Summary: A properly internationalized product will satisfy the cultural, linguistic, and moral expectations of any national group of users. This paper describes software internationalization and localization, and introduces techniques for internationalizing and localizing software.

Title: XPG-based Software Internationalization

Author: Scott Trent

Table of Contents:

1 Why is Software Internationalization Necessary?

Historically, the majority of computers and accompanying software were designed to work only in U.S. English. This was acceptable when typical computer users throughout the world were highly paid professionals who had sufficient understanding of the English language. However, with the extensive propagation of low end computers, this is no longer so. Computers are being put into the hands of ordinary people world-wide, who may or may not understand English. Thus suggesting the necessity to translate software messages, prompts, documentation, packaging, etc.

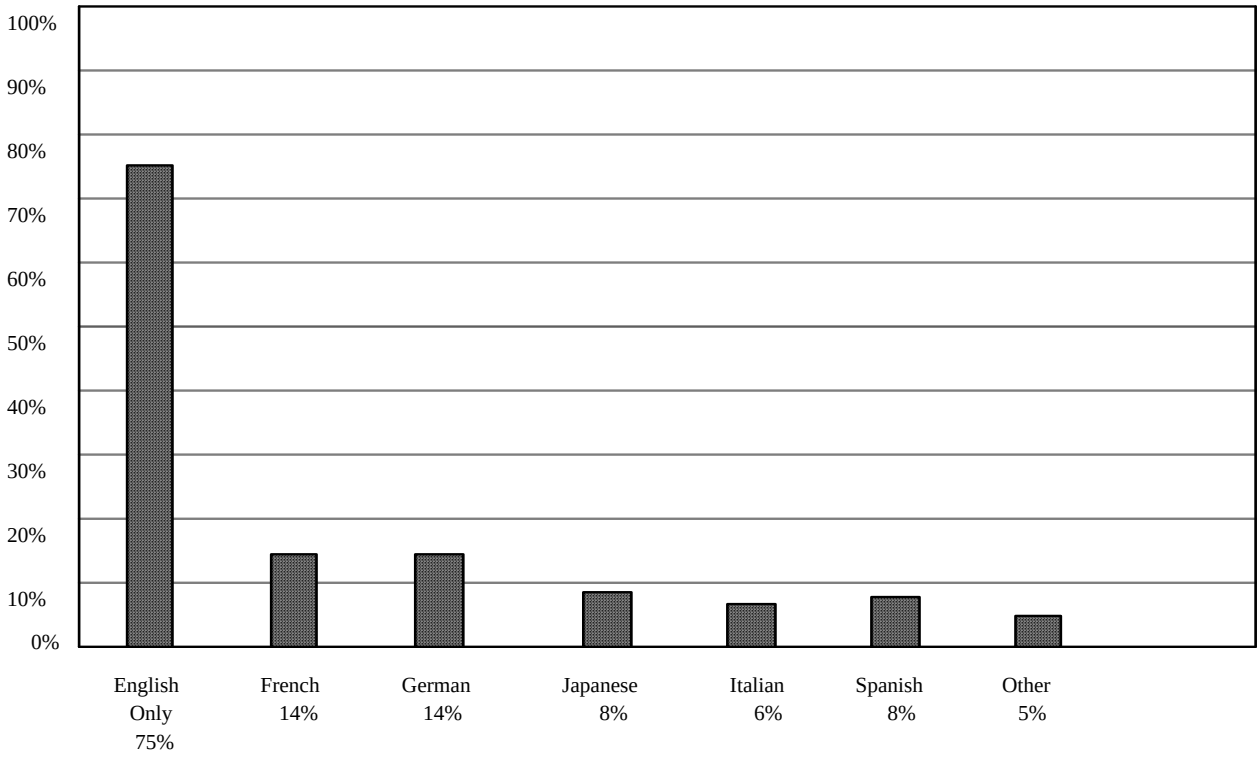
However there are more compelling reasons to internationalize products. It is increasingly becoming a requirement to enter the lucrative international market. Proper internationalization will help differentiate a product and will aid in increasing market share. There are many facts which underscore the potential of this market. As of 1989, Information Technology (IT) was a 200 billion dollar market. 100 billion of this was in the U.S. 57 billion was in Japan, and 40 billion in Germany, UK, France, and Italy. This suggests that a software developer could double his market by going international. Companies who are successful in international marketing do indeed see this.

Why is internationalization a requirement in this market? Even if a software user does not mind English messages and U.S. culturally oriented processing of sorting, character sets, formats, etc. chances are that his customer is expecting fully localized output. Is it conceivable that a user of a French telephone book would be satisfied with U.S. English sorting order? Would the reader of a document created with a Japanese word processor be satisfied if it could not contain kanji? Would accounting software that only displayed U.S. dollars and was written for U.S. accounting practices be useful in Italy? Software must meet not only the computer operators expectations, but also any expectations that the end user may have regarding such issues as language, spelling, hyphenation, collation/sorting, character set, page size, formatting of numbers, currency, date, time, etc. It should be obvious that if users are given the opportunity to use a system that runs in a language they know, and supports their cultural conventions, that they would choose that system over one that doesn't. Would an American user, feel comfortable using a piece of software that only generates prompts and messages in Japanese?

It is important to understand that users expect 100% full internationalization. Any necessary areas in a particular software package that are not internationalized are defects. For example, it is not enough for database software localized for Japan to display kanji, it must also sort in the proper order. To market software that is not fully internationalized would be like marketing a car that is not fully functional! There are all too many amusing anecdotes of companies who have only partially internationalized software or other products, or have made critical mistakes in localization. Some examples include a Japanese isotonic sports drink called, "Poccari Sweat", which did not sell in test marketing in the U.S. until they changed the name to "Poccari." The Ford "Nova" reportedly did not sell well in Spanish speaking countries until it's name was changed to Caribe'. (“Nova” can be interpreted as "doesn't go" in Spanish.) Even icons are subject to misinterpretation. Apparently the Macintosh garbage can looks quite similar to a British mail box. This could lead to an unpleasant situation if a British user attempted to send email files by dropping files into it. In fact, the rural mailbox used on Unix systems (and others) to indicate the arrival of mail, is not intuitively recognized to mean mail in any other country. Translation mistakes are another source of embarrassment to companies that need to work on quality control. One of my favorites comes from the translation of an AIX message, "Sorry, you are not a super user" (so you can not use this function). It was translated into Japanese as 「あなたは優れたユーザーではありません。」 or "You are not a top notch user".

In some cases, internationalization is a legal requirement for product entry to a given country. For example, in the past Japan has had legal restrictions banning the importation of computer systems that do not support kanji.

It is interesting to note that in a Price Waterhouse survey of software developers, 75% of the respondents stated that they supported only English. Correspondingly an average of 84% of their revenue came from the U.S. Even though successful firms receive over 50% of their revenue from overseas, the industry as a whole still has a long way to go. Even so the International Computer Professionals Association of Washington D.C. states that 80% of the software in use throughout the world originated from the U.S.



2 What is Software Internationalization?

There are as many definitions for internationalization as there are authors of papers and books on the subject. Internationalization can refer to the process of designing or modifying software to support the cultural and linguistic expectations of any of its potential users. Even broader yet, it can refer to all areas involved with producing a product which can be marketed in non-domestic markets. In a more restrictive sense, it can refer to the process of designing or modifying software so that it can support localization. Localization is the modification of (hopefully) internationalized software to support the cultural and linguistic expectations of potential users. Quite often the usage of these and other similar terms, such as National Language Support (NLS), or NLS enabled, tend to mean what the speaker or author is thinking of at the moment rather than a strict denotative meaning.

There are many different techniques which are used to internationalize software. These techniques will be introduced later range from retrofitting to the locale model. I will tend to focus on the locale model since it is generally accepted that the locale model is a good starting point for internationalization, and is the basis of the internationalization strategy of such standards as the X/Open Portability Guide (XPG), and POSIX. It is also the foundation for the internationalization of Unix operating systems marketed by IBM, HP, OSF, and others.

To successfully internationalize software, anything that is dependent on certain cultures or languages must be modified. The easiest way to do this is to isolate all such dependent information and add it to a locale database. Hence software need not be modified to support new locales, it simply accesses the proper database. Thus a locale would be created for each country or cultural grouping. Some examples of locales on the AIX operating system for the RISC System/6000 include "En_US" (English as used in the United States; this would have English messages with U.S. cultural conventions), "Ja_JP" (Japanese as used in Japan; this would have Japanese messages with Japanese cultural conventions), "Fr_CA" (French as used in Canada; this would have French messages, and use Canadian cultural conventions), and many more.

The use of locales like this makes it very easy for a user to switch from one locale to another. In the example below, the locale starts out as "En_US" (English), and the current date is displayed in a format acceptable in the U.S. Next the locale is changed to "De_DE" (German), and the date is displayed again with the same command, only this time it is in a format acceptable in Germany, using appropriate abbreviations for the month and day of the week. Next the locale is set to "Ja_JP" (Japanese) and a command is issued to display a non-existent file. This generates a "file not found" error message in Japanese. Lastly, the locale is set back to "En_US" and the same non-existent file access is attempted, only this time the message is in English. Similar simple examples could have demonstrated sorting, monetary format, etc.

```
[trent@musashi] /u/trent > export LANG=En_US
[trent@musashi] /u/trent > date
Tue Oct 26 10:33:06 CST 1999
```

```
[trent@musashi] /u/trent > export LANG=De_DE
[trent@musashi] /u/trent > date
Di 26 Okt 10:33:13 1999

[trent@musashi] /u/trent > export LANG=Ja_JP
[trent@musashi] /u/trent > cat this_file_does_not_exist
cat: this_file_does_not_existがオプーンできません。

[trent@musashi] /u/trent > export LANG=En_US
[trent@musashi] /u/trent > cat this_file_does_not_exist
cat: Cannot open this_file_does_not_exist.
```

The next logical question is, what exactly is culturally dependent information? Although I base my presentation of culturally dependent items on the six locale categories specified in XPG, I will illustrate why certain items are necessarily culturally dependent, and will point out some short comings in the XPG specification. There are many additional culturally dependent items which need to be addressed in the process of internationalization. The six XPG locale categories are collation (LC_COLLATE), character types (LC_CTYPE), messages (LC_MESSAGES), monetary notation (LC_MONETARY), numeric notation (LC_NUMERIC), and time notation (LC_TIME).

2.1 XPG: Character Classification Category (LC_CTYPE)

The first category, LC_CTYPE, defines character classification, case conversion, and other character attributes. XPG explicitly allows for the following character classifications: `upper`, `lower`, `digit`, `space`, `cntrl`, `punct`, `graph`, `print`, `xdigit`, and `blank`. The definition of which categories characters fall into can change from locale to locale. XPG also allows for additional classifications such as hiragana, katakana, kanji, etc. System provided API's such as `isctype()` can make use of this information. Case conversion tables are also specified in the locale database, since this is another area which changes depending on the locale. Case conversion can be accomplished with system functions such as `toupper()` and `tolower()`, which access these character mappings. Note that most non-alphabetic characters including kanji, kana, hangul, hanzi, etc. do not have uppercase or lowercase attributes.

Related issues include support for different character sets. There are tens if not hundreds of different character sets for the world's countries. An internationalized program should support all character sets which are used in the target countries. Many code sets, especially those for Western languages are Single Byte Code Sets (SBCS), that can be represented in a single byte. However there are code sets, notably, Japanese, Chinese (traditional and simplified), and Korean, which can not be represented in a single byte, and are thus referred to as Double Byte Code Sets (DBCS). Perhaps Multi-Byte Code Set (MBCS) is a more accurate in some cases, as certain code sets provide one, two, three, and four byte characters.

Supporting a code set implies the ability to enter its characters with the keyboard, display it on the screen, print it to a printer, and in general process it as one would expect to be able to. Additionally, data portability should be provided with the ability to convert characters from one code set to another. (This is addressed in XPG with the command `iconv` and the library function `iconv()`.)

2.2 XPG: Collation Category (LC_COLLATE)

The LC_COLLATE category provides a collation sequence definition that is used by many of the commands and library functions on an XPG compliant system. Collation can be much more complicated than the simple ASCII ordering of characters as we often find in English. For example, Spanish has multi-character collating symbols such as "ch", "cz" and "ll". "ch" must collate between c and d, unless it is in the middle of a word, in which case it collates after "cz" and before d. "ll" must collate after l and before m. And the character – collates between n and o. This results in a sorting order quite unlike that for English.

Many of the European languages also have different sorting requirements. In Germany the characters å , ä, and æ collate before b, however in Sweden they collate after z. In German, the sharp character must be treated as "ss", and the Ö character must be treated as "oe".

Japanese has three major methods for sorting text. (1) Telephone directory order (based on the first syllable of the canonical pronunciation of each kanji), (2) Dictionary order (based on the actual pronunciation of the word), and (3) JIS ordering (similar to ASCII order, in which each character is given a value, and characters are sorted according to that value. e.g. just as the character A has the value 65 in ASCII, and collates before the letter B (value 66), the hiragana character か (ka) has the JIS KUTEN value of 0411 and collates before the character き (ki) (value 0413).

2.3 XPG: Monetary Notation Category (LC_MONETARY)

The LC_MONETARY category defines the rules and symbols that are used to format monetary numeric information. This information is available from either the `localeconv()` or `nl_langinfo()` functions, and is used by the monetary

formatting function `strfmon()`.

XPG allows the specification for each locale of an international currency symbol for that locale (e.g. USD), the currency symbol (e.g. $ for the U.S. or ¥ for Japan), the radix character/decimal point (e.g. comma for many European countries), the character for the thousands separator, information on how to group digits (U.S. groups digits in groups of threes, but other combinations are possible), the character for the positive sign, the character for the negative sign, the number of digits to display to the right of the decimal point, where to put the currency symbol (e.g. prefix, infix, or postfix. Respectively, $11.34, 11$34, 11.34$), whether or not to put a space between the currency symbol and the amount, whether or not to change the position of the currency symbol if the value goes negative, where to place the positive and negative signs, and whether or not negative or positive values should be indicated by enclosing the value in parenthesis.

Areas not addressed by XPG include padding (Should it be done? If so, how and with which characters?) and rounding. Monetary rounding is done differently in Argentina and Switzerland than it is done in the U.S. In Switzerland, monetary values are rounded to the nearest .5 rather than the nearest unit. That is NN.(N-1)76 through NN.N25 round to NN.N0. NN.N26 through NN.N75 round to NN.N5. For example, 12.325 rounds to 12.30. 12.326 rounds to 12.35.

2.4 XPG: Numeric Notation Category (LC_NUMERIC)

The LC_NUMERIC category defines the rules and symbols that will be used to format non-monetary numeric information. This information is available through the `localeconv()` and `nl_langinfo()` functions. It is used by system routines as necessary.

XPG allows for the specification of a radix character, thousands separator, and grouping information in this category. The issue of padding and rounding is not addressed in XPG.

2.5 XPG: Time Format Category (LC_TIME)

The LC_TIME category defines the information used to display dates and times. This information is used by `strftime()`, `wcsftime()`, `strptime()`, `nl_langinfo()`, and other functions, as well as being used by many commands.

Information which is defined in this category includes:

- abbreviation for days of the week (e.g. Mon, Tue, Wed, ...)
- full name for days of the week (e.g. Monday, Tuesday, ...)
- abbreviation for names of the month (e.g. Jan, Feb, ...)
- full names for months (e.g. January, February, ...)
- appropriate formatting for dates, including delimiters, etc.
- appropriate formatting for times, including delimiters, etc.
- appropriate formatting for date/time, including delimiters, etc.
- strings to use for AM/PM (e.g. AM:PM for English, 午前:午後 for Japanese)
- appropriate formatting for 12 hour time, including delimiters, etc.
- description of eras (Used to display a date using Japan's emperor reign based era calendar.)
  - offset (what year the calendar starts with)
  - direction (does it count up like A.D. or down like B.C.?)
  - starting date of era (in U.S. date format!)
  - ending date of era (in U.S. date format!)
  - era name (this can be in kanji or any other character set)
  - formatting information for era based date, time, time/date.
- alternate digits (e.g. kanji digits, Hebrew digits, even roman numerals, etc.)

XPG allows for other mechanisms to support time zones, day light savings time, leap years, leap days, etc. However, detailed attention is missing for support of totally different calendars, such as Julian days or lunar calendars. Note however, that era support would allow use of calendars such as are used in Israel and Thailand, since they are essentially era based.

2.6 XPG: Message Category (LC_MESSAGES)

The LC_MESSAGES category defines the format and values for affirmative and negative responses. The message system falls under this category as well. The message system allows internationalized programs to read messages out of separate translated message files called message catalogs. The XPG message system is explained in the Section 3.4.

In addition to translated message catalogs, this category supports the specific denotation of regular expressions (a pattern to match strings) for “yes” and “no”, as well defining actual valid “yes” and “no” strings for a particular locale.

The XPG items discussed above may seem conclusive, but they are just the beginning of the many issues which must be analyzed and isolated for effective internationalization. The following sections describe some other culturally dependent

items.

2.7 Translation

It is obvious that certain text such as screen output, help screens, menus, user input, documentation, packaging, etc. must be translated. Care must be taken to use consistent terminology throughout the entire package to avoid user confusion. Additionally, it goes without saying that mistranslation and strict literal translation should also be avoided. Indeed, slang and unnecessary jargon should be avoided in the original English. Since these can sometimes be difficult for native speakers of English to understand, it is almost unreasonable to hope that translators will convey the proper meaning.

There are still many other things which need "translation". For instance, sound clips in multimedia packages may need translation. Especially if they contain voice messages. Even something as simple as sounds of emergency vehicles (e.g. fire engines) or the sound of a phone ringing can vary greatly from country to country and should be translated into sounds expected and understandable to people in each country or region. The same goes for icons, clip art, and all pictures in general. The "kitchen sink" icon used for the Emacs editor is a good example. In American English this implies that Emacs contains everything INCLUDING the kitchen sink, however this American idiom is not necessarily understood in other cultures. (I would think that a Swiss Army pocketknife would do a better job of portraying a software tool capable of nearly anything, but even this is certainly not universal.) Additionally offensive graphics, icons and packaging must be avoided. It has been pointed out that **any** usage of common body parts, such as a hand or head, is guaranteed to be considered offensive somewhere in the world. Likewise usage of animals or other common objects can have widely differing impressions in different countries. Symbols that suggest flags or ideologies may give offence, for example a graphic resembling the Star of David would be offensive to many Middle Eastern countries. Even usage of color varies greatly from country to country and should be considered a translatable item. For example, in the U.S. the color red can indicate danger, where as in China it generally signifies happiness and good luck. In fact the roles of red and green in the U.S. are opposite to those in Germany. An unpretentious green "on" light on a U.S. product would tell a German user that the product had broken down!

2.8 Other Data Formats

Differing data formats between countries must also be taken into account. Some simple examples include weights and measurements (e.g. Metric system, English system, traditional Japanese measurement system, etc.) The sizes of paper, address formats, phone number formats, and conventions for writing names can all vary drastically from country to country. Even such issues as page number placement should be considered.

2.9 Other Notational Conventions

Care must be taken in the use of mathematical notations and symbol usage. For example, in the U.S. the term "billion" means one thousand million, or 1,000,000,000 where as in Europe "billion" means one million million, or 1,000,000,000,000. Likewise in the U.S. a trillion is one million million, where as in British English it is considered to mean one million million million.

The ± character is a good example of a symbol which has different meaning in different countries. It means plus/minus in many countries, yet in Belgium it means "approximately". Likewise, while the tilde character ~ can be used in American English to mean “approximately”, in Japanese, it can be used to indicate range. The usage of the percent symbol also changes from country to country.

2.10 Hardware support

Hardware support is another critical issue. Initially, hardware and operating systems which may be quite popular in one country may be insignificant or even unheard of in another country. The software developer must be concerned with compatibility for these platforms. For example, in the 80’s and early 90’s NEC produced a popular PC line in Japan which ran a version of DOS that was largely incompatible with American MS-DOS. In fact, the disk formats were even different! Quite often each country will have its own keyboard standards and requirements which must be supported to allow input of that countries character set. Depending on the computer system and language used, a Front End Processor (FEP) may be utilized to provide a mechanism for the user to enter characters. High resolution monitors and printers are critical in displaying such complex characters as kanji, hanzi and hangul. In fact one of the reasons for the early proliferation of inexpensive 24 pin printers and 640x480 VGA resolution monitors and video cards is that they are required to process readable Japanese.

2.11 Text Formatting

The many issues involved in text formatting can get quite complex when attempting to support various languages. Some of these issues include text orientation (horizontal, vertical, right to left, left to right (e.g. bidi), etc.), punctuation requirements and rules, spacing (Japanese, Chinese and Korean do not use spaces to delimit words), capitalization rules, hyphenation, spelling, determination of word boundaries, character boundaries, paragraph boundaries, kerning, fonts, etc. Certain languages such as Arabic, Farsi, Urdu, and Hebrew also have to be concerned with presentation and layout. With these

languages, text is stored in a generic format in these languages, however on output each character may use one of four possible presentation forms depending on its position in a word.

2.12 Marketing Issues

Appropriate marketing activities are probably just as important as the internationalization of the software itself. Packaging and advertising must be in tune with cultural morals and expectations. Offensive graphics, symbols, colors, etc. must be avoided. One of the more amusing potential pitfalls are ads which show cause and effect. For example, consider a billboard with three frames showing a sick person, the same person using a medical product, and finally that person well and cured, sequenced from left to right. This would be fine for either the U.S. or England, however, it would convey the opposite meaning in many Arab countries where text is generally read from right to left.

Additionally, absolutely critical issues such as sales activities, product distribution, customer support, etc. must be tailored to meet end user/consumer expectations.

2.13 Mixed Locale Requirements

Another interesting requirement is that of mixed locales. I will illustrate this with the XPG locale model based on the 6 categories; LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, and LC_TIME. It is possible to set each category individually. Generally a user will simply set the LANG environment variable to the language that he wishes to use, and then accepts the cultural conventions/notations that come with that language. However, it is possible to use nearly any combination of these categories. For example, a user in France who spoke only English wishing to create output for use in Germany might set LANG to German to ensure that all categories not specifically set to another language will default to German, then he may set LC_MESSAGES to English so that screen prompts show up in English, and finally he may set LC_TIME to French out of a desire to see the date and time displayed in French convention. This ability to mix and match is one of the advantages of the locale model.

As nice as this seems, the multilingual model takes this one step further. I would like to illustrate this briefly by referring to the three levels of internationalization referred to in the article "What is the Internationalization of Applications" from SuperASCII.

Level l International (i.e. localized) versions of software are available, but are hard-coded to support exactly one country per executable.

Level 2 International versions of software support two countries/languages per executable. (e.g. possibly English and Japanese, or English and French.)

Level 3 Single international (i.e. internationalized) version supports a wide variety of languages, one at a time, by changing run time parameters. (e.g. XPG locale model.)

However, this approach can be taken one step further to include the multilingual model.

Level 4 International versions support a large number of languages (character sets, fonts, etc.) simultaneously.

This forth level would for example, allow the same document to contain Japanese, French, English, and Korean. The Unicode code set is an excellent starting place for operating systems and applications wishing to provide multiple concurrent language support.

3 How does one Internationalize and Localize Software?

There are two broad classifications of techniques to write code that is appropriately localized. These are retrofitting and the locale model. I will discuss the advantages and disadvantages of both. I will then give detailed examples of how one might implement messaging and character processing, two of the more critical areas in internationalization, both with and without the XPG locale model. Obviously, a truly internationalized program must take into account all of the issues brought up in Section 2, however a detailed coverage of how to implement all of those items is far beyond the scope of this paper.

3.1 Retrofitting

To retrofit a program is to make modifications to it such that it is localized for a specific region. At the simplest level, this means that if a software developer wishes to market a program to five countries, that he would need to produce five different versions of the same program, each with the necessary changes for the target country. In the U.S., each international version is generally based on the original English program. While as this technique does have many drawbacks, it is the easiest to understand, and may even be the most economical approach if the program only needs to be localized for one additional country.

The easiest way to illustrate this approach is with trivial examples:

English Version:

```
main()
{
printf("Hello.¥n");
}
```

Spanish Version:

```
main()
{
printf("Hola.¥n");
}
```

Japanese Version:

```
main()
{
printf("今日は¥n");
}
```

The English version has been modified in each version to include the appropriately translated hello message. This approach generally requires translators to either modify the source code itself, or to work very closely with the software developer. This has a number of drawbacks. In a complicated program, it is not obvious what should and should not be translated, which can lead to either output with mixed languages, or worse to a program which will not compile or run correctly. Modification of a program by a non-programmer can have disastrous results. Obviously, stray characters added in the wrong place can prevent the program from compiling, and something as innocuous as adding or removing a semicolon from the wrong location in a C program can drastically change its output and processing. Additionally many of the issues covered in Section 2 really can not be addressed by a translator alone. Either the person localizing the software must be a programmer or must work very closely with the programmer. Also, since source code is proprietary, it is desirable to minimize the number of people with access.

The result is `n` different versions of source code, and `n` different versions of the final executable for `n` countries. This quickly becomes very expensive to maintain. A bug found in the original code must now be fixed in `n` different **unique** programs. Distribution and user support also grows in complexity since `n` programs must be supported. This affects the customer too, since if the customer wishes to run the program in two or more languages, he must actually purchase and maintain several copies of the program.

This approach often leads to international releases lagging months or years behind the release of the original product. This is no longer an acceptable approach in today's software market. Where ever possible a product must be available worldwide at the same time it is available domestically.

3.2 Better Retrofitting

There are a number of ways that the original retrofitting approach can be improved short of adopting the locale model. For example, rather than creating `n` different versions of the source code, compiler directives such as `#ifdef` can be used to create `n` different executables based on a single source program. Even though the resulting single program is significantly more complex, maintenance costs should be reduced since there is only a single version of the source code. However, the problem of multiple executables remains.

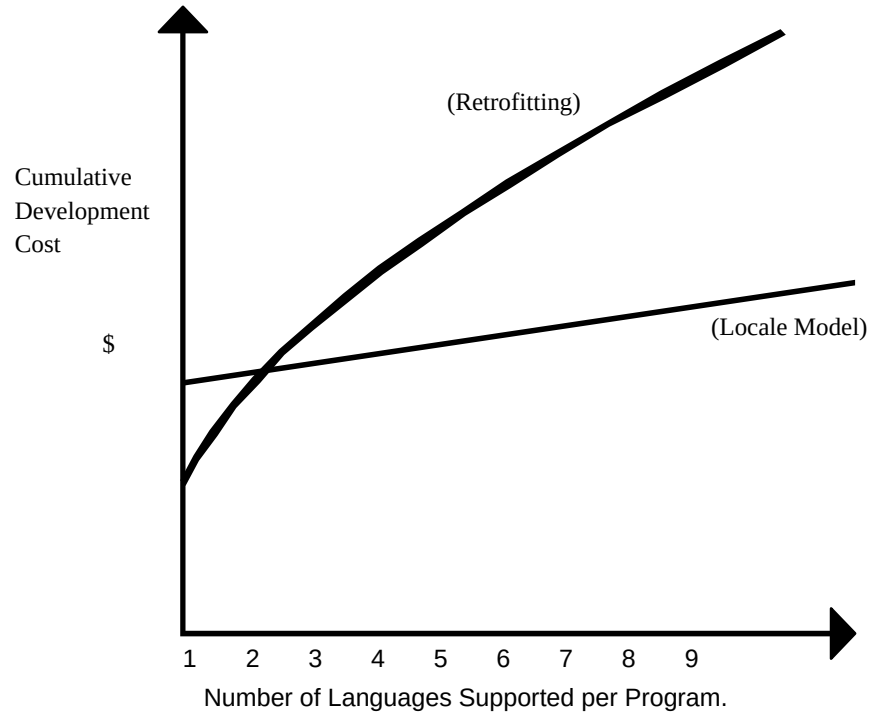
Example:

```
main()
{
#ifdef US_ENGLISH
printf("Hello.¥n");
#endif
#ifdef SPANISH
printf("Hola.¥n");
#endif
#ifdef JAPANESE
printf("今日は ¥n");
#endif
}
```

3.3 Locale Model

The best solution to date is the locale model, in which locale specific information is stored in a database or locale object that can be specified at run time. There are three steps to developing software with this model. First, the original program must be written such that it depends on the locale database for all cultural specific information, and should use a message catalog rather than hard-coded messages. This step need only be done once per program. Second, culturally specific information for each locale must be stored in the locale database. This step need only be done once per locale, and is often provided as part of the operating system for XPG compliant systems. Third, the messages for each program must be translated into each target language. Translation becomes somewhat easier, as all translated text is isolated in a message catalog.

Thus, if a program is properly written to make use of the locale model from the start, a vendor can ship a single executable worldwide accompanied by the appropriately translated message catalogs. Users can select which language they wish to use at run time.



3.4 Messaging with the XPG Locale Model

To use the messaging in the XPG locale mode, the developer will create a message catalog and modify the program to use this message catalog. For example, suppose this is the original program:

```
/* prog.c before message catalog usage */
main()
{
printf("message 1¥n");
printf("message 2¥n");
printf("message 3¥n");
}
```

The message source file for English would contain the messages as follows:

```
$ prog.msg  English
$ This is the source for the message catalog.

1 "message 1¥n"
2 "message 2¥n"
3 "message 3¥n"
```

The message source file for Japanese would contain the messages as follows:

```
$ prog.msg  Japanese
$ This is the source for the message catalog.

1 " メッセージ 1¥n"
2 " メッセージ 2¥n"
3 " メッセージ 3¥n"
```

Then, this message source file must be converted, using the `gencat` command, to a message catalog usable by XPG functions.

Next, the original C program must be modified to make use of message catalogs. This program can then be used without further modification to display messages in any language that has a corresponding locale entry and message catalog.

```
/* prog.c with message catalog usage */
#include <locale.h>
#include <nl_types.h>
```

```
main()
{
nl_catd catd;

(void)setlocale(LC_ALL,"");

catd=catopen("prog.cat",0);
printf(catgets(catd,1,1,"message 1¥n"));
printf(catgets(catd,1,2,"message 2¥n"));
printf(catgets(catd,1,3,"message 3¥n"));
}
```

The message text used as the fourth parameter to the `catgets()` function is a default message to be used if the message catalog for the user specified language is not found. AIX supports extensions which allow the programmer to replace the second and third parameters with mnemonic strings such as `"SET1"`, or `"message2"` which results in easier to understand code.

Sample output:

```
[trent@musashi] /u/trent > LANG=En_US prog
message 1
message 2
message 3

[trent@musashi] /u/trent > LANG=Ja_JP prog
メッセージ  1
メッセージ  2
メッセージ  3
```

There are a number of guidelines for writing understandable messages. One should avoid hard-coding any kind of English into a program, default messages should always be used in the `catgets()` routine, and allowance should be made for expansion of translated text. Messages tend to expand 20 to 30 percent or more. It is a very good idea to be liberal in commenting message source files. Copious explanations may clear up translator misunderstandings before they become a problem. Strings should not be concatenated together to form output. For example, a talented programmer may be tempted to write output like this, `"The printer is no%c ready."`, where `%c` is replaced with either the letter `"t"` or the letter `"w"` depending on context. This is clearly not translatable. Again, it may be tempting try a compromise with `"The printer is %s ready."`, where `%s` is replaced with either `"now"` or `"not"`. Yet this too is not translatable to all languages, **even** if the strings `"now"` and `"not"` are translated. The correct approach is to add the easily translatable strings `"The printer is not ready."` and `"The printer is now ready."` into the message catalog, even though it may require a few more bytes.

3.5 Messaging with Other Frameworks

Messaging can be accomplished within frameworks other than the XPG locale model. Perhaps the simplest example would be to simulate the XPG functions by creating a flat ASCII text file containing one message per line, and then writing subroutines that will display the specified line number. This would fulfil the requirement to keep hard-coded messages out of the source code and in a separate editable source file. In fact, the Turbo Pascal compiler from Borland (version 3.0) used this method in the early 80's to display error messages. There are commercial packages which remove strings from programs and automatically modify the programs to make use of the generated messages files, which can in turn be more easily translated than if the messages were still in the program. The MIT X-Window system uses resource files which can be used to contain translatable information.

3.6 Character Processing with the XPG Locale Model

The advantage of character processing with the XPG locale model is that it can be assumed that an infrastructure already exists to display and manipulate characters in various languages. This greatly simplifies the task of writing generalized programs which can process any system supported language, character set, code page, etc. With AIX and other XPG compliant systems, this gives the developer a wide range of languages that can be supported easily.

To fully support the locale model, a program can make no assumptions about the character set that it is processing. It must be able to handle them all. It can not assume single byte data. It can not assume double byte data. It can not even assume that all uppercase alphabetic characters come in order and have values 65 through 90, or that the value for the character B directly follows the value for the character A, as in ASCII. This makes the task of character processing slightly more challenging, but is required in order to allow programs to support any language.

This section only touches on a couple of the many issues in character processing. For more detailed coverage of internationalization and character processing using the XPG locale model please refer to the annotated bibliography at the end of this document.

To understand character processing under the XPG locale model, one must appreciate the differences between the traditional one byte character, multi-byte characters and wide characters, as well as the distinction between file codes and process codes. A program which is written to support any character set must necessarily be able to handle both traditional single byte characters as well as multi-byte characters for input and output. This means that when a program is scanning through data, it is not possible to simply increment a pointer by one byte for each character, nor is it possible to advance by any constant number of bytes since the input data could be a mixture of single and multi-byte characters. The concept of a wide characters was introduced to overcome this. Each single or multi-byte character (sometimes referred to as "file code", since they are the only valid characters within files) has a wide character representation (sometimes referred to as "process code", since they can only be used within processes, and can not be used directly for input/output.) To further complicate matters, even though there are XPG character manipulation functions for most operations that can use both file codes (single and multi-byte characters) and process codes (wide characters), there are certain operations which can only be performed on either process codes **or** file codes, but not both. Fortunately, there are functions which will convert data from file code to process code and back. One of the advantages of using wide characters is that it is possible to scan through data, knowing that each character (process code) has a fixed width, regardless of whether it was a single, double, triple, or quad byte character as a file code.

Many traditional programming assumptions about characters are no longer applicable in international programming. For example the terms byte and character used to be synonymous. This is obviously no longer always accurate. Interestingly enough with the recent introduction of triple and quadruple byte characters as found in EUC character sets, some internationalization assumptions that were valid not too long ago, are no longer valid. For instance, double byte characters used to always take up exactly two display positions on output, so the number of bytes in a character string was always the same as the number of display positions it took up. However, this is no longer true since there are three and four byte characters which take up only two display positions. Indeed, there are two byte characters which only take up a single display position (half width katakana in EUC). This necessitates the use of the XPG function `wcwidth()` which returns the number of display columns that a wide character will take up. This is an example of an operation that can only be done with wide characters. There is no corresponding function that will return the display width of multi-byte characters, without first converting them to wide characters.

Most of the nuances of international programming with regard to character processing can be summed up by stating that "characters can contain any number of bytes, can take up any number of display positions, and **must** be treated as whole units." Violation of this principle **will** result in data corruption.

3.6.1 Searching for Character Data

I will demonstrate several functions to illustrate how to search for character data. First, as a basis of comparison, let us look a traditional method for searching for a single byte character in single byte data.

```
/* searching for single byte character data    */
/* ptr points to target data on exit           */
/* if target not found ptr points to null      */
(void) find_char(char *ptr, const char target)
{
while ((*ptr !=0) && (*ptr != target))
       ptr++;
}
```

Next, let us look at the wide character version of the same function. It is the caller's responsibility to provide all of the data in wide characters. You will notice that this code fragment looks quite similar to the single byte version since wide characters have a constant width. This function is also significantly simpler than the multi-byte routines I will demonstrate later. The trade-off is that conversion to wide characters consumes some computer resources.

```
/* searching for wide character data           */
/* wc_ptr points to target data on exit        */
/* if target not found wc_ptr points to null   */
(void) find_char (wchar_t *wc_ptr, const wchar_t target)
{
while ((*wc_ptr !=0) && (*wc_ptr != target))
       wc_ptr++;
}
```

Next is an example of how **not** to search for multi-byte data. This code essentially treats the multi-byte character pointed to by `target` as a string, and searches for the first occurrence of that string in the search data. This may sound like a valid approach, but it is not since it does not treat each character in the scanned data as an atomic unit, and may return an invalid answer. To illustrate this further, suppose the input data was:

	Input Data:	Target Character:
Kanji	日　本　語	楔
JIS	467C 4B5C 386C	5C38

Clearly 楔 does not exist in the input data, **however**, the byte stream 5C38 does (the last byte of 本 and the first byte of 語), and this algorithm would mistakenly interpret that byte stream as the target character.

```
/* searching for multi-byte data (bad example) */
/* mb_ptr points to target data on exit        */
/* if target not found mb_ptr points to null   */
(void) find_char (char *mb_ptr, const char *target)
{
int target_len = mblen(target, MB_CUR_MAX);
int found = 0;
int i;

if (target_len <= 0)
       /* an invalid character was passed in, do error recovery */

while ((*mb_ptr != 0) && (found == 0)) {
       if (*mb_ptr == *target) {
              found = 1;
              for (i=1; i<target_len; i++)
                     if (mb_ptr[i] != target[i])
                            found = 0;
              }
       if (found == 0)
              mb_ptr++;
       }
}
```

The next algorithm shows the proper way to search for multi-byte data. The important distinction between this algorithm and the last one is that characters are treated as atomic units. `mblen()` is used to determine the length of each character in the input array, then this amount is used to increment the pointer after each character comparison. This ensures that we always start character comparisons at the beginning of a character. There are more efficient methods to search for character data, but this should be sufficient for pedagogical purposes.

```
/* searching for multi-byte data (good example)       */
/* mb_ptr points to target data on exit        */
/* if target not found mb_ptr points to null   */
(void) find_char (char *mb_ptr, const char *target)
{
int target_len = mblen(target, MB_CUR_MAX);
int found = 0;
int i, len;

if (target_len <= 0)
       /* an invalid character was passed in, do error recovery */

while ((*mb_ptr != 0) && (found == 0)) {
       len = mblen(mb_ptr, MB_CUR_MAX);
       if (len <= 0)
              {invalid character, do error recovery}
       if (len == target_len) {
              found = 1;
              for (i=0; i<target_len; i++)
                     if (mb_ptr[i] != target[i])
                            found = 0;
              }
       if (found == 0)
              mb_ptr += len;
       }
}
```

3.6.2 Deleting/Inserting Character Data

Character deletion must always be done on an entire character. Let us observe what happens if the first byte of a two byte character is deleted without deleting the second byte.

	Original Data:
Kanji	日 本 語
JIS	467C 4B5C 386C

	First byte deleted:
Kanji	<XX> 楔 l

JIS 7C4B 5C38 6C

Clearly the resulting data has been damaged. The character 7C4B represented by <XX> is not even a valid character and would be treated as invalid data, and the second remaining character was nowhere to be found in the original data. I would like to stress that this is not a contrived example, but that errors in character handling of this type will always result in data loss. It should be obvious that data loss will also occur if any kind of data is inserted (line splitting, character inserts, etc) in between the two bytes of a double byte character. Data insertion must be done on character boundaries.

3.6.3 Aligned Output

Since characters can take up varying number of display columns, such tasks as padding fields must be done slightly differently. I will illustrate with a function that returns the number of spaces needed to pad a field to a certain number of columns in single byte environment.

```
/* returns the number of spaces needed to      */
/* pad a field in single byte environment      */
int pad_chars(const char *str, int field_len)
{
if (strlen(str) >= field_len)
       return (0);
else
       return (field_len - strlen(str));
}
```

Since the length of a string in bytes no longer equates to its display width, the `wcswidth()` function will need to be used. Since this function requires wide character input, I will write this function to use wide character data.

```
/* returns the number of single-byte spaces needed    */
/* to pad a field in a multi-byte environment  */
int pad_chars(const wchar_t *wc_str, int field_len)
{
if (wcswidth(str) >= field_len)
       return (0);
else
       return (field_len - wcswidth(str));
}
```

3.6.4 Character Classification

Traditionally programmers using ASCII have been able to make assumptions about the underlying numeric values of characters. For instance it was possible to assume that if a character had a numeric value of 48 through 57, that it was a digit. If it was in the range of 65 to 90, it could be assumed to be an uppercase character. Lower case characters were from 97 to 122. One could even use tricks like OR'ing (adding) 32 to uppercase characters to make them lower case characters, or XOR'ing (subtracting) 32 to do the opposite. These are still valid assumptions **if** one is only working in an ASCII environment, but should be avoided in an internationalized program. XPG provides the following functions to determine the type of a multi-byte character: `isalnum()`, `isalpha()`, `isascii()`, `iscntrl()`, `isdigit()`, `isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`, and `isxdigit()`. The following functions will work on wide characters: `iswalnum()`, `iswalpha()`, `iswascii()`, `iswcntrl()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`, `iswpunct()`, `iswspace()`, `iswupper()`, and `iswxdigit()`. It is also possible to use the `iswctype()` function to determine if characters have language specific attributes, such as hiragana, katakana, kanji, etc. For example, this checks to see if character `wc` is a kanji: `iswctype(wc, wctype("kanji"))`. Conversions from to uppercase are done with `toupper()` and `towupper()`. Conversions to lowercase are done with `tolower()` and `towlower()`.

Use of these functions can actually make a program much more readable. Compare the following code fragments:

```
/* if character is uppercase ASCII, convert it to lowercase. (obscure code) */
if ((*ptr>=65) && (*ptr<=90))
       *ptr |= 0x20;


/* using XPG functions, convert uppercase character to lowercase */
if (iswupper(*wc_ptr))
       *wc_ptr = towlower(*wc_ptr)
```

ANNOTATED BIBLIOGRAPHY

Government Printing Office. *Office Style Manual*. 1984. (Among other things, contains reference information for 19 languages on such linguistic items as pronunciation, sample characters, word division, capitalization, punctuation, abbreviation, cardinal and ordinal numbers, months, days, seasons, time, etc.)

IBM Corporation. *AIX Version 3.2 for RISC System/6000: Internationalization of AIX Software -- A programmers' Guide*. 1992. (A technical reference manual written to teach UNIX C programmers to write properly internationalized code for AIX.)

----- . *AIX version 3.2 for RISC System/6000: National Language Support*. International Technical Support Centers. 1992. (Detailed user level description of what NLS features are available for AIX, and how to use them. Topics include locale system, code sets, message catalogs, customization, etc.)

----- . *Keys to Sort and Search for Culturally Expected Results*. International Technical Support Centers. 1990.

----- . *National Language Design Guide: Volume 1, Designing Internationalized Products*. National Language Technical Center. 1996. (A good, but brief introduction to the do's and don'ts of designing internationalized products with detailed examples.)

----- . *National Language Design Guide: Volume 2, National Language Support Reference Manual*. National Language Technical Center. 1994. (A rather thorough compilation of localization information for a number of different countries. Categories of information include character sets, currency/date/time/number formats, accepted languages, sorting information, keyboard information, etc.)

----- . *National Language Design Guide: Volume 3, National Language Support: Bidi Guide*. National Language Technical Center. 1995. (An introduction to bidirectional language processing and requirements for systems which support it.)

Lunde, Ken. *CJKV Information Processing*. O'Reilly and Associates, Inc., 1999. (This book is an encyclopedic coverage of the field of Chinese, Japanese, Korean, and Vietnamese information processing. Included are writing systems, character encodings, input/output issues, fonts, text processing, numerous resources available over Internet and otherwise, etc.)

中村正三郎, アプリケーションの国際化とは何か, (What is Application internationalization?). *SuperASCII*, Volume 3, #11. 1992. (An overview of software internationalization. Covering issues such as depth of internationalization, system environment, how to go about internationalization, code sets, fonts, input, output, sorting, editing, data compatibility, etc.)

Price Waterhouse and Massachusetts Computer Software Council, Inc. *1992 Software Industry Business Practices Survey*. 1992. (A survey of software producing companies. Contains a variety of information regarding nearly all aspects of the business. Relevant issues include figures on localized products, overseas revenue, international business practices, etc.)

Taylor, Dave. *Global Software: Developing Applications for the International Market*. Springer-Verlag. 1992. (Good introduction to the many facets of internationalization, not just the technical ones. Taylor's presentation of various marketing issues with numerous anecdotes makes for interesting reading.)

Tuthill, Bill. *Creating Worldwide Software: Solaris International Developer's Guide*. Prentice Hall. 1997. (A technical guide to NLS development on Sun platforms.)

Unicode Consortium, The. *The Unicode Standard: Worldwide Character Encoding, Version 3*. Addison-Wesley. 2000. (The definitive guide to the Unicode character set.)

Uren, Emmanuel. *Software Internationalization and Localization: An Introduction*. Van Nostrand Reinhold. 1993. (Solid introduction and reference to internationalization and localization. This book's strong point is the detail with which it treats many of the practical issues facing the software developer, including marketing in addition to technical concerns. It tends to focus on European localization for the PC and Mac arenas. Unix and Asian internationalization coverage are kept to a minimum.)

The Open Group. *Commands and Utilities, Issue 5*. The Open Group. 1997. (Standards which include the definition of the locale model. This volume defines commands one can expect from a compliant system. (including commands such as locale and localedef in the locale model))

The Open Group. *System Interfaces and Headers, Issue 5*. The Open Group. 1997. (Standards which include the definition of the locale model. This volume defines library routines and resources one can expect from a compliant system. (including such library routines such as setlocale(), and localeconv() in the locale model))

The Open Group. *System Interface Definitions, Issue 5*. The Open Group. 1997. (Standards which include the

definition of the locale model. This volume provides definitions that support the other guides. This also describes the foundation of the locale model in minute detail in chapters 4 and 5.)

The Open Group. *Distributed Internationalisation Services.* The Open Group. 1994. (A description of the limitations of the global locale based internationalization model and proposed solutions.)

The Open Group. *Internationalisation Guide Version 2.* The Open Group. 1993. (A very usable introduction and guide to internationalization using the XPG locale model.)