting strings is fundamental in computer systems, but not trivial. Sorting of strings is frequently used in user interfaces since humans prefer to look for data when it is presented in a certain order. The sorting process is also common in databases.

Collation is language and culture dependent. For example, Germans and Swedes sorts the same characters differently. In German ö goes before z while in Swedish z goes before ö. Another example is the character ø that in the majority of Latin languages is sorted as a variation of o and is sorted alongside o. In languages as Norwegian and Danish however, the ø is treated as a unique sorting element that comes after z.

Languages that uses non-alphabetic scripts can be sorted by its phonetics (the second) or the root (appearance) of a character.

The *Unicode Collation Algorithm* is a general purpose algorithm for sorting all Unicode based strings. The main algorithm has four steps.

1. Normalize each input string
2. Produce an array of collation elements for each string
3. Produce a sort key for each string from the collation elements
4. Compare sort keys with binary comparison

Briefly, the Unicode Collation Algorithm takes a Unicode encoded string and a produces a collation element, which is an ordered list of three or more 16-bit weights, that represents the string. From the collation element a so called sort key is created which is a binary string. Finally, two or more sort keys are binary compared to give the correct comparison between the strings for which they were generated.

The specification of the Unicode Collation Algorithm can be found here: http://www.unicode.org/reports/tr10/

Fonts

In computer systems, we are used to fonts like New Times Roman and Arial. Depending on the font you choose, characters typed in that font get a certain graphical look. A font can be defined as

“The visual representation of characters from a particular character set.”

or

“A description of how to display a set of characters that includes the shape of the characters, spacing between characters, type of characters (bold, italics, underline) and the size of the characters.”

Both of the definition are correct although the first one refers to the visual representation of the font while the second definition refers to the file that defines the visual font.

A (visual) font for a character normally consists of a set of images that are called glyphs. A glyph can constitute the whole visual representation of a character or just a part of it. Also, one glyph may be part of many characters. That means that the mapping between characters and glyphs is not often one to one, but many to many.

Fonts play a very important role in localization since a localized software that a computer is not able to render and visualize, does not make much sense. Having fonts that support all characters in a specific character set is a basic requirement for localization.

There exists many free and open source fonts that are freely available to download from the Internet. Some of them can be found here:

1. freedesktop.org: http://freedesktop.org/wiki/Software_2fFonts

2. Unicode Font Guide For Free/Libre Open Source Operating Systems: http://eyegene.ophthy.med.umich.edu/unicode/fontguide/

If no fonts for the target language exist, fonts need to be created (see section 'Creation of Fonts')

Font types

Bitmap Fonts

A bitmap is a matrix of pixels where each character or glyph is stored as an array of pixels. Bitmap fonts are easy to create and they do not require complex rendering as other types of fonts. The problem with bitmap fonts is their lack of scalability. They work fine in a terminal window, console or in a text editor, where scalability normally is not an issue, but do would not like to use them to create a document to print.

A bitmap font file normally has the extension bdf or pcf.

Outline Fonts

An outline font can, in opposite to bitmap fonts, be scaled to any size keeping the quality of the font. This operation of scaling an outline font needs sophisticated rendering technology which requires a lot of numerical processing. Opposite to defining a character by an array of bits, in outlines fonts the character or a glyph is defined as a set of lines and curves. By performing complex rendering operations on the font, a character represented as an outline font can be *hinted (reduced in size) and anti-aliased (enlarged in size).*

Below follows a list of various types of outline fonts that are commonly used.

Type1 Fonts

Type1 fonts are developed by Adobe Systems and are widely used vectorial fonts under UNIX and Linux.

A Type1 fonts contain two files, (1) a metric file (that contain information regarding kerning, ligatures, spacing etc.) and (2) an outline file (that contain information regarding the characters shape)

The outline fonts are stored in two formats, pfa (PostScript Font ASCII) and pfb (PostScript Font Binary) while the metric file are distributed in afm (Adobe Font Metric) format.

Before a visual representation of a Type1 Font can be created, the font must be *rendered* into a bitmap, either by the PostScript interpreter or by a specialized rendering engine, such as Adobe Type Manager.

One disadvantage of this type of fonts is that Type1 metrics are very limited for some complex scripts like Arabic, Thai etc.

TrueType Fonts

TrueType Fonts is an outline font standard that was originally designed by Apple Computers in the late 1980s as a competitor to Adobe Systems Type1 fonts that were used in Postscript. Later, the TrueType Fonts standard was extended by another big player, Microsoft.

The TrueType fonts are distributed in one single file (.ttf) which contains both metric and shape information.

TrueType fonts are supported by GNU/Linux, Microsoft Windows, and Apple Macintosh, and all BSD variants.

OpenType Fonts

OpenType is an outline font standard that was initially designed and developed by Microsoft and later joined by Adobe Systems. Since Adobe converted their entire font library to OpenType fonts in the end of 2002, there are now (2005) more than 7000 fonts available in OpenType format.

The OpenType font is an extension of the TrueType font format.

The aim of the OpenType format standard was to create an advanced font format with full support for internationalization.

The OpenType fonts are distributed in a single file (.otf) which contains both metric and shape information.

The OpenType font format offers cross-platform compatibility (the same font file works on Macintosh, Windows and Linux computers). Since the font encoding is based on Unicode, OpenType fonts can support a large amount of languages and also enable multiple languages at one.

FreeType Font Engine

FreeType is, despite its name, not a type of font, but a Free and Open Source Software font engine that is worth to be mentioned in this discussion. FreeType is designed to be a small, efficient and highly customizable and portable font engine.

FreeType 2 provides a simple and easy-to-use API to access font content in a uniform way, independently of the file format. Additionally, some format-specific APIs can be used to access special data in the font file.

The latest version of FreeType is FreeType 2.1 which by default support all the above mentioned type of fonts.

FreeType is licensed under the GNU General Public License and a license, created by FreeType, that is similare to the BSD license.

You can read more about FreeType at http://freetype.sourceforge.net/freetype2/index.html

Creation of Fonts

Thousands of fonts for various scripts are created and made available by various institutions. If you need to create fonts, a good hint is to start with existing fonts used for languages related to your target language and modify the existing fonts. In many cases, most characters can be common to other languages, while a few characters are unique to that language.

If the target languages is supported in Unicode, it is a good idea to develop a Unicode font instead of a font that only covers the specific encoding of the target language.

In order to create fonts, a generic *font editor* that supports editing in multiple font formats and multiple character encodings is needed. One free font editor is Fontforge which can be downloaded at http://fontforge.sourceforge.net/.

Font creation is a great topic in itself and could be offered as a separate course. Creation of fonts is not only a combination of visual design and mathematical science, it also requires an understanding of the thousand years of language development that has formed scripts to what they are today.

Input method

An input method is a software component that deals with all aspects of entering text to the computer screen such as typing keys, speaking or writing using a pen device to generate text input. These aspects includes layout description of your keyboard as well as mapping characters to keys or combinations of keys (keymap).

For most Latin languages advanced input methods are not needed, as each character in the encoding scheme is mainly represented by one key in the keyboard and hence, the mapping is one to one. These languages only needs a keymap, with defines how the keyboard input/output is processed.

There are some language, especially those that use ideographs, that have thousands of characters in their character repertoire. For obvious reasons, a keyboard can not contain thousands of keys, so a special input method that combines a number of keystrokes into a character is needed.

Some scripts also needs to be processed before they can be displayed. For example, the input method for Hindu, that uses the Devanagari script, is defined in a way that it reorders and transforms letters that are combined together and produces an output that is different from the original input characters.

Input methods is a complicated issue when it comes to ideographic languages. **Ideograms**[13] are graphical symbols that represents words and ideas. Each ideogram is composed by a number of visual elements that can be arranged in different ways.

In some ideographic languages, for example the CJK family[14], each ideogram represent an actual object and not sounds. This leads to a very large character repertoire and difficulties in mapping characters into keystrokes.

There are also ideographic languages where each ideograph is a combination of sounds. A keymap can be created by combining phonetics with keystrokes. In this group of languages, a character can be typed by inputing the set of keystrokes that reproduce its sound (phonems). However, it can be the case that more than one ideographic characters can have the same phonetic sound, which leads to a “one-to-many” situation. In that scenario, the user must be given the option to choose the character to display from a group of possible characters that matches the input phonetics.

There exist a number of frameworks that enables the collaboration between text editing components and input methods, for example X KeyBoard Extension (XKB), X Input Method (XIM), GTK+ IM and IIIMF Framework.

Currently, the IIIMF Framework (Internet/Intranet Input Method) developed by Hideki Hiura is a popular framework that is platform neutral, window system independent, implementation sortlanguage independent and offers a multilingual distributed IM infrastructure. IIIMF is based on UTF16.

[13] from Greek *ιδεα* ***idea*** "idea" + *γραφω* ***grapho*** "to write"

[14] CJK refers to Chinese, Japanese and Korean

Keyboards

In order to be able to input data to a computer in the target language, a keyboard with full character support is desired. For languages based on the Latin alphabet, that might not be a big issue since existing keyboards can be used and modified to fully fit the target language. Unfortunately that is not the case for all languages and a physical keyboard might need to be developed.

The first thing you need to consider, having a character encoding in hand, is to decide the mapping between the code points of the character encoding and the keystrokes. If your character encoding includes more characters than keystrokes available, that process requires some thinking and planning. The mapping of code points and keystrokes can be done in many different ways, some better than others. For example, a good keyboard layout should aim to minimize the distance the fingers have to travel to write a text, right? The usage (frequency) of characters in the language should also be considered, where the most frequently used characters should be easy to reach while seldom used characters can be placed in the periphery. Funny enough, the QWERTY keyboard that has become the most common modern-day keyboard layout, was designed in the complete opposite way! As a matter of fact, the original prototypes of QWERTY typewriters from 1874 had a problem with the bars colliding with each other and jamming. To avoid the problem with colliding bars, the keys were arranged in a way to slow typists down by putting frequently-used pairs of letters separated.

Keyboard Layout

Keyboard layout refers to the process of giving each character a distinct place on a keyboard. This section will present two different approaches to keyboard layout. As a proof of concept, let take the German character 'ü' as an example. Where should it be placed on a keyboard? Should it have a key of its own or should it be a combination of key strokes? Whatever solution, where on the keyboard should the key(s) be placed?

Mnemonic keyboard layout

Mnemonic means “memory aid” and in terms of computer keyboards, it refers to that the layout of keys is chosen in a way that helps the user to remember the keying of the character. Mnemonic keyboards is commonly used for Latin-bases scripts. In mnemonic keyboards, there is little difference in what the user want to type and what is painted on the keys.

So, how should the German 'ü' be represented in a Swedish mnemonic keyboard?

In the Swedish QWERTY keyboard the German 'ü' is considered as a 'u' with umlaut. It can be typed using the combination <¨>+<u>.

Positional keyboard layout

In positional keyboards the layout of the keys are defined positionally in relation to each other. It is not the character printed on the key that is important, but rather which key is located next to which key and on which row that are positioned. Typically, commonly typed letters will be placed in the keys in the middle of the keyboard while less frequently typed letters will be placed in the periphery. The Dvorak layout is an attempt to provide a positional keyboard layout which allows typists to type English faster. As mentioned before, the QWERTY keyboard is rather the opposite.

In a positional keyboard, we might should have chosen to represent 'ü' in one single key and placed it in the middle of the keyboard, easy to reach, if 'ü' is frequently used character or in the peripheries if it is not. In the German keyboard, the character 'ü' has an own key and is placed in the middle of the keyboard as it is a very common character in the German language.

This approach to keyboard design is commonly used when a whole keyboard needs to be designed opposite to when it is a matter of adding a single extra character to an existing keyboard.

Large Keyboards

Most languages requires more keys that can be offered by any normal size keyboard. There are many approaches to extend a keyboard to support more characters. Some of those approaches are presented below.

Modifier Keys

The most common way to extend an existing keyboard is to use modifier keys such as Shift, Ctrl, Alt, Alt-Gr etc. The problem with these modifier keys are that they are frequently used by applications as acceleration keys. An *accelerator key* is a key on your keyboard that you can press to quickly access a menu or a function of a application. This collision can lead to complications. Either, the acceleration key of the application will not be accessible and can not be used in the application. Or, the application will have first priority to the key and will not allow that key to be used for typing a combination of characters.

Dead keys

Dead keys are another approach to extend keyboards which is commonly used for Latin keyboards. The implementation of dead keys allows the user to type a single character as a combination of two or more keys (in a certain order). Only the last key will result in a character on the screen, which will be the visual representation of the combination of characters inserted. In that way, ê is inserted as

<^>+<e>.

The usage of dead keys requires a strong mnemonic relationship between the dead key, the input character and the output character for the user to memorize the combination. Also, the dead key sequence should not be too long. One problem with dead keys is the fact that some keystrokes does not result in any visual feedback requires a little bit of training not to confuse the user.

Operator keys

A slightly different approach to dead keys is to place the modifier (the dead key) after the key that it modifies instead of before. In the example of ê, we would type the combination <e>+<^> instead of <^>+<e>. In this way, the user would get a visual feedback of each keystroke.

The problem of this approach is the implementation. Among other things, you need a system that can go back and replace characters in a document when the user is typing, which is not trivial to implement. Tools as Keyman[15] has worked this out, but with some limitation.

[15]http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=keyman

Candidate Window

An approach to solve the problem of thousands of characters to map (for example Chinese) is the use of a “popup” window. As a special key is pressed, a window with a set of possible options is displayed and the user can select the desired one by either using the mouse, pressing the initial key or using the arrows. As the user types more keys, the set of possible options are changing and narrows down.

Using a candidate window is a powerful method of languages with large amount of characters. The drawback of the approach would be the screen space that it takes up.

Spell Checkers

For applications as OpenOffice.org and mail clients where text creation is the main purpose, the existence of a spell checker is important for a satisfactory usage of the application. For Linux there are four main spell checkers: Ispell, Aspell, MySpell and Hunspell.

Ispell is a very old program that was originally written in 1971, by R. E. Gorin. Since then, many people has contributed and evolved Ispell to what it is today. For example, international support has been added as Ispell today supports a large number of European languages.

Ispell is a fast screen-oriented spell-checker that displays errors and suggests possible corrections when the such can be computed. Ispell is used as a part of the GNU system.

GNU Aspell is a Free and Open Source spell checker that was designed to eventually replace Ispell. The GNU Aspell can be used as a library or as an independent spell-checker. Its primary advantage over Ispell and other existing spell-checkers is the suggesting of possible replacements for a misspelled word. Aspell has also the capability to spellcheck UTF8 encoded documents without the use of an additional dictionary. Also, Aspell includes support for multiple dictionaries at once, which Ispell does not do.

MySpell is a spellchecker based on Ispell. MySpell is used by OpenOffice.org and Firefox/Mozilla and works on both Windows and Linux.

Hunspell is a new spell checker and morphological analyzer library and program designed for languages with rich morphology and complex word compounding or character encoding. Hunspell is planned to replace MySpell since it offers more sophisticated spelling functionalities. The main features of Hunspell is:

- Unicode support (first 65535 Unicode character)
- Morphological analysis (in custom item and arrangement style)
- Support for a Max. of 65535 affix classes and twofold affix stripping (for agglutinative languages, like Azeri, Basque, Estonian, Finnish, Hungarian, Turkish, etc.)
- Support complex compoundings (for example, Hungarian and German)
- Support language specific algorithms (for example, handling Azeri and Turkish dotted i, or German sharp s)
- Handling conditional affixes, forbidden words, pseudoroots and homonyms.
- LGPL license

Dictionary and Affix file

All the above mentioned spell checkers include a dictionary file (.dic) and supports for affix compression be means of a second file (.aff). The dictionary (.dic) is a list of words with their corresponding affix rules. The affix file describes each of the prefix based on aspell/myspell based rules.

Affix is a linguistic element added to a word to produce an inflected or derived form. An affix can be placed at the beginning (prefix), middle (infix), or end (suffix) of the root or stem of a word, but in the spell checkers mentioned above, an affix can only be prefix or a suffix (not an infix).

The fact that dictionaries supports affix compression means that the dictionary need only to contain the root of a word, since the affix file specifies all possibles derived forms of the word. By using an affix file, the size of the dictionary decreases dramatically.

The dictionary and the affix file is compiled to a hash table to form the spell checker. The size of the hash table will affect the speed of the spell checker. So, the smaller the dictionary is (only roots) and the larger the affix file is (definition of derivate), the faster will the spell checker will be.

For example, one row in the English dictionary looks like this:

```
imply/S
```

That means that the rule S (defined in the affix file) applies to the word 'imply'.
The rule S in the affix file is defined as:

```
flag *S:
   [^AEIOU]Y   >      -Y,IES        # As in imply > implies
   [AEIOU]Y    >      S             # As in convey > conveys
   [SXZH]      >      ES            # As in fix > fixes
   [^SXZHY]    >      S             # As in bat > bats
```

The first line for the flag S implies the following:
Two conditions must be fulfilled in order for the rule to me applied
1. The word must finish by an 'y' (suffix) and
2. The character next to the last character in the word must NOT be any of the following a, e, i, o, u.

3. Any such word in the dictionary can be derived by the rule -Y,IES (remove 'y' and append 'ties').

How to create a MySpell spell checker

Below follows a very general methodology on how to create your own spell checker based on MySpell.

A. Create Dictionary

- Collect many correct words as possible in the target language.
- Select the roots of the words (unique) and remove the rest
- Apply affix rules to each word

B. Create the affix file based on the grammar of the target language

C. Create the installation packs for Myspell

D. Submit the results to the FOSS responsible at MySpell

The methodology to create a spell checker depends greatly on the existing resources that you have for the target language. For example the creation of the affix file requires linguistic knowledge since the file specifies great parts of the grammar of the language. Such vast linguistic resources might not be available. If that is the case, the spell checker may only consist of the dictionary and contain no affix file. Since the purpose of the affix file is to speed up string matching and reduce the size of the dictionary, the spell checker will not be as fast as it should have been with an affix file.

Furthermore, dictionaries, i.e. lists of words, might not exist in the target language, at least not in digital form. If that is the case, as it is for many minority languages, words have to be collected “manually” to form a very first dictionary.

Word Lists sources

Word list can be collected from many institutions and online resources but is can be a time consuming work. Below follows some hints on how word lists can be collected.

Webcrawler
'An *crúbadán* Corpus builder[16]' (by Kevin Scannell) is a web crawler that searches the Internet via Google for any document (HTML,DOC or PDF) containing words in a specific language and extract relevant words.
The goal of the software is automatic development of large text corpora for minority languages that lacks content in digital form. The 'An *crúbadán'* is designed to exploit text that is freely available on the web, which can be a large quantity even for minority languages.
The crawler is initially fed with a small amount of correct spelled words, called “seed” text (a few hundred words is sufficient), in the target language. From those words, queries based on combination of the seed words are passed on the Google API which returns a list documents that with high probability is written in the target language. Those documents are downloaded, converted into plain text and formatted. Statistical techniques based on the initial text in the target languages are applied to the plain text in order to determine what documents or parts of them, is actually written in the target language. The “seed text” will grow larger as more words in the target language has been identified. The crawler words recursively and follows links from documents that has been determined to contain text in the target language. As a set of new identified words in the target language has been identified, the process is repeated and new Google queries will be executed.

For example, over 8000 new words where included in the second release of the Swahili Spellchecker[17] by using this method.

Linguistic Institutions
Contact linguistic institutions and ask for dictionaries in various areas. The dictionaries does not need to be technical as they should serve the purpose of spell checking any kind of text. Normally, the Bible is a good reference text as it is normally translated and stored in digital form in many languages.

International Resources
Search for resources outside of the country where the target language is spoken. In many cases, expertise and research in a minority language is based in Western Universities.

[16] http://borel.slu.edu/crubadan/

[17] http://www.it46.se/downloads/openoffice/dictionary/sw_TZ/README_sw_TZ.txt

Localization of text

When you are able to type all characters of the target language and represent them in a computer according to a certain character encoding, you are ready for the actual text localization.

This sections will give you an overview of the different steps that text localization implies.

Extract and convert strings

This first thing to do is to identify and extract the strings that need to be translated out of the source code of the software. Different scenarios on how to extract the string are possible since software is coded differently depending on its vendor or developer. Here are some examples of formats that we can expect to find.

Text strings embedded in the source code

This it the worse of the scenarios. The only way to localize the software is to edit the strings embedded in the source code and do fully recompile the code. A software with embedded strings can be considered as not internationalized.

(Un)known non-standard formating of strings

In this scenario, there is normally a specific tool to extract the strings to translate. It is recommended to extract the strings and convert them to a known standard format that we can use in a common translation environment.

For example, OpenOffice.org uses a internal format known as GSI/SDF and a tool called transex3 to extract the strings out of the source code. Strings are embedded in a set of files called resources files.

Documentation

Sometimes documentation is written as pure user guides or help documents. In this case, we need to rewrite the whole file trying to keep layout and structure. (ie keeping the meta-code)

Portable Object (PO)

If the software is using PO format for localization (like Gnome or the KDE desktop) we can use functions from the 'gettext' framework to extract strings into Template files or POT files to later create PO files in the target language out of them (see 'The Gettext framework and PO files').

String Translation Process

Big projects as OpenOffice.org can include the translation of more than 40,000 strings. To handle this huge amount of strings, special editing tools are needed. The translation also requires procedures for translation peer reviews and in many cases the advice of external linguistic experts.

When it comes to translation tools, Poedit is freely available for Windows and Linux. Other tools are available under Linux only systems as Kbabel (KDE) and GTranslator (Gnome).

For a small community that together want to localize a software or for people that for some reason is not allowed to install certain software on his/her computer, a web-based translation tool can be used. Possible options are Pootle, Rosetta and Kartouche.

Some localization teams also like to do the job in the “hard way” and use no other editor than 'vi'. Some teams just convert POT files to Comma Separated Files (CSV) and edit them using a spreadsheet program. The best way to do it is the one that works better for team and the very concrete circumstances of every localization project.

XLIFF (XML Localization Interchange File Format) is an emerging translation interchange standard. We will see much more available in this format as converters are being created to move PO files to Xliff and a number of editors both GPL and commercial are being made available.

Compilation (building the software)

As the translation of the strings is completed and peer review and quality assurance has been performed, the PO files should be merged into the code. PO files are converted to a binary format as MO and then used by the the application at run-time.

Merging back the translation into the code and recompile a new version can be a very challenging task in some projects. Some softwares as Firefox allows the creation of languages packs with the translations and enables the possibility of adding new languages without the need of recompiling the source code. In other projects like KDE or Gnome, PO files are compiled into binary form and easily integrated in the system operative locale messages repository. In another projects like OpenOffice.org building a localize version will pass through recompiling the whole code.

Quality Assurance

The final quality assurance of the software must be done by using the software during some time trying to identify what translated strings needs modification or existing bugs in the system. During the QA phase you need to make sure that users can file bug reports and collect feedback. After the testing or evaluation period, the software should modified accordingly and finally released.

The Gettext framework and POT files

The 'gettext' framework is the GNU Internationalization (i18n) library that is commonly used for writing multilingual programs. Gettext was created to support software Internationalization, which means that an application/software can easily be localized/modified to another language. The 'gettext' framework has set up rules for how to standardize the way applications should be written/designed in order to facilitate adaption to a particular language.

The 'gettext' library offers an integrated set of tools and documentation to programmers, translators, and even users to support internationalization.

The functions *gettext* and *ngettext* are used to extract strings and create POT file out of them.

Today, almost 90% applications on Linux platform are localized using the *gettext* framework.

POT files

Extracting strings from an application that is based on gettext is trivial and results in a set of POT files (.pot).

The POT files are ready for the translator to edit, either by hand or using an editing tool. The original string in a POT file is represented by the keyword 'msgid' and the string between a pair of “”. Under each string, there is pair of empty “” and the keyword 'msgstr' to be filled up by the translators.

In general, a POT file looks the following:

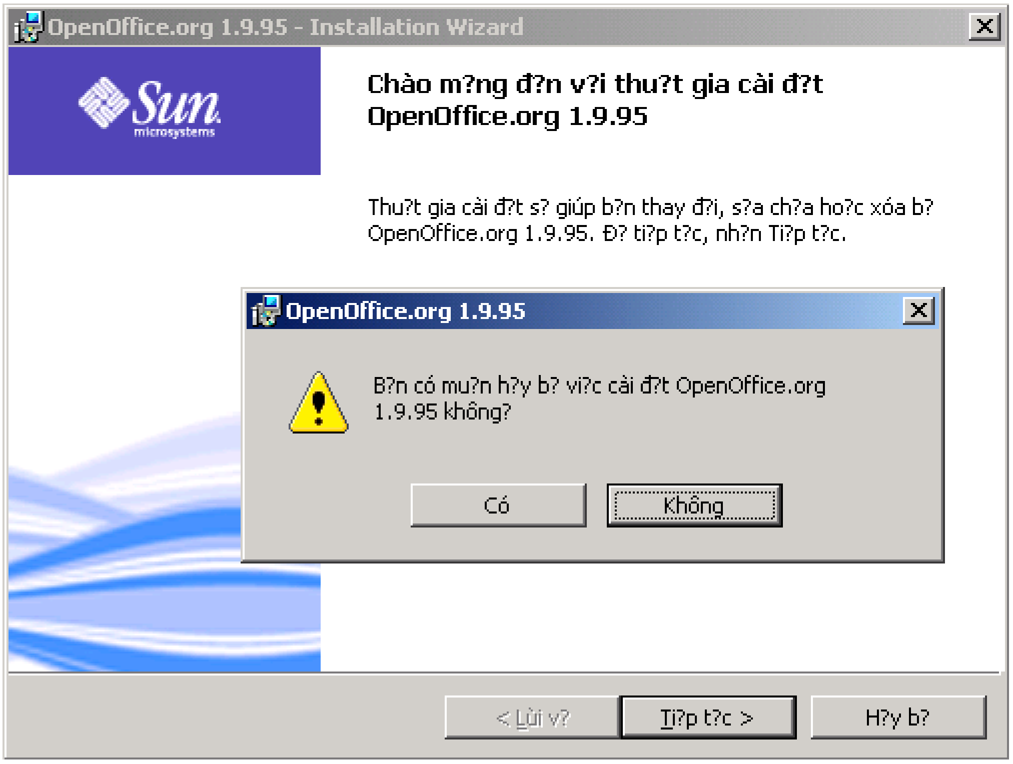
```
white-space
#  translator-comments
#. automatic-comments
#: reference...
#, flag...
msgid “string1”
msgstr “”

#: reference...
#, flag...
msgid “string2”
msgstr “”

etc.
```

As the translator fills up the missing word, a PO file is created. The difference between a POT file and a PO file lies only in the filename extensions and intended usage. POT file has empty msgstr fields and is intended to act as a template for the translators, while the PO file is the translated version of its corresponding POT file.

Creating a Software Installer

As the software has been translated and built, an installer for the application must be created. Sometimes a software installer is part of a third party development environment. If the software installer does not support your encoding, you might need to encode the strings of the "installation process" in an encoding supported by the installer.

The picture below shows a encoding problem in a beta-release of the Vietnamese OpenOffice.org.


OpenOffice.org 1.9.95 - Installation Wizard

Chào m?ng đ?n v?i thu?t gia cài đ?t OpenOffice.org 1.9.95

Thu?t gia cài đ?t s? giúp b?n thay đ?i, s?a ch?a ho?c xóa b? OpenOffice.org 1.9.95. Đ? ti?p t?c, nh?n Ti?p t?c.

OpenOffice.org 1.9.95

B?n có mu?n h?y b? vi?c cài đ?t OpenOffice.org 1.9.95 không?

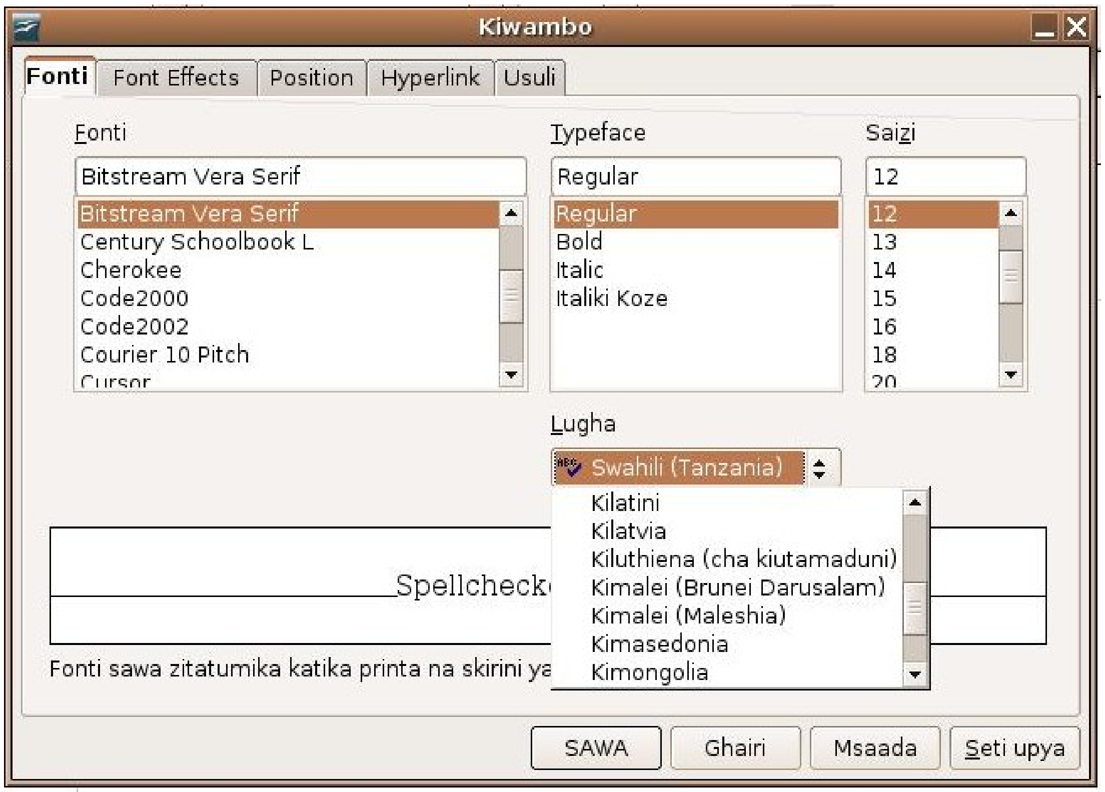
Image 5: Font encoding problems in a OpenOffice.org Installer (development phase)

During the translation

Sorting

The issue of sorting text is a language dependent problem that can cause trouble. Swahili is a good example for that. In Swahili, like in a Bantu languages, the nouns are arranged in a number of classes (15 classes for Swahili). Swahili uses a group of stems to which a number of prefixes are attached to. One of those prefixes are 'ki' that indicates that the stem that follows in a language. In that way the Swahili words for English and Spanish are composed as 'Kiingereza' and 'Kihispania'.

The image below shows an example how the languages are sorted in the Swahili version of OpenOffice.org. The prefix -ki does the sorting a bit difficult for the brain, right?



Fonti: Bitstream Vera Serif
Bitstream Vera Serif
Century Schoolbook L
Cherokee
Code2000
Code2002
Courier 10 Pitch
Cursor

Typeface: Regular
Regular
Bold
Italic
Italiki Koze

Saizi: 12
12
13
14
15
16
18
20

Lugha: Swahili (Tanzania)
Kilatini
Kilatvia
Kiluthiena (cha kiutamaduni)
Kimalei (Brunei Darusalam)
Kimalei (Maleshia)
Kimasedonia
Kimongolia

Spellcheck

Fonti sawa zitatumika katika printa na skirini ya

Image 6: Sorting problem in Vantu languages

Plurals

The issue of plural forms in English is trivial since there only exist only way of doing it, append a 's' in the end of the word. In Bantu languages the case are a bit more complex since plurals are created with prefixes. For example, the Swahili word 'mtoto' (child) is called 'watoto' in plural (children). Another example is 'kitabu' (book) and 'vitabu' (books).

In softwares you can see strings like 'Add file(s)' as an indication that you can add one or more files. The corresponding string in Swahili would be 'Ongeza (ma)faili'.

Introduction to ICT for Development
Course code: 2I4360, 2I1160, IB7000
23 October 2005, Creative Commons Deed. Attribution-NonCommercial-ShareAlike 2.0

How to Plan a localization projects

What is the goal of the project?

The first thing you need to do to start a localization project is to define a clear goal of the project that is suitable in terms of resources (technical, linguistic, financial). Also, the goal must be adopted to the potential users of the software, and not just to your own dreams, if you want to see an user adoption of the software. When you define the goal, you should not have a specific software in mind but rather choose a software depending on your goal.

For example, “Our goal is to localize KDE to Somalii”, just specifies What you will do, not Why you should do it. KDE is a deskop for Linux, is there really such a large Linux community among the Somalii speakers? Maybe is there other software that is more suitable for Somalii.

Avoid the use of very general and undefined goals as “Our goal is to make sure that Somalii speakers can use computers in their native language”. Consider the following: What is needed for a person to use a computer in his/hers native language? To write a document and being able to input all characters of her native language or to be able to change settings of her mail client in her native language?

Defining a clear and narrow goal is of utmost importance for the outcome of the project.

Our experience is, that if no software of any kind exists, but the issues about fonts, encoding systems and keyboards are solved, and you aim to localize a software that as many as possible has access to and will benefit from, the office suite OpenOffice.org or the browser Firefox is suitable softwares to begin with.

What to localize?

Depending on your goal there are a number of useful applications, desktops and linux distributions to localize. However, you should consider the following.

If there exists very little online content in the target language, is it really worth localizing a browser? As, mentioned before, consider how large a potential Linux community in the target language could be? Is it worth localizing a software that only runs in Linux or should you choose one that is cross-platform.

Our recommendations when it comes to defining the goal and the software to localize are the following:

1. End-user Focused
   Have in mind the desktop bound end-user and not the system administrator when deciding which software to localize. A system administrator is more likely to speak English or another international language, than the end user. Consider what application that the end user could benefit from the most (office suites, email programs, web-browsers, instant messaging) etc.
2. Free Software
   In order to reach out to as many as possible, hurdles like license fees must be eliminated, especially in developing countries. Therefore, the software that you localize should be Free of charge.
3. Cross-Platform
   No matter how much you will like to see a world wide Linux community for your native language, you must face the reality and consider a plan to make it happen. Investigate how the situation looks like for your native language first and take that information as a starting point. The reality is that most computers are using Microsoft Windows today. Because of that, it makes little sense to localize a product that can only be used in Linux. At the same time, you should NOT limit yourself to a product that can only be used in Windows. The best way is to choose a cross platform application that allows Windows users to use the product but still offers them to change to a Free Operating System as soon as they are ready for that.

Some major applications that are popular to localize are Mozilla, Firefox, Thunderbird and OpenOffice.org. Both OpenOffice.org and Mozilla fits well into the recommendations given above. GNOME, KDE and XFCE are localizable desktop systems for Linux (*BSD).

The Localization Team

After you decide what is the goal of your project and what you want to localize... you need a team, a localization team. You will need to identify which human resources you need and make a time plan for the activities that needs to be under taken.

In a localization team, one person often has many roles. Here follows a list of people that can be useful in a localization project.

- *Project manager (overall project coordination)*
- *Localization engineer (responsible for all technical matters)*
- *Translator (text translation)*
- *Linguist (glossary, creation of new terms etc.)*
- *Web designer/manager (project website, important for visibility)*
- *Testing engineer (software tester)*
- *Volunteers (testers of software,)*
- *Font designer*
- *QA Specialist (assurance of quality in the final product)*

In a big localization project, the key persons of the team are the overall project coordinator and the translators. Having a good project coordinator will ensure that the project is on track and moving in the right direction. Localization is not a one-day task, it requires perseverance and project planning.

Conclusion

This “Primer to Localization of Software” aims to provide to the reader with a basic understanding of the technical and human aspects that a full localization project requires. Far from what many think, localization is not just the simple translation of strings, but a complex task that includes several interrelated subcomponents. A good project management and planning is essential to achieve good results.

Finally, keep always in mind that tools need to adapt to people and not people to tools!