Internationalizing UNIX* Software Projects

Randall J. Scheer

AT&T's international products are providing interfaces using the customer's own language and cultural conventions. Providing internationalized systems that support the conventions of a country, its language, and culture can provide a competitive advantage. Identifying customer needs and developing the internationalization architecture at the beginning of a project cuts life-cycle costs associated with internationalization. This paper describes the problems and issues of cost-effectively producing internationalized software products that run under the UNIX* operating system, although this discussion is applicable to other operating systems.

Introduction

In international markets, customers want systems and documentation that use their own language and meet their own cultural conventions. Indeed, some countries *require* products to reflect their culture and language. Since most international software markets are growing faster than the U.S. market,[1] companies that intend to be competitive internationally must consider the cultural preferences of their customers.

This discussion is based on work since 1989 for the Multi-Function Operations System (MFOS), forward-looking work activities, and consultation on internationalization with other AT&T projects.

Most of the literature on the subject of internationalization deals with very small projects or single programs. This paper, however, views the internationalization process from a system-wide basis, focusing on the complete project life cycle, rather than just the development portion.

Internationalization Overview

Internationalization is the process of providing a computer system that handles a variety of language, country, and cultural conventions. A *locale* is an operating system database of language and country conventions. Developing software to support multiple locales is called *localization*. The combination of internationalization and localization is referred to as *globalization*.

Message catalogs are files used to store program input and output strings. All the program strings that a user can interact with should be contained in one or more message catalogs.

A *character set* is a collection that includes printable and non-printable letters, numbers, control characters, and symbols. *Coded-character sets* are used to represent character sets in a computer, such as *A* being encoded as a decimal 65 in the U.S. ASCII character set, and are classified in a locale. International standards now define the most widely used character sets. When a language has too many characters to fit into a single 8-bit byte, multiple bytes must be used, and some coded-character sets can require up to four bytes to represent a single character.

An internationalized system is one that can support different languages using per-language locales and per-language message catalogs. For further information, consult the references on software internationalization and localization,[1-5] individual program examples,[6,7] and converting legacy applications.[8]

Overview of Industry Standards

International customers expect that international standards will be followed, since they help define interfaces and consistencies across systems. Two such industry

Panel 1. Abbreviations, Acronyms, and Terms

8-bit flow through — capability to send characters with their highest order bit set throughout user input and output. Also called *8-bit clean.*

ASCII — American Standard Code for Information Interchange. In the U.S., ASCII usually refers to the 7-bit U.S. ASCII variant

character— a single item from a character set

character set — set of printable and non-printable symbols, numbers, ideographs, letters, control characters, etc.

coded-character set — character set mapping to an internal computer representation, possibly with display characteristics

input method — keystrokes required to enter a character

internationalization — process of providing a computer system that handles different language, country, and cultural conventions. Also called *I18N* for the 18 letters between the *i* and the *n*

JIS — Japanese Industrial Standard

locale — operating system database of language, country, and cultural conventions

localization — process of providing software to support multiple locales. Also called *L10N* for the 10 letters between the *l* and the *n*

message catalogs — binary or readable files containing program input and output strings accessed through standard subroutine calls

multiple-byte character set — coded-character set requiring more than one byte for at least some of the characters

single-byte character set — coded-character set that fits into one byte

Unicode — multilingual 16-bit coded-character set that defines character codes to support most languages. Unicode version 1.1 is a compliant subset of ISO/IEC 10646-1: 1993

wide character — a standard way of storing single or multiple-byte characters in the C language using a fixed *wchar_t* type

XPG4 — X/Open Portability Guide, Issue 4

standards are Unicode and the X/Open Common Application Environment (CAE).

Unicode, defined through the Unicode Consortium, is a multilingual 16-bit coded-character set that defines character codes to support most languages.[9,10] With Unicode, conversions between different coded-character sets are not required when changing between languages. However, switching between different message catalogs, locales, and font sets may be required.

The X/Open CAE provides, through the X/Open Portability Guide Issue 4 (XPG4), a common set of UNIX* system subroutine and program standards for providing internationalization support.[7] Using these interfaces can enhance software portability. XPG4 also includes standard interfaces for manipulating multiple-byte characters, referred to as *wide characters.*

Because different UNIX systems are at different stages of internationalization support, and since the standards are still evolving, it can be difficult to use the same software to provide internationalized systems across different operating systems.

Guidelines and Suggestions

When providing internationalized software, think globally from the start, considering how any issue will affect customers around the world. Develop a global mind-set and an international culture for your products.

The following sections discuss guidelines and suggestions that have evolved from our internationalization efforts for different portions of a project life cycle.

Product Feasibility and Commitment. Before internationalizing a system, first consider if it is worth the expense. Determine the market potential, market size, and customer need for the product. If a need is there, consider internationalizing gradually to gain experience. Assess what level of internationalization support is required, then estimate the expenses and feasibility. Include expenses for doing business in the markets, including duties, taxes, regulations, and support.

Determine whether the return of investment supports the cost of resources. If it does not support such costs, determine if there is a strategic need for internationalization, since internationalizing software

Panel 2. Potential Areas Of Internationalization Impact

This lists areas where internationalizing a system may impact a customer and require system changes to function in another language or coded character set.

- Output strings: sentence structure, phrase order, grammar, abbreviations, capitalization, plurals, hyphenation, string length, word delineation, line length, line breaking, help and error messages, punctuation, and white space.
- Input parameters: string length, database impacts, search criteria, accelerators, and delimiters.
- Date and time formats: time zones, daylight savings time, non-Gregorian calendars, abbreviations, capitalization, name and abbreviation lengths, 12- versus 24-hour time, and use of leading zeros.
- Numeric formats: separator, number of characters between separators, decimal point, negative numbers, percent indication, and rounding.
- Sorting and collation.
- Monetary formats and conventions.
- Character classification and conversion: upper/lower case conversion, and printable and non-printable characters.
- Presentation: terminal, window, form, and report layouts, tabular alignment, and form filling order.
- Orientation (direction for next character and next line): bidirectional (that is, direction depends on type of input) and unidirectional languages, and ligatures (characters that change fonts according to their position in words).
- Forms, menus, reports, electronic mail, and other tools.
- UNIX* system: shells, commands, file names, login sequence, regular expressions, metacharacters, spell checkers, editors, and encryption.
- Language and coded-character set.
- X Window System*: fonts, font sets, widgets, icons, menus, images, titles, resources, use of mouse, geometry, keyboard mapping, focus method, non-spacing marks, and context-sensitive rendering.
- Input method.
- Color and meanings associated with colors.
- Hardware: printers, keyboards, keyboard layouts, terminals, workstations, personal computers, input devices, screen resolution, ergonomics, power, local standards, labels, media, and patents.
- Audible devices: audible feedback, voice recognition, speech synthesizers, varying tone and pitch, voice gender, music, and meanings associated with sounds.
- Multimedia.
- Country/customer specific items: addresses, titles, telephone numbers, systems of measurement, temperature indications, page length and width, ordinal numbers, list separators, lucky and unlucky numbers, holidays, schedules, hours of operation, and business practice and etiquette.
- User and training documentation: promotional materials, illustrations, hypertext, copyrights, and trademarks.
- Interfaces to other systems.
- Product packaging and warranties.
- Vendor software packages: database management systems, spread sheets, word processors, operating systems, terminal emulators, and others.

can allow you to move more quickly into new markets.

Identify Country/Customer Needs. Many existing products, originally intended only for the U.S. market, are slowly moving into international markets. For any new project, planning for future internationalization should be included in the early stages of product development. As an example, if a product currently supports only single-byte European languages, converting applications to multiple-byte support, such as for Asian markets, can be complex and time consuming. However, properly planned, the added cost of providing this support for a future release may only be the additional time required to design the architecture and the additional training to use the XPG4 multiple-byte subroutines.[7]

Panel 2 presents a list of areas that may require system changes when providing an internationalized product. We have used this list to better understand what our customers may expect and, in addition, what system components may be impacted.

The way users in distinct countries and cultures perform and perceive system tasks may be different. *Operational scenarios*—end-to-end descriptions of user, system, and environment behaviors in particular situations—can provide a common understanding of a product and its features between you and your customer. We have used local human factors experts to help us create these operational scenarios. If such experts are not available, familiarize yourself with the customer's culture. Review the operational scenarios with the customer and use them as the basis for system requirements.

Consider performing usability testing with the product, since users in distinct markets may have different skill levels. Discover how customers perceive the usability of the translated user interface and documentation. Test the user interface with real users to determine if the look and feel meets their expectations. Can, for example, characters be freely entered and are they easily recognizable? Study whether elements of the look and feel, like input method and color, need to be customizable on a per-user basis. Involve user interface designers in this usability testing.

Entering Asian characters requires a graphical input method that can take significantly longer to perform. Using choice lists and other methods for non-keyboard input can reduce these problems.

There may be some users who do *not* require an internationalized product. Many times, system administrators can be trained in English to work with an English user interface. While this may significantly reduce the total effort, it may lead to decreased customer satisfaction, since users can better use and understand a system in their own language. Supplying the system and documentation in a user's *second* language, instead of the native language, could result in an underutilized system.

Always specify the language and coded-character set for each external interface used by the system, since different character set encodings, even those with the same language and character set, could prevent communication.

Identify Specific Customer Needs. Once internationalization requirements and needs have been identified for a product, per-customer specifications for locale and other variable information should be identified. This locale information will include, for example, date and time format preferences, numeric format preferences, and coded-character set.

Make sure the chosen coded-character set has all the characters needed by the customer and can be implemented by the system. Also, determine if the customer prefers certain display fonts, printer fonts, or keyboards.

Always get firm agreements with the customer on what documentation will and will not be translated, and which language and coded-character set will be used. As an example, who will translate the customer change requests? Promotional material may need to be translated before you even see the customer. Also, determine which languages will be used for product training.

Develop Project-Wide Goals. Each project should have a set of internationalization goals and expectations. The following list gives an admittedly lofty set of project internationalization goals. While not all these goals may be realizable for all projects, striving toward each will enhance product reusability:

- Develop multilingual products instead of monolingual or bilingual products.
- Allow switching between different locales and languages.
- Provide software that meets international standards.
- Provide locale-independent software that adjusts to different customers and markets.
- Provide coded-character set and font-independent software.
- Isolate differences between distinct markets and customers.
- Reuse translations whenever possible.
- Minimize expenses and developer intervention when adding new but "similar" languages or locales, but always retest each new language and locale.

An important user interface objective is to provide the look and feel of a local product, even though the product will be created outside the customer's country.

Software Architecture, Design, Implementation. Developing the internationalized architecture requires careful planning and design. First, identify the system internationalization capabilities needed through the project life span, since internationalizing non-internationalized projects can be expensive. Localizing legacy software usually involves re-architecting and re-engineering.

Architectures. There are two main architectures for providing internationalized software: *embedded* and *front-end.* The embedded approach—the standard method that we have seen used in most projects—requires strings to be accessed from message catalogs, and other internationalization functionality to be based from locales. It also requires each program to account for internationalization.

The front-end approach puts a *software user interface layer* above the English-only system software to provide internationalization. This approach becomes impractical with any large system and in-language data cannot be processed by the system. This method works best when the messages sent to and from the front end to the system are table indexes, instead of English.

These architectures may be intermixed. As an example, instead of passing strings between processes, pass message catalog indexes. Delay the conversion of an item into a string in a particular language until it is needed by the user interface. This can be helped by isolating the user interface from the main body of software.

Choose operating systems that support the standard XPG4 internationalization interfaces.[7] If your operating systems don't support these standard interfaces, provide your own interfaces that simulate the standards.

Design and Implementation. Using wide characters for multiple-byte characters will enhance your long-term software portability. However, in the near term, most operating systems haven't provided full wide character support. Wide characters require a completely different set of subroutines and string literals and may require developer retraining. They also need more program memory and require additional system performance.

Understand the internationalization capabilities of your operating systems. Be careful about trying to localize beyond the operating system's capabilities. If the operating system doesn't support multiple-byte operations, a system built on that operating system will have difficulty supporting multiple-byte operations.

Only a limited number of culture-specific items are stored in a locale. Items that are not in a locale, such as telephone number format (see Panel 2 for others), should be assessed from common message catalogs or other tables.

Do not put all program strings into message catalogs; use only those with which the user will interact. Many program strings, such as those used with debugging statements and those required for UNIX system subroutine calls, will not be seen by users. An analysis of the MFOS project showed that over 50 percent of the program strings were not visible to users. Including those additional strings into message catalogs would have increased translation and maintenance expenses and decreased system performance.

Be aware of internationalization performance issues. As an example, it takes longer to retrieve a string from a message catalog than it does out of internal memory. Removing message catalogs and using default message catalog strings can improve the performance of an English-only system. (But be careful that default message catalog strings and English message catalogs don't slowly diverge.) Measure the difference between using English message catalogs and using default message catalog strings to judge message catalog overhead.

Identify and create a common internationalization infrastructure, as we did for the BaseWorX™ platform, then build the rest of the system onto it. Consider creating a core world-wide feature set and putting per-customer features, locales, and message catalogs into separable feature packages. Train developers in the XPG4 and X Window System* standard internationalization subroutines and their use. Internationalizing a program usually means choosing one subroutine that supports internationalization over one that doesn't. Thus, much of software internationalization equates to training personnel and providing adequate quality assurance.

Integrate internationalization into your processes and methodologies. Develop internal project standards for internationalization. Include internationalization as part of your code inspection process.

Other Suggestions. Here are other suggestions that we have found will improve software architecture and design:

- Assign an architect to be responsible for the overall system internationalization infrastructure.
- Treat English as just another language.
- Build a software layer to hide the internationalization complexities from developers.
- Use one program source for all languages to reduce the costs associated with maintenance and documentation.
- Structure the system so that families of similar languages can be added without changing program source, even if only one language is initially needed.
- Be careful of byte orders when transferring interna-

tionalized data between systems.
- Understand the interoperability between different coded-character sets and languages.
- Consider if users should be able to switch between languages, intermix languages, or set and change their default language.
- Specify system accelerators and special command syntax characters in message catalogs. For example, many commands use the "\" character to escape the next character, but the "\" character is not defined in each coded-character set.
- Be aware of the differences between the number of bytes in a string, the number of characters in a string, and the number of display columns required. With multiple-byte characters, know where the character boundaries are located.
- Plan for the extra disk space needed for message catalogs in different languages. To save disk space, ship only the languages purchased by a customer.
- Reduce the number of conversions required between coded-character sets. Be aware that coded-character set conversions may result in data loss if there is not a one-to-one mapping between the coded-character sets.
- If you have users using different coded-character sets, adding local identifiers to file or database data can help distinguish which coded-character set is used.
- When possible, store database items in a language-independent manner. As an example, store date and time information in numeric format instead of character format. Storing this information in character format will require a conversion to display it in other locales.
- U.S. ASCII characters still may be required for certain operations, such as transferring data between systems and naming transferred files. As an example, mail may need to be transformed into U.S. ASCII, sent, and then transformed back into the original coded-character set. When required, do these transformations in a consistent way.
- Provide both readable and binary versions of message catalogs in the field, so minor adjustments can be made.
- Standards for bidirectional output support, needed for Middle Eastern markets, currently don't exist.
- When internationalizing across non-UNIX system platforms, bear in mind that standards and implementations are different with other types of operating systems.

Vendor Management. Internationalization support is a new concept for most software companies and many vendor software packages do not yet provide it. Since operating system internationalization support is also new, many software vendors have had to create their own, usually different, methods for providing this support. For each vendor software package that you use, or would like to use, evaluate whether it can fit into your software internationalization architecture. Our product team has included the following questions in its software vendor questionnaire:
- Which languages, locales, and coded-character sets are supported? Can new ones be added?
- Are there portions that do not provide internationalization support?
- Is only one language supported at a time, or can users switch between languages? If the latter, how does the package determine which language to use?
- Does the package provide flow-through and storage for 8-bit or multiple-byte coded-character sets?
- If this package is for customer use, does the supplier provide documentation in the customer's language? If not, can the required sections be translated? A word of warning: Do not translate someone else's documentation without a right-to-use agreement.
- Is training available in the customer's language?
- What is the delay from when the package is available in the vendor's local country to when it is available in other languages?
- How should vendor copyrights and trademarks be translated?

Note that vendor packages written for other countries may not have documentation available in English and may not be internationalized. In one project, a purchased software package from a Japanese firm had English documentation, but came with complex installation instructions in Japanese.

Finding vendor software packages that support multiple-byte characters is still difficult. If multiple-byte characters are required, choose your vendor software packages early, because they may require the use of a specific operating system or software architecture.

Quality Assurance. Always verify your system using native locales and message catalogs before shipping to a customer. Whenever possible, verify the system with individuals that have the same cultural background as your customers, which may require hiring people who can speak the customer's language. Consider outsourcing quality assurance along with translations, a service now being offered by some translations organizations. Test to verify that users will receive correct output in their language and not unintentionally see output in other languages.

Providing quality assurance in one language might find problems in other language versions, resulting in higher quality for each language. Our experience has been that we usually find some problems with the English version when we tested in other languages. The more you test, the more problems you will find for correction. Retest in each language when maintenance changes are made.

Verifying software to be 8-bit or multiple-byte clean is hard to automate and verify. It requires visiting every prompt and trying the new characters.

Message catalogs can be tested before they are translated. As an example, when using European character sets, we have taken English message catalogs and replaced unaccented characters with accented characters (as an example, replace *e* with an *é*). String lengths also may be expanded. The message catalogs will be infused with 8-bit characters and still read in English. Another example is to use the Roman characters in the Japanese JIS X 0208-1990 character set.

Allowing customers to change your supplied locales and message catalogs in the field isn't recommended, unless sufficient system protection and testing is performed. If changes are allowed, write and verify a front end that updates locale or message catalog databases and guarantees consistent results. As an example, don't allow message catalog strings to be longer than allocated memory space, since programs will fail. Also, use default locale information in help and user documentation, but describe how the system can diverge from what is documented.

Instead of regenerating automated test scripts, port them to other languages. Building automated test scripts from message catalogs allows one script to be used for different locales.

Deployment. The same issues apply for deploying an internationalized system outside the U.S. as with deploying an English-only system outside the U.S. Installers should understand the system in the customer's language. To assist installation and to allow developer access at a customer site, provide an English user interface option that can be used in the field.

After delivery, hold a post-mortem to identify the strengths and weaknesses of the system. Determine what capabilities and features the customers are and are not using. Use this information to develop capabilities for the next release.

Software Maintenance. When you change a program's default message catalog strings, you must also change the corresponding English message catalog strings, retranslate the corresponding non-English message catalog strings, and update where those strings appear in documentation. A tool that checks default message catalog strings against English message catalogs is useful as a quality assurance check.

On a project-wide basis, check to see which message catalogs have changed between releases and, thus, need to be retranslated. Mark all changes for translators, since they only need to translate strings that have changed.

Translation Issues

Everyone who has dealt with translating user documentation or system phrases has examples where cultural differences have changed the meaning of a phrase. In a recent MFOS translation, the phrase "Command line argument," which referred to different options on a command line, translated to "Line of command personal disagreement." One user thought this was only for military use. It wasn't. In another example, users were instructed after a processor failure to "Please shoe the computer," when, of course, they were supposed to "boot" the computer.

Translation should meet the quality expected with the rest of the system. To the customer, the quality of the translation is a big component of product quality. The utility of a system depends on its terminology and language being easily understood by its users. Care should be taken in choosing a translation organization, instead of just throwing the translation "over a wall" and hoping for the best.

The translation organization should be chosen early in the project life cycle, even before the requirements are completed. Many translations organizations also are familiar with internationalizing software, so they may be of some help.

When choosing a translation organization, investigate its experience, stability, linguistic capabilities, and use of tools, including machine translation. Most translation organizations now use some form of machine translation as a starting point.

Make sure the translator has an appropriate background and is competent in the language and culture. When possible, choose translators who are familiar with computer science terminology and technical vocabularies. Always review the translation results, preferably with someone knowledgeable with the target markets. Translation results should appear to users as if they were written in the user's language, instead of being translated, so the translated document may need to be edited to reflect the local document style.

Always supply a glossary of terms and abbreviations for the translators. Determine which acronyms should be maintained and which acronyms should be regenerated from translations, but be careful, as some acronyms may have negative meanings in other languages. Get agreement on the glossary translation before other translations begin.

When translating for more than one product, build cross-product dictionaries to add consistency. Use uniform terminology throughout the projects, and standard industry terminology[11] whenever appropriate.

Verify your ownership and copyrights for translated material. In some countries, the translation organization may also have rights to the material.

Plan the translation reuse from the start. To contain costs, only translate the parts of the product that have changed from one release to another. To do this, use some method, such as a *diff mark* capability, by which the translator can find and outline the changes between the different versions.

System Translation. System translation is the part of the system that requires translation when moving from one language to another. UNIX system translations are typically stored in message catalogs and distributed as part of the product. System translations are harder to produce, since the individual program strings do not give as much contextual information as a documentation paragraph.

Prepare message catalog comments for translators when you are creating or editing software, instead of later in the project life cycle—or not at all.

Develop tools to audit message catalog file formats and all other files to be translated, especially if you have any string length restrictions, and give these tools to your translation vendor. Develop an audit, as we did with our updates, to detect when message catalogs have changed after they have been sent to be translated. All changed portions will need to be retranslated.

Since system translations are part of the product, they must be available well before a product release, so the translation can be verified. Projects consistently overlook the fact that system translations are needed substantially before user documentation translations. Stabilize the system text as early as possible, so it can be translated earlier. To speed up this process, translate and generate source in parallel; some items can be translated while others are still waiting to be completed.

Document Translation. Translated user and training documentation should represent and describe the system. For systems that remain in English, consider keeping the system output in English, while translating the explanations of the system output. To do this, the documented system output must be easily distinguishable to translators.

Many times, documentation and system translation will use different coded-character sets, depending on the system capabilities and the typesetting fonts. Thus, the same terminology and coded-character sets should be used in the documented system output, as with the actual system. When system output is shown in the documentation, the document should have the exact look of the output.

Since system and documentation translation are performed at different times, retranslating any system translation located in the documentation will invariably differ with the original system translation. To solve this, build the documentation from the same translated message catalogs used in the system.

Anything you can do to make your documentation as clear and simple as possible will promote easier and clearer translations. When the source document is clear and concise, the translation process takes less time and results in a better final product. Restructure the documentation to avoid redundancy, and try to translate items only once.

Other Suggestions. We have found that the following suggestions can help improve translation results and reduce expenses:

- Understand the differences between U.S., British, and global English.
- Spend as much time to verify translated documentation as you would the English version.
- Make sure the translation organization understands the document formatter and the documentation standards and conventions.
- To reduce interruptions, we have a translation coordinator as a single point of contact for translation questions.
- Be aware of different dialects in the same language.
- Exercise care when using sorted lists in documentation. These need to be adjusted for each language. Use numeric indexes instead of sorted alphabetic indexes.
- Allow room in tables and figures for text expansion, and keep illustrations simple.
- Using one vendor for all your language translations is usually cheaper than using separate translation vendors for each language.
- Verify an early translation sample.
- As a quality check, some projects have taken their translation results, retranslated them back to the original language, and compared the results with the original version. However, this only works if you don't make style changes owing to cultural differences.
- Where we seldom get into disagreements on the choice of words—even made-up words—the choice of non-English words usually generates customer discussions.

Training. The same issues for user documentation also apply to training documentation. On MFOS, we build our training documentation from our user documentation, which eliminates the need to retranslate areas common to both, thereby reducing expenses.

Customers may require training in other languages, which will require trainers who understand the system and can speak the customer's language. Training video tapes can be refilmed in-language or remixed with translated voice-overs. If English training must still be given, remember to speak slowly and allocate additional training time. Users who are familiar with the system in English may need to be retrained to use the system in other languages.

Conclusion

Projects moving into the global marketplace will encounter an increasing number of customers requiring systems using their language and cultural conventions. Producing multilingual products that can readily add new languages and adjust to different cultural conventions will allow systems to move into new markets in a shorter time.

This paper has presented many of the project life-cycle activities needed when supplying an internationalized system. It has discussed guidelines and suggestions for reducing internationalization expenses by providing:
- Early knowledge of marketplace needs.
- Up-front planning, architecture, and decision making.
- Translation preparation and reuse.
- Suggestions for reducing the costs and complexities.

We will continue to see the evolution and implementation of industry standards and the increase of tools to reduce the internationalization complexity.

(Manuscript approved February, 1995)

*** Trademarks**

UNIX is a registered trademark of Novell in the United States and other countries, licensed exclusively through X/Open Company Limited.

X Window System is a trademark of The Massachusetts Institute of Technology.

References

1. Dave Taylor, *Global Software: Designing Applications For The International Market*, Springer-Verlag Inc., New York, 1992.
2. Ken Lunde, *Understanding Japanese Information Processing*, O'Reilly & Associates, Sebastopol, California, 1994.
3. Jakob Nielsen, *Designing User Interfaces For International Use*, Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.
4. Emmanuel Uren, Robert Howard, and Tiziana Perinotti, *Software Internationalization and Localization: An Introduction*, Van Nostrand Reinhold, New York, 1993.
5. *Guide To Macintosh Software Localization*, Apple Computer, Inc., Addison-Wesley Publishing Company, New York, 1992.
6. Sandra Martin, "Internationalization Explored," UniForum, 1992.
7. *Internationalisation Guide*, X/Open Company Ltd., U.K., February 1992.
8. Rob Pike and Ken Thompson, "Hello World," *Proceedings of the Winter 1993 USENIX Conference*, USENIX Association, 1993.
9. *Unicode Consortium, The Unicode Standard - Worldwide Character*

Encoding, Version 1.0, Volume 1, Addison-Wesley Publishing Company, New York, 1991.
10. *Unicode Consortium, The Unicode Standard - Worldwide Character Encoding*, Version 1.0, Volume 2, Addison-Wesley Publishing Company, New York, 1992.
11. *Telecommunications Glossary*, Select Code 350-207, AT&T, 1987.

Randall J. Scheer *is a member of technical staff in the Global Network Surveillance Architecture and Development Department at AT&T Bell Laboratories in Columbus, Ohio. He is responsible for architecture and software engineering for the Global Network Surveillance Product Team. He has a B.A. degree in computer and information sciences and mathematics from the State University College at Potsdam, New York, and an M.S. degree in computer and information sciences from Ohio State University in Columbus. He joined AT&T in 1981.*