Digital Technical Journal

Digital Equipment Corporation

Volume 5 Number 3
Summer 1993

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460. Subscriptions to the *Journal* are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to the *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically to DTJ@CRL.DEC.COM. Single copies and back issues are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent back issues of the *Journal* are also available on the Internet at gatekeeper.dec.com in the directory /pub/DEC/DECinfo/DTJ.

Digital employees may send subscription orders on the ENET to RDVAX::JOURNAL. Orders should include badge number, site location code, and address.

Comments on the content of any paper are welcomed and may be sent to the managing editor at the published-by or network address.

ISSN 0898-901X

Documentation Number EY-P986E-DP

Book production was done by Quantic Communications, Inc.

Cover Design

Scripts, symbols, and writing directions are elements of written communication that are addressed by product internationalization, the featured topic in this issue. Like engineering designs and standards for internationalization, the graphic design on the cover provides a framework that accommodates a rich diversity of the world's written languages.

The cover was designed by Joe Pozerycki, Jr., of Digital's Corporate Design Group.

Contents

Editor's Introduction

Jane C. Blake
Managing Editor

Engineering products for international markets is a multifaceted undertaking, as it entails the adaptation of computer technology to the unique and varied ways cultures communicate in written languages. Papers in this issue describe some of the cultural and technological challenges to software engineers and their responses. Topics include conventions of culture and language, internationalization standards, and design of products for local markets.

Product internationalization begins with identifying the cultural elements and user expectations that the software must accommodate. Tim Greenwood has written a tutorial that provides insight into the cultural differences and the complexities of written languages as they relate to product development. Among the topics he discusses are scripts and orthography, writing directions, keyboard input methods, conventions for values such as time, and user interfaces.

As a counterpoint to the complexity of languages and cultures, industry engineers and organizations have developed standards that lend simplicity and uniformity. Unicode, described here by Jürgen Bettels and Avery Bishop, is a significant internationalization standard that accommodates many more complex character sets than does 8-bit ASCII; software produced using Unicode character encoding can be localized for any language. The authors discuss the principles behind the 16-bit encoding scheme and considerations for application processing of Unicode text. They conclude with approaches for the support of Unicode and reference the Microsoft Windows NT implementation.

Wendy Rannenberg and Jürgen Bettels have written a paper on another important standard, the X/Open internationalization model. X/Open supports multibyte code sets and provides a comprehensive set of application interfaces. The authors examine benefits and limitations of the standard, referencing Digital's DEC OSF/1 AXP implementation, and close with proposed changes to the model.

René Haentjens' paper is not about a standard per se but about the ways in which various cultures order words and names and the methods used in computers to emulate this ordering. He examines the table-driven multilevel method for ordering universal character strings, its variations and its drawbacks. The implications of Unicode relative to ordering are also considered.

The development and adaptation of software for use in local markets is the common theme of three papers. Gayn Winters identifies several programming practices for the development of distributed systems and discusses the benefits of modularity in systems and in run-time libraries to reduce reengineering effort and costs. However, as Michael Yau notes in his paper, reengineering is necessary for systems designed when English was the only language supported in computer systems. Michael presents an overview of the engineering challenges encountered and resolved in the creation of local variants of the OpenVMS operating system to support the Japanese, Chinese, and Korean languages. A third paper, written by Hiro Yoshioka and Jim Melton, provides a case study of a coengineering project, i.e., a project in which engineers from the local environment (or market) join in the product development. The case references the internationalization of the DEC Rdb database (specifically for Asian markets) utilizing an SQL standard.

The concluding paper focuses on software designed to facilitate Japanese keyboard input and to reduce reengineering/localization effort. Takahide Honma, Hiroyoshi Baba, and Kuniaki Takizawa review the methods of Japanese keyboard input and then describe a three-layer, application-independent software implementation that is embedded in the operating system and offers users flexibility in the choice of an input operation.

The editors are grateful to Tim Greenwood, an architect of Unicode currently working in the Software Development Tools Group, for his help in coordinating the development of papers and to Gayn Winters, Corporate Consulting Engineer.

Note to Internet Users: Recent back issues of the DTJ are now available in ASCII and PostScript formats on gatekeeper.dec.com in the /pub/DEC/DECinfo/DTJ directory.

Jane Blake

Biographies

Hiroyoshi Baba Hiroyoshi Baba is an engineer in the Japanese Input Method Group in Digital Japan, Research and Development Center. He is currently developing the Japanese front-end input system on OpenVMS VAX and OpenVMS AXP and the Japanese language conversion server system. He received a B.S. (1989) and an M.S. (1991) in electronics engineering from Muroran Institute of Technology, Japan. He joined Digital in April 1991.

Jürgen Bettels Jürgen Bettels is an internationalization architect and the standards manager for the International Systems Engineering Group. Since 1986, he has worked on many internationalization architectures starting with DECwindows. He participated in the Unicode consortium, ECMA, and X/Open on internationalization. He contributed to the ISO/IEC WG2/SC2, whose work merged Unicode and ISO 10646 into a single universal character encoding. Prior to joining Digital, he was a physicist at the European particle laboratory, CERN. Jürgen has the degree of Diplom Physiker (physicist) from the University of Aachen.

F. Avery Bishop Avery Bishop is the program manager for Windows NT/Alpha internationalization. Prior to this position, he worked in ISE as Digital's representative to the Unicode consortium and the ANSI X3L2 technical advisory group on character encoding. He worked with ISO/IEC WG2/SC2, Unicode, and others in Digital to merge Unicode and ISO 10646 into a single universal character encoding. Prior to that, he managed projects at DECwest and worked as the product management manager for ISE in Japan. Avery has a Ph.D. in electrical engineering from the University of Utah.

Timothy G. Greenwood Since 1981, Tim Greenwood has held various positions related to internationalization at Digital. He was the architect for the Japanese and Chinese versions of DECwindows. This software introduced the compound string technology that was incorporated into Motif. Tim conceived of, managed, and wrote much of the software section of the internal version of the handbook on Producing International Products. He also participated in the design of international support on the X Window System. Tim is currently responsible for guiding the introduction of Unicode into Digital.

René Haentjens René Haentjens is a software consultant working for both Digital Consulting Belgium and Corporate Standards and Consortia. He was the Belgian local engineering manager for two years. Today, René is a member of the Belgian, the European (CEN), and the ISO committees on character sets and internationalization. He contributed significantly to the ISO/IEC 10646-1:1993 standard. He has a civil engineering degree (chemistry) from the University of Ghent and has contributed to publications on compiler portability, on software engineering, and on developing international software and user information.

Takahide Honma A senior software engineer, Takahide Honma leads the Japanese Input Method Group. He joined Digital in 1985 as a software service engineer. He has worked on systems such as real-time drivers, network system (P.S.I.), and database on VMS and was a consultant to customers. At the same time, he also took the role of a sales advisory support engineer. Since 1990, he has been with Research and Development in Japan and has worked on the Japanese input method. He has an M.S. (1983) in high-energy physics from Kyoto University and is a member of the Physics Society of Japan.

Jim Melton A consulting engineer with Database Systems, Jim Melton represents Digital to the ANSI X3H2 Technical Committee for Database. He represents the United States to the ISO/IEC JTC1/SC21/WG3 Working Group for Database. He edited the SQL-92 standard and continues to edit the emerging SQL3 standard. Jim also represents Digital to the X/Open Data Management Working Group and to the SQL Access Group. Jim is the author of *Understanding the New SQL: A Complete Guide,* published in 1992, and is a regular columnist (SQL Update) for *Database Programming & Design.*

Wendy Rannenberg Principal software engineer Wendy Rannenberg manages the UNIX Software Group's internationalization team. She is responsible for the delivery of Digital's internationalization technology on both the ULTRIX and the DEC OSF/1 AXP platforms. Prior to joining Digital in 1988, she held engineering positions with Lockheed Sanders Associates and the Naval Underwater Systems Center. Wendy holds a B.S. (1980) in engineering from the University of Connecticut at Storrs and is a member of IEEE, SWE, and ACM. She has written or contributed to numerous technical publications.

Kuniaki Takizawa Kuniaki Takizawa is an engineer with Digital Japan, Research and Development Center and is a member of the Japanese Input Method Group. He joined Digital in April 1991 and is currently developing and porting the henkan module and the input method library (IMLIB) on OpenVMS, ULTRIX, and OSF/1. He graduated from the University of Electronic Communications (Denki-Tsushin University) in Japan in 1991. His speciality area was the structure of operating systems.

Gayn B. Winters Corporate consulting engineer Gayn Winters has 25 years' experience developing compilers, operating systems, distributed systems, and PC software and hardware. He joined Digital in 1984 and managed the DECmate, Rainbow, VAXmate, and PC integration architecture. He was appointed Technical Director for Software in 1989 and contributes to the Corporate software strategy. From 1990 to 1992, Gayn led the internationalization systems architecture effort and is on the Board of Directors for Unicode, Inc. He has a B.S. from the University of California at Berkeley and a Ph.D. from MIT.

Michael M. T. Yau Michael Yau is a principal software engineer in the International Systems Engineering Group. Since 1984, he has worked on Asian language support in the OpenVMS operating system. He led and managed the development team in Hong Kong from 1986 to 1991. Currently, he provides architecture and product internationalization support to U.S. engineering groups. Prior to joining Digital, Michael worked for GEC Marconi Avionics (U.K.). Michael holds a B.Sc. (Hons) in mathematics and an M.Sc. in communication engineering from the Imperial College of Science and Technology, University of London.

Hirotaka Yoshioka A senior software engineer in the International Software Engineering Group, Hiro Yoshioka is the project leader of the CDD/Repository/Japanese. He is a member of the internationalization special committee of ITSCJ (Information Technology Standards Commission of Japan) and ISO/IEC JTC1 SC22/WG20 internationalization. During the past nine years, he has designed and implemented the Japanese COBOL, the Japanese COBOL generator, and the internationalized DEC Rdb. Hiro joined Digital in 1984, after receiving an M.S. in engineering from Keio University, Yokohama.

Foreword

Claude Henri Pesquet
Engineering Group Manager, International Systems Engineering

In the late 1970s, Digital began to ship its first office products outside the U.S. We realized then that it was no longer an option to provide users with the ability to input, view, edit, and print foreign language text; it was instead a necessity, as well as a passport for Digital into world markets.

The foreign-language requirement came as a shock to the application developers who had been trained in the late 1960s, at a time when the U.S. English-speaking market represented more than 70 percent of the total worldwide information technology market. Today's reality is quite different. The English-speaking IT market is below 40 percent, and trends indicate that it will continue to decline. This is not surprising, because only 8.41 percent of the world's population is native English speaking. Moreover, each year the commoditization of computers lowers the entry point for the acquisition of computer products; consequently the market is expanding to encompass a much broader socioeconomic community. Further, starting in the 1980s, the creation of global markets—for labor, materials, intellectual talent, financing, and distribution channels—has forced businesses to continually reach outside their domestic markets. Government mandates also have an impact, requiring that products sold within country boundaries have local-language capability. Together these factors will increase the demand for and requirements of international products—products that will provide users with linguistic choices.

In recent years, Digital has broadened its market focus to include not only the scientific/technical, mainly English-speaking markets, but also commercial markets—a large market comprising many languages. To serve these markets well, we are compelled to adopt a strategy for the internationalization* of our products.

The strategy, i.e., to develop products that "speak" the local language, has evolved from a fastidious reengineering of a product after the fact to an architectural definition that ensures products are designed originally to meet local-language requirements. Digital had three goals:

- Reduce development costs.
- Shorten the time to market.
- Increase product quality.

The cost of reengineering products that were designed based upon a North American paradigm is similar to the cost of maintaining an application that was designed without regard to future maintenance. Such costs could meet, if not exceed, the original product development cost. This was discouraging, because the markets outside the U.S. were smaller and emerging; producing the local product compared in cost to producing the original U.S. product. It became obvious that it was too expensive to continually rebuild products that sold only to a small market.

Local-language products were late to market when compared with availability of the same products in the U.S. This presented a twofold problem. It denied our multinational customers the capability of installing products and applications simultaneously in their worldwide operations. Further, product launches, training, selling, support, and retirement had to be addressed one country at a time because specific local-language components were not available simultaneously.

In addressing short-term "surface" issues, we had utilized the brute force of reengineering to produce one language version at a time. As a result, we delayed addressing the "deep" quality issue of originally designing and building into our products the internationalization features that would allow for easy adaptation to any language without modification of a product's core.

A vision on how to address the internationalization of products was developed by a worldwide team of architects led by Gayn Winters. The majority of this team was located outside the U.S. and had

*The term internationalization as it is used within the context of this *Journal* includes both the technologies and the processes applied to enable a product to meet the need of any local-language market without requiring modification of the base functionality of the product.

been closely involved in Digital's reengineering efforts for many years. The team's prime motivation was to eliminate the need for reengineering. The vision they developed is one in which all Digital integrated systems can process electronic information containing multiple languages and character sets, and can satisfy end-user linguistic preferences. An inherent part of this vision is to make all systems available simultaneously worldwide.

One of the major difficulties in implementing the vision was that internationalization was not aimed at specific products, rather it was a pervasive attribute required across systems. For product development groups trained to develop components, this represented a difficult change in mindset. The implementation also required a huge paradigm shift—

From one character and...	To one character and...
One input method	Many input methods
One cell	Multiple cells
One collation point	Several collation points
One geometry	Many geometries
Alphabet	Ideograms
"Frozen" alphabet	User-defined characters

The paradigm shift led to a redefinition of the elements to be incorporated in the basic design of new products. The strategy from a product perspective was to start with the base system (CPU, peripherals, network, and operating system), and then move to the application side. From an engineering-resource perspective, we would start with parallel internationalization development, and then inject internationalization expertise into the original product development group. The strategy from a process perspective was to customize code for specific countries, and then roll back the country-specific code into the original product code base and continue future development from this unique code base. The implementation has resulted in major achievements, for example, the simultaneous shipment of products to which this approach was applied.

To illustrate our progress, the latest version of Rdb (relational database application) was developed with the injection of internationalization expertise. The approach resulted in one common code base and achieved worldwide simultaneous shipment.

Many challenges remain. Standards have to be defined and implemented in areas such as naming conventions, user profiles, and character attributes. Emerging technologies such as object-oriented software and multimedia need to be addressed. And real-time multilinguality (the simultaneous translation from one language to another) must be tackled.

This issue of the *Journal* provides a broad sampling of our product internationalization efforts—from the concept of cultural differences to the specific internationalization of our Rdb product. The papers herein represent only a few of the hundreds of projects dedicated to the internationalization of Digital's products.

Timothy G. Greenwood

International Cultural Differences in Software

Throughout the world, computer users approach a computer system with a specific set of cultural requirements. In all cultures, they expect computer systems to accommodate their needs. A major part of interaction with computers occurs through written language. Cultural requirements, particularly written languages, influence the way computer systems must operate. Cultural differences concerning national conventions for the presentation of date, time, and number and user interface design for the components of images, color, sound, and the overall layout of the screen also affect the development of computer technology. Successful computer systems must respond to the multicultural needs of users.

Not chaos-like, together crushed and bruised,
But, as the world harmoniously confused:
Where order in variety we see,
And where, though all things differ, all agree.
— Alexander Pope

In the first years of the computer age, users adapted themselves to the requirements of the computer. They had to learn the language of the machine to interact with it. Now the computer is part of daily life, a tool to complete a task. Computer systems must be adapted to the needs of their users. Computer users approach a computer system with a specific set of cultural requirements. Successful systems respond to these requirements.

International Adaptation of Computer Systems

Each nation has developed its own culture, and some areas of the world share a cultural background. Adaptation of computer systems to different cultures uses processes known as localization and internationalization.

Localization is the process of changing products to suit users from different cultural backgrounds. Localization is achieved by taking the source code for a product developed for one country and modifying the source code and product to satisfy the needs of other countries. Often teams of developers in different countries are needed to adapt products. If the original product is not built with a view toward being localized, this can be a very expensive and time-consuming process. There is the direct cost of multiple development teams modifying the source code of the original product. This process also produces multiple code bases, which makes future development and maintenance more complex.

Building software that can be localized with minimal software changes is called internationalization, often abbreviated to I18N (the letter I followed by 18 letters and the letter N). The basis of internationalization is to identify those cultural elements that the software must accommodate and to simplify the task of adapting the product. This paper describes a set of these cultural elements. The remainder of this issue of the *Digital Technical Journal* details specific aspects of building international software.

Cultural Differences

Language is the most obvious cultural difference among people. Written language is an important method of communication with computers. This paper examines written languages and their representation in computer systems. It also presents cultural differences concerning national conventions for the presentation of date, time, and number and user interface design for the components of images, color, sound, and the overall layout of the screen. The base functions of a product may change in response to different needs around the world, and some examples of these differences are illustrated. Finally, with a look to the future, the paper presents deeper cultural differences that are only beginning to be represented in software.

Written Language

The written representation of spoken language requires a script and an orthography. The script is the set of symbols that represents the sound or meaning of components of the language. The orthography consists of the rules of spelling and pronunciation. Specific spelling and pronunciation rules may differ among locations or communities; for example, the American English orthography differs from the British English orthography. A script may be tied to a specific language, for example, Korean Hangul, but frequently a script can represent several languages. French and Italian both use the Latin script.

A written language may be broadly categorized into either an ideographic, a syllabic, or an alphabetic writing system. The category is determined by examining the relation between the symbols in the script and the unit of sound or meaning represented.

In writing systems based on ideograms, every symbol has a specific meaning that is not related to its pronunciation. The ideograms imported from Chinese, and used in Chinese, Japanese, and South Korean writing provide examples in current use.[1] Thus 人 represents a man or person, even though it is pronounced *ren* in Chinese, *zin* in Japanese, and *in* in Korean.[2] "3" represents "three" even though it is pronounced *tatu* in Swahili and *trwa* in French. Ideographic writing systems typically contain several thousand discrete symbols with a subset of approximately 2,000 symbols in frequent use. The users of this writing system continue to develop new symbols.

In the syllabic writing systems, each symbol represents a syllable. マ in Japanese *katakana* denotes the *ma* sound. There is a wide variation in the number of discrete symbols in a syllabic system. Japanese *kana* uses some 47 symbols; the writing of the Yi people (a minority nationality scattered through provinces in Southwest China) uses a standardized syllabary of 819 symbols.[3]

In alphabetic systems, each symbol or letter approximately matches a phoneme (the smallest unit of speech distinguishing meaning). Thus *M* in Latin script, מ in Hebrew, and մ in Armenian denote the *m* sound. Most alphabets have from 30 to 50 discrete letters.[4] The match between phonemes and letters is not exact, especially in English, which has about 40 phonemes.[5] Some phonemes are represented by letter sequences, such as the *th* in *thank*.

No written language uses a pure set of either alphabetic, syllabic, or ideographic symbols; each does use one set of symbols predominantly. The Latin script is primarily alphabetic, but numerals and certain signs such as & are ideograms.[6] Other languages use a more even mix. South Korean combines the native Hangul alphabet with Hanja, the Korean name for their ideographic characters. Japanese combines the *katakana* and *hiragana* syllabaries (collectively called *kana*) with the ideographic characters called *kanji* in Japanese. Written Japanese, especially technical and advertising material, also often uses the Latin alphabet, called *romaji*.

Character Placement In most European languages, basic symbols are written in a linear stream with each character placed on a baseline. In other writing systems, for example, Korean Hangul, the elements do not follow this linear layout. Rather than evolving piecemeal like most writing systems, Hangul is the result of deliberate, linguistically informed planning. It has been called "...probably the most remarkable writing system ever invented."[7] Korean uses an alphabet of 14 consonants and 10 vowels. These letters, called *jamo,* are blocked into syllable clusters. If the same technique were applied to English, *cat* might be written $^{ca}_{t}$. Figure 1 shows the Korean Hangul alphabet, and Figure 2 shows the *jamo* blocked into syllable clusters.

Thai also uses an alphabet and is written with the symbols arranged in a nonlinear fashion. Thai is a tonal language; different tones distinguish words that would otherwise be homonyms. Thai words consist of consonants, vowels, and tone marks. Each component is an atomic unit of the language. A vowel is written in front of, above, below, or behind the consonant to which it refers. A tone mark, if present, is usually placed above the consonant or above the upper vowel. Thai potentially has symbols at four levels, as shown in Figure 3. Level

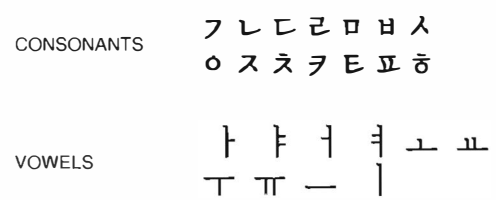


Figure 1 Korean Hangul Consonant and Vowel Signs

이 장치는 고주파를 생성, 사용 및 발산한다. 이 장치는 유형 검사를 거쳤으며, 고주파 방해를 가능한 한 방지하기 위해 제정된 FCC Rules, Part 15, Subpart J와 관련된 Class A 컴퓨팅 장치에 대한 제한 사항을 준수한다는 것이 입증되었다. 주거 지역에서 이 장치를 사용하면, 전파 방해가 발생할 수 있으며, 그러한 경우에 사용자는 전파 방해를 제거하는 데 필요한 조치를 취해야 한다.

Figure 2 Korean Hangul Text Showing Blocking of Jamo

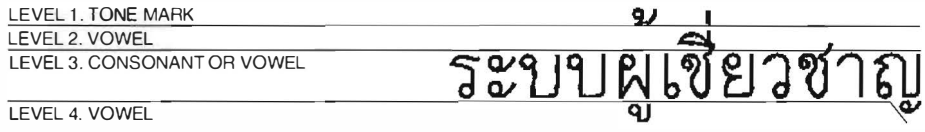


Figure 3 Thai Script

one is an optional tone mark. Level two is an optional vowel. Level three is either a consonant or a vowel preceding or following a consonant. Level four is another optional vowel. A consonant never has a vowel at both level two and level four.

Some level-three consonants have part of their glyph images rendered in another level. They either dip into level four or rise into level two. The last letter (*yo ying*) in Figure 3 is a level-three consonant, but it has a small (separate) portion written below the baseline. When this letter is written, this small portion is written at level four. When this letter is followed by a true level-four vowel, the vowel is shown instead of this portion.

In Figure 3, both tone marks are shown at level one to aid understanding of the script. In high-quality printing of Thai, if a cell does not have a level-two vowel, then the tone mark falls down to be positioned directly above the consonant; however, it is also currently acceptable for all tone marks to be physically positioned at level one. Thai mechanical typewriters position all tone marks at level one.

Diacritics and Vowels In Arabic and Hebrew alphabets, vowels are indicated by placing vowel points above, below, or beside the letter. (Arabic also uses the consonant letters *alif, ya,* and *waw* to represent the long vowels a, i, and u.) Vowels are normally used only in religious text and in teaching materials for people learning to read the language; in other texts, vowels are inferred by the reader. Since vowel points are used, written Hebrew with vowels is called pointed. Figure 4 shows pointed Hebrew from a children's comic and the same text with the vowels removed.

מְטַפֵּס הֶהָרִים הָאַמִּיץ תִּרגֵם: אָבִיב יוֹרֵק

לְפִינוֹקְיוֹ הִתְחַשֵּׁק לָצֵאת לְטִיוּל. וְלֹא סְתָם טִיוּל, כִּי אִם מַסָּע
אָרֹךְ שֶׁיִּגָּמֵר רַק עִם רֶדֶת הַחֲשֵׁכָה.
וְכָךְ הוּא יָצָא לַדֶּרֶךְ.

מטפס ההרים האמיץ תרגם: אביב יורק

לפינוקיו התחשק לצאת לטיול. ולא סתם טיול, כי אם מסע
ארך שיגמר רק עם רדת החשכה.
וכך הוא יצא לדרך

Figure 4 Pointed and Unpointed Hebrew

Other Symbols Most languages written with Latin letters have diacritical marks on some letters. In some cases, the use of a diacritic provides a stress or pronunciation guideline, as in the English word *coöperate*. Removing the diacritic does not change the meaning of the word. In other languages, a mark that appears to be a diacritic is a fundamental part of the letter. The Danish letter *å* is a separate letter in the alphabet and is not a variant of *a*. In German, three vowels have umlauts and are separate letters in the alphabet. The deletion of an umlaut can change the definition of a word; for example, *schwül* means hot or humid, and *schwul* means homosexual.

Presentation Variants The characters in the Arabic writing systems change form, depending on whether they are the first, last, or middle character of a word, or if they stand alone. Note that the abstract characters themselves do not change, only the glyph image. Figure 5, adapted from Nakanishi, shows the presentation variants of Arabic letters.[8]

Writing Direction

In English and many other writing systems, the letters are written from left to right, with lines progressing from top to bottom. Japanese, Chinese, and Korean may also be written in this form but are traditionally written vertically. The characters flow from the top of the page to the bottom, with lines advancing from right to left. The pages are ordered in the opposite direction to that used for English. Mongolian is also written vertically, but the columns of text advance from left to right. Consequently, pages of Mongolian text are ordered in the same direction as in English.

Figure 6 shows a portion of a newspaper printed in Taiwan. The newspaper exhibits many styles of format. Headlines may run horizontally from right to left, or left to right; the text of the article may run vertically; and advertisements and tables may run horizontally from left to right.

In Japanese writing, Latin characters (*romaji*) are interspersed with vertical *kanji* (Han script) characters. *Romaji* may be presented with each character in a horizontal orientation running vertically down, or they may be presented vertically, with each character rotated by 90 degrees. In addition, if one, two, or three Latin characters are mixed with vertical Han script characters, they may be presented horizontally in the vertical stream. Figure 7 shows mixed characters in a Japanese text.

INDEPENDENT FORM	INITIAL FORM	MEDIAL FORM	FINAL FORM	PHONETIC VALUE
ا			ـا	*a*
ب	بـ	ـبـ	ـب	*b*
ت	تـ	ـتـ	ـت	*t*
ث	ثـ	ـثـ	ـث	*t*
ج	جـ	ـجـ	ـج	*j*
ح	حـ	ـحـ	ـح	*x*
خ	خـ	ـخـ	ـخ	*kh*
د			ـد	*d*
ذ			ـذ	*dh*
ر			ـر	*r*
ز			ـز	*z*
س	سـ	ـسـ	ـس	*s*
ش	شـ	ـشـ	ـش	*sh*
ص	صـ	ـصـ	ـص	*s*
ض	ضـ	ـضـ	ـض	*d*
ط	طـ	ـطـ	ـط	*t*
ظ	ظـ	ـظـ	ـظ	*z*
ع	عـ	ـعـ	ـع	*[']*
غ	غـ	ـغـ	ـغ	*gh*
ف	فـ	ـفـ	ـف	*f*
ق	قـ	ـقـ	ـق	*q*
ك	كـ	ـكـ	ـك	*k*
ل	لـ	ـلـ	ـل	*l*
م	مـ	ـمـ	ـم	*m*
ن	نـ	ـنـ	ـن	*n*
ه	هـ	ـهـ	ـه	*h*
و	و		ـو	*w,u*
ى	يـ	ـيـ	ـي ـى	*y,i*

NUMERALS Arabic figures are written from right to left, but the figure written to the left shows the higher value: ٢٣ *23*

١	٢	٣	٤	٥	٦	٧	٨	٩	٠
1	2	3	4	5	6	7	8	9	0

Figure 5 Arabic Presentation Forms

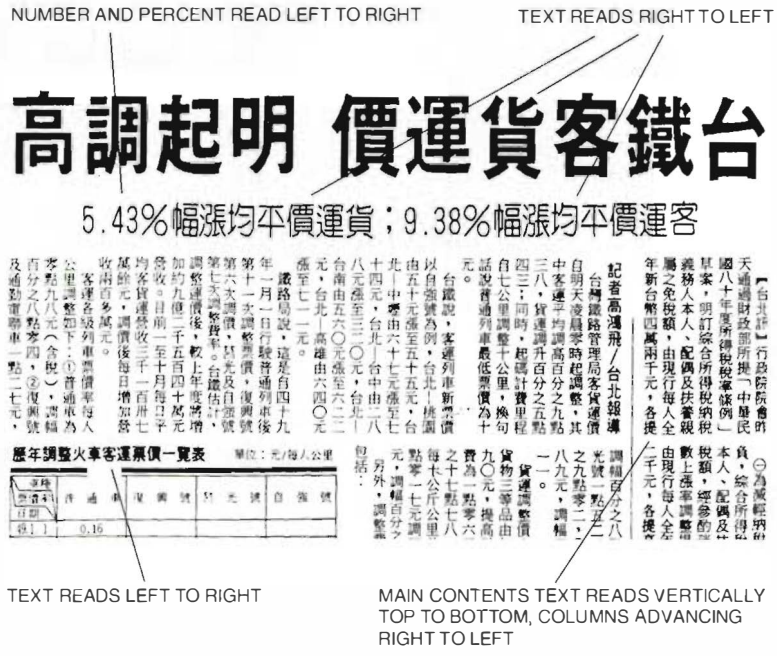


Figure 6 Taiwanese Newspaper

Figure 7 Japanese Text Showing Latin Characters Mixed with Kanji

Semitic language scripts (e.g., Hebrew and Arabic) are written horizontally from right to left, with lines advancing from top to bottom, but any numbers using Arabic numerals are written left to right.[9] Any fragments of text written in the Latin script are also presented left to right. This method leads to nesting segments of reversed writing direction as shown in Figure 8. The text in this figure reads "Attention: Kalanit (1984) Tel-Aviv, ISRAEL;" where "Kalanit (1984)" is a company name. Figure 9 is another example of combining left-to-right and right-to-left text. It shows a portion of the contents page from the EL AL airline magazine.

Text Input

The following section discusses techniques for addressing cultural differences in computer keyboard input.

Alphabetic writing systems typically have no more than 50 discrete symbols. Computer keyboards contain approximately 48 keys with symbols from the writing system inscribed. The depression of a key produces a code from the keyboard that is translated into a character coding according to some predefined coding. Input of a character not represented directly on the keyboard requires depression of several keys. For example, in terminals from Digital, the *á* character is input on non-German keyboards by pressing "Compose s s".

Latin keyboards typically have two possible characters available from each alphabetic key: lowercase letters are displayed by depressing the key alone, and uppercase letters are produced by depressing a shift modifier or a locking shift and the letter key.

Some keyboards have four levels, with three or four characters available from each key. Figure 10 shows the Arabic keyboard from Digital and the Khmer keyboard from Apple Computer. The user switches into the additional two groups of characters with an additional modifier or shift key. Note that the Arabic keyboard uses the additional group to support Latin characters as well as Arabic, but the Khmer keyboard uses all four groups for the Khmer characters only.[10] A four-group keyboard is now a national standard in Germany.[11]

The *katakana* and *hiragana* syllabaries have approximately 50 characters each. These can be input either directly from the keyboard or through a mapping of the syllable typed with the phonetically equivalent Latin characters. For example, the character マ (*ma*) can be input either by typing the マ key on a Japanese keyboard, or by typing *m* and *a* and using an input method to convert to マ.

Although some early keyboards had many ranks of individual keys, input of ideographic characters from modern keyboards always requires a multiple-stroke input method, with some user interaction.

Figure 8 Nested Bidirectional Text

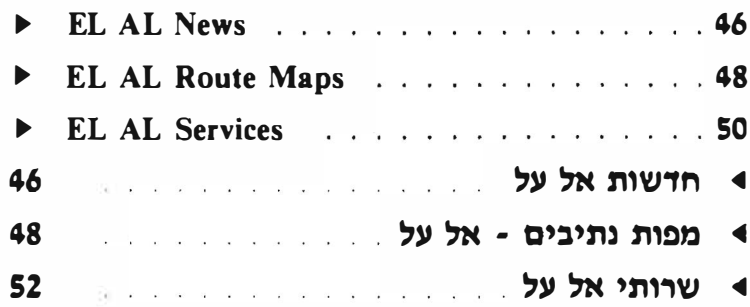


Figure 9 Combined Direction Text

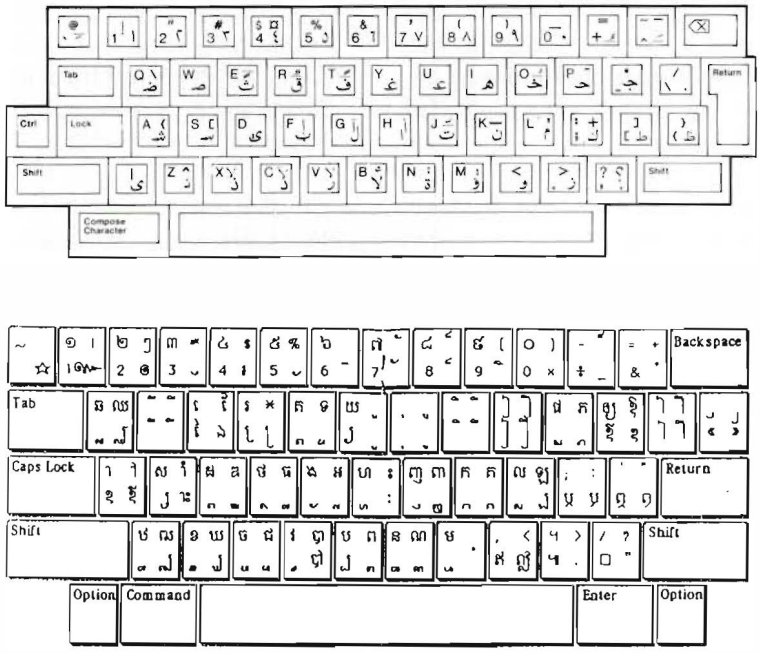


Figure 10 Arabic Keyboard (above) and Khmer Keyboard

Both Japanese and Korean have phonetic writing systems. Users of these languages primarily use phonetic methods to input ideographic characters. The Chinese-language user has many different input methods; these are based on phonetic input or on strokes or shapes in the character. Almost all of these methods display a list of possible candidates as a result of the string input, and the user selects the appropriate candidate. The implementation of Japanese input methods is detailed in a separate paper in this issue of the *Journal*.[12]

Bidirectional Text

Hebrew and Arabic user interfaces have an additional level of difficulty. As discussed earlier, the text is bidirectional; the primary writing direction progresses from right to left and includes left-to-right segments of numbers and non-Hebrew or non-Arabic text. As shown in Figure 8, these segments can nest. The order in which to read bidirectional text can be ambiguous and can depend on the semantics of the text. Figure 11 duplicates the information on a pair of signs displayed at parking lots in Tel-Aviv. Urban legend has it that at least one parking ticket was dismissed by the court on the argument that the sign indicated that parking was not allowed from 5:00 p.m. to 9:00 a.m.

To some extent the correct direction can be assigned automatically. Hebrew and Arabic characters have an implicit direction of right to left, and Latin text has an implicit left-to-right direction. Thus an output method can lay out simple combinations

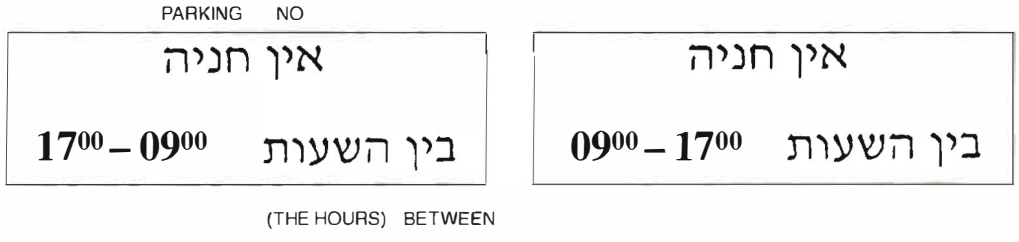


Figure 11 "No Parking" Signs in Tel-Aviv

of bidirectional text correctly. Beyond these characters, direction can be ambiguous. Punctuation marks are common to both Hebrew and Latin text. Thus a period or comma or space has no implicit direction; the software must wait for the next character to determine the direction of the segment. In more complex cases such as the nested directions shown in Figure 8, direction attributes must be explicitly assigned to the segments.[13] As discussed in the paper on Unicode in this issue, the Unicode and ISO 10646 characters sets do include a rich set of directional markers.[14]

Insertion of text should be performed in the way the user finds most convenient, which is not necessarily in accordance with the "correct" directional order of a segment. If entering a two-digit number in the "common" direction requires too many operations, or if the user was trained on a manual typewriter, most users would use the easiest typing order, i.e., entering the least-significant digit first and the most-significant digit second. "Smart" software, which puts the digits in the supposedly correct order, is not doing this user a service.

National Conventions

Various entities such as date, time, and numeric values can be presented differently. Such presentation differences develop both from national and from personal or company styles. These presentation differences are not only tied to different writing systems. For example, dates are presented differently in the United States and in England.

Date Formats

The ninth day of October 1990 is written 9/10/90 in Europe but 10/9/90 in the United States. The order of the day and month numerals is well defined for a particular culture, but there are no overall formats for the separator used, or indeed for the general style. The separator may be a slash, hyphen, colon, space, or another symbol, according to policy or personal preference. The style may be numeric date as shown or the name of the month may be spelled out, and the year may be two or four digits.

In Japan, dates are based on the reign of the emperor. As shown in Figure 12, 1990 was the second year of Heisei, the reign of the current emperor. (The first and last years of two eras may coincide. Showa, the previous era, ended January 7, 1989, and Heisei started on January 8, 1989.) This date format is routinely used in business in Japan. The Western date formats are also used, so a date parsing program should be able to process both formats.

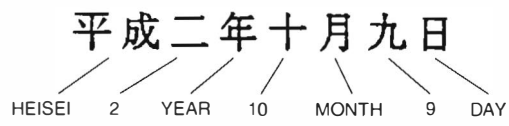


Figure 12 Japanese Date Format

Time-of-day Formats

Similarly, time-of-day formats vary according to personal and, to some extent, national preference. Possible time formats include

9.15am 09:15 0915 09:15:36 09 15 09h15

Time-zone abbreviations also change around the world. Two or more different abbreviations may indicate the same time zone. Eastern Standard Time (EST) is a U.S.-specific time-zone indicator. This zone is called HNE (*Heure Normale de l'Est*) in French-speaking Canada. Central European Time is known as HEC by the French-speaking populations and as MEZ by German speakers. The same time-zone abbreviation may stand for different time zones. AST is used for both Alaska Standard Time and Atlantic Standard Time, which are five hours apart. Time-zone abbreviations are not standardized and may change. Time zones are not all at one-hour intervals. Some countries have time zones at a 30-minute difference from a neighboring zone. Certain towns in Islamic countries use solar time and thus can have time differences of several minutes between towns within one time zone.

Number Formats

The separators used with numerals to express quantities vary as part of national and personal preferences. In the United Kingdom and the United States, the comma is a thousands separator, and the period is a decimal separator. In continental Europe, the opposite is true. Separators include

1,234.56 1.234,56 1 234.56 1234,56
1'234.56 1,234·56

Numbers written in Japanese or Chinese using Chinese ideograms sometimes include the unit indicator, as in the number 28 二十八 ("two","ten", "eight") and sometimes omit it 二八 .

Positive and negative indicators differ. The plus and minus signs may be used before or after the number. In accounting, negative numbers are usually enclosed in parentheses.

Currency Formats

In currency formats, the currency symbol may be one or several characters and may be placed before or at the end of the number, or used instead of the decimal point. Some examples are: ÖS 2,50 (Austria); 2,50 $ (French-speaking Canada); 2$50 (Portugal); and $2.50 (United States).

User Interface

As the point of contact between the user and the machine, the user interface is an obvious area for potential clashes of culture between the designer and the recipient. The interface must be localized to fit the cultural expectations of the end user.[15] The interface designer must be aware of issues of geometry management, images, symbols, color, and sound.

Geometry Management

Graphical interfaces in English use menu bars aligned at the left, with cascading menus falling from left to right. Menus in Hebrew and Arabic cascade from right to left. Figure 13 shows a menu from the Hebrew version of DECwindows XUI.

Although Japanese and Chinese are traditionally read from top to bottom with columns advancing from right to left, most technical material is presented with the same flow as English has. Consequently, user interfaces have the same left-to-right flow as English. This may be considered an aspect of new technology setting new cultural norms. Japanese and Chinese do present some geometry management challenges. A word processor for English uses the right scroll bar to advance from page to page. The analogy is from writing on a long scroll of paper, which is cut into pages. For a Japanese word processor, which enables the user to type in the traditional top-to-bottom orientation, does the bottom scroll bar control page advance by sliding the selection to the left? There is no one correct answer. A designer can keep consistent with the traditional horizontal scroll or with word processors for Latin-based writing systems.

Images

Some designers may consider that using images instead of text creates an international, culturally neutral product that requires no localization. This is only the case if the image is entirely abstract and chosen to be equally foreign to all cultures. This may meet the requirements of internationalization, but at the expense of good user interface design.

Most images are chosen to provide a cultural mnemonic to the action. This link may have little meaning in another culture. The rural mailbox image chosen for certain electronic mail systems is a good example. This image is unknown outside the United States, and some American city dwellers are unfamiliar with it as well. The convention of raising the flag on the mailbox to indicate that new mail has arrived is not common throughout American rural communities. It can instead indicate the presence of outgoing mail.

In addition, a graphic may be a play on words that will not translate. One personal computer product uses a musical note to indicate that a written note is associated with an item in its database.

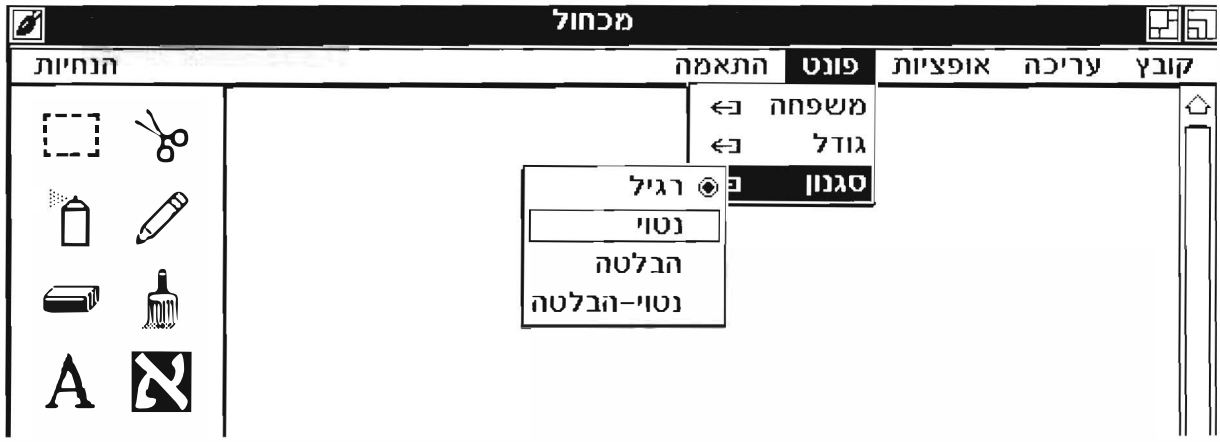


Figure 13 Hebrew DECwindows XUI

Symbols

Symbols commonly used in one culture may be misinterpreted by someone from another culture. For example, the cross ☒ is often used to indicate prohibition. However, in Egypt it does not have this connotation.[16] Designers should allow for the replacement of selection symbols such as ticks (checkmarks) and crosses found in many user interfaces.

☒ *Italic* ☑ **Bold**

Color

The significance of color varies greatly across cultures. Table 1, taken from Russo and Boor, gives the ideas associated with colors in six cultures.[17] For example, red means danger in the United States, but it has the connotation of life and creativity in India. Garland found that using a red "X" as a prohibitive symbol in Egyptian pictures was not effective because the color red is not associated with forbiddance, and the "X" is not understood as prohibitive.[16]

Sound

In the book *Global Software*, Dave Taylor relates that when Lotus localized its 1-2-3 spreadsheet for use in Japan, the developers had to remove all beeps from the program.[18] Japanese users, typically seated much closer together than their Western counterparts, did not appreciate the computer broadcasting to their colleagues every time they made an error. Since beeps can be irritating in open offices in all cultures, well-designed systems allow users to eliminate them or modify the volume.

Functional Differences in Software

Much of this paper has covered areas where the form of the information must change for different cultures. The software may also require functional changes for different cultures. Applications that manipulate text provide a set of operations linked to the nature of both the writing system and the code set. We have seen that typing Japanese and Chinese requires an indirect input method. Applications using the Latin script provide a user interface to an operation to change the case of a character. This operation is not applicable to Japanese, but a Japanese word processor has an operation to convert from *katakana* to *hiragana.*

A delete operation on a Latin letter deletes both the letter and the rectangular cell, a piece of the screen real estate, causing the adjacent text to close up. With the cursor to the right of a Korean syllable cluster or Thai consonant/vowel/tone combination, the user presses the delete key. What should be deleted? Thai and Korean do not have the union between a letter and its linear space that the Latin alphabet has. Two separate operations with different user interfaces may be required, whereas one suffices in English. The code set used also plays a part in determining the nature of the operation. The Thai code set independently codes every letter and tone, so deleting a single letter or tone is practical. The national Korean code set codes syllable

Table 1 Significance of Color across Cultures

	Red	Blue	Green	Yellow	White
U.S.	Danger	Masculinity	Safety	Cowardice	Purity
France	Aristocracy	Freedom Peace	Criminality	Temporary	Neutrality
Egypt	Death	Virtue Faith Truth	Fertility Strength	Happiness Prosperity	Joy
India	Life Creativity		Prosperity Fertility	Success	Death Purity
Japan	Anger Danger	Villainy	Future Youth Energy	Grace Nobility	Death
China	Happiness	Heavens Clouds	Ming Dynasty Heavens Clouds	Birth Wealth Power	Death Purity

clusters.[19] Deleting one letter from a cluster may produce a combination with no code. In Digital's Thai and Korean products, the action of the delete operation is as suggested by the code set. Thai deletes one letter or tone mark; Korean deletes the syllable cluster.

In unidirectional writing systems, the right arrow key navigates the cursor over the logical reading order of the text as it moves smoothly over the screen. The operation of logical movement and geometrical movement across the screen is identical within one line. This is not the case with bidirectional text. The following fragment is from a Hebrew application one two|אחד שתיים שלוש . Pressing the left arrow key moves the cursor to the left of the word "one" if the action means to follow reading order, or to the left of the "o" in "two" if the action is one of navigating screen real estate.

Functional differences may come from regulatory constraints. The United States has export prohibitions on certain encryption techniques. Non-U.S. versions of products may need to remove them or use different techniques. Standards and regulations for connection to external devices such as modems vary around the world.

Product features may also need to vary based on less tangible aspects of a culture. LYRE is a hypertext product developed in France. The product allows students to analyze a poem from various viewpoints selected by the teacher. Students are not allowed to add their own viewpoints. This is acceptable in France but not in Scandinavian countries, where independent discovery is highly valued.[15]

Correct and Incorrect Actions

Learning the rules concerning cultural sensitivity does not guarantee that a software developer from outside, or even inside, that culture will not make errors. Two examples illustrate this.

When Lotus localized its 1-2-3 product into Japanese, the developers were aware that the Japanese date counts the year from the ascension of the emperor to the throne. In their initial test of the product under development in Japan, they included the ability to reset the counter and to modify the field naming the reign. This appears to be admirable planning, sensitive to the needs of the local date format; however, the Japanese users strenuously requested that this feature be removed since it anticipated the demise of the emperor.[20]

In Arabic and Hebrew bidirectional text, deletion of one segment of text can cause the surrounding segments to be rearranged under certain circumstances. This follows from a logical analysis of ordering of the segments and was implemented in an early version of Hebrew DECwrite. Studies with users revealed that they found this rearrangement of text disconcerting and preferred to manually rearrange segments. The program was changed in a subsequent version. Note that this resolution is dependent on the specific product. One should not conclude that automatic reordering of text is always incorrect. Other bidirectional text systems perform this reordering.

Responding to and Setting Culture

New technology in computer applications must reflect the prevalent existing culture, but it also plays a part in creating new cultural norms. An earlier section described how users of a Hebrew word processor might enter digits into a stream of Hebrew by reversing the order of the digits. This cultural behavior was introduced during the days of manual typewriters or older computer systems, which required additional keystrokes to change writing direction. An older technology introduced a cultural expectation. As users in Israel grow more accustomed to word processors that enter the correct order automatically, and as the base of users exposed to older technology shrinks, we can anticipate that the standard expectation of the order in which to enter digits will change.

The Arabic and Khmer writing systems modify the shape of the written glyph based on surrounding characters. The Khmer keyboard (Figure 10) shows separate glyphs for each variation (implying separate codes). This design follows the lead of earlier typewriters and is familiar to users trained on such typewriters. It adds complexity to the keyboard and requires the user to manually enter the correct glyph. The Arabic keyboard is from a system that codes each character independently of glyph; the renderer selects the correct glyph to display based on context. This system may require a longer transition for users trained on manual typewriters, but it is the preferred use of a more advanced technology.

As described previously, written Thai and Korean both use syllable clusters, but the delete operation on each script differs due to the different methods by which the code set represents the script. Which is the correct action? The question does not have an easy answer. From a formal analysis of the language, one might argue that deleting the individual letter

is correct; but as we have seen, formal analysis need not yield an appropriate answer. Ultimately the correct answer is a delicate balance between users' expectations based on the past and the requirements of innovation. The users' expectations are set by previous implementations, which were derived from limitations in the technology of the time. We have a cycle of computers adapting to people adapting to computers.

Deeper Cultural Differences

Some of the cultural differences discussed in this paper such as the presentation of dates and currency are obvious even on a superficial examination of the culture. Others such as the cultural reaction to color are learned from deeper study.

We can expect the future development of software to consider as yet unexplored cultural differences. New features in user interfaces, the use of sound, voice, pen-based computers, and animation, will tie into aspects of cultural behavior that are currently little explored by researchers. Higher-resolution screens and the prevalence of color bring the ability to design applications that relate more directly to the user's sense of beauty.

The personal computer revolutionized personal productivity. Applications such as spreadsheets succeeded because they modeled individual user's existing work practices and extended their capability. A current trend is toward applications for the work group or collaborative computing. This style attempts to revolutionize the way groups work. Jeffrey Hsu reports that "Collaborative systems can meet stubborn resistance when they are introduced in a company, because they challenge the organizational culture with a new means of communication."[21] The difference in the business decision-making process between Japan and the United States is well documented, with Japanese groups valuing group decision and harmony or *wa* highly. We can expect the emerging "groupware" applications both to model existing styles of group work and to change those styles.

The future will also bring software agents.[22] This software will act as a collaborator with the user to process information in much the same way as a human personal assistant. As with a human assistant, we can anticipate that software agents will adapt to the specific requirements and habits of the user, a culture of one. We can imagine an agent recommending circulation lists for memos and aiding in correctly phrasing the mail. The forms of address will vary not only across national boundaries, but across companies. As the set of cultural differences to be addressed goes deeper, the circles of people sharing those cultures will shrink.

Techniques exist to build products with a high level of internationalization. These are described in other papers in this issue. These techniques will continue to develop and improve, but internationalization will never be a fully resolved consideration. The term may fall from use as the cultural differences being addressed have a decreasing relationship to national boundaries. Internationalization is simply making software easy to localize, and the essence of localization is meeting the individual needs of the customers. As computer systems become more powerful and software more sophisticated, adaptation to the individual will continue. Techniques to adjust software to fit personal preferences will continue to develop.

Acknowledgments

This paper is adapted from an earlier unpublished work, circulated within Digital Equipment Corporation. The author would like to thank Gayn Winters for initiating and driving the paper and for many valuable comments on early drafts. This paper took shape as reviews from colleagues around the world corrected errors and contributed many examples, some of which are seen in this paper. The author would like to acknowledge the input from John McConnell and Michael Yau in the United States; Jürgen Bettels in Switzerland; K. H. Chan and Fred Li in Hong Kong; Trin Tantsetthi in Thailand; Mike Feldman, Moti Huberman, and Moshe Loterman in Israel; Hirotaka Yoshioka in Japan; and Nai-peng Kuang in Taiwan.

References and Notes

1. South Korean writing uses two scripts. Hangul is an alphabetic system. Hanja is the set of ideograms imported from China and used as the sole script until the invention of Hangul. There exists a widespread misconception that Hangul is ideographic. The author wishes to stress that only the Hanja script uses ideograms.

2. The example shown contains no phonetic element. Many, more complex characters do have phonetic components. Some scholars disparage the use of the term ideograph to describe Japanese and Chinese writing,

asserting that the phonetic element is primary. (See references 3 and 5.) This paper uses the term ideograph since it is in common use.

3. J. DeFrancis, *The Chinese Language Fact and Fantasy,* Second Paperback Edition (Honolulu: University of Hawaii Press, 1989): 91.

4. M. Stubbs, *Language and Literacy: The Sociolinguistics of Reading* (London, Boston, and Henley: Routledge and Kegan Paul, 1980): 48.

5. J. DeFrancis, *Visible Speech: The Diverse Oneness of Writing Systems* (Honolulu: University of Hawaii Press, 1989).

6. & is an interesting character. It was originally formed as a ligature of *e* and *t* and is now used as an ideogram in many European written languages.

7. F. Coulmas, *The Writing Systems of the World* (Oxford: Basil Blackwell, 1989): 118.

8. A. Nakanishi, *Writing Systems of the World,* third printing (Rutland, VT, and Tokyo: Charles E. Tuttle Company, 1988): 112.

9. The numerals 1, 2, 3, etc., are generally known as, and referred to, as Arabic numerals; however, by one of those quirks of language, the Arabic script uses a different set of symbols for numerals, sometimes called Indic numerals.

10. Note that the Khmer keyboard has four registers because it is based on a glyph encoding of Khmer rather than a character encoding, which would use two registers at most. Also, the subscript Khmer glyphs on the keycaps, which are used in conjuncts, are not necessary if more sophisticated display software is used.

11. DIN 2137, German keyboard for typewriters, Allocation of Characters to Keys, Parts 1, 2, 6, and 11 (Deutsch Institut für Normung, 1988).

12. T. Honma, H. Baba, and K. Takizawa, "Japanese Input Method Independent of Applications," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 97–107.

13. There is some dispute in the industry on the need for explicit direction markers. Under certain circumstances, correct rendition of nested direction text can be computed. For example, a renderer could show the structure of the text in Figure 8 correctly without directional attributes. The Unicode and ISO 10646 character sets do include a rich set of directional markers.

14. J. Bettels and F. Bishop, "Unicode: A Universal Character Code," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 21–31.

15. J. Nielsen, "Usability Testing of International Interfaces," in *Designing User Interfaces for International Use,* edited by J. Nielsen (New York: Elsevier, 1990).

16. K. Garland, "The Use of Short Term Feedback in the Preparation of Technical and Instructional Illustration," in *Research in Illustration: Conference Proceedings Part II* (1982).

17. P. Russo and S. Boor, "How Fluent Is Your Interface? Designing for International Users," paper presented at INTERCHI, Amsterdam, April 1993.

18. D. Taylor, *Global Software: Developing Applications for the International Market* (New York, Berlin, Heidelberg, London, Paris, Tokyo, Hong Kong, Barcelona, Budapest: Springer-Verlag, 1992): 54.

19. The design of the Korean code set reflects compromises made among cultural, economic, and technological requirements. The structure of the writing system leads to independent coding of each *jamo,* with the display device rendering them into syllable clusters. Coding as syllable clusters greatly simplified the prevalent technology of the time and reduced the cost of the display device.

20. F. Hapgood, "A Journey East—The Making of 1-2-3 Release 2J," *Lotus: Computing for Managers and Professionals* (Cambridge, MA: Lotus Development Corporation, 1987).

21. J. Hsu and T. Lockwood, "Collaborative Computing," *BYTE Magazine* (March 1993): 120.

22. L. Tesler, "Networked Computing in the 1990s," *Scientific American* (September 1991).

Jürgen Bettels
F. Avery Bishop

Unicode: A Universal Character Code

A universal character encoding is required to produce software that can be localized for any language or that can process and communicate data in any language. The Unicode standard is the product of a joint effort of information technology companies and individual experts; its encoding has been accepted by ISO as the international standard ISO/IEC 10646. Unicode defines 16-bit codes for the characters of most scripts used in the world's languages. Encoding for some missing scripts will be added over time. The Unicode standard defines a set of rules that help implementors build text-processing and rendering engines. For Digital, Unicode represents a strategic direction in internationalization technology. Many software-producing companies have also announced future support for Unicode.

A universal character encoding—the Unicode standard—has been developed to produce international software and to process and render data in most of the world's languages. In this paper, we present the background of the development of this standard among vendors and by the International Organization for Standardization (ISO). We describe the character encoding's design goals and principles. We also discuss the issues an application handles when processing Unicode text. We conclude with a description of some approaches that can be taken to support Unicode and a discussion of Microsoft's implementation. Microsoft's decision to use Unicode as the native text encoding in its Windows NT (New Technology) operating system is of particular significance for the success of Unicode.

Background

In the 1980s, software markets grew throughout the world, and the need for a means to represent text in many languages became apparent. The complexity of writing software to represent text hindered the development of global software.

The obstacles to writing international software were the following.

1. Stateful encoding. The character represented by a particular value in a text stream depended on values earlier in the stream, for example, the escape sequences of the ISO/IEC 2022 standard.[1]
2. Variable-length encoding. The character width varied from one to four bytes, making it impossible to know how many characters were in a string of a known number of bytes, without first parsing the string.
3. Overloaded character codes and font systems. Character codes tended to encode glyph variants such as ligatures; font architectures often included characters to enable display of characters from various languages simply by varying the font.

In the 1980s, character code experts from around the world began work on two initially parallel projects to eliminate these obstacles. In 1984, the ISO started work on a universal character encoding. This effort placed heavy emphasis on compatibility with existing standards. The ISO/IEC committee published a Draft International Standard (DIS) in spring 1991.[2] By that time, the work on Unicode (described in the next section) was also nearing completion, and many experts were alarmed by the potential for confusion from two competing standards. Several of the ISO national bodies therefore opposed adoption of the DIS and asked that ISO and Unicode work together to design a universal character code standard.

The Origins of Unicode

In some sense Unicode is an offshoot of the ISO/IEC 10646 work. Peter Fenwick, one of the early

conveners of the ISO working group responsible for 10646, developed a proposal called "Alternative B," based on a 16-bit code with no restriction on the use of control octets. He presented his ideas to Joseph Becker of Xerox, who had also been working in this area.[3]

In early 1988, Becker met with other experts in linguistics and international software design from Apple Computer (notably Lee Collins and Mark Davis) to design a new character encoding. As one of the original designers, Becker gave this code the name Unicode, to signify the three important elements of its design philosophy:

1. Universal. The code was to cover all major modern written languages.
2. Unique. Each character was to have exactly one encoding.
3. Uniform. Each character was to be represented by a fixed width in bits.

The Unicode design effort was eventually joined by other vendors, and in 1991 it was incorporated as a nonprofit consortium to design, promote, and maintain the Unicode standard. Today member companies include Aldus, Apple Computer, Borland, Digital, Hewlett-Packard, International Business Machines, Lotus, Microsoft, NeXT, Novell, The Research Libraries Group, Sun Microsystems, Symantec, Taligent, Unisys, WordPerfect, and Xerox. Version 1.0, volume 1 of the 16-bit Unicode standard was published in October 1991, followed by volume 2 in June 1992.[4,5]

It was sometimes necessary to sacrifice the three design principles outlined above to meet conflicting needs, such as compatibility with existing character code standards. Nevertheless, the Unicode designers have made much progress toward solving the problems faced in the past decade by designers of international software.

The Merger of 10646 and Unicode

Urged by public pressure from user groups such as IBM's SHARE, as well as by industry representatives from Digital, Hewlett-Packard, IBM, and Xerox, the ISO 10646 and Unicode design groups met in August 1991; together they began to create a single universal character encoding. Both groups compromised to create a draft standard that is often referred to as Unicode/10646. This draft standard was accepted as an international character code standard by the votes of the ISO/IEC national bodies in the spring of 1992.[6]

As a result of the merger with ISO 10646, the Unicode standard now includes an errata insert called Unicode 1.0.1 in both volumes of version 1.0 to reflect the changes to character codes in Unicode 1.0.[7] The Unicode Consortium has also committed to publish a technical report called Unicode 1.1 that will align the Unicode standard completely with the ISO/IEC 10646 two-octet compaction form (the 16-bit form) also called UCS-2.

Relationship between Unicode and ISO/IEC 10646

Unicode is a 16-bit code, and ISO/IEC 10646 defines a two-octet (UCS-2) and a four-octet (UCS-4) encoding form. The repertoire and code values of UCS-2, also called the base multilingual plane (BMP), are identical to Unicode 1.1. No characters are currently encoded beyond the BMP; the UCS-4 codes defined are the two UCS-2 octets padded with two zero octets. Although UCS-2 and Unicode are very close in definition, certain differences remain.

By its scope, ISO/IEC 10646 is limited to the coding aspects of the standards. Unicode includes additional specifications that help aspects of implementation. Unicode defines the semantics of characters more explicitly than 10646 does. For example, it defines the default display order of a stream of bidirectional text. (Hebrew text with numbers or embedded text in Latin script is described in the section Display of Bidirectional Strings.) Unicode also provides tables of character attributes and conversion to other character sets.

In contrast with the Unicode standard, ISO 10646 defines the following three compliance levels of support of combining characters:

- Level 1. Combining characters are not allowed (recognized) by the software.
- Level 2. This level is intended to avoid duplicate coded representations of text for some scripts, e.g., Latin, Greek, and Hiragana.
- Level 3. All combining characters are allowed.

Therefore, Unicode 1.1 can be considered a superset of UCS-2, level 3.

Throughout the remainder of this paper, we refer to this jointly developed standard as Unicode. Where differences exist between ISO 10646 and Unicode standards, we describe the Unicode functionality. We also point out the fact that Unicode and ISO sometimes use different terms to denote the same concept. When identifying characters, we

use the hexadecimal code identification and the ISO character names.

General Design of Unicode

This section discusses the design goals of Unicode and its adherence to or variance from the principles of universality, uniqueness, and uniformity.

Design Goals and Principles

The fundamental design goal of Unicode is to create a unique encoding for the characters of all scripts used by living languages. In addition, the intention is to encode scripts of historic languages and symbols or other characters whose use justifies encoding.

An important design principle is to encode each character with equal width, i.e., with the same number of bits. The Unicode designers deliberately resisted any calls for variable-length or stateful encodings. Preserving the simplicity and uniformity of the encoding was considered more important than considerations of optimization for storage requirements.

A Unicode character is therefore a 16-bit entity, and the complete code space of over 65,000 code positions is available to encode characters. A text encoded in Unicode consists of a stream of 16-bit Unicode characters without any other embedded controls. Such a text is sometimes referred to as Unicode plain text. The section Processing Unicode Text discusses these concepts in more detail.

Another departure from the traditional design of code sets is Unicode's inclusion of combining characters, i.e., characters that are rendered above, below, or otherwise in close association with the preceding character in the text stream. Examples are the accents used in the Latin scripts, as well as the vowel marks of the Arabic script. Combining characters are allowed to combine with any other character, so it is possible to create new text elements out of such combinations.[8] This technique can be used in bibliographic applications, or by linguists to create a script for a language that does not yet have a written representation, or to transliterate one language using the script of another. An example in recent times is the conversion of some Central Asian writing systems from the Arabic to the Latin script, following Turkey's example in the 1920s (Kazakhstan).

An additional design principle is to avoid duplication of characters. Any character that is nearly identical in shape across languages and is used in an equivalent way in these languages is assigned a single code position. This principle led to the unification of the ideographs used in the Chinese, Japanese, and Korean written languages. This so-called CJK unification was achieved with the cooperation of official representatives from the countries involved.

The principle of uniqueness was also applied to decide that certain characters should not be encoded separately. In general, the principle states that Unicode encodes characters and not glyphs or glyph variations. A character in Unicode represents an abstract concept rather than the manifestation as a particular form or glyph. As shown in Figure 1, the glyphs of many fonts that render the Latin character A all correspond to the same abstract character "a."

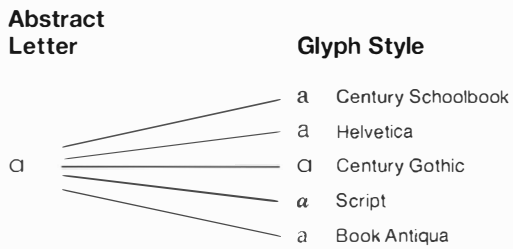


Figure 1 Abstract Latin Letter "a" and Style Variants

Another example is the Arabic presentation form. An Arabic character may be written in up to four different shapes. Figure 2 shows an Arabic character written in its isolated form, and at the beginning, in the middle, and at the end of a word. According to the design principle of encoding abstract characters, these presentation variants are all represented by one Unicode character.[9]

Since much existing text data is encoded using historic character set standards, a means was provided to ensure the integrity of characters upon conversion to Unicode. Great care was taken to create a Unicode character corresponding to each

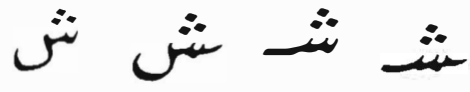


Figure 2 Isolated, Final, Initial, and Middle Forms of the Arabic Character Sheen

character in existing standards. Characters identical in shape appearing in different standards are identified and mapped to a single Unicode character. For characters appearing twice in the same standard, a compatibility zone was created. These characters are encoded as required to make round-trip conversion possible between other standards and Unicode. The Unicode Consortium has agreed to create mapping tables for this purpose.

Text Elements and Combining Characters

When a computer application processes a text document, it typically breaks down text into smaller elements that correspond to the smallest unit of data for that process. These units are called text elements. The composition of a text element is dependent on the particular process it undergoes. The Arabic ligature *lam-alef* is a text element for the rendering process but not for other character operations, such as sorting.

In addition, the same process applied to the same string of text requires different text elements depending on the language associated with the string. Figure 3 shows sorting applied to the string "ch." If this string is part of English text, the text elements for the process of sorting are "c" and "h." In Spanish text, however, the text element for sorting is "ch" because it is sorted as if it were a single character.

For other text-processing operations, text elements might constitute units smaller than those traditionally called characters. Examples are the accents and diacritical marks of the Latin script. These small text elements interact graphically with a noncombining character called a base character. The acute accent interacts with the base character A to form the character A acute. If a given font does not have the character A acute, but it does have A and acute accent as separate glyphs, the character A acute has to be divided into smaller units for the rendering process.

In Thai script, vowels and consonants combine graphically so that the vowel mark can be either before, above, below, or after a consonant, thus forming one display unit. This unit becomes the text element for purposes of rendering. For a process such as advance to next character, however, the individual vowels and consonants are the natural units of operation and are therefore the text elements.

There is no simple relationship between text elements and code elements. As we have shown, this relationship varies both with the language of the text and with the operation to be performed by the application. In earlier encoding systems such as ASCII or others with a strong relationship to a language, this problem was not apparent. When designing a universal character code, the Unicode designers acknowledged the issue and analyzed which character elements have to be encoded as code elements to represent the scripts of Unicode across multiple languages. Rather than burden the character code with the complexity of encoding a rich set of text elements, the Unicode Technical Committee decided that the mapping of code elements to more complex text elements should be performed at the application level.

Spanish	English
curra	charm
chasquido	current
dano	digit

Figure 3 Text Elements and Collation

Code Space Structure

The Unicode code space is the full 16-bit space, allowing for 65,536 different character codes. As shown in Figure 4, approximately 50 percent of this space is allocated. This code space is logically divided into four different regions or zones.

The A-zone, or alphabetic zone, contains the alphabetic scripts. The first 256 positions in the A-zone are occupied by the ISO 8859-1, or 8-bit ANSI codes, in such a way that an 8-bit ASCII code maps to the corresponding 16-bit Unicode character through padding it with one null byte. The positions corresponding to the 32 ASCII control codes 0 to 31 are empty, as well as the positions 0x0080 to 0x009F.

The characters of other alphabetic scripts occupy code space in the range from 0x0000 to 0x2000. Not all of the space is currently occupied, leaving room to encode more alphabetic scripts.

The remainder of the A-zone up to 0x4000 is allocated for general symbols and the phonetic (i.e., nonideographic) characters in use in the Chinese, Japanese, and Korean languages.

The second zone up to 0xA000 is the ideograph, or I-zone, which contains the unified Han characters. Currently about 21,000 positions have been filled, leaving virtually no room for expansion in the I-zone.

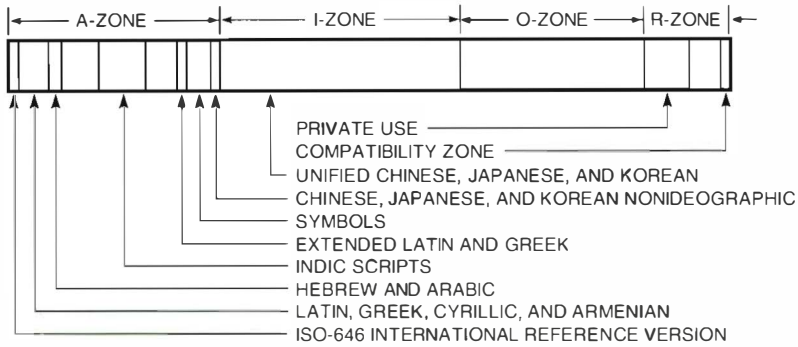


Figure 4 Code Space Allocation for Scripts

The third zone, or O-zone, is a currently unallocated space of 16K. Although several uses for this space have been proposed, its most natural use seems to be for more ideographic characters. However, even 16K can hold only a subset of the ideographic characters.

The fourth zone, the restricted or R-zone, has some space reserved for user-defined characters. It also contains the area of codes that are defined for compatibility with other standards and are not allocated elsewhere.

Processing Unicode Text

The simplest form of Unicode text is often called plain Unicode. It is a text stream of pure Unicode characters without additional formatting or attribute data embedded in the text stream. In this section, we discuss the issues any application faces when processing such text. Processing in this context applies to the steps such as parsing, analyzing, and transforming that an application performs to execute its required task. In most cases, the text processing can be divided into a number of primitive processing operations that are typically offered as a toolkit service on a system. In describing Unicode text processing, we discuss some of these primitives.

Code Conversion

One of the goals of Unicode is to make it possible to write applications that are capable of handling the text of many writing systems. Such an application would typically apply a model that uses Unicode as its native process code. The application could then be written in terms of text operations on Unicode data, which does not vary across the different writing systems.

Today, and for some time to come, however, the data that the application has to process is typically encoded in some code other than Unicode. A frequent operation to be performed is therefore the conversion from the code (file code) in which data is presented to Unicode and back.

One of the design goals of Unicode was to allow compatibility with existing data through round-trip conversion without loss of information. It was not a goal to be able to convert the codes of other character sets to Unicode by simply adding an offset. This would violate the principle of uniqueness, since many characters are duplicated in the various character sets. Most existing character sets therefore have to be mapped through a table lookup. These mapping tables are currently being collected by the Unicode Consortium and will be made available to the public.

It was, however, decided that the 8-bit ASCII, or ISO 8859-1 character set, was to be mapped into the first 256 positions of Unicode. Other character sets (or subsets), such as the Thai standard TIS 620-2529, could also be mapped directly, since character uniqueness was preserved. Also, one of the blocks of Korean syllables is a direct mapping from the Korean standard KSC 5601.

Some character sets contain characters that cannot be assigned code values under the Unicode design rules. Often these characters are different shapes of encoded characters, and encoding them would violate the principle of uniqueness. To allow round-trip conversion for these characters, a special code area, the compatibility zone, was set aside in the R-zone to encode them and to allow

interoperation with Unicode. For example, the wide forms of the Latin letters in the Japanese JIS 208 standard were invented to simplify rendering on monospacing terminals and printers.

Character Transformations

A frequently used operation in text processing is the transformation of one character into another character. For example, Latin lowercase characters are often transformed into uppercase characters to execute a case-insensitive search. In most traditional character sets, this operation would translate one code value to another. Thus, the output string of the operation would have the same number of code values as the input string, and both strings would have the same length. This assumption is no longer true in the case of Unicode strings.

Consider the Unicode characters, Latin small letter a + combining grave accent, i.e., a string of two Unicode characters. If this string were part of a French text (in France), transforming a to A would result in one Unicode character, Latin capital letter A. If the same string were part of a French Canadian text, the accent would be retained on the uppercase character. We can therefore make two observations: (1) The string resulting from a character transformation may contain a different number of characters than the original string and (2) The result depends on other attributes of the string, in this case the language/region attribute.

Another important character transformation operation is a normalization transformation. This operation transforms a string into either the most uncomposed or the most precomposed form of Unicode characters. As an example, we consider the different spellings of the combination:

Ǜ

Latin capital letter U
with diaeresis and grave accent

This letter has been encoded in precomposed form in the Additional Extended Latin part of Unicode. There are two additional spellings possible to encode the same character shape:

Ü + ◌̀

Latin capital letter U with diaeresis
+ combining grave accent

and

U + ◌̈ + ◌̀

Latin capital letter U
+ combining diaeresis
+ combining grave accent

The most uncomposed and the most precomposed forms of these spellings can be considered normalized forms. When processing Unicode text, an application would typically transform the character strings into either of these two forms for further processing.

Note that the spellings:

Ù + ◌̈

Latin capital letter U
with grave accent + combining diaeresis

and

U + ◌̀ + ◌̈

Latin capital letter U
+ combining grave accent
+ combining diaeresis

would result in a different character:

Ǜ

This result is due to the rule that diacritical marks, which stack, must be ordered from the base character outwards.

Byte Ordering

Traditional character set encodings, which are conformant to ISO 2022 and the C language multibyte model, consider characters to be a stream of bytes, including cases in which a character consists of more than one byte. Unicode characters are 16-bit entities; the standard does not make any explicit statement about the order in which the two bytes of the 16-bit characters are transmitted when the data is serialized as a stream of bytes.

The ordering of bytes becomes an issue when machines with different internal byte-order architecture communicate. The two possible byte orders are often called little endian and big endian. In a little-endian machine, a 16-bit word is addressed as two consecutive bytes, with the low-order byte being the first byte; in a big-endian machine, the high-order byte is first. Today all computers based on the Intel 80x86 chips, as well as Digital's VAX and Alpha AXP systems, implement a

little-endian architecture, whereas machines built on Motorola's 680xx, as well as the reduced instruction set computers (RISC) of Sun, Hewlett-Packard, and IBM, implement a big-endian architecture. In blind interchange between systems of possibly different byte order, Unicode-encoded text may be read incorrectly. To avoid such a situation, Unicode has implemented a byte-order mark that behaves as a signature. As shown in Figure 5, the byte-order mark has the code value 0xFEFF. It is defined as a zero-width, no-break space character with no semantic meaning other than byte-order mark.

The code value corresponding to the byte-inverted form of this character, namely 0xFFFE, is an illegal Unicode value. If the byte-order mark is inserted into a serialized data stream and is read by a machine with a different byte-order architecture, it appears as 0xFFFE. This fact signals to the application that the bytes of the data stream have been read in reverse order from that in which they were written and should be inverted. Applications are encouraged to use the byte-order mark as the first character of any data written to a storage medium or transmitted over a network.

Display of Bidirectional Strings

To facilitate internal text processing, a Unicode-compliant application always stores characters in logical order, that is, in the order a human being would type or write them. This causes complications in rendering when text normally displayed right to left (RL) is mixed with text displayed left to right (LR). Hebrew or Arabic is written right to left, but may contain characters written left to right, if either language is mixed with Latin characters. Numerals or punctuation mixed with Hebrew or Arabic can be written in either order.

The Default Bidirectional Algorithm

Unicode defines a default algorithm for displaying such text based on the direction attributes of characters. We outline the algorithm in this paper; for details, see both volumes of the Unicode standard.[4,5] (It is important to consult the second volume because it contains corrections to the algorithm given in the first volume.)

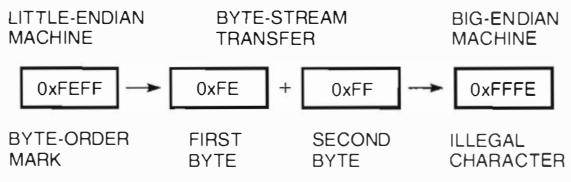


Figure 5 Byte-order Mark

All printing characters are classified as strongly LR, weakly LR, strongly RL, weakly RL, or neutral. In addition, Unicode defines the concept of a global direction associated with a block of text. A block is approximately equivalent to a paragraph. The first task of the rendering software is to determine the global direction, which becomes the default. Embedded strings of characters from other scripts are rendered according to their direction attribute. Neutral characters take on the attribute of surrounding characters and are rendered accordingly.

Directionality Control

Although the default algorithm gives correct rendering in most realistic cases, extra information occasionally is needed to indicate the correct rendering order. Therefore, Unicode includes a number of implicit and explicit formatting codes to allow for the embedding of bidirectional text:

Left-to-right mark	(LRM)
Right-to-left mark	(RLM)
Right-to-left embedding	(RLE)
Left-to-right embedding	(LRE)
Left-to-right override	(LRO)
Right-to-left override	(RLO)
Pop directional formatting	(PDF)

It must be pointed out that the directional codes are to be interpreted only in the case of horizontal text and ignored for any operation other than bidirectional processing. In particular, they must not be included in compare string operations.

The LRM and RLM characters are nondisplayable characters with strong directionality attributes. Since characters with weak or neutral directionality take their rendition directionality from the surrounding characters, LRM and RLM are used to influence the directionality of neighboring characters.

The RLE and LRE embedding characters and the LRO and RLO override characters introduce substrings with respect to directionality. The override characters enforce a directionality and are used to enforce rendering of, for instance, Latin letters or numbers from right to left. Substrings can be nested, and conforming applications must support 15 levels of nesting. Each RLE, LRE, LRO, or RLO character introduces a new sublevel, and the next following PDF character returns to the previous level. The directionality of the uppermost level is implicit or determined by the application.

Only correct resolution of directionality nesting gives the correct result. In general it cannot be assumed that a string of text that is inserted into other bidirectional text will have the correct directionality attributes without special processing. This may result in the removal of directional codes in the text or in the addition of further controls. As shown in Figure 6, particular care needs to be taken for cut-and-paste operations of bidirectional text.

Transmission over 8-bit Channels

Existing communication systems often require that data adheres to the rules of ISO/IEC 2022, which reserve the 8-bit code values between 0x00 and 0x1F (the C0 space), between 0x80 and 0x9F (the C1 space), and the code position DELETE.[1] Since Unicode uses these values to encode characters, direct transmission of Unicode data over such transmission systems is not possible.

The Unicode designers, in collaboration with ISO, have therefore proposed an algorithm that transforms Unicode characters so that the C0 and C1 characters and DELETE are avoided. This algorithm, the UCS transformation format (UTF), is part of the ISO 10646 standard as an informative annex. It is expected that it will be included in the revised Unicode standard.

The transformation algorithm has been conceived in such a way that the characters corresponding to the 7-bit ASCII codes and the C1 codes are represented by one byte (see Figure 4). Code positions 0x00A0 through 0x4015 (which include the remainder of the extended Latin alphabet) are represented by two bytes each, and three bytes each are used for the remaining code values.

Originally, UTF had been proposed for use in data transmission and to avoid the problem that embedded zero bytes represent for C language character strings in the char data type. Subsequently, it has been proposed to use UTF in historical operating systems (e.g., UNIX) to store Unicode-encoded system resources such as file names.[10]

Modifications of UTF have therefore been proposed to address other special requirements such as preservation of the slash (/) character.[11] It remains to be seen which of these various transformation methods will be widely adopted.

Handling of Combining Characters

In some of the operations discussed above, we have indicated that the presence of combining characters requires processing Unicode text differently from text encoded in a character set without combining characters. Normalization or transformation of the characters into a normalized form is usually a first helpful step for further processing. For example, to prepare a text for a comparison operation, one may wish to decompose any precomposed characters. In this way, multiple-pass comparison and sorting algorithms, which typically pass through a level that ignores diacritical marks, can be applied almost unchanged.[12]

For simple comparison operations, the application must decide on a policy of what constitutes equality of two strings. If the string contains characters with a single diacritical mark, it can choose

```
DESTINATION TEXT IN LOGICAL ORDER:          PLEASE SEND TO:_
DIRECTIONALITY NESTING:                    (              )
DESTINATION TEXT IN DISPLAY ORDER:          PLEASE SEND TO:_
                                                    ^
                                                insertion point

TEXT TO BE PASTED IN LOGICAL ORDER:         mr. j. smith, 12 john doe str
DIRECTIONALITY NESTING:                    (          (  )          )
TEXT TO BE PASTED IN DISPLAY ORDER:         rts eod nhoj  12 ,htims .j .rm


PASTED TEXT IN LOGICAL ORDER:               PLEASE SEND TO:_ mr. j. smith, 12 john doe str
DIRECTIONALITY NESTING:                    (              (          (  )          ))
PASTED TEXT IN DISPLAY ORDER WITHOUT NESTING: PLEASE SEND TO:_ htims .j .rm, 12 rts eod nhoj  incorrect!
PASTED TEXT IN DISPLAY ORDER WITH NESTING:  PLEASE SEND TO:_ rts eod nhoj 12 ,htims .j .rm
```

Note: Capital letters signify left-to-right writing. Small letters signify right-to-left writing.

Figure 6 Cut and Paste of Bidirectional Text

either strong matching, which requires the diacritical marks in both strings, or weak matching, which ignores diacritical marks. If the text includes characters with more than one diacritical mark for a medium-strong match, the presence of certain marks might be required but not of others. Strong matching is required for the Greek word for micromaterial μικροϋλικά and the Greek diminuitive form of small μικρούλικα. Without the diacritical marks, the words would be identical.

Unicode requires that combining characters follow the base character. This solution was chosen over the alternatives of (1) precede and (2) precede and follow, for various reasons.[13] Text-editing operations must take into account the presence and ordering of diacritical marks. A user-friendly application should be consistent in its choice of text element on which operations such as next character or delete character operate. This choice should feel natural to the user. For example, in Latin, Greek, and Cyrillic, the expectation would be that accented characters are the unit of operation, whereas in Devanagari and Thai, where several combining characters and a base character combine into a cell, the natural unit is the individual character.

Implementation Issues

In this section we describe some of the approaches that can be taken to support Unicode. As a concrete example, we describe how the Microsoft Windows NT operating system uses Unicode as the native text encoding and maintains compatibility with existing applications based on a different encoding.

General Considerations in Adding Unicode Support

Informal discussions with vendors planning to support Unicode indicate that the following data types and data access are being considered when using the C programming language.

1. A new data type would be designated for Unicode only. It would be directly accessible by the application, e.g., typedef unsigned short UNICHAR.

   The Unicode-only data type has the advantage of being unencumbered with preconceptions about semantics or usage. Also, since the application knows that the contents are in Unicode, it can write code-set-dependent applications.

   The major disadvantage is that the data type would vary from one vendor or platform to another and would therefore have no standard string-processing libraries.

2. An existing data type, such as wchar_t in C would be used. (Note that the char data type is appropriate only if char is defined as 16 bits, or if the string is given some further structure to define its length by means other than null termination. Similar issues may exist in other languages.)

   The use of an existing data type has the advantage of being widely known and implemented; however, it also has the disadvantage of preexisting assumptions about behavior and/or semantics.

3. An opaque object would be used. Since the data in these objects is not visible to the calling program, it can only be processed by routines or by invoking its member functions (e.g., in C++).

Use of an opaque object has the advantage of hiding much of the complexity inherent in the world's writing systems from the application writer. It has the disadvantages common to object-oriented systems, such as the need for software engineers to learn a new programming paradigm and a set of class libraries for the Unicode objects.

How Windows NT Implements Unicode

The Windows NT design team started with several goals to make an operating system that would preserve the investment of customers and developers. These goals affected their decisions regarding the data types and migration strategies described in the previous section.

The goals related to text processing were to

1. Provide backward compatibility
   a) Support existing MS-DOS and 16-bit MS Windows applications, including those based on 8-bit and double-byte character set (DBCS) code pages.
   b) Support the DOS file allocation table file system.
2. Provide worldwide character support in
   a) File names
   b) File contents
   c) User names

As described later in this section, these conflicting goals were met under a single Windows NT

architecture, if not simultaneously in the same application and file system, then by clever segregation of Windows NT into multitasking subsystems. These goals also affect the way Microsoft recommends developers migrate their existing applications to Windows NT.

The Basic Approach Microsoft's overall approach is close to that of using a standard data type that accesses data mainly through string-processing functions. In addition, Microsoft defined a special set of symbols and macros for application developers who wish to continue to develop applications based on DOS (e.g., to sell to those with 286 and 386SX systems), while they migrate their products to run as native Win32 applications on Windows NT. The developer can then compile the application with or without the compiler switch -DUNICODE to produce an object module compiled for a native Windows NT or a DOS operating environment, respectively.

Dual-path Data Types To select the appropriate compilation path, Microsoft provides C language header files that conditionally define data types, macros, and function names for either Unicode or traditional 8-bit (and DBCS) support, depending on whether or not the symbol UNICODE has been defined. An example of a data type that illustrates this approach is TCHAR. If UNICODE is defined, TCHAR is equivalent to wchar_t. Otherwise, it is the same as char. The application writer is asked to convert all instances of char to TCHAR to implement the dual development strategy.

String-handling Functions Similarly, the macro TEXT is defined to indicate that string constants are wide string constants when UNICODE is defined, or ordinary string constants otherwise. Application writers should surround all instances of a string or character constant with this macro. Thus, "Filename" becomes TEXT("Filename"), and 'Z' becomes TEXT('Z'). The compiler treats these as a wide string or character constant if UNICODE is defined, and as a standard char based string or character otherwise.

Finally, there are symbol names for each of the various string-processing functions. For example, if UNICODE is defined, the function symbol name _tcscmp is replaced by wcscmp by the C preprocessor, indicating that the wide character function of that name is to be called. Otherwise, _tcscmp is replaced with the standard C library function strcmp. Details of this procedure can be found in *Win32 Application Programming Interface*.[14]

Procedures for Developing/Migrating Applications in the Dual Path In his paper "Program Migration to Unicode," Asmus Freytag of Microsoft explains the steps used to convert an existing application to work in Unicode and retain the ability to compile it as a DOS or 16-bit Windows application.[15] The basic idea is to remove the assumptions about how a string is represented or processed. All references to string-related objects (e.g., char data types), string constants, and string-processing functions are replaced with their dual-path equivalents. The following steps are then taken.

1. Replace all instances of char with TCHAR, char* with LPSTR, etc. (For a complete listing, see "Program Migration to Unicode.")[15]
2. Replace all instances of string or character constants with the equivalent using the TEXT macro.[16] For example,

```
char filemessage[] = "Filename";
char yeschar = 'Y';
```

   becomes

```
TCHAR filemessage[] = TEXT("Filename");
TCHAR yeschar = TEXT('Y');
```

3. Replace standard char based string-processing functions with the Win32 functions. (See page 221 of *Win32 Application Programming Interface* for a complete listing.)[14]
4. Normalize string-length computations using sizeof( ) where appropriate. For example, direct computation using address arithmetic should take the form: string_length = (last_address − first_address)*sizeof(TCHAR);
5. Mark all files with the byte-order mark.[17]
6. Make other, more substantial changes.

Most character-code-dependent processing should be taken care of by step 3, assuming the developer has used standard functions. If the source code makes assumptions about the encoding, it will have to be replaced with a neutral function call. For example, the well-known uppercasing sequence

```
char_upper = char_lower + 'a' -- 'A';
```

implicitly assumes the language and the uppercasing rules are English. These must be replaced with a function call that accesses the Windows NT Natural Language Services.

Summary

A universal character encoding—the Unicode standard—has been developed to produce international software and to process and render data in most of the world's languages. The standard, often referred to as Unicode/10646, was jointly developed by vendors and individual experts and by the International Organization for Standardization and International Electrotechnical Commission (ISO/IEC). Unicode breaks the (incorrect) principle that one character equals one byte equals one glyph. It stipulates the use of text elements that are dependent on the particular text operation. A number of software vendors are now moving to support Unicode. Microsoft's implementation supports Unicode as the native text encoding in its Windows NT operating system. At the same time, it maintains compatibility with existing applications based on 8-bit encoding.

Acknowledgments

The authors would like to express their thanks to Asmus Freytag of Microsoft Corporation and Masami Hasegawa (ISO/IEC 10646 editor) for their efforts in reviewing this paper.

References and Notes

1. *Information Processing—ISO 7-bit and 8-bit Coded Character Sets—Code Extension Techniques, ISO 2022:1986* (Geneva: International Organization for Standardization, 1986).
2. *Information Technology—Multiple-Octet Coded Character Set, ISO/IEC DIS 10646: 1990* (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).
3. J. Becker, "Multilingual Word Processing," *Scientific American,* vol. 251 (July 1984): 96–107.
4. *The Unicode Standard, Version 1.0, Volume 1* (Reading, MA: Addison-Wesley Publishing Company, 1991).
5. *The Unicode Standard, Version 1.0, Volume 2* (Reading, MA: Addison-Wesley Publishing Company, 1992).
6. *Information Technology—Universal Multiple-Octet Coded Character Set (UCS),* ISO/IEC DIS 10646-1.2:1991 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1991).
7. *Unicode 1.0.1 Errata Insert for The Unicode Standard, Version 1.0, Volume 1 and Volume 2* (Reading, MA: Addison-Wesley Publishing Company, 1992).
8. ISO/IEC 10646 restricts the use of combining characters. See the definitions of level 2 and level 3 in the section Relationship between Unicode and ISO/IEC 10646.
9. Some of the presentation variants are encoded for compatibility with existing standards. For a discussion, see the section Code Conversion.
10. R. Pike and K. Thompson, "Hello World," *Usenix Conference,* 1993.
11. *File System Safe—UCS Transformation Format* (Reading: X/Open Company Limited, 1993).
12. A. LaBonté, "Multiscript Ordering for Unicode," *Proceedings of the Fourth Unicode Implementors Workshop,* Sulzbach (Unicode Inc., 1992).
13. Private communication, Joseph D. Becker, 1993.
14. *Win32 Application Programming Interface* (Redmond, WA: Microsoft Press, 1992).
15. A. Freytag, "Program Migration to Unicode," *Proceedings of the Second Unicode Implementors Workshop,* Merrimack (Unicode Inc., 1992).
16. String constants in source code should be avoided in all cases. They violate one of the fundamental design rules of software internationalization, i.e., that objects dependent on language and/or culture should be isolated into easily accessible modules for the purpose of localization.
17. Unicode defined the code value 0xFEFF to have the semantic byte-order mark (BOM) and encourages software developers to place it as the first character in a Unicode file. (For details, see the section Byte Ordering.)

Wendy Rannenberg
Jürgen Bettels

The X/Open Internationalization Model

Software internationalization standards allow developers to create applications that are neutral with respect to language and cultural information. X/Open adopted a model for internationalization and has revised the model several times to expand the range of support. The latest version of the X/Open internationalization model, which supports multibyte code sets, provides a set of interfaces that enables users in most of Europe and Asia to develop portable applications independent of the language and code set. One implementation of this model, the internationalized DEC OSF/1 AXP version 1.2 (based on OSF/1 release 1.2) supports complex Asian languages such as Chinese and Japanese.

Software internationalization standards initiatives began in the late 1980s. This paper provides a brief history of internationalization standards activities followed by a description and an analysis of the X/Open model for internationalization. The Open Software Foundation's OSF/1 release 1.2 and Digital's DEC OSF/1 AXP version 1.2 internationalization implementations serve as reference software for the description. The analysis covers both the strengths and the limitations of the model. The paper concludes with a discussion of current and future relationships between this model and other work in the field.

Internationalization Standards

The International Organization for Standardization (ISO) is the primary group that is currently publishing or developing internationalization specifications, including code sets, programming languages, and frameworks. Before the ISO adopts emerging specifications, much work is done by other groups. In the case of interfaces that support the development of international applications, the Uniforum Internationalization Technical Work Group, the X/Open Internationalization Work Group, the Unicode Consortium, and the X Consortium have been instrumental.

Internationalization is generally considered to be the processes and tools applied to create software that is neutral with respect to language and cultural information. This neutrality can be accomplished by providing a set of application interfaces designed to isolate sensitivity to language and culture-specific information. Such interfaces include functionality to

- Attain character attributes independent of coded character sets, i.e., code sets
- Order relationships of characters and strings
- Process culturally sensitive format conversion (e.g., date, time, and numbers)
- Maintain user messages for multiple languages

Standardization of internationalization interfaces began predominantly in the UNIX environment. Companies such as Hewlett-Packard and AT&T provided early proprietary solutions.[1]

When X/Open announced its intention to include support for internationalization in Issue 2 of its X/Open Portability Guide (XPG2), Hewlett-Packard submitted its Natural Language Support System as a proposal for an internationalization model. X/Open further developed this proposal and published the guide in 1987.[2] Some principles developed for these solutions found their way into the emerging C programming language standard (ISO/IEC 9899) and the POSIX operating system interface specification (ISO/IEC 9945-1).[3,4]

The subsequent version of the X/Open Portability Guide, XPG3, published in 1989, demonstrated further improvement in internationalization support.[5] The guide was aligned with the ISO/IEC C standard and the ISO/IEC POSIX specification, both of which meanwhile had been finalized.

A major drawback of the XPG3 specification is that it is limited to single-byte code sets. Such code sets are used primarily for western European languages and preclude use of the X/Open internationalization model for Asian and eastern European languages.

The Japanese UNIX Advisory Group developed specifications to extend support to character sets that are encoded in more than one byte. These code sets are generally known as multibyte code sets. The Multibyte Support Extensions developed by this group are now included in an addendum to the ISO/IEC C programming language standard.[6] This work was also adopted by X/Open for inclusion in Issue 4 of the X/Open Portability Guide (XPG4), which was published in 1992.[7,8,9]

However, the underlying model used by X/Open and POSIX does not fully meet the needs of distributed and multilingual computing environments. Therefore, in 1992 X/Open and Uniforum created a joint internationalization work group, commonly referred to as the XoJIG. This group investigated internationalization requirements for distributed and multilingual environments and, in November 1992, published a revised model for internationalization.[10]

The X/Open Internationalization Model

When X/Open first investigated the need for internationalization services, several needs were identified:

- Meet the market requirements of the X/Open member companies. (Many of these requirements were based on the needs of the European Economic Community [EEC].)
- Support more than one language and cultural environment, including messages and date/time.
- Provide for data transparency, i.e., remove 7-bit, U.S. ASCII restrictions from the environment.

As discussed previously, X/Open adopted a model for internationalization and has updated and revised the model many times. The next section describes the current X/Open model.

Overview of the X/Open Portability Model, Issue 4

There are five components to the current X/Open internationalization model, X/Open Portability Guide, Issue 4 (XPG4):

1. Locale announcement mechanism
2. Locale databases
3. Internationalization-specific library routines
4. Internationalized interface definitions for standard C language library routines
5. Message catalog subsystem

The locale announcement mechanism provides a way for an application to load, at run time, a specific set of data that describes a user's native language and cultural information. An application user can specify a language, a territory, and a code set by means of environment variables. The locale announcement mechanism checks the environment variables. If the variables are set, the application attempts to load the locale-specific data. If the environment variables are not set, most applications default to the use of the POSIX (i.e., C language) locale or an implementation-defined locale. The POSIX locale definition is based on the U.S. ASCII code set and the U.S. English language.

In conjunction with locale databases, the announcement mechanism provides access to code set specification data, character collation information, date/time/numerical/monetary formatting information, negative/affirmative responses, and application-specific message catalogs.

Figure 1 shows the relationships among the components of the X/Open internationalization model.[11] Refer to Figure 1 throughout this section, as the various elements of the figure are described.

The locale announcement mechanism is based on the setlocale( ) function

```
char *setlocale(int category,
  const char *locale)
```

The categories correspond to components of the locale database and have a set of corresponding user environment variables. The announcement mechanism supports an order of precedence when querying the user's environment to establish the preferred locale. Table 1 shows the environment variables specified by XPG4.

The *LC_ALL* environment variable has precedence over all others, whereas the *LANG* environment variable has no precedence. The other *LC_** environment variables are of equal weight.

Although it does not provide a naming convention for locales, the X/Open model does specify the locale argument as a pointer to a string in the form

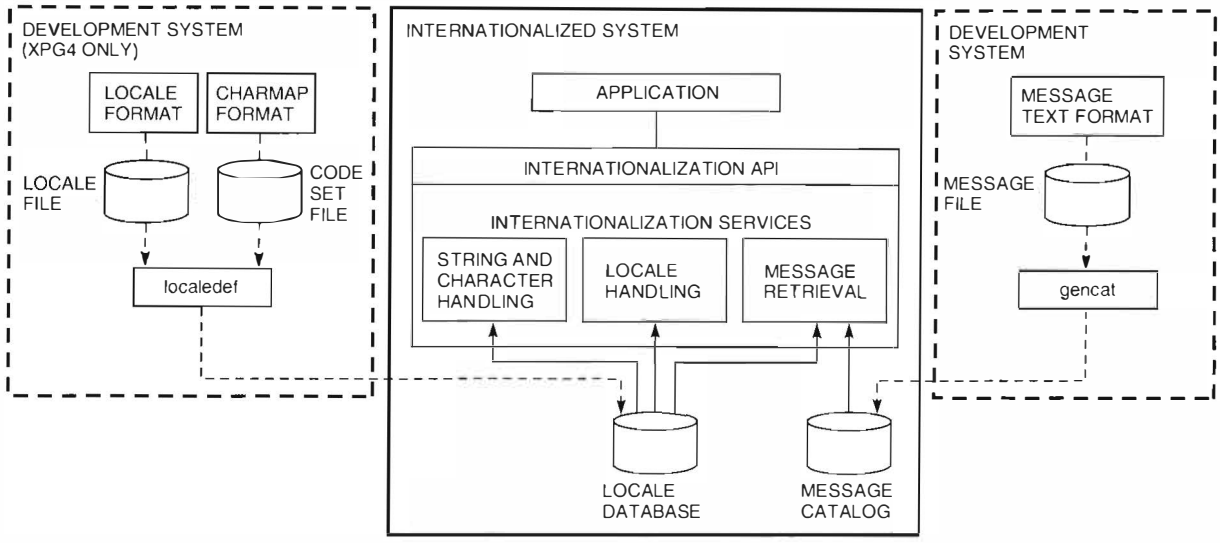


Figure 1 Components of the X/Open Internationalization Model

Table 1 Locale-specific Environment Variables

Variable	Use
LC_ALL	For all categories
LC_COLLATE	For collation
LC_CTYPE	For character classification
LC_MESSAGES	For responses and message catalogs
LC_MONETARY	For monetary information
LC_NUMERIC	For numeric information
LC_TIME	For date/time information
LANG	If no others are set

XPG3:

```
language[_territory[.codeset]][@modifier]
```

XPG4:

```
language[_territory][.codeset][@modifier]
```

Examples of environment variable settings are

LANG = en_US.ISO8859-1

and

LC_COLLATE = ja_JP.jpEUC

The modifier is sometimes used to specify a particular instance of a language or cultural information for a locale. For instance, if support for a particular sort order is necessary, in a German locale the user might specify

LC_COLLATE = de_DE.ISO8859-1@phone

to sort alphabetically according to the telephone directory rather than the dictionary.

Locale databases can be provided by either the system vendor or an application developer. A description of utilities that convert a source format specification of a locale to a binary file follows.

The setlocale( ) function accesses the binary locale databases and provides a global locale within a given application. The global locale is similar to a global variable in that it is shared by all of an application's procedures. Locale switching can be done within an application, but within the scope of the XPG4 model such locale switching is unnecessarily complex and costly, in terms of performance. A later section discusses additional limitations of this mechanism.

The set of interfaces shown in Table 2 supports international application development and was first introduced as part of the ISO/IEC C and the XPG2 and XPG3 specifications. These interfaces are used primarily to access data in the locale databases or to manipulate locale-sensitive data.

The XPG3 specification is based on the use of ISO/IEC 8859-1 as the transmission code set.[12] Some implementations use this as an internal code set, instead of the ASCII code set.

A limited set of functions that support multibyte characters is also available: mblen( ), mbtowc( ), mbtowcs( ), wctomb( ), and wcstombs( ). Each of these functions is based on the ISO/IEC C wide character (wchar_t) data type. The size of the data type is not specified by the standard and can vary

Table 2 Interfaces for International Application Development

Interface	Use
localeconv( )	For retrieving locale-dependent formatting parameters
nl_langinfo( )	For extracting information from the locale database
setlocale( )	For locale announcement
strcoll( )	For locale-based string collation
strftime( )	For converting date/time formats based on locale
strxfrm( )	For transforming a string for collation in current locale

from one implementation to the next, depending on the code set support offered by a particular vendor. This multibyte function set does not provide adequate support for Asian language application development.

In addition to the mb* and wc* functions, the X/Open internationalization model specifies a set of extensions for many library functions and commands. These extensions enable the support of 8-bit characters as well as provide the functionality required to meet the original goal of ensuring data transparency. For example, changes to the printf( ) and scanf( ) families of functions allow the ordering of arguments to be specified in translated message catalogs. In addition, about 80 commands, including sort and date, were modified to support the locale categories.

The XPG specifications include a message catalog subsystem. Although not very sophisticated, this subsystem provides much needed functionality. Minor updates have been made with each new issue of the Portability Guide. The subsystem comprises only three functions: catopen( ), catclose( ), and catgets( ). A command, gencat, is used to convert a message source file into a binary message catalog that is accessed at run time by an application. The behavior of the catopen( ) function is dependent on the user's chosen locale allowing selection of translated messages.

XPG4 Specification and the OSF/1 Release 1.2 Implementation

This section discusses the XPG4 model in terms of the OSF/1 release 1.2 implementation. Topics include code set support, the locale definition utility (the utility for handling data in mixed code sets), worldwide portability interfaces, and local language support.

Code Set Support As mentioned in the previous section, the XPG3 specification primarily supports code sets based on the ISO/IEC 8859-1 specification. The XPG4 model goes beyond this by including additional interface specifications to support multibyte locales and internationalized commands.

The XPG4 model is a superset of the five basic components of the XPG3 model. The use of the wchar_t data type is a key feature of the new interface specifications, because this data type supports multibyte code sets. In the internationalized DEC OSF/1 AXP version 1.2 system, the size of wchar_t is 32 bits, which enables the support of complex Asian languages such as Chinese. This implementation is based on the OSF/1 release 1.2, which is itself designed to support 8-, 16-, or 32-bit wchar_t definitions. The X/Open internationalization model is based on the concept of process and file codes. In the internationalized DEC OSF/1 version 1.2 implementation, the wchar_t data type is used as process code. That is, internal to an application, characters are converted to the wchar_t data type before use. File code, i.e., on-disk data, is always stored as multibyte characters. An application converts all internal process code (i.e., wchar_t data type) character to multibyte character prior to storing it on disk. This enables file compression and enforces the use of a constant width for the processing of character information. The mb* and wc* functions convert between the two types of data. The size of the wchar_t data type combined with the capability to support multiple encoding schemes provides the flexibility required to have a code set-independent implementation.

Restrictions exist on the use of certain characters in the second and subsequent bytes of a multibyte character so that full code set independence is difficult to achieve. An example of such a restriction is the slash character /. The UNIX file system uses this character as a delimiter in absolute and relative pathname specifications. Implementations based on OSF/1 release 1.2 restrict the use of characters in the range 0x00–0x3F to the ASCII code set. However, even with this restriction, it is possible to build robust systems that support a wide range of multibyte code sets.

To gain the necessary flexibility, the Open Software Foundation introduced an object-oriented architecture for the internationalization subsystem.

This architecture specifies the various components of the X/Open model as subclasses. At run time, an application instantiates objects built from these subclasses by means of the setlocale( ) function call.

localedef, iconv, and Code Set Independence XPG3 does not provide a utility for describing locales. Therefore, the number of different approaches to the problem matched the number of vendors. Introduced in the POSIX specification ISO/IEC DIS 9945-2 and hence adopted by X/Open, the localedef utility provides a mechanism for specifying a locale in a portable manner.[13] For each code set supported in the internationalized DEC OSF/1 AXP system, there is a corresponding charmap file and one or more corresponding locale definition files that adhere to the POSIX specifications. Combined with a set of locale-specific methods and code set converter modules, these subclasses provide the foundation for the OSF internationalization architecture.

Locale-specific methods provide a way for the ISO/IEC C language mbtowc( ), wctomb( ) family of functions to work in a multiple code set environment. The wchar_t encoding of a multibyte character in the Japanese SJIS code set is different from that for a character in the Super DEC Kanji code set. At execution time, the correct method is instantiated based on the user's choice of locale. An example of such an instantiation is shown in Figure 2.

A user-level utility (iconv) and several library functions (iconv( ), iconv_open( ), and iconv_close( )) provide a way to handle data that may be in mixed code sets. Internationalized DEC OSF/1 version 1.2 provides an extensive set of code set conversion modules. New conversion methods are easily added to the system.

Worldwide Portability Interfaces The XPG4 internationalization architecture parallels the XPG3/ISO C model. For example, XPG4 specifies a family of isw* functions similar in design to the is* functions

```
LANG = ja_JP.SJIS
  mbtowc( ) → sjis_mbtowc( )
or
LANG = zh_TW.eucTW
  mbtowc( ) → eucTW_mbtowc( )
```

Figure 2 Instantiation of mbtowc( )

(e.g., isalpha) specified in the ISO/IEC C standard. As mentioned previously, the XPG3 model does not include all the interfaces necessary for application developers to handle multibyte code sets. A new set of interfaces, which parallels the set of ISO/IEC C 8-bit interfaces, was developed and integrated into the XPG4 specification. The final version of the interface specification was proposed to the ISO/IEC C committee as the Multibyte Support Extensions.

Cultural Data/Local Language Support Local language support is achieved through the use of locale databases and message catalogs. The catalogs enable translation of user messages. Locale databases have two components: the charmap file and the locale definition file. These databases are created by means of the localedef command.

The charmap file contains a POSIX-compliant specification of the code set, i.e., a one-to-one mapping from character to code point. The locale definition file contains the cultural information. Various sections of the definition file correspond to the categories referenced by the setlocale( ) function. The definition file contains collation specifications, numeric and monetary formatting information, date/time formats, affirmative/negative response specifications, and character classification information. In the OSF/1 release 1.2 implementation, these definition files are independent of the code set. For example, the definition for Japanese (ja_JP) can be combined with multiple charmap files such as SJIS or eucJP.

Strengths of the X/Open Model

The greatest strength of the X/Open internationalization model is that it is in place today and enables the development of portable, language- and code set-independent applications. The internationalized DEC OSF/1 AXP version 1.2 system provides support throughout the commands and utilities for 20 code sets that represent major European and Asian languages. All this is accomplished using XPG4 application programming interfaces (APIs). In addition, the programming paradigm is consistent with ANSI C, making it easier for application developers to modify existing applications for international support.

Limitations of the X/Open Model

As described previously, the X/Open model for internationalization provides a comprehensive set of application interfaces, thus enabling the

development of applications that can be used worldwide. Yet, as with many standards, there are limits to what can be accomplished. In this case, limitations manifest themselves in several areas:

- C language API
- Distributed computing environments
- Multithreaded applications
- Multilingual applications[14]
- Unicode and ISO/IEC 10646 support[15,16]

Because the X/Open and POSIX specifications are based on UNIX implementations, the APIs are specified only for the C programming language. For programming languages such as COBOL, FORTRAN, and Ada, it is not necessarily possible to match the syntax and semantics of the API. The remainder of this section explores generic problems with the global locale model and addresses specific issues in more detail.

Global Locale Issues

The X/Open model is based on the concept of a global locale. This aspect of the model is achieved through the use of locale data that is maintained in a private, process-wide global structure. The use of a global locale is one of the more severe drawbacks to using the overall model.

When working with this model, application developers typically assume that a single language-territory-code set combination is in use at a given time and will remain constant on a per-process basis. Although it is possible to use the announcement mechanism to determine the run-time locale of a process, this mechanism is cumbersome. The application must both save and restore the locale information.

Another drawback of the X/Open model is that existing APIs do not include a way to share locale-specific information between processes. This, combined with the difficulty of locale switching, limits the ability to support multilingual and distributed applications.

Distributed Processing Issues

In a client-server environment, the problem of supporting multiple locales becomes a serious issue. Consider the following examples:

- A server gets requests from various clients, each running their own locale. These requests are processed using the locale of the client. The process includes returning locale-specific user messages to the client and processing user-locale-sensitive date/time formats, collation information, and string manipulation.
- A window manager that supports multiple clients displays menus for a client based on the client's locale. The user error messages displayed are based on the locale of the server.

When a client sends a request to a server, the request parameters that are passed between the client and the server imply an associated locale. Since the global locale is not an explicit argument in any of the XPG4 functions, this locale is difficult to pass to the server. Consider the specific case of remote procedure calls (RPCs), where an interface definition language (IDL) might be used to generate client stubs. Because of the global nature of the locale, insufficient information is available to the IDL to determine if the locale information needs to be used as an argument to any generated functions. Thus, the server may need to change its locale for each client request, which may be unacceptable in terms of system performance.

Using the current XPG model, synchronizing the use of a specific locale between a client and server may not be possible. Even if a client could specify a locale as part of the request, the locale may not be available at the server side or may be replicated incorrectly on the server side. This situation exists because locale names and content are not standardized.

Although the XPG4 specification includes the localedef command for specifying the content of a locale database, there is no provision for standardizing the content. The only locale for which an X/Open specification exists is the POSIX or C locale. In addition, there is no specification for explicitly naming a locale. Locale names are composed of language, territory, and code set components. Many vendors use ISO/IEC 639 and ISO/IEC 3166 for the language and territory components, but there is little agreement on code set naming conventions.[17,18] This naming scheme is not sufficient for uniquely identifying locales, as is required in a client-server model.

Another problem with the X/Open model that impacts application performance and the ease with which an application can be internationalized is related to the process code. The representation of the process code, i.e., wchar_t, is implementation defined, and the mapping of multibyte characters to wide character codes may be locale sensitive.

Therefore, wchar_t-encoded data cannot be exchanged freely between the client-server pair. The only exception would be if the end user guaranteed that the process code was identical for a given locale for each part of the client-server pair. The XPG4 specification does not include functionality to identify or to interrogate the wchar_t encoding scheme used.

Multithreaded Applications

The problems encountered in a distributed processing environment become more complex if the application is also multithreaded. Using POSIX threads, commonly referred to as pthreads, more than one thread is in the execution phase at the same time.[19] Again, a problem with the global, process-wide locale is evident. The application cannot maintain the state of the global locale, accomplished by a save/restore process, without blocking all other threads. Likewise, execution of locale-sensitive functions requires locking all threads to ensure that the global state is not altered prior to completion. The need to continually lock and unlock threads, in addition to being undesirable, results in a performance problem for internationalized applications. Another approach is to make locale data thread-specific.

Multilingual Applications

The X/Open internationalization model is oriented toward the development of monolingual applications. Therefore, the model does not provide functions to handle data that consists of an arbitrary mixture of languages and code sets.

The following are some examples of applications that may require multilingual services:

- Applications that simultaneously interact with a number of users (e.g., transaction processing systems), where each user can choose a language
- A word processing application for multilingual texts that need language-sensitive formatting, hyphenation, etc.

Unicode Support

With the arrival of the Unicode universal character code and the adoption of ISO/IEC 10646 as its form, both POSIX and X/Open have to address the issues of support.[15,16] The X/Open Internationalization Working Group is preparing a paper on Unicode support within the existing specifications; this publication should be available in late 1993. Some of the issues that the C language, POSIX, and XPG4 are facing to support Unicode or ISO/IEC 10646 are character compatibility, code restrictions, and valid character strings.

Unicode characters are incompatible with the C language char* data type used in the POSIX and X/Open models. Unicode characters are 16-bit entities, whereas the POSIX and X/Open characters are in practice 8-bit bytes, even though theoretically the byte size is implementation dependent. Most APIs defined in the POSIX and X/Open models implicitly assume 8-bit characters. This principle is extended to cover Asian multibyte characters by considering each character to be a sequence of 8-bit char data elements. Unicode characters, however, cannot be broken down into sequences of valid 8-bit char* data elements.

The POSIX character model requires that the code values for char* data protect the code ranges for control characters between 0x00–0x1F and 0x80–0x9F, the code position DELETE, and the slash character /. No such restrictions exist in Unicode.

The C language postulates that a null character terminates a char* string. Since the Unicode string most likely contains zero bytes, these bytes would be interpreted as string terminators. In principle, the C language would allow a compiler to define the char* data type to be of 16-bit width. However, given the prevailing assumption in POSIX and XPG4 that one character equals one 8-bit byte, a Unicode character string cannot be a valid char* string.

For these reasons, Unicode cannot be a valid file code as defined by the POSIX and X/Open specifications. Unicode is not usable as an XPG4 process code either. Unicode and ISO/IEC 10646 allow the combining of 16-bit characters.[15] However, in many operations the combining character (e.g., in the French character set, the grave accent) and the base character (e.g., the letter *e*) have to be processed together. This situation contradicts the XPG4 model, where each character of the process code is individually addressed and processed.

Using a well-defined encoding as XPG4 process code would also violate the principle that the process code is opaque, implementation defined, and not valid outside the current process. For all these reasons, the X/Open Joint Internationalization Group decided to propose using Unicode in a modified form of the universal multiple-octet coded character set (UCS) transformation format (UTF).[16,20]

Proposed Changes to the Model

The XPG4 model limitations described in the previous sections are well understood in the internationalization community. X/Open has published a Snapshot specification for a set of distributed internationalization services.[10] This specification does not solve all the problems identified in this paper. It does, however, address the problems associated with the use of the global locale mechanism, locale identification, and text object manipulation. Note that these are proposed changes and have not been adopted by any standards organization.

The proposed changes include

- A locale naming specification that enables the identification of a given locale in a distributed environment
- Definition and support of a locale registry
- A new set of APIs that enables application software to
  - Concurrently manage and use many different locales
  - Manipulate opaque text objects[21]
  - Support stateful and nonstateful encodings and file codes that are excluded by the current standards (e.g., nonzero byte terminators used in the Unicode code set)

Locale Naming and the Locale Registry

In an internationalized environment, the server must replicate the client's locale. If the client's locale can be uniquely identified, the remote code can replicate the locale by obtaining it and specifying this information as part of the operation. To solve the locale replication problem, the XoJIG developed a locale naming scheme, referred to as the locale specification.

The locale specification is a character string that contains the locale name for each category that exists within the locale. The syntax for locale names is a list of keyword-value pairs, where each pair defines a locale category. Certain keywords, such as code set name, encoding name, and owner or vendor name, are standardized as part of the registration process. Table 3 shows two examples of locale specifications.

Although this naming scheme provides for unique identification of locales, the names are long. The specification calls for the use of ASCII characters to name locales. The American English locale specification is over 200 bytes in length. A shorthand notation called network locale specification token has been proposed.

The network locale specification token is an unsigned integer value that can be represented within four bytes. The two most significant bytes represent the registration authority. Under the proposal, national and international standards bodies, companies, and consortia, etc., that wish to use network locale specification tokens will receive unique identifiers. A block of values will be reserved for private use between consenting systems. A set of new functions will allow conversion

Table 3 Network Locale Naming Specifications

American English Locale Using the ISO/IEC Latin-1 Code Set

```
CTYPE=ANSI;en_US;01_00;ISO-88591-1987;;/
COLLATE=ANSI;en_US;01_00;ISO-88591-1987;;/
MESSAGES=ANSI;en_US;01_00;ISO-88591-1987;;/
MONETARY=ANSI;en_US;01_00;ISO-88591-1987;;/
NUMERIC=ANSI;en_US;01_00;ISO-88591-1987;;/
TIME=ANSI;en_US;01_00;ISO-88591-1987;;/
```

Japanese Locale Using Japanese Extended UNIX Code (EUC) Encoding

```
CTYPE=ISO;ja_JP;01_00;JIS-X0208-1987,JIS-X0201-1987,JIS-X0212-1991;EUC;/
COLLATE=ISO;ja_JP;01_00;JIS-X0208-1987,JIS-X0201-1987,JIS-X0212-1991;EUC;/
MESSAGES=ISO;ja_JP;01_00;JIS-X0208-1987,JIS-X0201-1987,JIS-X0212-1991;EUC;/
MONETARY=ISO;ja_JP;01_00;JIS-X0208-1987,JIS-X0201-1987,JIS-X0212-1991;EUC;/
NUMERIC=ISO;ja_JP;01_00;JIS-X0208-1987,JIS-X0201-1987,JIS-X0212-1991;EUC;/
TIME=ISO;ja_JP;01_00;JIS-X0208-1987,JIS-X0201-1987,JIS-X0212-1991;EUC;/
```

between the full locale specification and the locale specification token.

The locale specification proposal solves the problem of unique naming for locales. Combined with a locale registry, this proposal overcomes some of the limitations of the current X/Open model. Within the registry, each locale will have a name defined according to the new syntax. Assuming vendors add these registered locales to their systems, language-sensitive operations in a distributed environment will obtain the same results across systems. This registry has been established by X/Open, and several locales have been submitted.

Multilocale Support

A new set of interfaces, the set of o* functions, has been proposed. These interfaces provide capabilities similar to those defined by the XPG4 model. These new functions address many of the model's limitations, including multithreaded applications, distributed systems, and multilingual applications.

Most of the o* functions utilize three new data types: locale object, attribute object, and text object. To overcome the limitation imposed by a global, per-process locale, the fundamental XPG4 programming paradigm is altered to define localization on a per-call rather than a per-process basis. This change is accomplished by defining a new opaque data type called a locale object. A locale object identifies the locale and can be passed as an argument to locale-sensitive functions on a per-call basis. In this way, the basic programming paradigm becomes

1. Perform operation *X* on data *Y* using locale *Z*

and not

1. Set global locale *Z*
2. Perform operation *X* on data *Y*

An attribute object is a generic opaque object that serves as a container to other opaque objects, such as a locale object. Use of an attribute object in the proposed APIs provides a solution that is not specific to solving internationalization problems. It is anticipated that objects, in addition to the locale object, will be identified. The additional objects might result from requirements in such areas as multimedia, network security, and X11-specific extensions to the locale.

A text object is a new data type that replaces the character (char) and wide character (wchar_t) data types used in the XPG4 internationalization model. As previously defined, a text object refers to a collection of text characters that may or may not have metadata associated with them. Support for directionality, as required for right-to-left languages such as Hebrew, is an example of when such metadata would be introduced. If a text object has a locale defined as part of the metadata (i.e., self-announcing data), the locale specified as part of the data supersedes the locale passed as an argument to the o* functions. The locale that is passed as a function argument acts as a default locale for operations that require it. All o* functions allow a locale identifier to be passed as an argument. This capability eliminates the limitations of the XPG4 global locale. The support of metadata associated with text objects is implementation defined.

A text object data type is represented by a text pointer of type txt_ptr. A text pointer represents all the information associated with a particular character position within the text object. This information is sufficient to perform any kind of operation, such as classification, extraction, or uppercasing.

In summary, the o* functions allow text objects to be classified, converted, transferred to and from files, etc. The functionality of the o* functions is designed to parallel the character-handling functionality provided by the X/Open internationalization model. For example, functions for manipulating text pointers and for concatenating text objects are tuned to the multilocale model. Interfaces have also been introduced to provide management functions for new objects.

Conclusions

When introduced, the X/Open Portability Guide Issue 3 model for internationalization met about 90 percent of the known requirements in the western European market. The introduction of the XPG4 worldwide portability interfaces expanded the region to include Asia, Japan, and eastern Europe. Consequently, application developers can write portable code that supports a variety of languages. The use of the worldwide portability interfaces for computer-aided design applications that are distributed worldwide is one example of such code.

However, the use of the client-server model expanded greatly in the time it took to develop these standards. Also, the need to support truly multilingual applications in a distributed environment became evident. New code set specifications (i.e., Unicode) have been adopted, and systems

supporting Unicode as both file and process code have been implemented. Application vendors are beginning to see their markets expand into every corner of the world.

The XPG4 model will continue to provide much-needed interfaces for quite some time. Yet, to meet the challenges of the truly distributed environment, a new API, similar to the o* functions presented here, must be developed and accepted.

Acknowledgments

Thanks to Mike Feldman, Richard Hart, and Dave Lindner, among others, who spent their time providing comments and recommendations during the writing of this paper.

References and Notes

1. *UNIX System V Release 4 Multi-National Language Supplement (SVR4 MNLS) Product Overview* (Japan: American Telephone and Telegraph Co., 1990).

2. *X/Open Portability Guide, Issue 2* (Reading, U.K.: X/Open Company Ltd., 1987).

3. *Programming Languages—C,* ISO/IEC 9899: 1990 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).

4. *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language],* ISO/IEC 9945-1:1990 (Geneva: International Organization for Standardization/ International Electrotechnical Commission, 1990).

5. *X/Open Portability Guide, Issue 3* (Reading, U.K.: X/Open Company Ltd., 1989).

6. *Multibyte Support Extensions,* ISO/IEC 9899:1990/Amendment 3:1993(E) (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1993).

7. *X/Open CAE Specification, System Interface Definitions, Issue 4,* ISBN 1-872630-46-4, C204 (Reading, U.K.: X/Open Company Ltd., 1992).

8. *X/Open CAE Specification, Commands and Utilities, Issue 4,* ISBN 1-872630-48-0, C203 (Reading, U.K.: X/Open Company Ltd., 1992).

9. *X/Open Internationalisation Guide* (Reading, U.K.: X/Open Company Ltd., 1992).

10. *Distributed Internationalization Services (Snapshot)* (Reading, U.K.: X/Open Company Ltd., 1992).

11. L. Laverdure, P. Srite, and J. Colonna-Romano, *NAS Architecture Reference Manual* (Maynard, MA: Digital Press, 1993): 255–264.

12. *Information Processing—8-bit, Single-byte Coded Graphic Character Sets—Part 1: Latin Alphabet No. 1,* ISO/IEC 8859-1 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1987).

13. *Information Technology—Portable Operating System Interface (POSIX)—Shell and Utilities,* ISO/IEC DIS 9945-2 (Geneva: International Organization for Standardization/ International Electrotechnical Commission, 1992).

14. Multilingual applications can process multiple languages at the same time, whereas implementations of the X/Open model can process several languages but only on an individual basis.

15. J. Bettels and F. Bishop, "Unicode: A Universal Character Code," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 21–31.

16. *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane,* ISO/IEC 10646-1 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1993).

17. *Codes for the Representation of Names and Languages,* ISO/IEC 639 (Geneva: International Organization for Standardization/ International Electrotechnical Commission, 1988).

18. *Codes for the Representation of Names of Countries,* ISO/IEC 3166 (Geneva: International

Organization for Standardization/International Electrotechnical Commission, 1988).

19. *Information Technology—Portable Operating System Interface (POSIX)—Threads Extension for Portable Operating Systems,* IEEE 1003.4a/D7 (New York: The Institute of Electrical and Electronics Engineers, 1993).

20. *File System Safe—UCS Transformation Format* (Reading, U.K.: X/Open Company Ltd., 1993).

21. As defined in the X/Open Draft Internationalization Services Snapshot: A text object is an implementation-defined representation of a fragment of text that consists of zero or more text characters.

General Reference

S. Martin and M. Mori, *Internationalization in OSF/1 Release 1.1* (Cambridge, MA: Open Software Foundation, Inc., 1992).

René Haentjens

The Ordering of Universal Character Strings

In the countries of the world, people have developed various methods to order words and names based on their cultures. Many challenges and problems are associated with developing ways for computers to emulate human ordering methods. An efficient computer method for obtaining a quality ordering has been devised as an extension to the single-step compare. It solves many but not all of the problems. A universal code now exists to store words and names written in many languages and scripts, but there is no universal way to order words and names. Hence, formal specification methods are needed for computer users to describe culture-specific ordering rules. This area is still open to research. Meanwhile, international standardization committees endeavor to formulate sensible proposals for multicultural contexts.

Today, when we access information stored in computers, we often ask the computer to present us lists of items arranged in an order that is meaningful to us and easy to use. In the future, will the computer render obsolete the lists of words and names ordered for human reference? Will the computer look up all information in our place? Will we no longer need the skills to find our way around in dictionaries, telephone directories, and the like? These things are not impossible, but we ourselves might not live to see them happen.

If ordering for human consumption is to stay around for a while, then the next question that we might ask is whether or not it would be possible to harmonize the ways in which lists are ordered around the world. Most people are aware that alphabetic order may differ from one country to another. The same is true for scripts that are not based on an alphabet: although the Chinese Han characters are used to write Japanese and Korean, lists with Han characters are not in the same order in the People's Republic of China, Japan, Korea, and Taiwan, Republic of China.

Can we change to a universal ordering system or at least make ordering the same where the same script is being used? If the order of words were the same, life would surely be easier for the traveler! Unfortunately (if the reader permits that expression), the way in which we work with ordered lists is a cultural aspect and is related to the languages that we use. A proposal to change ordering habits is a bit like proposing a spelling reform. Everyone is in favor of simplification as long as it applies to other groups of people, but we see no reason to change things for ourselves. In fact, looking back to the roots of our own culture, we find many good reasons why things are as they are today, so a change is seldom perceived as an improvement.

The conclusion is, for the time being, that we may as well use the computer to help us organize lists and to take into account that the task of ordering lists is not universally the same.

This paper explores the issues involved with ordering and the ways the computer can deal with them. It describes how people order words and names, and consequently, how they expect words and names to be ordered if a computer does the ordering. It presents examples of ordering in various cultures. This paper concentrates on the ordering of words and names; it does not include a discussion of numerical ordering.

Words, Names, and Character Strings

Computers store words and names as character strings. The symbols that we use for writing are mapped to bit patterns in computers, and these patterns are chained together. For pragmatic reasons, the bit patterns do not correspond to graphic symbols in a simple one-to-one fashion. Attributes such

as the font in which the symbol is presented and the size of the symbol are usually stored in separate areas, and the bit pattern for the specific character that represents the symbol remains the same. Also, several characters or bit patterns can sometimes be represented by the same graphic symbol. For example, the characters LATIN CAPITAL LETTER A and GREEK CAPITAL LETTER ALPHA can be rendered with the same graphic symbol A. Finally, the chaining of characters to strings may not completely agree with the visual arrangement of corresponding graphic symbols.

In other words, there are differences between how people order words and names and how computers order the corresponding character strings. People combine knowledge about words and names (for example, how to read and pronounce them) with visual aspects of the written or printed words and names. Computers must work with the bit patterns.

With regard to character coding, the *International Standard ISO/IEC 10646-1:1993, Universal Multiple-Octet Coded Character Set,* and the de facto standard, Unicode version 1.1, are considered state of the art. These two coding methods can conveniently be considered as identical, and the same abbreviation, UCS, refers to both of them. With UCS coding, words and names can be stored in many of the scripts of the world, and Chinese Han characters can be chained together with Latin, Greek, Cyrillic, Hebrew, and Arabic letters and many more.

Before discussing the complexities of UCS coding, this paper explores some important aspects of ordering of character strings in the next section.

Lexical Ordering

With lexical ordering, the computer takes into account only the kinds of characters that appear in the strings and the arrangement of these characters. Apart from the ordering algorithm and the associated data, the computer uses no other knowledge that it might have about the words in the character strings. For example, it does not use an electronic dictionary or rules about natural language syntax, phonetics, and semantics. The idea is to see how computers can work with reasonably efficient techniques, while staying close to how people work. Meaning-based ordering and searching with the computer is an interesting subject in itself, but is too broad a scope for this paper.

When people order words or names or when they are looking for them in an ordered list, they often use (unconsciously sometimes) the meanings of these words or some other knowledge about the words or names. For example, when looking for the name *McMillan* in a telephone directory, they might try to find it between *MacLeod* and *MacNeville,* knowing that *Mc* is the same as *Mac.* They might even look between *Melbourne* and *Murphy,* ignoring the *Mc* of *McMillan* altogether. If the computer has only a character string that represents the letters of the name *McMillan,* then it lacks the knowledge to look up the name any other way. Lexical ordering cannot incorporate expanding or ignoring prefixes and abbreviations; there is no lexical rule to determine what part of the character string might be a prefix or an abbreviation.

As another example, in Japanese many Han characters (called *kanji* by the Japanese) are pronounced in a different way depending on the context. Japanese dictionaries for general use are ordered by pronunciation; therefore, if the computer has only the *kanji* character in the character string, it cannot order or look up in the same way as people do in Japan. The character for rice, for example, is pronounced *mai* in a form such as *gai mai* (imported rice), but as *bei* in a form such as *bei koku* (America). The difference is due to the historical background of the character or when, in its specific context, it was borrowed from the Chinese. When *kanji* are used in proper names, such as names of persons and geographical names, there may be no context information, and human intervention might be needed to know the correct pronunciation.

In these cases, since the computer must mimic how people order and is limited to lexical techniques, more than codes for the letters or for the *kanji* must be stored in the character strings. For example, the computer might have a character string that contains a *kanji* character plus its pronunciation represented with *kana* characters. Or the computer might have strings such as *(Mc)Millan* with the convention that the parentheses indicate parts to be ignored for ordering and searching.

Modern dictionaries and telephone directories use lexical techniques as much as possible, which is better in a multicultural environment. It is much easier to understand and apply lexical rules for searching than to acquire intuitive knowledge of an unfamiliar culture.

Words, Not Individual Letters

It is important to understand that people order words and names, not just the individual letters and symbols. Consequently, good-quality lexical ordering that comes close to how people work cannot be achieved by looking at all the characters in a string only once, from the first one through the last one. This concept can best be illustrated with alphabetic scripts, and some English examples are given below.

When one looks for *SOS* in a modern English dictionary, one expects to see it between *sort* and *soul*. Now, to find *SOS* between *sort* and *soul*, one must ignore that *SOS* is in uppercase letters and *sort* and *soul* are in lowercase. This type of lookup is achievable by looking at all the letters once.

Now consider the abbreviation *CAT*, meaning *clear air turbulence*. *CAT* is listed between *casual* and *catalyst*. In this case, we cannot ignore the difference between *CAT* and *cat*. The dictionary lists both words, and some dictionaries consistently list lowercase words before uppercase words (or vice versa), so the order using lowercase first would be *casual, cat, CAT, catalyst*. It is not possible to devise an algorithm or method that would arrange these four words in the correct order by looking at all the letters once. To guarantee the correct order in all cases, a first step is needed in which uppercase is considered equal to lowercase; the two words *cat* and *CAT* must be placed in the correct order in a second step, in which uppercase and lowercase make a difference.

Dealing with uppercase and lowercase is not the only issue for alphabetic ordering. Many languages use letters with diacritical marks such as accents. Words and names may also contain spaces or special symbols, such as hyphens, apostrophes, and points. Examples are *big bang, best-seller, rock 'n' roll,* and *P.S.* When ordering is strictly alphabetic, as is the case in many dictionaries, then accents on letters, spacing, and special symbols are ignored in the first step, but they are taken into account to resolve a tie. For example, the correct order in French might be *denier, dénier, dernier;* or *Nb, NB, N.B., Nd, n.d., N.D.* in English.

Table-driven Multilevel Ordering

The heart of ordering methods is the comparison of two character strings. If we have an algorithm to determine whether one string should precede, follow, or be considered equal to a second string, then arranging a list of strings in the correct order is straightforward.

Single-step or One-level Compare

The single-step compare or one-level ordering algorithm is known by most readers:

Compare the first characters of the two strings; if equal, then compare the second characters; continue until a difference is found or until at least one string is exhausted. If a difference is found, then the character-collating sequence determines which string precedes the other. (Example: *words* precedes *working* because *d* precedes *k*.) If one of the two strings is exhausted, then the shorter string precedes. (Example: *word* precedes *words*.) If both strings are exhausted, then they are considered equal.

Multiple-step or Multilevel Compare

The state-of-the-art computer method for comparing character strings is a generalization of the single-step compare. If, after using the above algorithm with the first collating sequence, both strings are found to be equal, then in the second step the algorithm is repeated. Both strings are compared again, starting from their first characters, now using the second collating sequence. The second step may be followed by a third step and so on, one step for each collating sequence.

To be precise, the one collating sequence of all characters is replaced by a matrix of collating weights and collating weight sequences for each weight (W) column. Consider the following example:

	W1	W2	W3
LATIN CAPITAL LETTER D	<D>	<NONE>	<UC>
LATIN SMALL LETTER E	<E>	<NONE>	<LC>
LATIN SMALL LETTER E WITH ACUTE	<E>	<ACUTE>	<LC>
LATIN SMALL LETTER E WITH GRAVE	<E>	<GRAVE>	<LC>
LATIN CAPITAL LETTER E	<E>	<NONE>	<UC>
LATIN CAPITAL LETTER E WITH ACUTE	<E>	<ACUTE>	<UC>
LATIN CAPITAL LETTER E WITH GRAVE	<E>	<GRAVE>	<UC>
LATIN SMALL LETTER F	<F>	<NONE>	<LC>

The collating sequence for W1 is <A>, <B>, <C>, etc. This means that, with the example matrix, all variants of Latin letter E are equal in the first comparison step. The collating sequence for W2 is <NONE>, <ACUTE>, <GRAVE>, which means that in the second step, the accents make a difference, but there is no distinction between lowercase and uppercase variants. That distinction is made in the third step: the collating sequence for W3 is <LC>, <UC>.

The weight matrix and the collating sequences can be placed in tables that are used by the ordering algorithm, hence the name table-driven multilevel ordering.

If this example matrix is extended in a similar way, then the multilevel algorithm would place the following words (most of which are real French words) in this correct order: *dénie, DÉNIE, denier, DENIER, dénier, DÉNIER, dènier, dernier.*

The method that is described here is also used in POSIX (*ISO/IEC 9945-2.2 Shell and Utilities, LC_COLLATE Definition*).[1] Rolf Gavare was among the first to publish a paper on multiple-step comparisons.[2] Alain LaBonté was the first to describe it as explained in this paper, and he also implemented it as a Canadian Standard (CSA Z243.4.1-1992). LaBonté devised a complete and predictable ordering method that corresponds to very fine detail with the best examples of French and English dictionary ordering.[3]

Generate Comparison Key

With the multilevel method, it is also possible to have the algorithm generate a comparison key for a specific character string rather than always compare two strings. These comparison keys can be stored with the character strings; a one-level comparison of keys then gives the same result as a multilevel comparison of the original character strings. For example, and again extending the example matrix given above, the comparison key for *dénie* could be a convenient numerical representation of <D><E><N><I><E><nil><NONE><ACUTE><NONE><NONE><NONE><nil><LC><LC><LC><LC><LC>.

The <nil> precedes all other weights. Its presence at the end of the comparison key subfields guarantees that shorter strings precede longer strings. Efficient compression techniques exist for such comparison keys.

Variations of the Multilevel Method

The following section expands upon the multilevel method and gives examples of changes necessary to accommodate cultural differences in word order.

Special Symbols

With a small extension, the multilevel method can also handle special characters such as the hyphen and the apostrophe to mimic traditional human alphabetic ordering. Another weight column must be added to the matrix given above to distinguish letters from special characters:

```
LATIN SMALL
  LETTER E    <E>      <NONE> <LC>    <LTR>
...
HYPHEN-MINUS IGNORE  IGNORE  IGNORE  <HPH>
```

The IGNORE indicates that the character is skipped in the comparison algorithm in the first three steps. A collating sequence for W4, in which <LTR> precedes all symbols for special characters such as <HPH>, guarantees that words and names without special characters precede the ones with exactly the same letters, but with special characters.

A four-level ordering such as the one suggested here is sufficient for a good-quality, complete, and predictable alphabetic ordering with the Latin alphabet.

Additional Letters

For most languages written in Latin characters, the correct order of words would be *senior, señorita, sentimental, separable.* To achieve this order, W1 would be ..., <M>, <N>, <O>, ..., and the matrix would include LATIN SMALL LETTER N WITH TILDE, where W1 is <N>, W2 is <TILDE>, and W3 is <LC>.

In Spanish, the N WITH TILDE is considered a letter to be ordered between N and O and the correct order is *senior; sentimental, señorita, separable.* To achieve this type of ordering, W1 would be ..., <M>, <N>, <NTILDE>, <O>, ..., and the matrix would add LATIN SMALL LETTER N WITH TILDE, where W1 is <NTILDE>, W2 is <NONE>, and W3 is <LC>.

Ligatures

The multilevel method can also handle ligatures by allowing each matrix element to be a sequence of weights, rather than one weight. For *Æ* in French, the matrix would include LATIN SMALL LIGATURE AE, where W1 is <A><E>, W2 is <LG><LG>, and W3 is <LC><LC>. In these languages, LIGATURE AE is equivalent to two letters when ordering words. In Norwegian, the *Æ* is a letter on its own. W1 is ..., <Y>, <Z>, <AE>, <OSTROKE>, <ARING>. For the matrix element, LATIN SMALL LIGATURE AE, W1 is <AE>, W2 is <NONE>, and W3 is <LC>.

Logograms

Some special symbols, sometimes called logograms, can be seen as short notations for words: & + %. A culture-specific ordering may replace such symbols by the corresponding words. If the language is English, for example, then Research & Development can be ordered as Research and Development. As long as a fixed rule exists for replacing symbols by

equivalent words, the extension that was introduced for Æ can be applied in a similar way to obtain the desired ordering. On the other hand, if the replacement word depends on the language used in the rest of the string, then lexical ordering cannot do the job properly without more information coded in the character strings.

Fine Tuning for the Accents

The table-driven multilevel method, as explained so far, would place French words in this order: *cote, coté, côte, côté, maçon, mâcon.* In a traditional, correct ordering, they should be in the following order: *cote, côte, coté, côté, mâcon, maçon.* (In general, accents at the end of a French word are more important for understanding than other accents.)

To obtain the desired ordering, another extension of the multiple-step method is needed: for the second step, the one that discriminates between quasi-homographs (words that differ only in their diacritical marks), the comparison algorithm should start from the end of the strings rather than from the beginning. For the other Western languages that use the Latin alphabet, this reverse processing for the accents is not needed. On the other hand, it does not hinder either, so the French method is acceptable as well.

French is not the only language with such quasi-homographs. In new-Greek, with the modern *monotoniko* spelling, all multisyllabic words have one accent that indicates the stressed syllable. New-Greek has many quasi-homographs, including the following examples, which use a simple transcription of Greek letters to Latin letters: *árguros, argurós, diakonía, diakoniá, métro, metró, pára, pará.* The French method of reverse processing produces acceptable results for new-Greek as well.

Fine Tuning for the Special Symbols

With the tables extended as explained in the section Special Symbols, the multiple-step algorithm would order words as follows: *unionized, union-ized, un-ionized.* For the exceptional cases such as this one, in which two words are identical except for the placement of a special symbol, the order *unionized, un-ionized, union-ized* may seem more appropriate. Usually, the hyphen is perceived as a word break, not on the first level, but on a subsequent level, and with word breaks, shorter words always come first.

To obtain the latter ordering, one could use the same technique as for the diacritical marks: have the algorithm start from the end of the strings for the level that deals with the special symbols. POSIX has a small extension to the multilevel method that gives similar results while still moving forward. This extension adds the position of the symbol to its table weight during comparison.

Special Symbols in Combination with Uppercase and Lowercase Characters

This section does not introduce a new extension but reconsiders the extension for the special symbols. This method adds a fourth weight column:

```
LATIN SMALL
  LETTER E    <E>     <NONE> <LC>    <LTR>
...
HYPHEN-MINUS IGNORE IGNORE IGNORE <HPH>
```

With W3 for uppercase and lowercase and W4 for the special characters, the distinctions between uppercase and lowercase are considered more important than the presence or absence of spacing and special symbols. In many cultures, this is indeed the case with proper names of people. The following order is desired with names that differ in use of uppercase or lowercase letters: *deGroot, de Groot, Degroot, De groot, DeGroot, De Groot.*

For some geographical names, it could be argued that special symbols are more significant than the difference between lowercase and uppercase. For example, the desired order is *Sanssouci, SANSSOUCI, Sans Souci, SANS SOUCI, Sans-Souci, SANS-SOUCI. (Sanssouci* is a castle near Potsdam in Germany; *Sans Souci* is a city in South Carolina, U.S.A., and a suburb of Sydney, Australia; and *Sans-Souci* is a historical place on Haiti.) To obtain this order, W3 and W4 must be switched.

Some Problems with the Multilevel Method

To obtain the correct order, changes are sometimes necessary to the multilevel method. This section discusses cases in which it is less easy to adapt the table-driven multilevel method.

Digraphs and Collating Elements

CH and LL have special placement in the Spanish alphabet. Spanish is not unique in this respect; combinations of letters also have special placement in the Albanian, Hungarian, Vietnamese, and Welsh alphabets. The Welsh ordering alphabet, for example, is A B C CH D DD E F FF G NG H I J L LL M N O P PH R RH S T TH U W Y, and the following list of words is correctly ordered in Welsh: *acw, achos, adwy,*

addas, agwedd, angau, almon, allan, anfynych, anffodus, antur; anthem.

Before the multilevel method can be applied, it is necessary to replace the multiple-character combinations by pseudo-characters. In POSIX LC_COLLATE, such a mechanism is foreseen. One can declare combinations such as LATIN SMALL LETTER C followed by LATIN SMALL LETTER H to be collating elements and give them a name that can be used in the matrix.

At first it would seem that this solves the problem. One complication, however, is that the two letters together do not always represent the special alphabet letter. In Welsh, for example, the *N* and *G* are separate letters in the Welsh words *melyngoch, dangos, gwyngalchu,* and *mwynglawdd.* The word *melyngoch* then is among words starting with *melyn,* not after the words with *melyg.* More information must be coded in the character strings that represent Welsh words to define a correct lexical ordering.

A similar problem exists with Danish. In most Danish words, *aa* is semantically and phonetically equivalent to *å.* Danes expect *aa* and *å* to be ordered together, after *Z, Æ,* and *Ø.* But in words of foreign origin, *aa* is just A + A.

The reader with a knowledge of programming complexity will probably also see that the collating-element extension makes the table-driven multilevel method less straightforward to implement. If there are only a few collating-element extensions, then simple workarounds might help, but what if there are thousands of them? (Improbable? Wait to form your opinion until you read the section Added Complexity with UCS Coding.)

Sequences, However Long

Other ordering requirements are difficult to accommodate with the matrix method. For example, the British standard on ordering, BS 1749:1985, requires that (in the first step) spaces, dashes, hyphens, and diagonal slashes and sequences of them be treated as a single space (which is significant), except at the beginning of an entry, where they should be ignored. Making a space significant for ordering is easy, but the collating-element extension unfortunately does not allow recursive definitions, so it cannot incorporate the sequences of spaces, etc.

Other Problems

Context dependencies illustrate another problem for collating-element extensions. The Japanese language has several DUP characters, the weights for which depend on the context. For first-level ordering, a DUP character in a Japanese word or name can be considered equivalent to the character that precedes it. Hence, if X represents a Japanese character, then X followed by DUP is equivalent to X followed by X in the first comparison step. Tie breaking is done in a subsequent step: X DUP then precedes X X. If collating-element definitions are used, definitions for all possible combinations are required.

Added Complexity with UCS Coding

The concepts discussed in this section have existed in other coded character sets for some time. For example, ISO 6937 has combining characters, and ISO/IEC 8859-7 contains Latin and Greek letters. With UCS, script mixing and combining characters will for the first time be implemented on a wide scale, not only geographically speaking, but also when counting the number and the importance of the computer platforms on which UCS coding will exist.

UCS has room for some 65,000 characters in the currently defined basic multilingual plane. The first and most obvious implication is that the tables for the multilevel method will be huge with UCS.

Mixing Scripts

With UCS coding, many scripts can be used in a single character string. Although all languages with a non-Latin script have some tradition of incorporating words and names written in Latin letters, there are not many rules about ordering in such a context. For example, where should the Latin-letter abbreviation *SOS* be placed in a Greek, Russian, or Chinese dictionary? The problem with computers, of course, is that everything must be specified, including the unusual situations.

Ordering Han Characters

As previously stated, UCS also codes Han characters. The people who use them for writing characterize a Han character with attributes such as its main radical, the number of pen strokes to draw the character, and its Chinese or Japanese pronunciation. (A radical is a constituent part of the character.)

For example, the Han characters with Japanese pronunciation *tera* (temple), *kata* (type), and *shiro* (capital) all have the same main radical. *Tera* has six strokes; *kata* and *shiro* have nine. The Chinese pronunciations are *ji, kei,* and *jyou.*

A popular ordering is by radical first, then by number of keystrokes, and finally by Chinese pronunciation. With this ordering, *tera* comes first (it has only six strokes), and *kata* precedes *shiro* because of the Chinese pronunciation. If this were the one and only way of ordering Han characters, then the computer would not need to know about the radicals, pen strokes, etc. Each Han character has a different code (bit pattern), so a single (but long) collation order for the corresponding codes would be sufficient.

Significantly, each dictionary of Han characters has developed its own tradition for ordering. Depending on the application, audience, school, or political considerations, the preferred ordering may be different. For example, the *onyomi* ordering is also in popular use in Japan. It is by Chinese pronunciation first, then by stroke count. With *onyomi* ordering, *kata* comes first, then *tera,* and *shiro* is the last one.

Han characters are always ordered character by character, so the multilevel method that applies multiple weights in multiple steps involving complete strings is not required. Han characters require multiple weights with a specific combination that is dynamically selected for a single-step ordering.

It is not evident how this dynamic single step can be combined with the standard multiple-step method, which is needed for UCS strings containing Han characters mixed with other ones.

Combining Characters

UCS also contains the concept of combining characters. In the example matrices given above, it was assumed that letters with accents such as LATIN SMALL LETTER E WITH ACUTE are coded as one character. UCS indeed has such one-character codings, but it allows a letter with an accent to be coded as two characters as well. The sequence of two characters LATIN SMALL LETTER E followed by COMBINING ACUTE is also valid in UCS.

UCS does not state that LATIN SMALL LETTER E WITH ACUTE is the same as LATIN SMALL LETTER E followed by COMBINING ACUTE; it leaves it to applications to consider them equivalent or not. Needless to say, many application developers will want users to have the possibility of considering both forms equivalent, at least for ordering.

The notion of equivalence becomes quite intricate with two or more diacritical marks. See the paper on Unicode in this issue for a discussion on transformations between equivalent spellings.[4]

For our extended matrix method, not only thousands, but an unlimited number of collating elements would have to be defined. UCS allows any number of combining characters to follow a noncombining character.

Logical Order and Coding Order

With UCS coding, the order of the characters in a string is the logical or reading order, not the order in which the symbols have to be printed or displayed. Hence, UCS encoded text is difficult to display and print, but relatively easy to be processed, e.g., for ordering.

In Thai, unfortunately, this approach was not implemented totally. The vowels and diacritics that appear above or under a consonant are coded in logical (reading) order, but Thai has five so-called pre-positioned vowels that are written and coded before the consonant after which they have to be pronounced. This corresponds to current computing practices in Thailand and was incorporated in UCS coding as a sort of backward compatibility. For example, the word written and encoded as E + CH + N (ignoring vowel shortener and tone mark) is pronounced *chên* and ordered accordingly. To allow correct ordering for UCS-encoded Thai, some preprocessing is necessary to arrange the Thai vowels in the correct position for the ordering step.

Formatting Characters

Many coded character sets contain characters that do not correspond to some written symbol but have some control function, often for output formatting. For ordering, these formatting characters can usually be handled in the same ways as special characters.

The characters ZERO WIDTH JOINER and ZERO WIDTH NON-JOINER are among the UCS formatting characters. Their primary purpose is to influence the display of characters of a cursive script such as Arabic. Before UCS was finalized, some people suggested that ZERO WIDTH NON-JOINER might be used to indicate the absence of special digraphs such as in the Welsh word *melyngoch.* It has also been proposed that ZERO WIDTH JOINER might be used to create new letters such as unusual or newly invented ligatures. Today, this is no longer considered a valid use of these formatting characters.

Toward a Formal Description of Ordering

Excellence for computer applications means not only that the application incorporate a different

way of ordering for each culture, but also that it give freedom to its users to define variations and use different approaches to ordering. This is important for some cultures. Not so long ago, the use of multiple letter fonts was considered specialized work for professional printers; today every word processor must allow it. Flexibility with regard to ordering may also become commonplace a few years from now. But how can such flexibility be provided in a computer-digestible yet user-friendly way?

Many documents describe ordering in an informal way. National standards on ordering are seldom formal definitions. They contain directives such as each unbroken sequence of digits, disregarding commas, spaces, and stops is considered as one character; or multiple hyphens collate as one; or *ij* is ordered as *i* + *j*; or ß = *ss*. Such directives are vague for computers. They are imprecise: Is the hyphen to be understood as the character HYPHEN-MINUS only, or also as related, but distinct characters in UCS coding such as HYPHEN, MINUS SIGN, and others? They are also incomplete: *ij* is ordered, but not *IJ*, *Ij*, and *iJ*. They use graphic symbols, where the computer wants to know things about characters: Does ß stand for LATIN SMALL LETTER SHARP S or for GREEK SMALL LETTER BETA?

On the other hand, the descriptions for POSIX LC_COLLATE are quite formal. They are more or less bound to a specific implementation, in this case the table-driven multilevel method described above. A more simple formal description is sometimes sufficient. For example, if the data to be ordered is filtered and contains only uppercase Latin letters, then the POSIX syntax may seem an overkill. In other cases, the LC_COLLATE formalism lacks expressive power, as we have seen.

Is it possible to design a formal specification method that falls between the descriptive texts in country standards and the almost algorithmic parameters such as POSIX LOCALEs?

ISO/IEC 10646-1:1993 may provide a first step to build formal definitions. It is the most comprehensive repertoire of characters to date and a strict superset of many earlier repertoires and coded character sets. Moreover, it establishes a unique and authoritative naming for characters. This paper uses character names such as LATIN CAPITAL LETTER E WITH ACUTE. ISO has decided that the 10646 names will be used in all future character set standards and standard updates. In a certain sense, ISO/IEC 10646-1:1993 is a character reference manual, and formal definitions about ordering can be built upon its content.

Preprocessing

Preprocessing a character string, transforming it into text elements or linguistic units in a logical sequence, is a second concept that deserves elaboration. It was mentioned in relation to Thai with its pre-positioned vowels in a preceding section.

Breaking down a string into the smallest units to be processed by an ordering algorithm and arranging these units in the desired processing order is a powerful mechanism. It could also be used to detect collating elements, to replace Japanese DUP characters, or to transform character sequences that contain combining characters. This mechanism would then allow the table-driven multilevel method to be used to its full extent on preprocessed strings.

Preprocessing might change the character string: units are rearranged, characters are replaced by other ones, etc. It is possible that two originally different character strings could be preprocessed to an identical intermediate form. If ordering is to be complete and predictable, preprocessing must generate additional tags that are taken into account by the multilevel method.

Consequently, the output of the preprocessing phase might be more than pieces of character strings. The lines used in the matrices for the multilevel method have (names of) characters as labels. If preprocessing were designed to generate an output that is easier to consume by the multilevel method, the labels could be anything that seems suitable.

The problem, again, is how to allow for the specification of preprocessing in a formal yet user-friendly way. Transformations based on regular expressions and finite state machines are a possible path. These techniques allow an efficient implementation. P. J. Plauger has published material about using them for ordering with the C language.[5,6]

Conclusions

The evolution of computer systems is progressing toward a better quality interaction with people. An aspect of that interaction is the ordering of words and names. Efficient methods exist today for obtaining a quality ordering. Although some software uses these methods, many applications perform computer-friendly ordering rather than human-friendly ordering. There is no technical limitation to improve on that aspect; for example, a multilevel

algorithm with user-specified tables can replace a single-step bit-code ordering.

For some cultures and in multicultural environments, not all ordering problems are solved. Research is needed, as well as formal rules to allow users to specify ordering preferences.

Some useful ordering techniques are in place. The table-driven multilevel method is an important one. Preprocessing can solve some problems, but a convenient formalism is needed to specify it. UCS coding provides many new challenges; but at the same time it offers a new fixed point, from which it may be possible to derive user-friendly formal definitions.

Appendix: International Standardization Efforts

Many countries have developed a standard on ordering. These standards are not listed in this section.

ISO/IEC JTC1/SC22/WG15 (Programming Languages) is the committee and work group that is discussing the POSIX work (ISO 9945).

ISO/IEC JTC1/SC22/WG20 (Internationalization) is working on a Technical Report that will provide a framework for internationalization. The work group is also preparing documents on the registry of cultural elements, specification methods for defining string comparison, and a default-tailorable ordering for 10646.

CEN (European Standardization Committee) BTS7 (Technical Bureau on IT)/TC304 (Character Set Technology) has a project on European character string ordering rules. The scope is to establish procedures for the registration of national and regional ordering rules and to prepare multilingual character ordering rules for European scripts (Latin, Greek, and Cyrillic).

ISO TC37/SC2/WG2 is currently working on multilingual ordering for terminological and lexicographical purposes. ISO TC46/SC9 has similar work but for bibliographical purposes. The approach is application oriented, whereas the other ISO and CEN efforts mentioned above are computer-oriented approaches.

To allow for some level of synchronization of these efforts and to avoid overlaps, liaisons have been established between all these committees.

Acknowledgments

Alain LaBonté of the Gouvernement du Quebec, Direction Générale des Technologies de l'Information, has been the inspiration for many things written in this paper. He has on many occasions encouraged me to continue with my explorations of ordering. I also owe thanks to Johan van Wingen, independent consultant in Leiden, the Netherlands, who has gathered and made available much background information on coded character sets and ordering practices. A special word of thanks goes to Kevin P. Donnelly, to Denis Garneau, and to P. J. Plauger for reviewing this paper and for providing many useful comments and suggestions.

Of the many colleagues in Digital who have helped me, I want to especially mention Masahiro Morozumi of International Systems Engineering in Japan, with whom I could exchange many mails about ordering in Japanese and about Digital's implementation of XPG4. I also want to mention Tim Greenwood of International Systems Engineering in the U.S., who has done a lot of coordination work for this issue of the *Digital Technical Journal,* and for my contribution to it in particular. And I know that I'm doing an injustice by not naming the many other colleagues who contributed.

References

1. *X/Open CAE Specification,* System Interface Definitions, Issue 4, X/Open Doc N C204 (London: X/Open Company Limited, 1992).
2. R. Gavare, "Alphabetical Ordering in a Lexicological Perspective," *Data Linguistica 18* (Almquist & Wiksell, 1988).
3. A. LaBonté, *Regles du classement alphabetique en langue française et procedure informatisee pour le tri* (Ministere des Communications du Quebec, 1988).
4. J. Bettels and F. Bishop, "Unicode: A Universal Character Code," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 21–31.
5. P. Plauger, "Translating Multibyte Characters," *The Journal of C Language Translation* (June 1991).
6. P. Plauger, *The Standard C Library* (Englewood Cliffs, NJ: Prentice Hall, 1992).

General References

G. Adams, "Introduction to Unicode," *Proceedings of the Fourth Unicode Implementors Workshop* (Mountain View, CA: Unicode Consortium, 1992).

B. Comrie, ed., *The World's Major Languages* (Oxford: Oxford University Press, 1990).

F. Coulmas, *The Writing Systems of the World* (Oxford: Basil Blackwell, 1989).

J. DeFrancis, *The Chinese Language* (Honolulu: University of Hawaii Press, 1984).

D. Garneau, *Keys to Sort and Search for Culturally Expected Results* (Ontario: IBM National Language Technical Center, 1990).

S. Jones et al., *Developing International User Information* (Burlington, MA: Digital Press, 1992).

K. Katzner, *The Languages of the World* (London: Routledge, 1989).

C. Kennelly, *Digital Guide to Developing International Software* (Burlington, MA: Digital Press, 1991).

E. Kohl, "The Art of Arranging Files," *ISO Bulletin* (December 1986).

A. LaBonté, "Multiscript Ordering for Unicode," *Proceedings of the Fourth Unicode Implementors Workshop* (Mountain View, CA: Unicode Consortium, 1992).

A. Nakanishi, *Writing Systems of the World* (Rutland, VT, and Tokyo: Charles E. Tuttle, Co., 1980).

G. Sampson, *Writing Systems* (Hutchinson, 1985).

STRI TS73, *Nordic Cultural Requirements on Information Technology* (Reykjavik: Idntaeknistofnum Islands, 1992).

U. Warotamasikkhadit and D. Londe, *Computerized Alphabetization of Thai,* Technical Memo TM-BA-1000/000/01 (Santa Monica, CA: System Development Corp., 1969).

Unicode 1.0.1, Report from the Unicode Consortium (Mountain View, CA: Unicode, Inc., 1992).

Gayn B. Winters

International Distributed Systems—Architectural and Practical Issues

Building distributed systems for international usage requires addressing many architectural and practical issues. Key to the efficient construction of such systems, modularity in systems and in run-time libraries allows greater reuse of components and thus permits incremental improvements to multilingual systems. Using safe software practices, such as banishing the use of literals and parameterizing user preferences, can help minimize the costs associated with localization, reengineering, maintenance, and design.

The worldwide deployment of computer systems has generated the need to support multiple languages, scripts, and character sets simultaneously. A system should focus on natural ease of use and thus allow end users to read system messages in the language of their choice, to have natural menus, forms, prompts, etc., and to enter and display data in their preferred presentation form.

Digital envisions a computer system that not only is distributed but is distributed geographically across the world. A single site may have end users with varying language and cultural preferences. For example, a Japanese bank in Tel Aviv may have employees whose native languages are Arabic, English, Hebrew, Japanese, or Russian, and may conduct business in one or several of these languages. Figure 1 could represent a portion of their network. The client software, e.g., a mail client and the local windowing system, could be completely monolingual. Networking, database, and printing services, for instance, should be multilingual in that they support the various end users by providing services independent of the natural languages, scripts, or character sets used.

This paper surveys many of the architectural and practical issues involved in the efficient construction of international distributed systems. We begin by discussing some economic issues and pitfalls related to localization and reengineering. Many of these topics can be addressed by straightforward good engineering practices, and we explore several important techniques. The structure of application-specific and system-level run-time libraries (RTLs) is a key issue. We therefore devote several sections

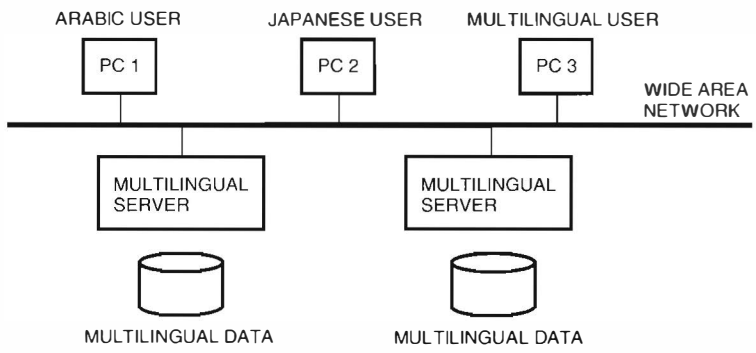


Figure 1 A Portion of a Multilingual Network

of this paper to preferred RTL structures, data representations, and key RTL services. Distributed systems cause some special problems, which we briefly discuss, commenting on naming, security, management, and configuration. In particular, a desire for client software designed for monolingual distributed systems to work without change in a multilingual distributed system led to a new system model. In the model, the host servers and the system management provide the interfaces and conversions necessary for these clients to interface with the multilingual world. Finally, we observe that all the preceding techniques can be delivered incrementally with respect to both increasing functionality and lowering engineering cost.

Localization and Reengineering

When a system component is productized for some local market, the process of making it competitive and totally acceptable to that market is called localization. During this process, changes in the design and structure of the product may be required. These changes are called reengineering. For example, U.S. automobiles whose steering linkages, engine placement, console, etc., were not designed to allow the choice of left- or right-hand steering were not competitive in Japan. Reengineering these automobiles for right-hand steering was prohibitively expensive, so manufacturers had to redesign later models.

Computer systems have problems similar to the automobile left-hand–right-hand steering problem. A good architecture and design is necessary to avoid expensive reengineering during localization. The following are examples of areas in which a localization effort may encounter problems: user-defined characters and ligatures; geometry preferences, such as vertical or right-to-left writing direction, screen layout, and page size; and policy differences, such as meeting protocols and required paper trails. Building limiting assumptions into a software or hardware product can often lead to costly reengineering efforts and regional time-to-market delays.

On the other hand, an internal survey of reengineering problems associated with Digital's software indicates that simple, easy-to-avoid problems are strikingly frequent. In fact, it is amazing how many ways a U.S. engineer could find to make use of the (ultimately erroneous) assumption that one character fits into one 8-bit (or even more constrictive, one 7-bit) byte!

Safe Software Practices

Many well-known, straightforward programming practices, if adopted, can dramatically reduce reengineering efforts.[1-7] Even for existing systems, the cost of incrementally rewriting software to incorporate some of these practices is often more than recovered in lower maintenance and reengineering costs. This section discusses a few key practices.

Probably the most fundamental and elementary safe software practice is to *banish literals,* i.e., strings, characters, and numbers, from the code. Applying this practice does not simply redefine YES to be "yes" or THREE to be the integer 3. Rather, this practice yields meaningful names, for instance, affirmative_response and maximum_alternatives, to help anyone who is trying to understand how the code functions. Thus, not only does the practice make the code more maintainable, but it also makes it easier to parameterize or generalize the data representation, the user interface preferences, and the functionality in ways the original programmer may have missed. These definitions can be gathered into separate declaration files, message catalogs, resource files, or other databases to provide flexibility in supporting clients of different languages.

The abstraction of literals extends to many data types. In general, it is best to *use opaque data types* to encapsulate objects such as typed numbers (e.g., money and weight), strings, date and time of day, graphics, image, audio, video, and handwriting. Providing methods or subroutines for data type manipulation conceals from the application how these data types are manipulated. The use of polymorphism can serve to overload common method and operation names like create, print, and delete. *Support* for *multiple presentation forms* for each data type should allow additional ones to be added easily. These presentation forms are typically strings or images that are formatted according to end-user preferences. Both input and output should be factored first into transformations between the data type and the presentation form, and then into input and output on the presentation form. For example, to input a date involves inputting and parsing a string that represents a presentation form of the date, e.g., "17 janvier 1977," and computing a value whose data type is Date.

The concepts of character and of how a character is encoded inside a computer vary dramatically worldwide.[2,7-11] In addition, a process that works with a single character in one language may need to

work with multiple characters in another language. One simple rule can prevent the problems that this variation can cause: *Banish the Character data type* from applications, and use an opaque string data type instead. This rule eliminates the tempting practice of making pervasive use of how a character is stored and used in the programmer's native system. The Array of Character data type is nearly as insidious, because it is tempting to use the *i*th element for something that will not make sense in another natural language. One should only extract substrings *s*[*i*:*j*] from a string *s*. Thus, when in a given language the object being extracted is only one code point *s*[*i*:*i*], the extraction is obviously a special case. The section Text Elements and Text Operations discusses this concept further.

Another safe software practice is to *parameterize preferences, or better yet, to attach them to the data objects.* As discussed previously, a "hardwired" preference such as writing direction invariably becomes a reengineering problem. The language represented by the string, the encoding type, the presentation form of the object, and the input method for the object are all preferences. In servers and in all kinds of databases, tagging the data with its encoding type is desirable. In general, the data type of the object should contain the preference attributes. The client that processes the object can override the preferences.

Geometry preferences should be user selectable. Some geometry preferences affect the user's working environment, e.g., the ways in which dialog boxes work, windows and pop-up menus cascade, and elevator bars work.[1] These preferences are almost always determined by the end user's working language. Other geometry preferences relate to the data on which the user is working, e.g., paper size, vertical versus horizontal writing (for some Asian languages), how pages are oriented in a book, layouts for tables of contents, and labels on graphs.

Computer programs, in particular groupware applications, mix policy with processing. "Policy" refers to the sequence or order of processing activities. For example, in a meeting scheduler, can anyone call a meeting or must the manager be notified first? Is an invoice a request for payment or is it the administrative equivalent of delivered goods requiring another document to instigate payment? Often such policy issues are not logically forced by the computation, but they need to be enforced in certain business cultures. A sequence of processing activities that is "hardwired" into the program can be very difficult to reengineer. Thus, *policy descriptions should be placed into an external script or database.* The advent of workflow controllers, such as those in Digital's EARS, ECHO, and TeamRoute products, makes it easy to do this.

Applications should not put date formatting, sorting, display, or input routines into their mainline code. Often such operations have been coded previously, and a new application's code will probably not be international and may well contain other bugs. Therefore, programmers should *construct applications to use, or more precisely reuse, run-time libraries,* thus investing in the quality and the multilingual and multicultural capabilities of these RTLs. When the underlying system is not rich enough and/or competition dictates, the existing RTL structures must be augmented.

Run-time Library Structure

A common theme for internationalizing software and for the safe programming practices discussed in the previous section is to keep the main application code independent of all natural language, script, or character set dependencies. In particular, the code must use only RTLs with universal application programming interfaces (APIs), i.e., the name of the routine and its formal parameter list must accommodate all such variants. Digital's early localization efforts typically made the mistake of replacing the U.S.-only code with code that called RTLs specific to the local market. This practice generated multiple versions of the same product, each of which needed to be changed whenever the pertinent part of the U.S. version was changed. A better structure for run-time libraries is shown in Figure 2.

The application illustrated in Figure 2 calls an RTL routine through the routine's universal APIs. This routine may in turn call another language-specific routine or method, or it may be table driven. For example, a sort routine may be implemented

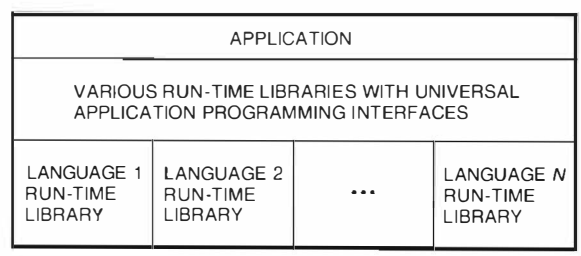


Figure 2 Modular Run-time Library Structure

using sort keys rather than compare functions for better performance. With this structure, localization to a new language involves only the addition of the new language-specific RTL or the corresponding new table entries.

Note that the application must pass sufficient structure to the RTL to guarantee that the APIs are universal. For example, to sort a list of strings, a call

```
sort_algorithm(list_pointer,sort_name,sort_order)
```

could be created. The sort_order parameter is of the type {ascending,descending}. The sort_name parameter is necessary because in many cultures numerous methods of sorting are standard.[1,12] In some RTL designs, notably those specified by X/Open Company Ltd., these extra parameters are passed as global variables.[5,6,7] This technique has the advantage of simplifying the APIs and making them almost identical to the APIs for the U.S. code. Such RTLs, however, do not tend to be thread-safe and have other problems in a distributed environment.[5,13,14] An alternative and far more flexible mechanism is more object oriented—using a subtype of the List of String data type when alternate sorts are meaningful. This subtype has the additional information (e.g., sort_name and sort_order) used by its Sort method.[12,14]

The next three sections discuss the organization and extensibility of RTLs with this structure.

Data Representation

Data representation in RTLs incorporates text elements and text operations, user-defined text elements, and document interchange formats.

Text Elements and Text Operations

A text element is a component of a written script that is a unit of processing for some text operation, such as sorting, rendering, and substring search. Sequences of characters, digraphs, conjunct consonants, ligatures, syllables, words, phrases, and sentences are examples of common text elements.[10,15] An encoded character set E represents some particular set of text elements as integers (code points). Typically, the range of E is extended so that code points can represent not only text elements in multiple scripts but also abstractions that may or may not be part of a script, such as printing control codes and asynchronous communication codes.[16] More complex text elements can be represented as sequences of code points. For example, *Û* may be represented by two code points <U> <^>, and a ligature such as *Œ* may be represented as three code points <O> <joiner> <E>, where a "joiner" is a special code point reserved for creating text elements. Less complex text elements, i.e., subcomponents of the encoded text elements, are found by using the code point and the operation name to index into some database that contains this information. For example, if <é> is a single code point for *é*, then the base character *e* is found by applying some function or table lookup to the code point <é>. The same is true for finding a code point for the acute accent. When a sequence of code points represents a text element, the precise term "encoded text element" is often abbreviated as "text element."

An encoded character set of particular importance is Unicode, which addresses the encoding of most of the world's scripts using integers from 0 to $2^{16}-1$.[11,17] The Unicode universal character set is the basis of ISO 10646, which will extend the code point interval to $2^{31}-1$ (without using the high-order bit).[9] Unicode has a rich set of joiner code points, and it formalizes the construction of many types of text elements as sequences of code points.

Processing text elements that are represented as sequences of code points usually requires a three-step process: (1) the original text is parsed into operation-specific text elements, (2) these text elements are assigned values of some type, and (3) the operation is performed on the resulting sequence of values. Note that each step depends on the text operation. In particular, a run-time library must have a wide variety of parsing capabilities. The following discussion of rendering, sorting, and substring searching operations demonstrates this need.

In rendering, the text must be parsed into text elements that correspond to glyphs in some font database. The values assigned to these text elements are indexes into this database. The rendering operation itself gets additional data from a font server as it renders the text onto a logical page.

The sorting operation is more complicated because it involves a list of strings and multiple steps. A step in most sorting algorithms involves the assignment of collation values (typically integers) to various text elements in each string. The parsing step has to take into account not only that multiple code points may represent one character but also that some languages (Spanish, for example) treat multiple characters as one, for the purposes of sorting. Thus, a sorting step parses each string into text

elements appropriate for the sort, assigns collation values to these elements, and then sorts the resulting sequences of values. Note that the parsing step that takes place in a sorting operation is somewhat different from the one that occurs in a rendering operation, because the sort parse must sometimes group into one text element several characters, each of which has a separate glyph.

Searching a string *s* for a substring that matches a given string *s'* involves different degrees of complexity depending on the definition of the term "matches." The trivial case is when "matches" means that the substring of *s* equals *s'* as an encoded substring. In this case, the parse only returns code points, and the values assigned are the code point values. When the definition of "matches" is weaker than equality, the situation is more complicated. For example, when "matches" is "equal after uppercasing," then the parsing step is the same one as for uppercasing and the values are the code points of the uppercased strings. (Note that uppercasing has two subtle points. The code point for a German sharp *s*, <ß>, actually becomes two code points <S><S>. Thus, sometimes the values assigned to the text elements resulting from the parse consist of more code points than in the original string. In addition, this substring match involves regional preferences, for example, uppercasing a French *é* is *E* in France and *É* in Canada.) The situation is similar when "matches" equals "equal after removing all accents or similar rendering marks." A more complex case would be when *s'* is a word and finding a match in *s* means finding a word in *s* with the same root as *s'*. In this case, the operation must first parse *s* into words and then do a table or dictionary lookup for the values, i.e., the roots.

User-defined Text Elements

When the user of a system wishes to represent and manipulate a text element that is not currently represented or manipulated by the system, a mechanism is required to enable the user to extend the system's capabilities. Examples of the need for such a mechanism abound. Chinese ideograms created as new given names and as new chemical compounds, Japanese *gaiji* (user-defined characters), corporate logos, and new dingbats are often not represented or manipulated by standard systems.

User-defined text elements cause two separate problems. The first problem occurs when E, the encoded character set in use, needs to be extended so that a sequence of E's code points defines the desired user-defined text element. The issues related to this problem are ones of registration to prevent one user's extensions from conflicting with another user's extensions and to allow data interchange.

The second, more difficult problem concerns the extensions of the text operations required to manipulate the new text element. For each such text operation, the parsing, value mapping, and operational steps discussed earlier must be extended to operate on strings that involve the additional code points of E. When tables or databases define these steps, the extensions are tedious but often straightforward. Careful design of the steps can greatly simplify their extensions. In some cases, new algorithms are required for the extension. To the extent that these tables, databases, or algorithms are shared, the extensions must be registered and shared across the system.

Document Interchange Formats

Compound documents (i.e., documents that contain data types other than text) use encoded character sets to encode simple text. Although many new document interchange formats (DIFs) will probably use Unicode exclusively (as does Go Computer Corporation's internal format for text), existing formats should treat Unicode as merely another encoded character set with each character set being tagged.[18] This allows links to be made to existing documents in a natural way.

Many so-called revisable DIFs, such as Standard Generated Mark-up Language (SGML), Digital Document Interchange Format (DDIF), Office Document Architecture (ODA), Microsoft Rich Text Format (RTF), and Lotus spreadsheet format (WKS), and page description languages (PDLs), such as PostScript, Sixels, or X.11, can be extended to provide this Unicode support by enhancing the attribute structure and extending the text import map Strings(E)→DIF for each encoded character set E. In doing so, however, many of the richer constructs in Unicode, e.g., writing direction, and many printing control codes are often best replaced with the DIF's constructs used for these features instead.[19] In this way, both processing operations are easier to extend and facilitate the layout functions DIF→PDL and the rendering functions PDL→Image.

Presentation Services

The practice of factoring input and output of data types into a transformation T↔T_Presentation_Form and performing the I/O on the presentation form allows one to focus on each step separately. This factorization also clarifies the applicability of various user preferences, e.g., a date form preference applies to the transformation, and a font preference applies to how the string is displayed. As mentioned in the section Safe Software Practices, preferences such as presentation form are best attached to the end user's copy of the data. Data types such as encoded image, encoded audio, and encoded video pose few international problems except for the exchangeability of the encodings and the viability of some algorithms for recognizing speech and handwriting. Algorithms for presentation services can be distributed, but we view them as typically residing on the client.[20] In Figure 1, we presume that the local language PCs have this capability.

Input

Existing technology offers several basic input services, which are presented in the following partial list of device-data type functions:

- Keystrokes →Encoded Character
- Image→Encoded Image
- Audio Signal→Encoded Audio
- Video Signal →Encoded Video
- Handwriting→Encoded Handwriting

The methods for each input service depend on both the device and the digital encoding and often use multiple algorithms. Whereas for some languages the mapping of one or more keystrokes into an encoded character (e.g., [compose] + [e] + [´] yielding *é*) may be considered mundane, input methods for characters in many Asian languages are complex, fascinating, and the topic of continuing research. The introduction of user-defined text elements, which is more common among the Asian cultures, requires these input methods to be easily extendable to accommodate user-defined characters.

Output

The basic output services are similar to the input services listed in the previous section.

- Strings→Image
- DIF →PDL →Image
- Encoded Image→Image
- Encoded Audio→Audio Signal
- Encoded Video→Video Signal
- Encoded Handwriting→Image

These output services also vary with encoding, device, and algorithm. Figure 3 illustrates the sequence DIF →PDL→Image. Optional parameters are permitted at each step. A viable implementation of Strings→Image is to factor this function by means of the function Strings →DIF, which is discussed in the Data Representation section. Alternatively, the data type Strings can be simply viewed as another DIF to be supported.

A revisable document begins in some DIF such as plain text, Strings(Unicode), SGML, or DIDIF. A layout process consumes the document and some logical page parameters and creates an intermediate form of the document in some PDL such as PostScript, Sixels, or even a sequence of X.11 packets. To accomplish this, the layout process needs to get font metrics from the font server (to compute relative glyph position, word and line breaks, etc.). In turn, the rendering process consumes the PDL and some physical media parameters to create the image that the end user actually sees. The rendering process may need to go back to the font server to get the actual glyphs for the image. Rendering, layout, and font services are multilingual services. The servers for these services are the multilingual servers envisioned in Figure 1.

Computation Services

To build systems that process multilingual data, such as the one shown in Figure 1, a rich variety of text operations is necessary. This section categorizes such operations, but a complete specification of their interfaces would consume too much space in this paper. Text operations require parsing, value mapping, and operational functions, as described earlier.

Text Manipulation Services

Text manipulation services, such as those specified in C programming language standard ISO/IEC 9899:1990, System V Release 4 Multi-National

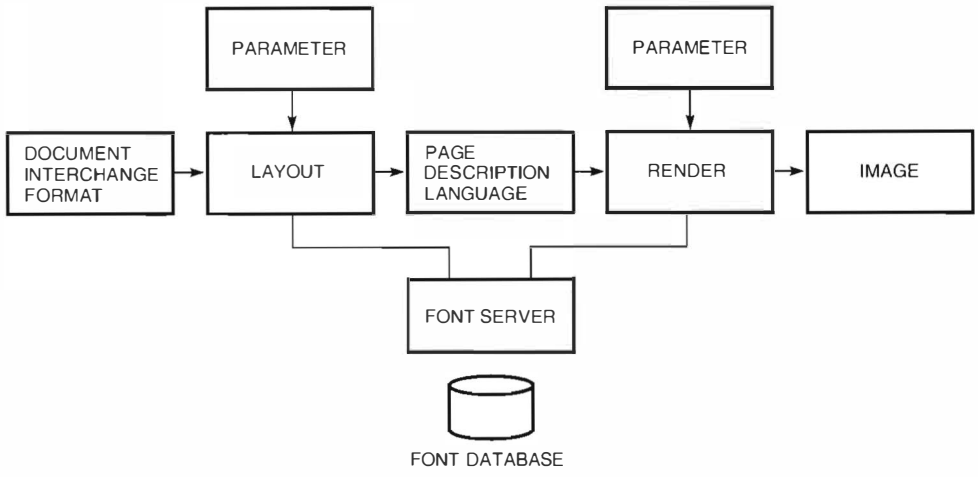


Figure 3 Layout and Rendering Services

Language Supplement (MNLS), or XPG4 run-time libraries (including character and text element classification functions, string and substring operations, and compression and encryption services) need to be extended to multilingual strings such as Strings(Unicode) and other DIFs, and to various text object class libraries.[6,8,13]

Data Type Transformations

Data type transformations (e.g., speech to text, image-to-text optical character recognition [OCR], and handwriting to text) are operations where the data is transformed from a representation of one abstract data type to a representation of another abstract data type. The presentation form transformations T↔T_Presentation_Form and the fundamental input and output services are data type transformations. Care needs to be taken when parameterizing these operations with user preferences to keep the transformation thread-safe. Again, this is best accomplished by keeping the presentation form preferences attached to the data.

Encoding Conversions

Encoding conversions (between encoded character sets, DIFs, etc.) are operations where only the representation of a single data type changes. For example, to support Unicode, a system must have for each other encoded character set a function to_uni:Strings(E)→Strings(Unicode), which converts the code points in E to code points in Unicode.[11] The conversion function to_uni has a partial inverse from_uni:Strings(Unicode)→Strings(E), which is only defined on those encoded text elements in Unicode that can be expressed as encoded text elements in E. If *s* is in Strings(E), then from_uni(to_uni(s)) is equal to *s*. Other encoding conversions Strings(E)→Strings(E') can be defined as a to_uni operation followed by a from_uni operation, for E and E' respectively. Another class of encoding conversions arises when the character set encoding remains fixed, but the conversion of a document in one DIF to a document in another DIF is required. A third class originates when Unicode or ISO 10646 strings sent over asynchronous communication channels must be converted to a Universal Transmission Format (UTF), thus requiring Strings(Unicode)↔UTF encoding conversions.

Collation or Sorting Services

Another group of computation services, collation or sorting services, sorts lists of strings according to application-specific requirements. These services were discussed earlier in the paper.

Linguistic Services

Linguistic services such as spell checking, grammar checking, word and line breaking, content-based retrieval, translation (when existent), and style checking need standard APIs. Although the implementation of these linguistic services is natural language-specific, most can be implemented with the structure shown in Figure 2.

Also, large character sets such as Unicode and other multilingual structures require a uniform exception-handling and fallback mechanism

because of the large number of unassigned code points. For example, a system should be able to uniformly handle exceptions such as "glyph not found for text element." Mechanisms such as global variables for error codes inhibit concurrent programming and therefore should be discouraged. Returning an error code as the return value of the procedure call is preferred, and when supported, raising and handling exceptions is even better.

System Naming, Synonyms, and Security

The multilingual aspect of Unicode can simplify system naming of objects and their attributes, e.g., in name services and repositories. Using encoded strings tagged with their encoding type for names is too rigid, because of the high degree of overlap in the various encoded character sets. For example, the string "ABC" should represent one name, independent of the character set in which the string is encoded. Two tagged strings represent the same name in the system if they have the same canonical form in Unicode according to the following definitions.

Unicode has the property that two different Unicode strings, u and v, may well represent the same sequence of glyphs when rendered.[11] To deal with this, a system can define an internal canonical form $c(u)$ for a Unicode string u. $c(u)$ would expand every combined character in u to its base characters followed by their assorted marking characters in some prescribed order. The recommended order is the Unicode "priority value."[11,21] The canonical form should have the following property: When $c(u)$ is equal to $c(v)$, the plain text representations of u and v are the same. Ideally, the converse should hold as well.

Thus, u and v represent the same name in the system if $c(u)$ is equal to $c(v)$. In any directory listing, an end user of a language sees only one name per object, independent of the language of the owner who named the object. Further restrictions on the strings used for names are desirable, e.g., the absence of special characters and trailing blanks. In a multivendor environment, both the canonical form and the name restrictions should be standardized. The X.500 working groups currently studying this problem plan to achieve comparable standardization.

Since well-chosen names convey useful information, and since such names are entered and displayed in the end user's writing system of choice, it is often desirable for the system to store various translations or "synonyms" for a name. Synonyms, for whatever purpose, should have attributes such as long_name, short_name, language, etc., so that directory functions can provide easy-to-use interfaces. Access to objects or attribute values through synonyms should be as efficient as access by means of the primary name.

In a global network, public key authentication using a replicated name service is recommended.[22] One principal can look up another in the name service by initially using a (possibly meaningless) name for the object in some common character set, e.g., {A-Z,0-9}. Subsequently, the principals can define their own synonyms in their respective languages. Attributes for the principals, such as network addresses and public encryption keys, can then be accessed through any synonym.

System Management and Configuration

The system management of a multilingual distributed system is somewhat more complicated than for a monolingual system. The following is a partial list of the services that must be provided:

- Services for various monolingual subsystems
- Registration services for user preferences, locales, user-defined text elements, formats, etc.
- Both multilingual and multiple monolingual run-time libraries, simultaneously (see Figure 2)
- Multilingual database servers, font servers, logging and queuing mechanisms, and directory services
- Multilingual synonym services
- Multilingual diagnostic services

Since a system cannot provide all the services for every possible situation, registering the end users' needs and the system's capabilities in a global name service is essential. The name service must be configured so that a multilingual server can identify the language preferences of the clients that request services. This configuration allows the servers to tag or convert data from the client without the monolingual client's active participation. Therefore, the name service database must be updated with the necessary preference data at client installation time.

Typically, system managers for different parts of the system are monolingual end users (see Figure 1) who need to do their job from a standard PC.

Thus, both the normal and the diagnostic management interfaces to the system must behave as multilingual servers, sending error codes back to the PC to be interpreted in the local language. Although the quality of the translation of an error message is not an architectural issue, translations at the system management level are generally poor, and the system design should account for this. Systems developers should consider giving both an English and a local-language error message as well as giving easy-to-use pointers into local-language reference manuals.

Data errors will occur more frequently because of the mixtures of character sets in the system, and attention to the identification of the location and error type is important. Logging to capture offending text and the operations that generated it is desirable.

Incremental Internationalization

Multilingual systems and international components can be built incrementally. Probably the most powerful approach is to provide the services to support multiple monolingual subsystems. Even new operating systems, such as the Windows NT system, that use Unicode internally need mechanisms for such support.[25] Multidimensional improvements in a system's ability to support an increasing number of variations are possible. Some such improvements are making more servers multilingual, supporting more multilingual data and end-user preferences, supporting more sophisticated text elements (the first release of the Windows NT operating system will not support Unicode's joiners), as well as adding more character set support, locales, and user-defined text elements. The key point is that, like safe programming practices, multilingual support in a distributed system is not an "all-or-nothing" endeavor.

Summary

Customer demand for multilingual distributed systems is increasing. Suppliers must provide systems without incurring the costs of expensive reengineering. This paper gives an overview of the architectural issues and programming practices associated with implementing these systems. Modularity both in systems and in run-time libraries allows greater reuse of components and incremental improvements with regard to internationalization. Using the suggested safe software practices can lower reengineering and maintenance costs and help avoid costly redesign problems. Providing multilingual services to monolingual subsystems permits incremental improvements while at the same time lowers costs through increased reuse. Finally, the registration of synonyms, user preferences, locales, and services in a global name service makes the system cohesive.

Acknowledgments

I wish to thank Bob Ayers (Adobe), Joseph Bosurgi (Univel), Asmus Freytag (Microsoft), Jim Gray (Digital), and Jan te Kiefte (Digital) for their helpful comments on earlier drafts. A special thanks to Digital's internationalization team, whose contributions are always understated. In addition, I would like to acknowledge the Unicode Technical Committee, whose impact on the industry is profound and growing; I have learned a great deal from following the work of this committee.

References

1. D. Carter, *Writing Localizable Software for the Macintosh* (Reading, MA: Addison-Wesley, 1991).

2. *Producing International Products* (Maynard, MA: Digital Equipment Corporation, 1989). This internal document is unavailable to external readers.

3. *Digital Guide to Developing International Software* (Burlington, MA: Digital Press, 1991).

4. S. Martin, "Internationalization Made Easy," OSF White Paper (Cambridge, MA: Open Software Foundation, Inc., 1991).

5. S. Snyder et al., "Internationalization in the OSF DCE—A Framework," May 1991. This document was an electronic mail message transmitted on the Internet.

6. *X/Open Portability Guide, Issue 3* (Reading, U.K.: X/Open Company Ltd., 1989).

7. *X/Open Internationalization Guide, Draft 4.3* (Reading, U.K.: X/Open Company Ltd., October 1990).

8. *UNIX System V Release 4, Multi-National Language Supplement (MNLS) Product Overview* (Japan: American Telephone and Telegraph, 1990).

9. *Information Technology—Universal Coded Character Set (UCS) Draft International Standard,* ISO/IEC 10646 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).

10. A. Nakanishi, *Writing Systems of the World,* third printing (Rutland, Vermont, and Tokyo, Japan: Charles E. Tuttle Company, 1988).

11. The Unicode Consortium, *The Unicode Standard—Worldwide Character Encoding,* Version 1.0, Volume 1 (Reading, MA: Addison-Wesley, 1991).

12. R. Haentjens, "The Ordering of Universal Character Strings," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 43–52.

13. *Programming Languages—C,* ISO/IEC 9899: 1990(E) (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).

14. S. Martin and M. Mori, *Internationalization in OSF/1 Release 1.1* (Cambridge, MA: Open Software Foundation, Inc., 1992).

15. J. Becker, "Multilingual Word Processing," *Scientific American,* vol. 251, no. 1 (July 1984): 96–107.

16. *Coded Character Sets for Text Communication, Parts 1 and 2,* ISO/IEC 6937 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1983).

17. J. Bettels and F. Bishop, "Unicode: A Universal Character Code," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 21–31.

18. Go Computer Corporation, "Compaction Techniques," *Second Unicode Implementors' Conference* (1992).

19. J. Becker, "Re: Updated [Problems with] Unbound (Open) Repertoire Paper" (January 18, 1991). This electronic mail message was sent to the Unicode mailing list.

20. V. Joloboff and W. McMahon, *X Window System, Version 11, Input Method Specification,* Public Review Draft (Cambridge, MA: Massachusetts Institute of Technology, 1990).

21. M. Davis, (Taligent) correspondence to the Unicode Technical Committee, 1992.

22. M. Gasser et al., "Digital Distributed Security Architecture" (Maynard, MA: Digital Equipment Corporation, 1988). This internal document is unavailable to external readers.

23. H. Custer, *Inside Windows NT* (Redmond, WA: Microsoft Press, 1992).

Michael M. T. Yau |

Supporting the Chinese, Japanese, and Korean Languages in the OpenVMS Operating System

The Asian language versions of the OpenVMS operating system allow Asian-speaking users to interact with the OpenVMS system in their native languages and provide a platform for developing Asian applications. Since the OpenVMS variants must be able to handle multibyte character sets, the requirements for the internal representation, input, and output differ considerably from those for the standard English version. A review of the Japanese, Chinese, and Korean writing systems and character set standards provides the context for a discussion of the features of the Asian OpenVMS variants. The localization approach adopted in developing these Asian variants was shaped by business and engineering constraints; issues related to this approach are presented.

The OpenVMS operating system was designed in an era when English was the only language supported in computer systems. The Digital Command Language (DCL) commands and utilities, system help and message texts, run-time libraries and system services, and names of system objects such as file names and user names all assume English text encoded in the 7-bit American Standard Code for Information Interchange (ASCII) character set.

As Digital's business began to expand into markets where common end users are non-English speaking, the requirement for the OpenVMS system to support languages other than English became inevitable. In contrast to the migration to support single-byte, 8-bit European characters, OpenVMS localization efforts to support the Asian languages, namely Japanese, Chinese, and Korean, must deal with a more complex issue, i.e., the handling of multibyte character sets. Requirements for the internal representation, input, and output of Asian text are radically different from those for English text. As a result, many traditional ASCII programming assumptions embedded in the OpenVMS system are not valid for handling Asian multibyte characters.

Since the early 1980s, Digital's engineering groups in Asia have been localizing the OpenVMS system to support Asian languages. The resultant Asian language extensions allow Asian-speaking users to interact with the OpenVMS system in their native languages. These extensions also provide a platform for developing Asian applications. This paper presents a high-level overview of the major features of Chinese, Japanese, and Korean support in the OpenVMS operating system and discusses the localization approach and techniques adopted.

Asian Language Variants of the OpenVMS System

The following five separate Asian language variants of the OpenVMS operating system are available in the Pacific Rim geographical area:

Language	Country	OpenVMS Variant
Japanese	Japan	OpenVMS/Japanese
Chinese	People's Republic of China	OpenVMS/Hanzi
Chinese	Taiwan, Republic of China	OpenVMS/Hanyu
Korean	Republic of Korea (South Korea)	OpenVMS/Hangul
Thai	Thailand	OpenVMS/Thai

This paper covers the first four variants, omitting the Thai variant because of space limitations. Each

Asian language variant of the OpenVMS system is designed to be installed and to run as a separate system. Currently, no provision exists to formally support multiple Asian languages simultaneously on a single OpenVMS system. Each variant provides a bilingual system environment of English and one Asian language. Such an environment, called Asian OpenVMS mode in this paper, supports ASCII and one multibyte Asian character set. The variants are available on the VAX and the Alpha AXP platforms with identical features. Throughout the paper, the generic name Asian OpenVMS variant denotes any of the Asian language variants of the OpenVMS operating system, regardless of the hardware platform.

To achieve full downward compatibility for existing users, applications, and data from the standard OpenVMS system, each Asian OpenVMS variant is a superset of the standard OpenVMS system. In fact, a user can operate in the standard OpenVMS mode, i.e., the 1-byte DEC Multinational Character Set (DEC MCS), on an Asian OpenVMS variant without noticing any difference in the functional behavior compared to a standard OpenVMS system. The components of an Asian OpenVMS variant are installed on a standard OpenVMS system in a manner similar to that of a layered product; files (executable images and other data files) are added and replaced on the standard OpenVMS system. In general, three types of components are supplied in an installation:

1. A standard OpenVMS component supplanted by an Asian localized version that includes the standard OpenVMS mode as a subset. At the process level, the user can set the component to run in either standard OpenVMS mode or Asian OpenVMS mode. The DCL and the terminal driver are examples of this type of component.
2. A standard OpenVMS component supplemented by an Asian localized version that runs only in Asian OpenVMS mode. Both versions of the component run simultaneously on the system. Examples are the TPU/EVE editor and the MAIL utility.
3. A new Asian-specific component created to provide functionality for Asian processing that does not exist in the standard OpenVMS system. An example of this type of component is the character manager (CMGR), which is discussed later in this paper.

Overview of Asian Writing Systems

Before looking at specific features of the Asian OpenVMS variants, this paper briefly reviews the Chinese, Japanese, and Korean writing systems. For a more detailed discussion of the differences among these writing systems, refer to Tim Greenwood's paper in this issue of the *Journal*.[1]

The Chinese Writing System

The Chinese writing system uses ideographic characters called Hanzi, which originated in ancient China more than 3,000 years ago. Each ideographic character (or ideogram) is a symbol made up of elementary root radicals that represent ideas and things. Some ideograms have very complex glyphs that consist of up to 30 brush strokes. Over 50,000 Chinese ideograms are known to exist today; however, a subset of 20,000 or less is typically sufficient for general use. Two or more ideograms are often strung together to represent more complex thoughts.

Ideographic writing systems have characteristics that are quite different from those of alphabetical writing systems, such as the Latin languages. For instance, the concept of uppercase and lowercase does not apply to ideographic characters, and collation rules are built on different attributes. The input of ideographic characters on a standard keyboard requires additional processing.

Two forms of Chinese characters are in use today: Traditional Chinese and Simplified Chinese. Traditional Chinese is the original written form and is still used in Taiwan and Hong Kong. In the 1940s, the government of the People's Republic of China (PRC) launched a campaign to simplify the writing of some traditional Chinese characters in an effort to speed up the learning process. The resulting simpler set of Chinese characters is known as Simplified Chinese and is used in the PRC, Singapore, and Hong Kong.

The Japanese Writing System

The Japanese writing system uses three scripts: Chinese ideographic characters (called *kanji* in Japan), *kana* (the native phonetic alphabet), and *romaji* (the English alphabet used for foreign words). The *kanji* script commonly used in Japanese includes about 7,000 characters. There are two sets of *kana* scripts, namely, *hiragana* and *katakana*; each comprises 52 characters that represent syllables in the Japanese language. *Hiragana* is used

extensively intermixed with *kanji*. *Katakana* is used to represent words borrowed from other languages.

The Korean Writing System

The Korean writing system uses two scripts: Hangul (the native phonetic characters) and Hanja (Chinese ideographic characters). The Hangul script was invented in 1443 by a group of scholars in response to a royal directive. Each Hangul character is a grouping of two to five Hangul letters (phonemes) that forms a square cluster and represents a syllable in the Korean language. The modern Hangul alphabet contains 24 basic letters—14 consonants and 10 vowels. Extended letters are derived by doubling or combining the basic letters.

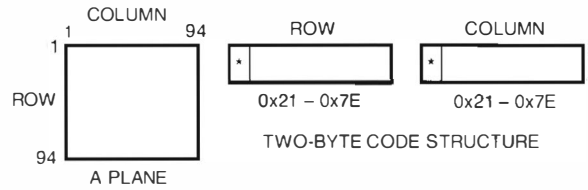
Asian Character Sets

During the early days of Asian language computerization when de jure standards did not exist for Asian character sets, individual vendors in the local countries invented their own local character sets for use in their Asian language products. Although most vendors have migrated to conform with the national standards, a variety of local character sets still exists today in legacy systems, thus creating interoperability issues. This paper reviews only the national standard character sets that are supported by the Asian OpenVMS variants.

National Standards

National standards bodies in each of the Asian Pacific geographies have established character set standards to facilitate information interchange for their local characters. For languages that use Han characters (which are large in number) in their writing scripts, the character set standards all share a similar structure, which is illustrated in Figure 1. Characters are assigned to a 94-row by 94-column structure called a plane. Each character is represented by a 2-byte (7-bit) value in the range of 0x21 to 0x7E. A plane, therefore, has a total of 8,836 code points available. Such a structure avoids the ASCII control code values, thus preventing conflicts with existing operating systems and communication hardware and software.

Japan Japan was the first country to announce a 2-byte character set standard, the Code of the Japanese Graphic Character Set for Information Interchange (JIS C 6226-1978).[2] This standard has since been revised twice, in 1983 and 1990, and renamed JIS X 0208. The JIS X 0208-1983 standard



* Note that the first bit of each row and column can be either 0 or 1.

Figure 1 Code Structure in Asian Character Set Standards

includes 6,353 *kanji* characters divided into two levels, according to frequency of usage.[3] Level 1 has 2,965 characters, and level 2 has an additional 3,388 characters. This standard also includes complete sets of characters for *hiragana* and *katakana*, ASCII, and the Greek and Russian scripts—a total of 453 characters. The 1990 revision, JIS X 0208-1990, added two characters to the standard.[4] An additional plane of *kanji* characters became standard in 1990 with the announcement of JIS X 0212-1990.[5]

Prior to the introduction of the 2-byte standards, Japanese systems that support *katakana* used the JIS X 0201-1976 standard for a 1-byte, 8-bit character set.[6] Today, there is still a demand to support this standard, in addition to the 2-byte standards, due to its pervasive use primarily in legacy mainframe systems.

People's Republic of China In 1980, China announced a 2-byte standard, Chinese Character Coded Character Set for Information Interchange—Basic Set (GB 2312-1980).[7] Its structure, which follows that of the Japanese standard, includes two levels of Hanzi. Level 1 has 3,755 characters, and level 2 has an additional 3,008 characters. The standard also has 682 characters, including ASCII, Greek, Russian, and the Japanese *kana* characters. Subsequently, China has announced additional character set standards.

Taiwan, Republic of China The Taiwanese national standard, Standard Interchange Code for Generally Used Chinese Characters (CNS 11643-1986) was first announced in 1986.[8] Again, the structure is similar to the Japanese and PRC standards. It defines two planes of characters with a total of 13,051 Hanzi, 651 symbols, and 33 control characters. The standard was revised in 1992 and renamed Chinese Standard Interchange Code (CNS 11643-1992).[9] An additional five planes were defined in this revision, adding 34,976 characters.

Republic of Korea (South Korea) The latest version of the Korean 2-byte character set standard is the Korean Industrial Standards Association Code for Information Interchange (KS C 5601-1987), announced in 1987.[10] This standard includes 2,350 precomposed Hangul characters, 4,888 Hanja (Chinese characters), and 352 other characters such as ASCII, the Hangul alphabets, Japanese *kana*, Greek, Russian, and special symbols.

User-defined Characters

Character set standards do not always encode all known characters of the writing scripts for which the standards are intended. For instance, when the total number of known characters exceeds the available code space, only subsets of the most common characters are included. In addition, new characters are invented over time to describe new ideas or objects, such as new chemical elements. The concept of user-defined characters (UDCs), sometimes known as *gaiji* in Japan, was introduced to address the user's need for characters that are not coded in a character set standard. Many computer vendors, including Digital, provide extended code areas for assigning UDCs and vendor-defined nonstandard characters. Attributes of these characters for various operations such as display fonts, collation weights, and input key sequence are often made available, e.g., by registering them in a system database. From an end-user and application perspective, UDCs should be processed transparently and in the same way as standard characters.

Asian OpenVMS System Overview

From an operating system perspective, three basic issues need to be addressed to support Asian character processing, namely, internal representation, (i.e., how Asian characters are represented and stored inside the computer), basic text input, and output.

Internal Representation

Asian OpenVMS variants support the respective national standard character sets. To achieve full compatibility with existing ASCII data, each Asian OpenVMS variant simultaneously supports one multibyte Asian character set and ASCII. A variety of schemes can be used to combine multiple coded character sets. In general, the schemes fall into one of the following three types:

1. Shift code–based representation. In this scheme, the 1-byte code is combined with a 2-byte code by inserting shift control codes to switch between the two code sets. A 1-byte "shift out" control code changes the mode from 1- to 2-byte, while a 1-byte "shift in" control changes the mode back to 1-byte characters. This scheme is in common use in mainframes.
2. ISO 2022–based representation. The ISO 2022 Code Extension Techniques allow a designated character set to consist of two, three, or four 7-bit bytes in addition to the 7-bit sets.[11] The only requirement is that all bytes of a character have the same high-order bit setting (all 0 or all 1). A simple scheme of simultaneously supporting ASCII and one 2-byte character set can be achieved by statically designating ASCII to G0 and invoking it to graphics left (GL) and designating a local 2-byte set (e.g., one of the Chinese, Japanese, or Korean sets) to G1 and invoking it to graphics right (GR). The resulting mixed 1-byte/2-byte representation is shown in Figure 2.

   The high-order bit of each 8-bit byte provides self-identifying information for the local 2-byte set. This scheme can be further extended to include two additional character sets by statically designating them to G2 and G3 and invoking them by the single shift codes SS2 and SS3. The Extended UNIX Code (EUC) scheme employs this additional extension.
3. Shift range–based representation. This scheme, a hybrid of the previous two schemes, is used by the "Shift JIS Code" on PC-based systems in Japan. Bytes with codes 0 to 127 are interpreted as 1-byte ASCII, codes 160 to 191 and 192 to 223 are interpreted as 1-byte *katakana* (as specified by the JIS X 0201 standard), and codes 128 to 159 and 224 to 255 are combined with the byte that

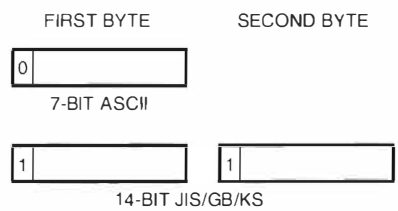


Figure 2 Example of an ISO 2022-based Representation That Combines Multiple Coded Character Sets

follows to form a 2-byte code that is interpreted as a *kanji* character (as specified by the JIS X 0208 standard). This scheme allows more single-byte characters to be represented at the expense of the number of 2-byte characters allowed.

Asian OpenVMS variants employ the ISO 2022-based representation for Digital's Asian code sets (the DEC Asian code sets) and are named respectively DEC Kanji, DEC Hanzi, DEC Hanyu, and DEC Hangul for the Japanese, Simplified Chinese, Traditional Chinese, and Korean character sets. This encoding scheme maintains full downward compatibility with all existing ASCII software and data. In particular, a string or record that consists of only ASCII characters has the form of simple ASCII. Because there is no need to keep state information about the data, this scheme simplifies processing, when compared to the shift code-based scheme. However, without explicit support for coded character set designation, simultaneous support for Chinese, Japanese, and Korean is not possible.

To support UDCs, each DEC Asian code set contains an extended code area for their assignment. The high-order bit of the second byte no longer has to be set; thus, an additional 94 by 94 plane of code positions is available. The disadvantages of this approach are that synchronizing a character boundary requires scanning forward from the beginning of the string and that the second byte can now conflict with the ASCII characters.

The DEC Asian code set internal representation corresponds to mapping a character plane (94 by 94) to one of the (1,1) and (0,1) quadrants of the 2-byte code space in Figure 3. The exact mappings of individual DEC Asian character sets supported by Asian OpenVMS vary. Table 1 provides a summary of the common code range assignments.

DEC Kanji The DEC Kanji (OpenVMS/Japanese) code set currently supports ASCII, JIS X 0208-1983, and an area for UDCs, as shown in Table 1. The UDC area is further divided into the two subareas described in Table 2.

Recently, Super DEC Kanji, a revision and extension to the DEC Kanji code set, has been proposed

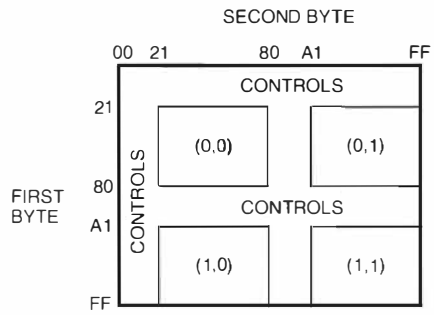


Figure 3 DEC Asian Code Set Internal Representation

Table 1 Summary of DEC Asian Code Range Assignments

Code Range	DEC Kanji	DEC Hanzi	DEC Hanyu	DEC Hangul
(0xxxxxxx)	ASCII	ASCII	ASCII	ASCII
(1xxxxxxx 1xxxxxxx)	JIS X 0208	GB 2312-1980	CNS 11643-1986(1)*	KS C 5601-1987
(1xxxxxxx 0xxxxxxx)	UDC	UDC	CNS 11643-1986(2)†	—
(000xxxx)	C0 Control	C0 Control	C0 Control	C0 Control
(100xxxx)	C1 Control	C1 Control	C1 Control	C1 Control

Notes:

* denotes plane 1 of CNS-11643-1986.

† denotes plane 2 of CNS-11643-1986.

Table 2 The DEC Kanji UDC Area

Area Usage	Quadrant	Rows	Number of Characters	Code Range
User Area	(1,0)	1–31	2,914	0xA121–0xBF7E
DEC Reserved	(1,0)	32–94	5,922	0xC021–0xFE7E

to support additional character sets such as JIS X 0201-1978 and JIS X 0212-1990, which are specified as follows:

Code Range	Additional Planes
(SS2 1xxxxxxx)	JIS X 0201
(SS3 1xxxxxxx 1xxxxxxx)	JIS X 0212

The redefined UDC area includes both a user/vendor-defined area (UDA) and a user-definable character set (UDCS), as described in Table 3.

DEC Hanzi The DEC Hanzi (OpenVMS/Hanzi for Simplified Chinese) code set supports ASCII, GB 2312-80, and a UDC area described in Table 4.

DEC Hanyu The DEC Hanyu (OpenVMS/Hanyu for Traditional Chinese) code set currently supports ASCII, CNS 11643-1986 (first and second planes), and the Digital Taiwan Supplemental Character Set (DTSCS). The DTSCS supplements the characters defined in CNS 11643-1986 with an additional collection of characters that address customer needs. Currently, the DTSCS defines the 6,319 characters recommended by the Electronic Data Processing Center (EDPC) of the Executive Yuan, a Taiwanese government body. The CNS 11643-1992 standard includes the DTSCS.

To support the additional DTSCS, the mixed 1-byte/2-byte scheme is extended to a 1-byte/2-byte/4-byte scheme. Each DTSCS character maps to a 4-byte code, in which a fixed leading 2-byte code (0xC2CB) combines with the following 2-byte code to form a 4-byte code. Of course, the code point 0xC2CB is reserved for this purpose. This scheme makes available another two 94 by 94 planes of code positions:

(0xC2CB 1xxxxxxx 1xxxxxx)
(0xC2CB 1xxxxxxx 0xxxxxx)

Table 5 shows the current definition of the DTSCS. An additional area is available for UDCs in the CNS planes, as defined in Table 6.

DEC Hangul The DEC Hangul (OpenVMS/Hangul) code set supports ASCII and KS C 5601-1987 (with the exception of UDCs).

Asian Text Input

The most widely used computer input device remains the keyboard. Because it is impossible to assign thousands of ideographic characters to a standard QWERTY keyboard, new methods must be devised to facilitate the Asian text input process. In this context, an input method is basically an algorithm that takes keystroke input representing certain attributes (e.g., phonetics) of a character or string and uses a lookup table to find characters or strings that have those attribute values. Typically, a user inputs several keystrokes and selects the desired character or string from a candidate list by means of an iterative dialog with the input method. This process is sometimes referred to as preediting. Depending on the physical location of where the dialog takes place, a preediting user interface can be one of three styles: off-the-spot, over-the-spot,

Table 3 The Super DEC Kanji UDC Area

Area Usage	Quadrant	Rows	Number of Characters	Code Range
JIS X 0208 UDA	(1,1)	85–94	940	0xF5A1–0xFEFE
JIS X 0212 UDA	SS3 (1,1)	78–94	1,598	(SS3 + 0xEEA1)–0xFEFE
UDCS	(1,0)	1–94	8,836	0xA121–0xFE7E

Table 4 The DEC Hanzi UDC Area

Area Usage	Quadrant	Rows	Number of Characters	Code Range
DEC Reserved	(1,1)	88–94	658	0xA1A1–0xFEFE
User Area	(1,0)	1–87	8,178	0xA121–0xF77E
DEC Area	(1,0)	88–94	658	0xF821–0xFE7E

Table 5 The DEC Hanyu DTSCS Area

Area Usage	Quadrant	Rows	Number of Characters	Code Range
EDPC Recommended Characters	C2CB (1,1)	1–68	6,319	0xC2CBA1A1–0xC2CBE4B5
Reserved	C2CB (1,1)	68–94	2,517	0xC2CBBEB6–0xC2CBFEFE
Reserved	C2CB (1,0)	1–94	8,836	0xC2CBA121–0xC2CBFE7E

Table 6 The DEC Hanyu UDC Area

Area Usage	Quadrant	Rows	Number of Characters	Code Range
UDC	(1,1)	93–94	145	0xFDCC–0xFEFE
UDC	(1,0)	82–94	1,186	0xF245–0xFE7E

or on-the-spot. Different input methods may have different preedit interface requirements. Usually, several screen areas are needed for the preediting dialog to take place. Input methods differ from culture to culture and from script to script.

The major difference in the implementation of input method support among the Asian OpenVMS variants is in the character cell terminal environment. Some input methods are suitable for programming into the terminal firmware. The Chinese and Korean input methods supported on the OpenVMS/Hanzi, OpenVMS/Hanyu, and OpenVMS/Hangul systems are examples of such methods. Other input methods are too complex or require so many resources as to make them too costly for firmware implementation. This is true of the Japanese input method, which needs to be implemented on the host. Such implementation causes a number of technical issues with the traditional ASCII character cell terminal-oriented application programming model, where an application does not have to be concerned with input methods and expects to receive character codes directly. The following three alternatives have been used to implement host-based input methods on the OpenVMS/Japanese system:

1. Application. All Japanese applications directly program the input method themselves. An application must call low-level routines (a set of *kana-kanji* input method routines are available in the JSY Run-time Library) to access the input method dictionary and directly controls the preedit interface in relation to its own screen management. This method is used by applications such as text editors, which need to directly manage the screen display. The method requires substantial reengineering of an ASCII application to support a Japanese input method.
2. Run-time library (RTL). Japanese applications call special text line input routines, which handle the Japanese input method. This method is suitable for applications that require simple line input of text. The RTL method hides the details of the input method from the application but lacks the flexibility to control the preedit user interface. The reengineering needed to handle the Japanese input method is shifted from the application to the RTL routines. This approach requires less application reengineering, but all standard line input routine calls in the application must be replaced by Japanese line input routine calls.
3. Front-end input processor (FIP). The Japanese input method is embedded as a front-end process inside the terminal queued I/O (QIO) system service. This method of implementation benefits existing high-level RTL text line input routines and requires little application or RTL reengineering to support the Japanese input method in the single-line input of Japanese text.[12]

The Asian OpenVMS graphical user interface on workstations is called Asian DECwindows/Motif. Current input method support is provided through a Digital extension implemented as an X client. With release 5 of the X11 standard, the implementation will migrate to using the standard X input method (XIM) support in the Xlib library routines.

Most Asian PCs have a front-end processor implementation of input methods resident on the PC. Therefore, PC desktop computers can send Asian characters directly when communicating with an Asian OpenVMS host.

The following is an overview of the input methods supported by each Asian OpenVMS variant.

Japanese Input Method *Kana* characters can be typed directly on a standard keyboard using a ***kana*** keyboard layout. For ***kanji*** characters, the de facto standard input method is called the ***romaji/kana***-to-***kanji*** conversion, which is based on phonetic conversion. The process of entering a ***kanji*** string involves typing the ***kana (hiragana*** or ***katakana)*** or the ***romaji*** pronunciation of the string. The input method then looks in a conversion dictionary for the list of ***kanji*** strings that have the same pronunciation. Since most Japanese words have homonyms, the user usually needs to go through a selection process to find the desired ***kanji*** string. More advanced implementations involve performing syntactic and semantic analysis of the sentence to increase the efficiency of the input method. On the OpenVMS/Japanese system, the ***kana***-to-***kanji*** input method has a provision for separating conversion units into word, clause, and sentence. The method also has a learning capability that reorders the candidate list entries by means of a personal dictionary, putting the characters selected at the top of the list so that more frequently used words appear first in the homonym list.

Chinese Input Method No standard exists for the Chinese input method. The large number of input methods that have been proposed over the years can be classified into one of two major types:

1. Pattern decomposition-based method. Each character is decomposed into basic strokes or patterns. Each stroke or pattern, e.g., a root radical, is assigned a code (mapped to a key) and each character is retrieved by inputting a sequence of such codes.
2. Phonetic-based method. Each character is transcribed into phonemic letters and retrieved by this phonemic transcription. The system used in Taiwan is based on the National Phonetic Alphabet (Bopomofo), whereas the PRC uses Roman alphabets based on the Wade-Giles system.

The OpenVMS/Hanzi system supports the following Chinese input methods:

- Five stroke
- Five shape
- Pinyin
- Telex code
- GB 2312 code

The OpenVMS/Hanyu system supports the following Chinese input methods, which are implemented by firmware on the Digital VT382 series Chinese terminals:

- Tsang-Chi
- Quick Tsang-Chi
- Phonetic
- Internal code
- Phrase

Korean Input Method Hangul characters are composed by directly typing the individual Hangul letters. The composition sequence always starts with a consonant, is followed by a vowel, and finishes with a consonant, if present. The input method validates the composition sequence keyed in by the user at each step. The display device updates the intermediate rendering of the partially formed Hangul character as the shape and position of each letter changes during composition. Hanja characters are entered by typing their Hangul pronunciation. The input method displays a list of all possible Hanja characters (homonyms). More sophisticated implementations can perform Hangul-to-Hanja conversion in word units similar to that of the ***kana***-to-***kanji*** conversion. On the Digital VT382 Korean terminal, both the Hangul and the Hanja input methods are implemented by firmware.

Asian Text Output

Asian character fonts are usually displayed or printed as bit-map graphics. To meet the requirements of specific applications such as scaling and plotting, these fonts can also be defined as outline fonts using vector representation. International codes of Asian language characters are mapped to the corresponding font data when needed for output. Predefined character fonts are usually stored in the read-only memory (ROM) of terminals and printers for better performance. As for the English alphabet, different standards, styles, and sizes exist for Asian language character fonts. The following list

contains some of the more popular font styles used in the respective markets:

Market	Font Style
Japan	Mincho, Gothic, Round-Gothic
Korea	Myuncho, Gothic
PRC	Song, Quasi-Song, Hei (boldface), Kai
Taiwan	Sung, Hei (boldface), Kai

In general, Asian ideographic characters require high-definition fonts, i.e., at least a 24-by-24 dot matrix, to achieve acceptable visual quality. As a result, memory requirement is a major issue when supporting Asian fonts.

Hardware

Supporting Asian language processing requires modifying the standard video terminals and printers. In general, software products need to recognize the different functional characteristics of Asian terminals and printers. For example, the character set designation and invocation defaults differ from those of standard terminals.

Workstations do not require any modifications (except for exchanging a local language keyboard for the standard one), because input and display are directly supported by software.

Asian Video Terminals The traditional character cell terminal provides certain local display and input functions on behalf of a software program. For example, the terminal firmware preprocesses scan codes generated by keyboard input and converts them to character code before sending them to an application. Similarly, character fonts are usually stored in the terminal ROM. Digital has developed a variety of video terminals to support Asian language processing.

Some major hardware considerations for Asian video terminals are

- High-resolution video display. Ideographic characters have complex glyphs, which require at least a 24-by-24 dot matrix cell to be of acceptable display quality. Such a cell would occupy two ASCII columns. As a result, to maintain a 26-line (40 ideograms per line) display requires a screen resolution of at least 960 by 780 pixels. Typically, Digital's Asian video terminals use monitors that run at a 60-hertz noninterlaced mode, a mode substantially higher than that of standard ASCII terminal monitors.
- Font memories and loading protocol. The terminal requires additional ROM to hold the fonts of standard characters in an Asian character set, typically 7,000 to 20,000 characters. Also, for characters outside the standard set, i.e., UDCs, the terminal requires random-access memory (RAM) to downline load the fonts from the host. Digital Asian terminals support font-loading protocols that work with the host software to downline load fonts into RAM either on demand or on a preloading basis. The font cache in Digital's Asian terminals can usually hold about 400 characters at once.
- Input method. Implementing input methods on a video terminal requires additional hardware modification. The input method algorithms must be programmed into the firmware together with extra memory for the input method lookup tables. In addition to the main display area, one extra line on the screen is needed as an input method work area, e.g., for displaying candidate lists for user selection. Some keys must be assigned permanently for invoking different input methods. The printing of legends on the tops of the keys is now more complex, because the keytops must include additional legends for the input method keyboard layout. For example, on Digital's Hanzi terminals, four ideograms must appear on the tiny area of one keytop.

Asian Printers Digital supports a range of Asian printers. Similar to Asian video terminals, Asian printers must support font-loading protocols to downline load fonts for UDCs by either preloading or on-demand-loading methods. Additional RAM is required to hold these fonts. Also, Digital's Asian printers generally support multiple font typefaces and sizes.

Asian OpenVMS Structure

The components provided by the Asian OpenVMS variants on top of the standard OpenVMS system can be divided into five main groups:

1. System support for transparent processing of UDCs
2. An enhanced OpenVMS terminal I/O subsystem to support Asian terminal devices
3. A set of run-time libraries to facilitate Asian application development on Asian OpenVMS systems

4. A set of localized utilities and commands for users to perform common tasks on OpenVMS systems in their native languages
5. A utility to set the operating modes (standard OpenVMS mode or Asian OpenVMS mode) of the localized components

Figure 4 summarizes the Asian OpenVMS system structure.

Asian OpenVMS Components

This section reviews the major components of the Asian OpenVMS variants.

User-defined Character Support—The Character Manager

Attributes of characters in the standard character sets supported on an Asian OpenVMS system are known and fixed. Therefore, attribute support can be built into the system statically. In contrast, UDCs usually require their attributes to be dynamically defined and accessed. A new utility called the character manager (CMGR) enables users to create, manage (modify and update), and retrieve UDCs and their attributes. UDC support is currently offered on the OpenVMS/Japanese, OpenVMS/Hanzi, and OpenVMS/Hanyu systems. In the OpenVMS/Hanyu system, the CMGR also supports Digital-defined characters, e.g., the DTSCS and DEC Recommended Characters (DRC).

The CMGR manages a set of systemwide databases that store UDC attributes. Two UDC attributes are currently supported, glyph images and collating values.

To represent the UDCs in the computer, the CMGR allows a user to assign each UDC a code point in the designated UDC area. Currently, UDC characters are entered by directly typing their binary code. The code point serves as the key in the CMGR databases for retrieving other attributes of the character.

The CMGR utility provides a user interface to create and manage the UDC attribute database. The user interface includes a font editor for users to create the glyph image of a UDC and entries for other attributes. To allow applications to retrieve the UDC attributes, the CMGR has a set of application programming interfaces (APIs) used to access the individual attribute databases. In particular, the on-demand font loading of UDCs supported by the Asian terminal I/O subsystem employs the CMGR font databases, and the SORT/MERGE utility uses the collation databases for UDC sorting.

CMGR Font Database To output a UDC to a display or printing device, the UDC's glyph image must first be defined. The CMGR provides a screen font editor for users to create the glyph images. The CMGR supports multiple typefaces (e.g., Hei, Sug, and Default) and font sizes (e.g., 24 by 24, 32 by 32, and 40 by 40) in multiple databases. There are two ways to load the UDC fonts to Asian output devices, namely, preloading and on-demand loading.

Fonts can be preloaded by sending a file that contains the appropriate control sequences and font patterns, which are discussed in more detail later in this section. The CMGR provides a command that generates a preload file from the font database for required UDCs.

On-demand font loading is a more complicated mechanism, which involves an on-demand loading protocol. Font patterns are retrieved from the font database through the CMGR callable interface by a font-handling process.

CMGR Collation Attribute Database To facilitate the sorting of data, including UDCs, the collation weights of the characters must be defined.

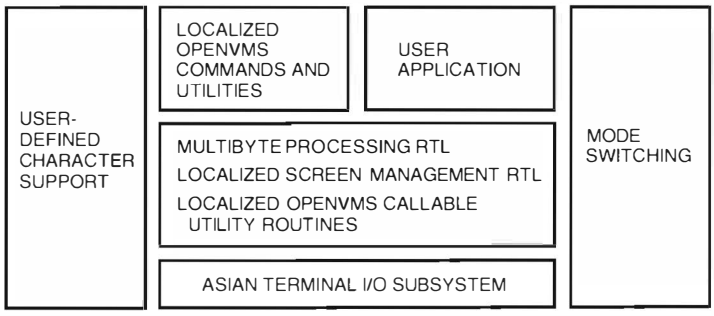


Figure 4 Asian OpenVMS System Structure

Currently, only the OpenVMS/Hanzi and OpenVMS/Hanyu systems offer this feature.

Asian Terminal I/O Subsystem

The Asian terminal I/O subsystem is an extension of the standard OpenVMS terminal I/O subsystem. It consists mainly of the OpenVMS terminal class drivers/port drivers, auxiliary class drivers, and server processes, and handles both standard and Asian terminals simultaneously. For Asian terminals, the subsystem provides extended functions to support multibyte character handling in the terminal QIO system service, input method, code set conversion, and font loading.

Terminal QIO System Service/Multibyte Character Handling The enhanced terminal QIO system service can handle mixed ASCII and multibyte Asian characters in line input calls. Line editing (e.g., character echo, cursor movement, character deletion, character insertion, word delimiters, and character overstrike), line wrapping, uppercasing, and read verifying will handle Asian characters correctly. Because the QIO system service is the lowest-level routine that handles terminal I/O, all other text I/O routines such as LIB$GET_INPUT, $GET RMS service, and the text I/O facility of programming languages such as C, Fortran, and COBOL are layered on it. The enhancements automatically benefit all of these higher-level routines.

Font Loading Asian terminal devices have writable font memory (WFM), and the firmware supports font-loading sequences and logic. A text file is scanned by a utility program prior to output to a terminal or printer. The Asian terminal I/O subsystem then creates a preloading file, which contains the font-loading sequence for all nonresident characters found in the file. Next, the subsystem sends this preloading file to the terminal or printer, causing the required fonts to be loaded in the font memory. Finally, the text file is output to the terminal or printer. This method is limited by the size of the font memory, typically 300 to 500 characters. The font preloading method is used mainly in batch operations, such as line printers, where performance is an important factor.

When an Asian video terminal or printer receives an Asian character code and determines that it is a UDC, the terminal firmware automatically halts the current processing and generates a font request to the OpenVMS system. The terminal driver traps this request and passes it on to a process called the font handler. On behalf of the terminal, the font handler retrieves the font bit map of the requested character from the system font database and sends it back to the terminal or printer, which in turn loads it into its RAM and resumes the display processing. Because it involves XON/XOFF flow control, which is done at a very low level of the system, the process requires modifications to device drivers. The amount of UDC font is not limited by WFM capacity, because the terminal firmware automatically updates the memory.

Front-end Input Process (FIP)[12] One of the biggest differences between Japanese and other Asian language (e.g., Chinese and Korean) support on the OpenVMS system is in the implementation of the input method. The nature of the *kana*-to-*kanji* input method makes it unsuitable for implementation in terminal firmware. The method requires a huge input method dictionary (about 1 megabyte in size) and a dynamic memory work area for syntactic and semantic analysis. Also, updating an input method dictionary that is implemented in firmware is a very costly operation.

Code Set Conversion Prior to the introduction of the Asian OpenVMS variants, Digital's customers used video terminals and printers that support proprietary local language code sets from third-party vendors. To protect customer investments and to ensure a smooth migration path for legacy equipment, the Asian terminal I/O subsystem provides an application-transparent, code set conversion facility. This facility is based on the terminal fallback facility (TFF) introduced in OpenVMS version 5.0, which provides a similar function for conversion between 7-bit National Replacement Character Sets (NRCSs) and the 8-bit DEC MCS. TFF provides a mid-driver that converts both incoming and outgoing data from one code set to another. For the Asian OpenVMS variants, the conversion logic is extended to support 16-bit character entities. Currently, TFF supports the conversion between the DOOSAN code and the DEC Hangul code on the OpenVMS/Hangul system and the MITAC TELEX code and the DEC Hanyu code on the OpenVMS/Hanyu system.

In addition, code set conversion is necessary between heterogeneous systems because of the proliferation of encoding schemes used by different vendors. For instance, Chinese PCs in Taiwan use the BIG 5 code. To facilitate the communication

between the OpenVMS system and PC desktop computers, the OpenVMS/Hanyu system supports the conversion between the BIG 5 code and the DEC Hanyu code.

Asian Application Programming Support

To help software developers write Asian applications on Asian OpenVMS variants, Digital provides a set of common Asian multibyte character processing RTL routines to supplement the standard OpenVMS RTLs. In particular, our Asian localization effort to develop OpenVMS layered products utilizes these RTLs. Functions provided by the Asian language RTL (approximately 240 routines) are classified into the following categories of routines:

- Character conversion
- String
- Read/write
- Pointer
- Comparison
- Search
- Count
- Character type
- Date/time
- Code set conversion

The majority of the routine interfaces are common to all Asian countries. Currently, one library image supports the Hanzi, Hanyu, and Hangul language variants. Language-specific code is hidden under this generic multibyte interface and switched at run time by a system logical name defined during system start-up.

The OpenVMS/Japanese system has a set of routines for handling *kana*-to-*kanji* conversion, both high level and low level. The high-level routines, such as JLBGET_INPUT, JLBGET_COMMAND, and JLB$GET_SCREEN (Japanese versions of LIBGET_INPUT, LIBGET_COMMAND, and LIB$GET_SCREEN), hide the *kana*-to-*kanji* input method details from the application. These routines use the off-the-spot preediting that usually takes place at the last line of the screen; however, the flexibility of the preedit user interface is limited. A set of low-level routines performs primitive functions such as opening the conversion dictionary, finding the next candidate *kanji* string, and getting the contents of the internal buffer. The *kana*-to-*kanji* input method is programmed by calling a sequence of these routines. This implementation gives the application the ability to directly control the screen management and allows flexibility in the design of the preedit user interface; however, the application must deal with every detail of the input method, which is a disadvantage. In addition, the library IMLIB helps the application customize the keyboard mapping for *kana*-to-*kanji* conversion.[12]

The screen management (SMG) RTL on the OpenVMS system provides a suite of routines for designing, composing, and keeping track of complex images on a character cell video terminal in a device-independent manner. The standard SMG version supports only the ASCII and DEC Special Graphics character sets and cannot correctly handle multibyte Asian characters. For example, operations such as screen update optimization, boundary processing (clipping on borders), and cursor movements operate on part of a multibyte Asian character and cause screen corruption because of the "one-character-is-equal-to-one-byte" assumption.

The Asian OpenVMS variants provide an extended version of SMG (about 20 percent of the original routines have been extended) to support multibyte character sets and DEC MCS, in addition to ASCII and DEC Special Graphics. To maintain downward compatibility, most routine entries remain identical, with an optional character set argument added at the end of the argument list to indicate desired character set operations. Alternatively, users can define a logical name SMG$DEFAULT_CHARACTER_SET without explicitly passing the character set argument in the routine call. Existing ASCII applications run unmodified with the Asian SMG. New Asian applications that use multibyte features relink with the new library.

Asian Commands and Utilities

The OpenVMS user interface determines the way an end user interacts with the system. The interface includes such components as the DCL command line interpreter, system help and messages, and all the system utilities provided by the OpenVMS system. Selected user interface components of the OpenVMS system have been localized to support Asian characters on the Asian OpenVMS variants. A description of some of these localized components follows.

DCL Command Line Interpreter The algorithms in the standard DCL that assume characters to be equal to 1 byte and interpret these characters as ASCII/DEC MCS are enhanced for the following DCL primitives in the Asian code set modes:

- Command parsing. Parsing of command input in single-byte units causes data corruption, because part of some multibyte Asian characters can be mistaken for one of the special DCL ASCII characters such as !, @, or ". Command parsing is now done in character units instead of byte units, and operations such as terminator, delimiter checks, and quotation mark compression are skipped on Asian characters, since the DCL special characters are all in ASCII.
- Character uppercasing and lowercasing. Uppercasing and lowercasing are applied only to ASCII characters, because the concept of uppercase/lowercase does not exist in Asian character sets. Uppercasing/lowercasing in single-byte units corrupts Asian character data, because part of an Asian character can be indiscriminately uppercased/lowercased.
- Symbols and labels. Certain 8-bit values (those with no character assigned in the DEC MCS) are currently disallowed for DCL symbol names, symbol values, and labels. This restriction has been removed in the Asian modes to allow all Asian characters in DCL symbols and labels. The enhanced algorithms maintain separate symbol tables for each of the code set modes, because of the possibility of code collision issues across different code sets.

The Asian DCL command line interpreter is currently supplied with the OpenVMS/Hanzi, OpenVMS/Hangul, OpenVMS/Hanyu, and OpenVMS/Thai systems in the same binary image, i.e., a single image supports multiple code sets. The default code set mode for DCL for a particular system is established during system start-up by means of a defined logical name supplied with the start-up procedure of each Asian OpenVMS variant. Switching the code set mode between DEC MCS and the particular Asian code set of the system is accomplished through a utility, e.g., HANZIGEN in the OpenVMS/Hanzi system. The Asian DCL is not supplied with the OpenVMS/Japanese system, because until only recently the Japanese input method was not available at the DCL level.

System Help and Messages The OpenVMS/Hanzi, OpenVMS/Hanyu, and OpenVMS/Hangul systems include a translated Asian language version of the OpenVMS system help library (accessed by typing HELP at the $ prompt). The Asian version of the system help library is placed in a directory that is separate from the original English one but that has the same file name. The user can switch the language (English or the particular Asian language) of system help by using the ASIANGEN utility, which redefines the file specification logical to point to the appropriate file.

The OpenVMS/Japanese system provides a translated Japanese version of the system messages (SYSMSG.EXE), which is placed in a subdirectory of SYS$MESSAGE. Users can switch the language of the system messages by using the SET LANGUAGE command, which reloads the message file into memory.

In addition, most of the localized original utilities and Asian-specific utilities provide bilingual help and messages.

SORT/MERGE Collation rules in the Asian languages are very different from those of the Latin languages.[13]

- Asian collation sequences. An Asian character has different collation sequences based on different attributes. The SORT/MERGE command is extended as follows to include new subqualifiers for the Asian collating sequences: /KEY=(POS:m, CSIZE:n, <collating sequence subqualifier>). The Asian OpenVMS SORT/MERGE utility supports the Asian collating sequences shown in Table 7.
- Collation weights. Unlike ASCII, the collation weights of the Asian collating sequences cannot be derived by virtue of the code value. Rather, the string comparison for Asian collation sequences are driven by collation weight tables. For the standard characters, these tables are built into binary images that are linked with the utility for fast access.
- Multibyte characters. String comparison in the original SORT/MERGE operation is done in byte units, because a character is assumed to be equal to 1 byte. For the Asian SORT/MERGE, a comparison operation must be aligned by character, i.e., multibyte, units rather than by byte units. The operation must be able to handle the case in which the start position of a sort key (specified by a byte position) in a record is in the middle of

Table 7 Asian Collating Sequences Supported by the OpenVMS User Interface

Collation Sequence Type	OpenVMS/Japanese	OpenVMS/Hanzi	OpenVMS/Hanyu
Pronunciation	Onyomi* Kunyomi† Kokugo‡ Kana8bit	Pinyin	Phonetic_Code
Radical	Bushu	Radical	Radical
Stroke Count	Sokaku	Stroke	Stroke
Internal Code	JIScode	QuWei	QuWei

Notes:
* denotes a Chinese reading.
† denotes a Japanese reading.
‡ denotes a Kana reading.

a multibyte character. Also, to avoid a truncation problem at the key boundary, the size of the sort key (mixed ASCII and multibyte characters are allowed) is specified as a number of characters instead of a number of bytes.

- Multiple passes. Sorting Asian characters by any of the individual collating sequences (except QuWei) may not produce a unique sort order. In general, multiple successive passes using different collating sequences are needed to do so. Thus, the Asian OpenVMS SORT/MERGE utility allows a sort key specified with multiple passes of different collating sequences. In addition, if the /STABLE qualifier is not specified, QuWei collation is always added last to the sort key to further classify records with identical collation values.
- User-defined characters. The Asian OpenVMS SORT/MERGE utility supports collation of UDCs. When a UDC is encountered, the SORT/MERGE operation retrieves the collation weight from a system database maintained by the CMGR utility with the value defined by a user when the character was registered.

MAIL Most of the work involved in localizing the MAIL utility enhances the user interface to use Asian characters. String search enhancements allow processing by character units instead of by byte units. String uppercasing is not applied to Asian characters. The subject field, the personal name field, and the folder names can all contain Asian characters. The listing of mail folders can be displayed in sorted order in any of the supported collation sequences using the new command qualifier DIR/FOLD/COLLATING_SEQUENCE=(<collating sequences>).

The MAIL utility invokes the Asian text editors by default instead of invoking the standard ones. The OpenVMS/Japanese system incorporates the Japanese input method to allow users to enter Japanese characters.

EDT The Asian OpenVMS EDT editor was localized and enhanced for Asian text editing. Much of the work involved driving the terminal display correctly for Asian characters. In addition, the editor has a large number of new editing features.

TPU/EVE Localization of TPU and EVE deals mainly with managing the screen update for mixed ASCII and Asian characters, such as cursor movement and screen boundary handling. Both the TPU editing engine and the EVE interface were modified. Asian-specific TPU built-in procedures were added, and existing ones were enhanced. String search is now aligned at the character boundary rather that on byte units.

For the Japanese TPU/EVE, one of the most difficult tasks is to incorporate the Japanese input method. This requires managing overlap windows in a character cell terminal between the input method working area and the background editing area.

DECwindows System With the increasing emphasis on internationalization features in the X11 and OSF/Motif standards, OpenVMS DECwindows systems provide these features and the localization

features demanded by the market. For a description of the latest internationalization support in the X Window System standard, refer to the book by Scheifler and Gettys.[14]

Asian OpenVMS Localization Issues

The Asian OpenVMS effort has been addressing various technical and engineering issues.[15,16,17] This section discusses the major ones.

Technical Issues

Localization of the OpenVMS components to support the Asian languages requires reengineering the program codes and text translation. The need to reengineer source code arises for two main reasons.

1. OpenVMS components make fundamental programming assumptions and practices based on the ASCII and DEC MCS character sets. For example,
   - OpenVMS components assume the character set to be ASCII (plus DEC MCS in some cases), and blindly uppercase and lowercase characters, validate characters against the DEC MCS, and define printable characters according to the ASCII and DEC MCS encodings.
   - OpenVMS components assume characters to be 1 byte and use string manipulation algorithms based on 1-byte units.
   - OpenVMS components assume the display width of a character to be of fixed length (1 byte) and use screen display management algorithms based on the assumption that 1 byte equals one display column.
   - OpenVMS components assume that the character count, the byte count, and the display width are the same, and use string manipulation algorithms and character cell terminal screen display management based on this assumption.
2. Some functionality that is required to support Asian languages is missing in the standard OpenVMS environment, For example,
   - Keyboard input of Asian characters requires more complicated input method processing than is available in the standard OpenVMS environment.
   - Collation rules of Asian languages are radically different from English collation rules, on which the standard OpenVMS environment is based.
   - The standard OpenVMS environment does not support the application-transparent processing of UDCs.
   - The writing direction of Asian languages can be vertical, i.e., from top to bottom. The standard OpenVMS environment assumes horizontal, left-to-right languages.

Engineering Issues

Historically, the Asian localization of the OpenVMS system has been organized as an engineering effort that is separate from mainstream development. As a result, a number of engineering constraints and overhead costs exist.

- Single language support. The design goal for the Asian OpenVMS variants, as driven by the local market requirements, has been targeted at supporting a single language on one system, i.e., one language variant per system. As a result, no special design considerations are given to supporting multiple languages on one system.
- Full upward compatibility. The top design requirement is to keep full downward compatibility with original ASCII/DEC MCS OpenVMS systems. All ASCII/DEC MCS applications with existing data must be able to run unchanged on the Asian OpenVMS variants. In fact, an Asian OpenVMS system can, at any time, be reset to operate in the original DEC MCS mode, if desired. Therefore, most localized components must be able to switch between the standard and Asian code paths. System mechanisms for determining the current language variant and operating mode are required.
- Optimal performance. Another design goal is to minimize any performance impact on standard English components. As a result, Asian codes are designed around standard code paths. For example, branches for Asian code are placed at the end of a conditional statement, and Asian code branches out from the main line code using special hooks.
- Limited or no kernel changes. Since Asian code changes are not merged into the mainstream, kernel changes in Asian code would be very difficult to maintain with new OpenVMS releases. In addition, any kernel changes in the standard OpenVMS release will likely break the Asian code. This puts a constraint on supporting Asian languages in OpenVMS kernel components.

- Commonality. Because the Asian languages share a lot of commonality, techniques such as common source are used for most Asian localized components to maximize engineering return by sharing common Asian localization code.

Conclusions

Local language processing has become a mandatory functionality for computer systems sold in Asian markets. From the OpenVMS operating system perspective, the basic local Asian language processing requirements are being addressed by its Asian language variants in a single-language-for-a-single-locale manner. With global trade and the technology trend of distributed computing systems, the challenge for the future is to be able to provide OpenVMS services simultaneously to multiple clients operating in different languages and code sets. Such a requirement leads to the concept of a multilingual operating system, which allows software applications to run irrespective of the language and/or code set they support. With the availability of the ISO 10646 Universal Character Set (UCS) standard, the set of tools for building such a multilingual operating system has been enhanced.[18]

From an engineering perspective, the current Asian localization approach of OpenVMS has been adopted historically because of a number of factors and constraints, such as the organization of engineering resources and the initial need to bring the capability rapidly to the market. The reengineering techniques are geared toward the character set encoding schemes currently supported. The arrangement of performing localization remotely and independently from the original mainstream development has meant costly reengineering and maintenance overheads in the long term. With the industrial trend of shipping global software simultaneously satisfying multiple different local market requirements, an international product engineering approach must be taken to minimize the cost of worldwide system engineering to deliver a global product. In particular, the original product must be internationalized from the ground up, so that no separate reengineering is needed during localization to support a local market. In addition, to achieve simultaneous worldwide delivery, concurrent engineering of localization needs to be performed closely in parallel with the product development.

References

1. T. Greenwood, "International Cultural Differences in Software," *Digital Technical Journal*, vol. 5, no. 3 (Summer 1993, this issue): 8–20.
2. *Code of the Japanese Graphic Character Set for Information Interchange*, JIS C 6226-1978 (Tokyo: Japanese Standards Association, 1978).
3. *Code of the Japanese Graphic Character Set for Information Interchange*, JIS X 0208-1983 (Tokyo: Japanese Standards Association, 1983).
4. *Code of the Japanese Graphic Character Set for Information Interchange*, JIS X 0208-1990 (Tokyo: Japanese Standards Association, 1990).
5. *Code of the Supplementary Japanese Graphic Character Set for Information Interchange*, JIS X 0212-1990 (Tokyo: Japanese Standards Association, 1990).
6. *Code for Information Interchange*, JIS X 0201-1976 (Tokyo: Japanese Standards Association, 1976).
7. *Code of Chinese Graphic Character Set for Information Interchange*, GB 2312-1980 (Beijing: Technical Standards Publishing, 1981).
8. *Standard Interchange Code for Generally-used Chinese Characters*, CNS 11643-1986 (Taipei: National Bureau of Standards, 1986).
9. *Chinese Standard Interchange Code*, CNS 11643-1992 (Taipei: National Bureau of Standards, 1992).
10. *Code for Information Interchange (Hangul and Hanja)*, KS C 5601-1987 (Seoul: Korean Industrial Standards Association, 1989).
11. *Information Processing—ISO 7-bit and 8-bit Coded Character Sets—Code Extension Techniques*, 3d ed., ISO 2022 (Geneva: International Organization for Standardization/ International Electrotechnical Commission, 1986).
12. T. Honma, H. Baba, and K. Takizawa, "Japanese Input Method Independent of Applications," *Digital Technical Journal*, vol. 5, no. 3 (Summer 1993, this issue): 97–107.

13. R. Haentjens, "The Ordering of Universal Character Strings," *Digital Technical Journal*, vol. 5, no. 3 (Summer 1993, this issue): 43–52.

14. R. Scheifler and J. Gettys, *X Window System, X11, Release 5*, 3d ed. (Burlington, MA: Digital Press, Order No. EY-J802E-DP-EEB, 1992).

15. *Introduction to Asian Language Software Localization* (Maynard, MA: Digital Equipment Corporation, Order No. AD-PG0AA-TE, December 1990).

16. *Technical Guide to Asian Language Software Localization* (Maynard, MA: Digital Equipment Corporation, Order No. AD-PG0BA-TE, December 1990).

17. Addendum to Technical Guide to Asian Language Software Localization (Maynard, MA: Digital Equipment Corporation, Order No. AD-PG0CA-TE, December 1990).

18. *Information Technology—Universal Multiple-Octet Coded Character Set (UCS)—Part 1: Architecture and Basic Multilingual Plane*, ISO/IEC 10646-1 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1993).

Hirotaka Yoshioka
Jim Melton

Character Internationalization in Databases: A Case Study

Character internationalization poses difficult problems for database management systems because they must address user (stored) data, source code, and metadata. The revised (1992) standard for database language SQL is one of the first standards to address internationalization in a significant way. DEC Rdb is one of the few Digital products that has a complete internationalization (Asian) implementation that is also MIA compliant. The product is still evolving from a noninternationalized product to a fully internationalized one; this evolution has taken four years and provides an excellent example of the issues that must be resolved and the approaches to resolving them. Rdb can serve as a case study for the software engineering community on how to build internationalized products.

Internationalization is the process of producing specifications and products that operate well in many languages and cultures.[1] Internationalization has several different aspects such as character set issues, date and time representation, and currency representation. Most of these affect many areas of information technology where the solutions are reasonably similar; for example, solutions to currency representation are equally applicable to database systems and to programming languages. Database systems, however, are affected in several unique ways, all of which deal with character sets. In this paper, we focus on the issues of character set internationalization in database management systems (DBMS) and do not address the other aspects of date and time, currency, or locales.

To better understand the problems and solutions associated with character internationalization of database systems, we present an overview of the solutions found in the SQL standard and report a case study of implementing those solutions in a commercial product. We first discuss the character internationalization features supported in the recently published revision of the standard for *Database Language SQL* (ISO/IEC 9075:1992 and ANSI X3.135-1992).[2] We then describe in some detail the application of those features in DEC Rdb, Digital's relational database product. The internationalization of DEC Rdb serves as a case study, or a model, for the internationalization of Digital's software products in general.

Internationalization in the SQL Standard

Like most computer languages, SQL came into being with the minimal set of characters required by the language; vendors were free to support as many, or as few, additional characters as they perceived their markets demanded. There was little, if any, consideration given to portability beyond the English language customer base. In 1989, after work was completed on ISO 9075:1989 and ANSI X3.135-1989 (SQL-89), significant changes were proposed for the next revision of the SQL database language to address the requirement for additional character set support. (Unfortunately, this put SQL in the vanguard, and little support existed in the rest of the standards community for this effort.)

Character Set Support

SQL must address a more complex set of requirements to support character sets than other programming languages due to the inherent nature of database systems. Whereas other programming languages have to cover the character set used to encode the source program as well as the character set for data processed by the program, database systems also have to address the character set of the metadata used to describe the user data. In other words, character set information must be known within three places in a database environment.

1. The user data that is stored in the database or that is passed to the database system from the application programs.

   In SQL, data is stored in tables, which are two-dimensional representations of data. Each record of data is stored in a row of a table, and each field in a row corresponds to a column of a table. All the data in a given column of a table has the same data type and, for character data, the same character set.

2. The metadata stored in the database that is used to describe the user data and its structure.

   In SQL databases, metadata is also stored in tabular form (so that it can be retrieved using the same language that retrieves user data). The metadata contains information about the structure of the user data. For example, it specifies the names of the users' tables and columns.

3. The data management source code.

   Data management statements (for querying and updating the database) have to be represented as character strings in some character set. There are three aspects of these statements that can be independently considered. The key words of the language (like SELECT or UPDATE) can be represented in one character set—one that contains only the alphabetic characters and a few special (e.g., punctuation) characters; the character string literals that are used for comparison with database data or that represent data to be put into the database; and the identifiers that represent the names of database tables, columns, and so forth.

   Consider the SQL statement

```
SELECT EMP_ID FROM EMPLOYEES
    WHERE DEPARTMENT = 'Purchasing'
```

In that statement, the words SELECT, FROM, and WHERE; the equals sign; and the two apostrophes are syntax elements of the SQL language itself. EMP_ID, EMPLOYEES, and DEPARTMENT are names of database objects. (EMPLOYEES is a table; the other two are columns of that table.) Finally, Purchasing is the contents of a character string literal used to compare against data stored in the DEPARTMENT column.

That seems straightforward enough, but what if the database had been designed and stored in Japan so that the names of the table and its columns were in Japanese *kanji* characters? Furthermore, what if the name of some specific department was actually expressed in Hebrew (because of a business relationship)? That means that our database would have to be able to handle data in Hebrew characters, metadata in Japanese characters, and source code using Latin characters!

One might reasonably ask whether this level of functionality is really required by the marketplace. The original impetus for the character internationalization of the SQL standard was provided by proposals arising from the European and Japanese standards participants. However, considerable (and enthusiastic) encouragement came from the X/Open Company, Ltd. and from the Nippon Telephone and Telegraph/Multivendor Integration Architecture (NTT/MIA) project, where this degree of mixing was a firm requirement.[3]

The situation is even more complex than this example indicates. In general, application programs must be able to access databases even though the data is in a different character encoding than the application code! Consider a database containing ASCII data and an application program written in extended binary coded decimal interchange code (EBCDIC) for an IBM system, and then extend that image to a database containing data encoded using the Japanese extended UNIX code (EUC) encoding and an application program written in ISO 2022 form. The program must still be able to access the data, yet the character representations (of the same characters) are entirely different. Although the problem is relatively straightforward to resolve for local databases (that is, databases residing on the same computer as the application), it is extremely difficult for the most general case of heterogeneous distributed database environments.

Addressing Three Issues

To support internationalization aspects, three distinct issues have to be addressed: data representation, data comparison, and multiple character set support.

Data Representation How is the data (including metadata and source code) actually represented? The answer to this question must address the actual repertoire of characters used. (A character repertoire is a collection of characters used or available for some particular purpose.) It must also address the form-of-use of the character strings, that is, the ways that characters are strung together into character strings; alternatives include fixed number of

bits per character, like 8-bit characters, or variable number of bits per character, like ISO 2022 or ASN.1. Finally, the question must deal with the character encoding (for example, ASCII or EBCDIC). The combination of these attributes is called a character set in the SQL standard.

It is also possible for the data to be represented in different ways within the database and in the application program. A column definition that specifies a character set would look like this

```
NAME CHARACTER VARYING (6)
    CHARACTER SET IS KANJI,
```

or

```
NAME NATIONAL CHARACTER VARYING (6),
```

(which specifies the character set defined by the product to the national character set), while a statement that inserts data into that column might be

```
INSERT INTO EMPS(NAME)
    VALUES (..., _KANJI'従業', ...);
```

If the name of the column were expressed in *hiragana,* then the user could write

```
INSERT INTO EMPS(_HIRAGANA 2な ま え)
    VALUES (..., _KANJI'番号', ...);
```

Data Comparison How is data to be compared? All character data has to be compared using a collation (the rules for comparing character strings). Most computer systems use the binary values of each character to compare character data 1 byte at a time. This method, which uses common character sets like ASCII or EBCDIC, generally does not provide meaningful results even in English. It provides far less meaningful results for languages like French, Danish, or Thai.

Instead, rules have to be developed for language-specific collations, and these rules have to resolve the problems of mixing character sets and collations within SQL expressions.

Applications can choose to force a specific collation to be used for comparisons if the default collation is inappropriate:

```
WHERE :hostvar = NAME COLLATE JAPANESE
```

Multiple Character Set Support How is the use of multiple character sets handled? The most powerful aspect of SQL is its ability to combine data from multiple tables in a single expression. What if the data in those tables is represented in different character sets? Rules have to be devised to specify the results for combining such tables with the relational join or union operations.

What if the character sets of data in the source program are different from those in the database? Rules must exist to provide the ability for programs to query and modify databases with different character sets.

Components of Character Internationalization

SQL recognizes four components of character internationalization: character sets, collations, translations, and conversions. Character sets are described above; they comprise a character repertoire, a form-of-use, and an encoding of the characters. Collations are also described above; they specify the rules for comparing character strings expressed in a given character repertoire.

Translations provide a way to translate character strings from one character repertoire to a different (or potentially the same) repertoire. For example, one could define a translation to convert the alphabetic letters in a character string to all uppercase letters; a different translation might transliterate Japanese *hiragana* characters to Latin characters. By comparison, conversions allow one to convert a character string in one form-of-use (say, two octets per character) into another (for example, compound text, a form-of-use defined in the X Window System).

SQL provides ways for users to specify character sets, collations, and translations based on standards and on vendor-provided facilities. The current draft of the next version of the SQL standard (SQL3) also allows users to define their own character sets, collations, and translations using syntax provided in the standard.[4,5] If these facilities come to exist in other places, however, they will be removed from the SQL standard (see below). SQL does not provide any way for users to specify their own conversions; only vendor-provided conversions can be used.

Interfacing with Application Programs

Application programs are typically written in a third-generation language (3GL) such as Fortran, COBOL, or C, with SQL statements either embedded in the application code or invoked in SQL-only procedures by means of CALL-type statements.[6] As a result, the interface between the database system and 3GL programs presents an especially difficult problem in SQL's internationalization facilities. Figure 1 illustrates the procedure to invoke SQL from C; Figure 2 shows SQL as it is invoked from C; and Figure 3 shows SQL schema.

```
main()
{
  #include <stdio.h>
  #include <stdlib.h>
  #include "SQL92.h"  /* Interface to SQL-92 */

  static sqlstate char[6];
  static employee_number char[7];
  static employee_name wchar_t[26];
  static employee_contact char[13];

  /* Assume some code here to produce an appropriate
     employee number value */

  LOCATE_CONTACT (employee_number, employee_name,
                  employee_contact, sqlstate);

  /* Assume more code here to use the result */

}
;
```

Figure 1 Invoking SQL from C

```
MODULE i18n_demo NAMES ARE Latin1
  LANGUAGE C
  SCHEMA personnel AUTHORIZATION management

  PROCEDURE locate_contact
      ( :emp_num           CHARACTER (6) CHARACTER SET Ascii,
        :emp_name          CHARACTER VARYING (25) CHARACTER SET Unicode,
        :contact_name      CHARACTER VARYING (6) CHARACTER SET Shift_jis,
        SQLSTATE)
    SELECT name, contact_in_japan
      INTO :emp_name, :contact_name
      FROM personnel.employees
      WHERE emp_id = :emp_num;
```

Figure 2 SQL Invoked from C

```
CREATE SCHEMA personnel AUTHORIZATION management
  DEFAULT CHARACTER SET Unicode


CREATE TABLE employees (
    emp_id                 CHARACTER (6) CHARACTER SET Ascii,
    name                   CHARACTER VARYING (25),
    department             CHARACTER (10) CHARACTER SET Latin1,
    salary                 DECIMAL (8,2),
    contact_in_japan       CHARACTER VARYING (6) CHARACTER SET Shift_jis,
    ...,
    PRIMARY KEY (emp_id) )
```

Figure 3 SQL Schema

In these figures, all the metadata values (that is, the identifiers) are expressed in Latin characters; this resolves the data representation issue. The reader should compare the character sets of the data items in the EMPLOYEES table and the corresponding parameters in the SQL procedure. The difficulties arise when trying to achieve a correlation between the parameters of the SQL procedure and the arguments in the C statement that invokes that procedure.

The C variable employee_number corresponds to the SQL parameter :emp_num; the C data type char is a good match for CHARACTER SET ASCII. The C variable employee name corresponds to the SQL parameter :emp_name; the C data type wchar_t is chosen by many vendors to match CHARACTER SET Unicode. However, CHARACTER SET Shift_jis is more complicated; there is no way to know exactly how many bytes the character string will occupy because each character can be 1 or 2 bytes in length. Therefore, we have allocated a C char that permits up to 13 bytes. Of course, the C run-time library would have to include support for ASCII data, Unicode data, and Shift JIS data.

Typically, 3GL languages have little or no support for character sets beyond their defaults. Consequently, when transferring data from an internationalized SQL database into a noninternationalized application program, many of the benefits are lost. Happily, that situation is changing rapidly. Programming language C is adding facilities for handling additional character sets, and the ISO standards group responsible for programming languages (ISO/IEC JTC1/SC22) is investigating how to add those capabilities to other languages as well.

The most difficult issue to resolve concerns the differences in specific character sets (especially form-of-use) supported by SQL implementations and 3GL implementations. As with other issues, purely local situations are easy to resolve because a DBMS and a compiler provided by the same vendor are likely to be compatible. Distributed environments, especially multivendor ones, are more complicated. SQL has provided one solution: it permits the user to write SQL code that translates and converts the data into the form required by the application program as long as the appropriate conversions and translations are available for use by SQL. Of course, once the data has been transferred into the application program, the question remains: What facilities does it have to manipulate that data?

Remote Database Access Issue

As mentioned, a distributed environment presents significant difficulties for database internationalization. A simple remote database access scenario illustrates these problems. If an application program must access some (arbitrary) database via a remote (e.g., network) connection, then the remote database access facility must be able to deal with all the character sets that the application and database use together; it may also have to deal with differences in available character sets. (See Figure 4.)

An ISO standard for remote database access (ISO/IEC 9579-1 and 9579-2) uses the ASN.1 notation and encoding for transporting SQL commands and database data across remote connections.[7] ASN.1 notation, as presently standardized, provides no way to use various character sets in general. Recently work has begun to resolve this problem. The revised standard must allow a character set to be specified uniquely by means of a name or identifier that both ends of the connection can unambiguously interpret in the same way. The individual characters in ASN.1 character strings must be similarly identifiable in a unique way.

This problem has not yet been resolved in the standards community, partly because several groups have to coordinate their efforts and produce compatible solutions.

Hope for the Future

In the past, programming languages, database systems, networks, and other components of information management environments had to deal with character sets in very awkward ways or use vendor-provided defaults. The result has been an incredible mess of 7-bit (ASCII, for example) and 8-bit (Latin-1, for example) code sets, PC code pages, and even national variants to all of these. The number of code variants has made it very difficult for a database user to write an application that can be executed on any database system using recompilation only. Collectively, they make too many assumptions about the character set of all character data.

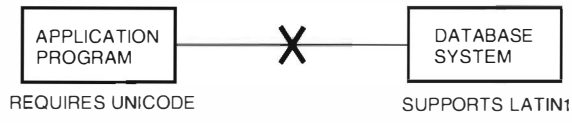


Figure 4 Remote Database Access

The future outlook for database internationalization was improved dramatically by the recent adoption of ISO 10646, Universal Multiple-Octet Coded Character Set (UCS) and an industry counterpart, Unicode.[8] The hope is that Unicode will serve as a "16-bit ASCII" for the future and that all new systems will be built to use it as the default character set.

Of course, it will be years—if not decades—before all installed computer hardware and software use Unicode. Consequently, provisions have to be made to support existing character sets (as SQL-92 has done) and the eccentricities of existing hardware and software (like networks and file systems). As a result, several different representations of Unicode have been developed that permit transmission of its 16-bit characters across networks that are intolerant of the high-order bit of bytes (the eighth bit) and that permit Unicode data to be stored in file systems that deal poorly with all the bit patterns it permits (such as octets with the value zero).

In the past few years, many alternative character representations have been considered, proposed, and implemented. For example, ISO 2022 specifies how various character sets can be combined in character strings with escape sequences and gives instructions on switching between them.[9] Similarly, ASN.1-like structures, which provide fully tagged text, have been used by some vendors and in some standards, e.g., Open Document Architecture.[10] None of these representations has gained total acceptance. Database implementors perceive difficulties with a stateful model and with the potential performance impact of having a varying number of bits or octets for each character. UCS and Unicode appear to be likely to gain wide acceptance in the database arena and in other areas.

Future Work for the SQL Standard

One should not conclude that the job is done, that there is nothing left to work on. Instead, a great deal of work remains before the task of providing full character set internationalization for database systems is completed.

At present, the working draft for SQL3 contains syntax that would allow users to define their own character sets, collations, and translations using a nonprocedural language.[4,5] In general, the SQL standards groups believe that it is inappropriate for a database standard to specify language for such widely needed facilities. Consequently, as soon as the other responsible standards bodies provide a language for these specifications, it is probable that this capability will be withdrawn from the SQL3 specification. This decision would completely align the SQL character internationalization capabilities with the rest of the international standards efforts.

After other standards for these tasks are in place, however, the remote data access (RDA) standard will have to be evolved to take advantage of them. RDA must be able to negotiate the use of character sets for database applications and to transport the information between database clients and servers. In order for RDA to be able to do this, the ASN.1 standard will have to support arbitrary named character sets and characters from those sets.

As a result, relevant standards bodies will need to provide (1) names for all standardized character sets and (2) the ability for vendors to register their own character sets in a way that allows them to be uniquely referenced where needed. Still other bodies will need to provide language and services for defining collations and translations. Finally, registries will need to be established for vendor-supplied collations, translations, and conversions.

Of course, the greatest task will be to provide complete support for all these facilities throughout the information processing environment: operating systems, communication links, CPUs, printers, keyboards, windowing systems, file systems, and so forth. Healthy starts have been made on some of these (such as the X Window System), but much work remains to be done.

DEC Rdb: An Internationalization Case Study

DEC Rdb (Rdb/VMS) is one of the few Digital products that has an internationalized implementation that is also compliant with the multivendor integration architecture (MIA).[11,12] Its evolution from a noninternationalized product to a fully internationalized one has taken four years to achieve. The design and development of Rdb can serve as a case study for software engineers on how to build internationalized products. In this half of our paper, we present the history of the reengineering process. Then we describe some difficulties with the reengineering process and our work to overcome them. Finally, we evaluate the result.

Localization and Reengineering

The localization process comprises all activities required to create a product variant of an application that is suitable for use by some set of users with

similar preferences on a particular platform. Reengineering is the process of developing the set of source code changes and new components required to perform localization. DEC Rdb had to be reengineered to support several capabilities that are mandatory in Japan and other Asian countries.

Our experience has shown that the reengineering process is very expensive and should be avoided. If the original product was not designed for internationalization or localization, however, reengineering is a necessary (and unavoidable) evil. Typically, reengineering is required; so we decided to develop a technology that would avoid reengineering and to build a truly internationalized product.

Most engineering groups follow the old assumptions about product design. These assumptions include the following:

1. The character set is implicitly ASCII.
2. Each character is encoded in 7 bits.
3. The character count equals the byte count and equals the display width in columns.
4. The maximum number of distinct characters is 128.
5. The collating sequence is ASCII binary order.
6. The messages are in English.
7. The character set of the source code is the same as it is at run time.
8. The file code (the code on the disk) is the same as the process code (the code in memory).

Different user environments require different product capabilities. Japanese *kanji* characters are encoded using 2 bytes per character. If a product assumes that the character set is 7-bit ASCII, that product must be reengineered before it can be used in Japan. On the other hand, internationalized products can operate in different environments because they provide the capabilities to meet global requirements. These capabilities include the following:

1. Multiple character sets ensure that the customer's needs are met.
2. Each character is encoded using at least 8 bits.
3. The character count does not equal the byte count or the display width.
4. The maximum number of unique characters is unknown.
5. The collating sequence meets the customer's needs.
6. The messages are in the language the customer uses.
7. The character set of the source code is not necessarily the same as it is at run time.
8. The file code is not necessarily the same as the process code.

The reengineering process has two significant drawbacks: (1) the high cost of reengineering and (2) the time lag between shipping the product to the customer in the United States and shipping to the customer in Japan. The time lag can be reduced but cannot be eliminated as long as we reengineer the original product. If a local product is released simultaneously with the original, both Digital and the customers will benefit significantly.

In the next section, we follow the DEC Rdb product through the reengineering process required to produce the Japanese Rdb version 3.0.

Reengineering Process

DEC Rdb version 3.0 was a major release and consequently was very important to the Japanese market. The International System Engineering Group was asked to release the Japanese version by the end of 1988, which was within six months of the date that it was first shipped to customers in the United States.

Japanese and Asian Language Requirements to VAX Rdb/VMS

Japanese and Asian language requirements apply to DEC Rdb and other products as well. The requirements common to Asian languages are 2-byte character handling, local language editor support, and message and help file translation.

Japanese script uses a 2-byte code; therefore 2-byte character handling is mandatory. For example, character searches must be performed on 2-byte boundaries and not on 1-byte boundaries. If a string has the hexadecimal value 'A1A2A3A4', then its substrings are 'A1A2' and 'A3A4'. 'A2A3' must not be matched in the string.

Digital's Asian text editors, e.g., the Japanese text processing utility (JTPU) and Hanzi TPU (for China), must be supported as well as the original TPU, the standard EDT editor, and the language-sensitive editor.

Messages, help files, and documentation must all be translated into local languages.

The country-specific requirements include support for a Japanese input method. *Kana*-to-*kanji* input methods must be supported in command lines. In addition, 4-byte character handling is required for Taiwan (Hanyu). Finally, NTT/MIA SQL features must be added for Japan.

Since there are not many requirements, one might conclude that the reengineering task is not difficult. However, reengineering is complicated, expensive, and time consuming; and thus should be avoided.

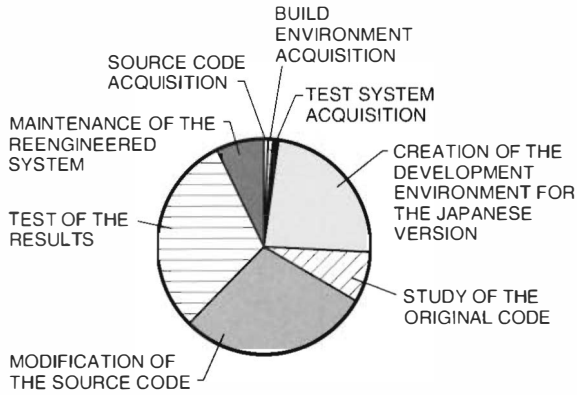
Reengineering Japanese Rdb Version 3.x

A database management system like DEC Rdb is very complex. The source code is more than 810,000 lines; the build procedures are complicated; and a mere subset of the test systems consumes more than one gigabyte of disk space. Consequently, the reengineering process is complicated. The process encompasses more than modifying the source code. Instead, a number of distinct steps must be accomplished:

1. Source code acquisition
2. Build environment acquisition
3. Test system acquisition
4. Creation of the development environment for the Japanese version
5. Study of the original code
6. Modification of the source code
7. Test of the results, including the new Japanese functionality and a regression test of the original functionality
8. Maintenance of the reengineered system

Figure 5 shows the development cost in person-weeks for each of the eight steps. Two engineers stabilized the development environment—compile, link/build, and run—for version 3.0 of DEC Rdb in approximately four months. It is likely that the process required four months because it was our first development work on DEC Rdb. In addition, approximately two months were needed to be able to run the test system. It was not an easy task.

Each step had to be repeated for each version of the original. (Project time decreased a little.) Every version required this reengineering, even if no new functionality was introduced. The cost of building



Note: Each segment of the chart represents the project time (person-weeks) required to complete each step in the reengineering process.

Figure 5 Reengineering Process for Japanese Rdb Version 3.x

the environment became cheaper after the first time. The other steps such as modifying the source code, testing, and maintenance remained at almost the same cost.

Reengineering Metric

We modified about 10 percent of the original source modules during reengineering. Most of the modification occurred in the front end, e.g., SQL and RDML (relational database manipulation language). The engine parts, the relational database management system (RDMS), and KODA (the kernel of the data access, the lowest layer of the physical data access) were not modified very much. Table 1 gives the complete reengineering metrics.

$$\text{Reengineering metric} = \frac{\text{(modified modules + new created modules)}}{\text{(original + modified + new created modules)}}$$

Coengineering Process: No More Reengineering

To reduce and eliminate reengineering, we have taken a conservative, evolutionary approach rather than a revolutionary one. We used only proven technologies. The evolution can be divided into three phases:

1. Joint Development with Hong Kong. Our development goal was to merge Japanese, Chinese (People's Republic of China and Taiwan), and Korean versions into one common Asian Rdb source code.

Table 1 Reengineering Metrics

Facility	Reengineering Metric	Modified Modules	Total Modules	Size in Kilo Lines
SQL	6.3%	8	128	226.0
RDML	11.7%	11	94	188.3
RDMS	3.1%	4	127	154.0
KODA	0.6%	1	157	109.8
RMU	0.0%	0	41	80.5
Dispatcher	0.0%	0	30	60.9

Notes:

RMU is the Rdb management utility; it is used to monitor, back up, restore, and display DEC Rdb databases.

The reengineering metric for JCOBOL (a Digital COBOL compiler sold in Japan) is 47/258 = 18.2%; the size is 225.0 kilo lines.

2. Coengineering Phase I. Our goal was to merge Asian common Rdb into the original master sources for version 4.0. The merger of J-Rdb and Chinese-Rdb into Rdb would eliminate reengineering and create one common executable image.
3. Coengineering Phase II. In the final phase, our goal was to develop the internationalized product for version 4.2 by adding more internationalization functionality, SQL-92 support, MIA support for one common executable, and multiple character set support.

Coengineering is a development process in which local engineers temporarily relocate to the Central Engineering Group in the United States to develop the original product jointly with Central Engineering. The engineers from a non-English-speaking country provide the user requirements and the cultural-dependent technology (e.g., 2-byte processing and input methods), and Central Engineering provides the detailed knowledge of the product. This process promotes good experiences for both parties. For example, the local engineers learn the corporate process, and the corporate engineers have more dedicated time to understand the requirements and difficulties of local product needs, what internationalization means, and how to build the internationalized product. Coengineering minimizes the risks associated with building internationalized products.

Asian Joint Development

Our goal for the Asian joint development process was to provide a common Asian source code for Japan, People's Republic of China (PRC), Taiwan, and Korea. One common source code would reduce reengineering costs in Asia. To achieve our goal, we devised several source code conventions. The purposes of the conventions were

1. To identify the module for each Asian version by its file name
2. To make it possible to create any one of the Asian versions (for Japan, the PRC, Taiwan, or Korea) or the English version from the common source codes, using conditional compilation methods
3. To identify the portions of codes that were modified for the Japanese version
4. To facilitate an engineer in Hong Kong who is developing versions for the PRC, Taiwan, and Korea

We developed the Japanese Rdb version 3.0 in Japan. The files were transferred to Hong Kong to develop versions for the PRC, Taiwan, and Korea. The modified versions were sent back to Japan to be merged into one common Asian source file.

Since we had one common Asian source file, reengineering in Hong Kong was reduced. Reengineering in Japan, however, was still necessary. We used compilation flags to create four country versions, that is, we had four sets of executable images. As a result, we needed to maintain four sets of development environments (source codes, tests, and so forth). We wanted to further simplify the process and therefore entered the coengineering phases.

Coengineering Phase I

The integration of Asian DEC Rdb into the base DEC Rdb product took place in two phases. In the first phase, we integrated the Asian code modifications into the source modules of the base product.

Consequently, the specific Asian versions of the product can be attained by definition and then translation of a logical name (a sort of environment variable). No conditional compilation is necessary. In all releases of DEC Rdb version 3.x, source modules of the base product were conditionally compiled for each Asian version, which created separate object files and images.

The process steps in this phase were

1. Merge the source code
   a. Create one executable image
   b. Remove Japanese/Asian VMS dependency
   c. Remove *kana*-to-*kanji* input method
2. Transfer the J-Rdb/C-Rdb tests

Source Code Merge (Rdb Version 4.0) To create a single set of images, we removed the compilation flags and introduced a new way of using the Asian-specific source code. We chose to do this by using a run-time logical name; the behavior of DEC Rdb changes based on the translation of that logical name.

We removed the Japanese/Asian VMS dependencies by using Rdb code instead of JSYSHR calls. (JSYSHR is the name given to the OpenVMS system services in Japanese VMS.)

We removed the *kana*-to-*kanji* input method: By calling LIB$FIND_IMAGE_SYMBOL (an OpenVMS system service to dynamically link library routines) to invoke an input method, the image need not be linked with JVMS; even an end user can replace an input method.

Run-time Checking We removed the compilation flags, but introduced a new logical name, the RDB$CHARACTER_SET logical, to switch the behavior of the product. For example, if RDB$CHARACTER_SET translates to DEC_KANJI, then the symbol ARDB_JAPAN_VARIANT is set true. This would indicate that all text would be treated as if it were encoded in DEC_KANJI. The code would behave as if it were DEC J-Rdb. This translation must occur at all levels of the code, including the user interface, DEC Rdb Executive, and KODA.

Since DEC Rdb checks the value of the logical name at run time, we do not need the compilation flags; that is, we can have one set of executable images.

Figure 6 shows the values that are valid for the RDB$CHARACTER_SET logical.

The DEC J-Rdb source contains code fragments similar to those shown in Figure 7, which were taken from RDOEDIT.B32 (written in the BLISS programming language). This code was changed to use a run-time flag set as a result of translation of the logical RDB$CHARACTER_SET, as shown in Figure 8.

Remove Japanese VMS (JVMS) Dependency The Japanese version of DEC Rdb version 3.x used the JVMS run-time library (JSY routines). The JSY routines are Japanese-specific character-handling routines such as "get one *kanji* character" and "read one *kanji* character." The library is available only on JVMS; native VMS does not have it, so DEC Rdb cannot use it. To remove the JVMS dependency, we modified all routines that called JSY routines so that they contain their own code to implement the same functions.

The J-Rdb/VMS source contains code fragments similar to the ones shown in Figure 9. The code was changed to remove references to the JSY routines as shown in Figure 10. This example does not use JSY routines like JSY$CH_SIZE or JSY$CH_RCHAR.

Remove Kana-to-kanji Input Method The dependency on JVMS can be eliminated by making the 2-byte text handling independent of JSY routines, but the input method still depends on JSYSHR for

```
$ DEFINE RDB$CHARACTER_SET -
    { DEC_KANJI | DEC_HANZI | DEC_HANGUL | DEC_HANYU }

  DEC_KANJI          Japanese
  DEC_HANZI          Chinese
  DEC_HANGUL         Korean
  DEC_HANYU          Taiwan

$ SET LANGUAGE JAPANESE ! If you use Japanese VMS
```

Figure 6 RDB$CHARACTER_SET Logical

```
! This example switches the default TPU shareable
! image (TPUSHR). If the Japanese variant is set,
! then the default editor should be JTPUSHR.
!
%IF $ARDB_JAPAN_VARIANT
%THEN
        TPU_IMAGE_NAME = ( IF (.TPU_NAME EQL 0)
        THEN $DESCRIPTOR ('TPUSHR')
        ELSE $DESCRIPTOR ('JTPUSHR'));
%ELSE
        TPU_IMAGE_NAME = $DESCRIPTOR ('TPUSHR');
```

Figure 7 Compilation Flag in DEC Rdb Version 3

```
 ! This code could be translated to the following
 ! which might contain redundant code but should work:
 !
IF.ARDB_JAPAN_VARIANT ! If ARDB_JAPAN_VARIANT flag is true,


THEN ! then Rdb/VMS should use the J-Rdb/VMS behavior.
      TPU_IMAGE_NAME = ( IF (.TPU_NAME EQL 0)
      THEN $DESCRIPTOR ('TPUSHR')
      ELSE $DESCRIPTOR ('JTPUSHR'))
ELSE
      TPU_IMAGE_NAME = $DESCRIPTOR ('TPUSHR');
```

Figure 8 Run-time Checking in Version 4

```
%IF $ARDB_COMMON_VARIANT %THEN
!+
! ARDB: Advance character pointer.
!
! JSY$CH_SIZE counts the size of the character.
! If it is ASCII, return 1,
! If it is Kanji, return 2.
! CP is a character pointer
CP = CH$PLUS( .CP, JSY$CH_SIZE( JSY$CH_RCHAR( .CP ) ) );
!-
%ELSE
CP = CH$PLUS( .CP, 1 );
%FI !$ARDB_COMMON_VARIANT
```

Figure 9 Using JSY Routines in DEC Rdb Version 3

kana-to-*kanji* conversions. To remove this dependency, we developed a new method to invoke the *kana*-to-*kanji* conversion routine. Figure 11 shows the new input method.

Since LIB$FIND_IMAGE_SYMBOL is used to find the Japanese input at run time, JSYSHR does not need to be referenced by the SQL$ executable image.

We created a shareable image for the input method, using the SYS$LA GUAGE logical to switch to the Japanese input method or to other Asian language input methods. Since an input method is a shareable image, a user can switch input methods by redefining the logical name to identify the appropriate image.

```
!*******************run time checking

IF $RDMS$ARDB_COMMON THEN
!+
! ARDB: Advance character pointer.
!
! If the code value of CP is greater than 128,
! then it means the first byte of Kanji, so
! advance 2, else it is ASCII, advance 1.
!
CP = CH$PLUS( .CP, (IF CH$RCHAR( .CP) GEQ 128
                    THEN
                        2
                    ELSE
                        1) );
!-
ELSE
CP = CH$PLUS( .CP, 1 );
FI !$RDMS$ARDB_COMMON

where $RDMS$ARDB_COMMON is a macro.
```

Figure 10 Removing JSY Routines in Version 4

```
SQL$.EXE
   |
   + (default) -> SMG$READ_COMPOSED_LINE
   |
   + (if Japanese Input is selected)
     LIB$FIND_IMAGE_SYMBOL
          |
          +------> (shareable for Japanese Input).EXE
```

Figure 11 Input Method for Version 4: Kana-to-kanji Conversion (Japanese Input) Shareable Image

Note that the input method is a mechanism to convert alphabetic characters to *kanji* characters. It is necessary to permit input of ideographic characters, i.e., *kanji,* through the keyboard. Asian local language groups would be responsible for creating a similar shareable image for their specific input methods.

Transfer DEC J-Rdb and DEC C-Rdb Tests To ensure the functionality of Japanese/Asian DEC Rdb, we transferred the tests into the original development environment. We integrated not only the source modules but also all the tests. Consequently, the Asian 2-byte processing capabilities have now been tested in the United States.

Kit Components and J-Rdb Installation Procedure The original DEC Rdb version 4.0 has the basic capability to perform 2-byte processing. Japanese and other Asian language components must be provided for local country variants. The localization kit for Japan contains Japanese documentation such as messages and help files, an input method, and the J-Rdb license management facility (LMF). As a result, we need not reengineer the original product any more. The installation procedure is also simplified. Users worldwide merely install DEC Rdb and then install a localization kit if it is needed.

The localization kits contain only the user interfaces, so no reengineering is necessary; however, translation of documentation, message files, help files, and so on to local languages still remains necessary. Nonetheless, the reengineering process is eliminated.

In version 4.0, we achieved the main goal, to integrate the Asian source code into the base product

to avoid reengineering. The Japanese localization kit was released with a delay of about one month after the U.S. version (versus a five-month delay in version 3.0). The one-month delay between releases is among the best in the world for such a complex product.

Coengineering Phase II

In the second phase of integration, we redesigned the work done in Phase I and developed a multilingual version of Rdb/VMS.

In version 4.0, we introduced the logical name RDB$CHARACTER_SET to integrate Asian functionality into DEC Rdb. In Phase II, we created an internationalized version of DEC Rdb. We retained the one set of images and introduced new syntax and semantics. We also provided support for the NTT/MIA requirements.

The following are the highlights of the release. The details are given in the Appendix.

- NTT/MIA SQL Requirements
  - NATIONAL CHARACTER data type
  - N'national' literal
  - *Kanji* object names
- Changes/extensions to the original DEC Rdb
  - Add a character set attribute
  - Multiple character set support
- Dependencies upon other products
  - CDD/Plus, CDD/Repository: Add a character set attribute
  - Programming languages: COBOL, PIC, N

Since we are no longer reengineering the original product, we now have time to develop the new functionality that is required by NTT/MIA. The new syntax and semantics of the character-set handling are conformant with the new SQL-92 standard. As far as we know, no competitor has this level of functionality.

If we had to continue to reengineer the original, we would not have had enough resources to continue development of important new functionalities. Coengineering not only reduces development cost but also improves competitiveness.

We introduced the RDB$CHARACTER_SET logical during Phase I to switch the character set being used. Since the granularity of character set support is on a process basis, however, a user cannot mix different character sets in a given process. In Phase II, we implemented the CHARACTER SET clause, defined in SQL-92, to allow multiple character sets in a table.

Database Character Sets The database character sets are the character sets specified for the attached database. Database character set attributes are default, identifier, and national.

SQL uses the database default character set for two elements: (1) database columns with a character data type (CHARACTER and CHARACTER VARYING) that do not explicitly specify a character set and (2) parameters that are not qualified by a character set. The user can specify the database default character set by using the DEFAULT CHARACTER SET clause for CREATE DATABASE.

SQL uses the identifier character set for database object names such as table names and column names. The user can specify the identifier character set for a database by using the IDENTIFIER CHARACTER SET clause for CREATE DATABASE.

SQL uses the national character set for the following elements.

- For all columns and domains with the data type NATIONAL CHARACTER or NATIONAL CHARACTER VARYING and for the NATIONAL CHARACTER data type in a CAST function
- In SQL module language, all parameters with the data type NATIONAL CHARACTER or NATIONAL CHARACTER VARYING
- For all character-string literals qualified by the national character set, that is, the literal is preceded by the letter N and a single quote (N')

The user can specify the national character set for a database by using the NATIONAL CHARACTER SET clause for CREATE DATABASE.

The following example shows the DEFAULT CHARACTER SET, IDENTIFIER CHARACTER SET, and NATIONAL CHARACTER SET clauses for CREATE DATABASE.

```
CREATE DATABASE FILENAME ENVIRONMENT
        DEFAULT CHARACTER SET DEC_KANJI
        NATIONAL CHARACTER SET KANJI
        IDENTIFIER CHARACTER SET DEC_KANJI;

CREATE DOMAIN DEC_KANJI_DOM CHAR(8);
CREATE DOMAIN KANJI_DOM NCHAR(6);
```

DEC_KANJI_DOM is a text data type with DEC_KANJI character set, and KANJI_DOM is a text data type with KANJI character set. The database default character set is DEC_KANJI and the national character set is KANJI.

As previously stated, the user can choose the default and identifier character sets of a database. Consequently, users can have both text columns that have character sets other than 7-bit ASCII and national character object names (i.e., *kanji* names, Chinese names, and so on).

In Rdb version 3.1 and prior versions, the character set was ASCII and could not be changed. In Rdb version 4.0, users could change character sets by defining the RDB$CHARACTER_SET logical. It is important to note that the logical name is a volatile attribute; that is, the user must remember the character set being used in the database in his process. On the other hand, the database character sets introduced in version 4.2 are persistent attributes, so the user is less likely to become confused about the character set in use.

Session Character Sets The session character sets are used during a session or during the execution of procedures in a module. The session character set has four attributes: literal, default, identifier, and national.

SQL uses the literal character set for unqualified character string literals. Users can specify the literal character set only for a session or a module by using the SET LITERAL CHARACTER SET statement or the LITERAL CHARACTER SET clause of the SQL module header, DECLARE MODULE statement, or DECLARE ALIAS statement.

Session character sets are bound to modules or an interactive SQL session, and database character sets are attributes of a database. For example, a user can change the session character sets for each SQL session; therefore, the user can attach to a database that has DEC_MCS names and then attach to a new database that has DEC_HANZI names.

Octet Length and Character Length In DEC Rdb version 4.1 and prior versions, all string lengths were specified in octets. In other words, the numeric values specified for the character-column length or the start-off set and substring length within a substring expression were considered to be octet lengths or offsets.

DEC Rdb version 4.2 supports character sets of mixed-octet and fixed-octet form-of-use. For this reason and to allow an upgrade path to SQL-92 (where lengths and offsets are specified in characters rather than octets), users are allowed to specify lengths and offsets in terms of characters. To change the default string-length unit from octet to characters, users may invoke the following:

```
SET CHARACTER LENGTH 'CHARACTERS';
```

Multiple Character Sets Examples Users can create a domain using a character set other than the database default or national character sets with the following sequence:

```
CREATE DOMAIN DEC_KOREA_DOM CHAR(6)
    CHARACTER SET DEC_KOREAN;

CREATE TABLE TREES
        (TREE_CODE TREE_CODE_DOM,
         QUANTITY INTEGER,
         JAPANESE_NAME CHAR(30),
         FRENCH_NAME CHAR(30)
            CHARACTER SET DEC_MCS,
         ENGLISH_NAME CHAR(30)
            CHARACTER SET DEC_MCS,
         KOREAN_NAME CHAR(30)
            CHARACTER SET DEC_KOREAN,
         KANJI_NAME NCHAR(30));
```

The table TREES has multiple character sets. This example assumes the default character set is DEC_KANJI and the national character set is KANJI. Users can have object names other than ASCII names specifying the identifier character set. The database engine uses the specific routines to compare data, since the engine knows the character set of the data. With DEC Rdb version 4.2, all three issues of data representation, multiple character-set support, and data comparison have been resolved.

Conclusions

By replacing reengineering with coengineering, we reduced the time lag between shipping DEC Rdb to customers in the United States and in Japan from five months for version 3.0 in July 1988 to two weeks for version 4.2 in February 1993. Figure 12 shows the decrease in time lag for each version we developed. We also eliminated expensive reengineering and maintenance costs. Finally, we increased competitiveness.

It has taken more than four years to evolve from a noninternationalized product to an internationalized one. If the product had originally been designed to be internationalized, this process would have been unnecessary. When DEC Rdb was originally created, however, we did not have an internationalization model, the architecture, or mature techniques. Reengineering is unavoidable under these circumstances.

By sharing our experience, we can help other product engineering groups avoid the reengineering process.

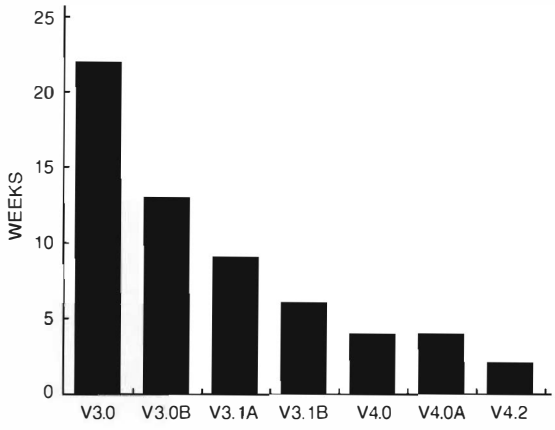


Figure 12 Time Lag between U.S. and Japanese Shipment of DEC Rdb

Future Work for DEC Rdb

Coengineering has proved that an evolutionary approach is not only possible, but that it is the most reasonable approach. Additional work, however, remains to be done for DEC Rdb.

DEC Rdb must support more character sets like ISO 10646-1. We think that the support of new character sets would be straightforward in the DEC Rdb implementation. DEC Rdb has the infrastructure for supporting it. SQL-92 has the syntax for it, that is, the character set clause. Furthermore, the DEC Rdb implementation has the attribute for a character set in the system tables.

Collations on Han characters should be extended. The current implementation of a collation on Han characters is based on its character value, that is, its code value. We believe the user would also like to have collations based on dictionaries, radicals, and pronunciations.[13]

Summary

There are significant difficulties in the specification of character internationalization for database systems, but the SQL-92 standard provides a sound foundation for the internationalization of products. The application of SQL-92 facilities to DEC Rdb is quite successful and can serve as a case study for the internationalization for other software products.

Acknowledgments

The authors gratefully acknowledge the help and contributions made by many people during the development of DEC Rdb's internationalization facilities and those of the SQL standard. In particular, Don Blair, Yasuhiro Matsuda, Scott Matsumoto, Jim Murray, Kaz Ooiso, Lisa Maatta Smith, and Ian Smith were particularly helpful during the DEC Rdb work. During the internationalization of SQL, Laurent Barnier, David Birdsall, Phil Shaw, Kohji Shibano, and Mani Subramanyam all made significant contributions.

References

1. G. Winters, "International Distributed Systems—Architectural and Practical Issues," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 53–62.

2. *American National Standard for Information Systems—Database Language SQL,* ANSI X3.135-1992 (American National Standards Institute, 1992). Also published as *Information Technology—Database Languages—SQL,* ISO/IEC 9075:1992 (Geneva: International Organization for Standardization, 1992).

3. W. Rannenberg and J. Bettels, "The X/Open Internationalization Model," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 32–42.

4. *Database Language SQL (SQL3),* Working Draft, ANSI X3H2-93-091 (American National Standards Institute, February 1993).

5. *Database Language SQL (SQL3),* Working Draft, ISO/IEC JTC1/SC21 N6931 (Geneva: International Organization for Standardization, July 1992).

6. J. Melton and A. Simon, *Understanding the New SQL: A Complete Guide* (San Mateo, CA: Morgan Kaufmann Publishers, 1992).

7. *Information Technology—Remote Database Access—Part 1: Generic Model, Service, and Protocol,* ISO/IEC 9579-1:1993, and *Information Technology—Remote Database Access—Part 2: SQL Specialization,* ISO/IEC 9579-2:1993 (Geneva: International Organization for Standardization, 1993).

8. J. Bettels and F. Bishop, "Unicode: A Universal Character Code," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 21–31.

9. *Information Processing—ISO 7-bit and 8-bit Coded Character Sets—Code Extension Techniques,* ISO 2022:1986 (Geneva: International Organization for Standardization, 1986).

10. *Information Processing, Open Document Architecture,* ISO/IEC 8613:1989 (Geneva: International Organization for Standardization, 1989).

11. *DEC Rdb, SQL Reference Manual* (Maynard, MA: Digital Equipment Corporation, Order No. AA-PWQPA-TE, January 1993).

12. *Multivendor Integration Architecture,* Version 1.2 (Tokyo: Nippon Telegraph and Telephone Corporation, Order No. TR550001, September 1992).

13. R. Haentjens, "The Ordering of Universal Character Strings," *Digital Technical Journal,* vol. 5, no. 3 (Summer 1993, this issue): 43–52.

Appendix: Syntax of Rdb Version 4.2

Format of CHARACTER SET Clause

```
<character data type> ::=
    <character string type>
        [ CHARACTER SET <character set specification> ]
  | <national character string type>


<character string type> ::=
    CHARACTER [ VARYING ] [ ( <length> ) ]
  | CHAR [ VARYING ] [ ( <length> ) ]
  | VARCHAR ( <length> )


<national character string type> ::=
    NATIONAL CHARACTER [ VARYING ] [ ( <length> ) ]
  | NATIONAL CHAR [ VARYING ] [ ( <length> ) ]
  | NCHAR [VARYING ] ( <length> )


<character set specification> ::=
        <character set name>


<character set name> ::= <name>
```

Character Set Names

```
  DEC_MCS
| KANJI
| HANZI
| KOREAN
| HANYU
| DEC_KANJI
| DEC_HANZI
| DEC_KOREAN
| DEC_SICGCC
| DEC_HANYU
| KATAKANA
| ISOLATINARABIC
| ISOLATINHEBREW
| ISOLATINCYRILLIC
| ISOLATINGREEK
| DEVANAGARI
```

Continued on next page.

Example of CHARACTER SET

```
CREATE DATABASE FILENAME ENVIRONMENT
   DEFAULT CHARACTER SET DEC_KANJI
   NATIONAL CHARACTER SET KANJI
   IDENTIFIER CHARACTER SET DEC_KANJI;

CREATE DOMAIN NAMES_GENERAL CHAR(20);

CREATE DOMAIN NAMES_PRC CHAR(20)
   CHARACTER SET IS HANZI;

CREATE DOMAIN NAMES_MCS CHAR(20)
   CHARACTER SET IS MCS;

CREATE DOMAIN NAMES_KOREAN CHAR(20)
   CHARACTER SET IS HANGUL;

CREATE DOMAIN NAMES_JAPAN NCHAR(20);
```

Format of Literals

```
<character literal> ::=
     <character string literal>
   | <national character string literal>

<character string literal> ::=
     [ <introducer><character set specification> ]
     <quote>[ <character representation>... ]<quote>

<character representation> ::=
     <nonquote character>
   | <quote symbol>

<nonquote character> ::= !! See the Syntax Rules.

<quote symbol> ::= <quote> <quote>

<national character string literal> ::=
     N <quote>[ <character representation>... ]<quote>
```

Example of National Object Name

```
CREATE TABLE 従業員
(従業員番号     CHAR(6),
 従業員氏名     NATIONAL CHARACTER(10),
 給与           DECIMAL(10),
 郵便番号       DECIMAL(5),
 住所           NCHAR(30),
 PRIMARY KEY 従業員番号);

SELECT従業員番号
FROM 従業員
WHERE 郵便番号    = 100 AND
  給与  > 300000 AND
  従業員氏名   LIKE N'芝野 %';
```

Takahide Honma
Hiroyoshi Baba
Kuniaki Takizawa

Japanese Input Method Independent of Applications

The Japanese input method is a complex procedure involving preediting operations. An application that accepts Japanese from an input device must have three systems for the input method: a keybinding system, a manipulator for preediting, and a kana-*to*-kanji *conversion system. Various keybinding systems and manipulators accelerate input operations. Our implementation separates an application from the Japanese input method in three layers. An application can use a front-end input processor to perform all operations including I/O. An application can use the* henkan *(conversion) module and implement I/O operation itself. An application can execute all operations except keybinding, which is handled by an input method library.*

In this paper, we first present an overview of the technical environment of the Japanese input method implementation. Based on this overview, we briefly describe the critical engineering issues for conversion of Digital's products for the Japanese user. Our most critical engineering issue was the reduction of similar (but slightly different) work to localize products. Another issue was to satisfy customers' requests for the ability to use the many input styles familiar to them. We describe our approach to the development of a Japanese input method that overcomes these issues by separating the input method from an application in three layers.

Overview of the Japanese Input Method

In this section, we describe Japanese input and string manipulation from the perspective of both the user and the application. Based on these descriptions, we present a brief overview of reengineering a product for Japanese users and a summary of the industry's complex techniques developed for Japanese input methods.

Japanese Input

The Japanese writing system uses hundreds of ideograms called *kanji*. In addition, Japanese uses a phonetic system of *kana* characters (*hiragana* and *katakana*) and has accepted *romaji*, which is the use of Latin letters to spell Japanese words. Figure 1 summarizes the Japanese character systems. Japanese input requires users to operate in a "preediting" mode to convert *kana* or *romaji* into a *kana-kanji* string.[1,2]

The computer keyboard used for Japanese input has multiple character assignments. Almost all keys on the Japanese keyboard are assigned both a Latin alphabet character and a Japanese *kana* character. The Japanese user must first choose between *kana* key input or alphabet input. A user in an engineering

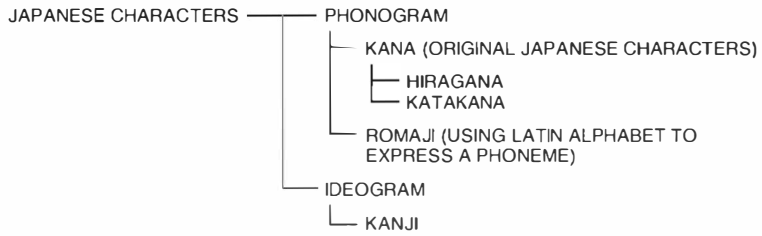


Figure 1 Japanese Character Systems

area generally uses *romaji* (alphabet) key input. In the office environment, however, a user prefers *kana* key input because it requires half as many keystrokes as *romaji* input.

Preediting Operation

The user inputs the phonetic expression in either *kana* or *romaji* that represents the statement the user wants to input. Then the user presses the conversion key to convert the phonetic expression to a *kana-kanji* mixed string. At this time, the user checks the accuracy of the conversion result. Sometimes the user needs to select the correct word from a system-generated list of several homonyms. Moreover, a user may also need to determine the separation positions in the phonetic expression to ensure a meaningful grammatical construction.

Japanese has no word separator equivalent to the space character in English. To obtain the correct or expected separation of grammatical elements, the user must sometimes move the separation position. After the user constructs a corrected statement, he or she finishes preediting and fixes the statement. The user repeats these complex steps to construct Japanese documents. Figure 2 shows the preediting steps for the Japanese user.

Various techniques have been developed to accelerate Japanese input operations. They include UNDO, COPY, zip code conversion, and categorized expert dictionary.

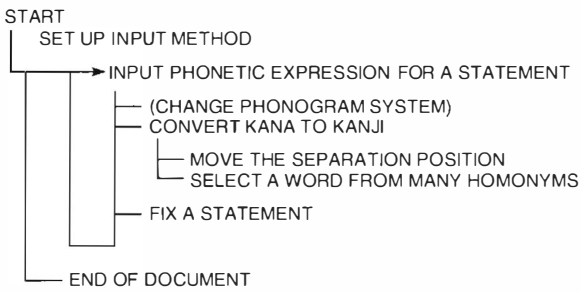


Figure 2 Preediting Japanese Input

Japanese Application Capabilities

The Japanese application has two special capabilities for Japanese processing. First, the application must be capable of handling multibyte characters. This subject itself is interesting as it involves wchar_t and Unicode character sets; however, this paper focuses on the second capability, the implementation of the input method. An application that accepts Japanese from an input device must have, at least, three additional systems for the input method. These are the so-called keybinding system, a manipulator for preediting, and the *kana*-to-*kanji* conversion system.

Keybinding System This system analyzes the key input from a user and determines which of the key's functions the user wants to do. It defines the user interface and the way a user operates with keystrokes. It also defines the preediting conversion key. We imagine there are as many keybinding systems as there are word processors.

Preedit Manipulator System This system not only echoes the input characters on the screen but also controls the video attribute that expresses the preediting area. This capability allows the user to distinguish preediting strings from background fixed strings. The user must be able to recognize the preediting string for more processing (for example, to convert the input to another expression such as *kana* to *kanji*). In addition, the user can set this system to convert input to another expression dynamically (for example, automatic conversion of *romaji* to *kana*).

Kana-to-kanji Conversion System This system analyzes the input string, gets the word from a dictionary, and constructs the correct statement grammatically. Many personal computer (PC) vendors have invested in systems that use this input method. In Japan, some vendors have introduced artificial intelligence technology, but this system essentially has only statistical rules.[3,4]

Figure 3 summarizes Japanese processing as handled by applications.

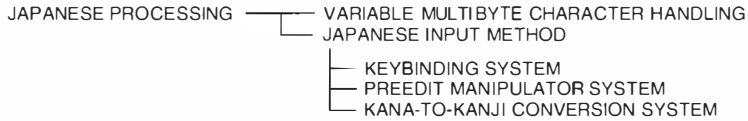


Figure 3 Japanese Processing by Applications

Method of Japanese Conversion

As mentioned above, to convert a product for use in Japanese, we must implement both a Japanese-string manipulator and an input method. To retain the "look-and-feel" of the original Digital product, the interface is designed so the Japanese user does not need to explicitly enter the preediting session with the special-purpose key (Enter Preedit) but is automatically entered. With most applications on other systems, a user must explicitly enter the preediting session by using the special keystroke. This implementation has the advantage that it completely separates the input method from the application, but it requires the user to remember to perform an extra step.

To eliminate the conflict between the original product's key function and the additional Japanese input function, each product has to have a slightly different keybinding system for Japanese. As a result, a user must learn more than one Japanese input operation when using multiple products.

User Environments

PCs are widely used in many offices and are popular devices for Japanese input. Naturally a user wants to operate with a familiar PC keystroke for Japanese input even in integrated systems (in some servers). When PCs, which use front-end processors, are integrated into environments with VMS and UNIX systems, a user often prefers the PC's interface. The more integrated a user's environment is, the more requirements a user has.

In addition, a distribution kit for the X Window System in a UNIX environment has some sample implementations of the Japanese input method. This kit gives a user more choices for input at no cost.

The market for the Japanese input method separates vendors into two main groups. One is the PC front-end processor manufacturer who implements more advanced techniques but at a high price. The other is the UNIX system vendor who supplies input implementations free (without guarantee) and thus reduces the maintenance cost.

In the next section, we present our approach to the development of an application-independent Japanese input method. The goals of our design were (1) to include the PC keybinding system in integrated environments so users could select their preferred input method, (2) to supply a tool that would easily convert products for the Japanese user, and (3) to provide a way to access the interfaces of several Japanese engines and thus capture the free software capabilities.

Application-independent Approach

As described in the previous section, the Japanese input method includes complex techniques. Many PC software vendors (but not manufacturers) decided against developing their own methods and incorporated a popular input method for their applications. This decision, of course, reduces their development cost. Our approach also seeks to reduce development cost. We separated the input method from the application to the greatest extent possible, as long as the separation did not adversely affect the application.

The PC system is designed as a single-task system, but Digital's operating systems (OpenVMS, ULTRIX, and DEC OSF/1 AXP) are designed as multitasking systems. Therefore we could not adopt many of the PC techniques that were implemented in the driver level. For example, access to dictionary and grammatical analysis of the input string are too expensive in the driver level of a multitasking system because they use system resources that are common to all tasks on the system.

Our approach divides the input method into three layers. Each layer is dependent on any lower layer. Consequently, any application using the highest layer also uses the functionalities of the other two layers.

Strategy of Three Layers

The criteria of our layering strategy were (1) to reduce the cost of reengineering products for the Japanese user, (2) to unify the input method user interface, and (3) to protect the user's operational investment in a keybinding system.

These criteria led us to set the keybinding system into the lowest layer. We named our system the input method library (IMLIB) and released it on VMS/Japanese version 5.5 and ULTRIX/Japanese version 4.3. We also ported it to the Alpha AXP system, and this facility is available on any Japanese platform. Any application using our method needs to use IMLIB.

In essence, this keybinding system allows a user to change the input method of operation to any style by changing the IMLIB definition files. If an application supports IMLIB, a user can change the application's input operation by changing IMLIB once. As a result, an application's key customization function can move into IMLIB.

At this point, we considered the simplest method of separating the input method from applications. The intermediate process, also called the front-end method, processes all the input and then passes it to an application. Many front-end implementations use the pseudo-terminal driver (pty in UNIX or FT in OpenVMS). The intermediate process gets all I/O to and from an application, processes it, and finally passes it to an application or a device. This implementation cannot recognize the application input requests, but works only by a user's operation. To change this operation, we set the hook inside the terminal driver to get all application-request information. Our front-end process recognizes application requests and works without conflict.

One advantage of this front-end implementation is a complete independence of applications. This can also be a disadvantage since an application cannot control the input method closely. For example, this implementation can alter the user interface of an editor system.

We continued to study another layer for separation. The prediting operation, that is, all the input string manipulations except I/O to devices, was a candidate. All applications pass the input from input devices into the Japanese input manipulator and then pass the output from this manipulator onto output devices. By using this system, we can unify the input operation except for device-dependent operations and reduce the cost to implement this kind of functionality.

Our development process started at the lowest layer (IMLIB), proceeded to the highest layer (front-end), and finished at the middle layer (prediting manipulator). In the following sections, we describe the functionalities in each layer from the lowest to the highest layer.

Implementation of IMLIB

IMLIB is a utility that supplies the keybinding definition function and other information for customizing the Japanese input operation. This capability enables us to supply user-friendly keybinding systems. A user can change the input sequences and the look-and-feel of the user interface by modifying databases. We introduced two databases, KEYBIND for keybinding and PROFILE for look-and-feel and an application's usage. We also supplied the KEYBIND compiler for improved performance and the elimination of the grammatical error at run time.

As mentioned in the introduction, there are many implementations of Japanese input styles on PCs or some word processors, and some text editors on various operating systems. If a user needs to use a different editor, he or she needs to learn another operation. Our method unifies the input operation. We studied several types of input styles and recognized that we could build the general model for this input operation. The IMLIB manual describes this model in detail.[5,6] In this paper, we discuss it briefly.

KEYBIND Database

In the Japanese input operation, entering the key input causes several conversion actions and state transitions. Figure 4 shows the multiple transitions incurred during input. We needed to define the conversion actions and some state transitions as a single key input action. We implemented this function through the KEYBIND database and language. Figure 5 is an example of the KEYBIND database. A user builds an input style by changing the KEYBIND database with the KEYBIND language.

IMLIB allows the user to change the keybinding and to choose a different input sequence with a different state transition vector. For the user's convenience, IMLIB provides some KEYBIND databases of the major Japanese input styles in default.

When an application calls the ImSetKeybind function, it loads a KEYBIND binary file into memory. Each time the application gets the key input, it queries the key's function from IMLIB. IMLIB searches the KEYBIND file for the key's definition and returns that information, called an action, to the application. Each action is a set of orders that has a different procedure for Japanese conversion. For example, the action CONVERT means to convert an input string to a *kanji* string. At that time, IMLIB also maintains Japanese input states and, if necessary, changes the state.

PROFILE Database

The Japanese input operation has many parameters to determine its look-and-feel, such as the video

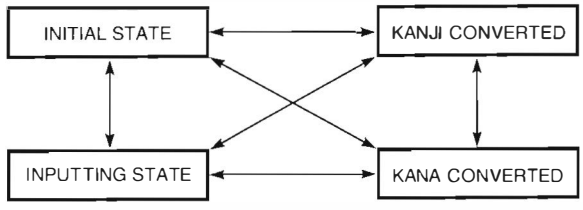


Figure 4 State Transition

```
        !
        ! JVMS 変換キー定義ファイル (システムテンプレート) v1.0
        ! (JVMS conversion key definition file (system template) ver 1.0)
        !
            gold            = CTRL_G;               ! Gold key; used as a PREFIX key
            kakutei         = CTRL_N;               ! Finish without any conversion
            kanji_henkan    = NULL, gold + CTRL_K;  ! Convert to Kanji / next candidate
            hiragana_henkan = CTRL_L;               ! (Convert clause) to Hiragana
            katakana_henkan = CTRL_K;               ! (Convert clause) to Katakana
            zenkaku_henkan  = CTRL_F;               ! Convert to full width characters
            hankaku_henkan  = gold + CTRL_F;        ! Convert to half width characters
            kigou_henkan    = GS;                   ! Symbolic code conversion
            oomoji          = VOID;                 ! Convert to upper characters
            komoji          = VOID;                 ! Convert to lower characters
            ji_bunsetsu     = CTRL_P;               ! Move to next clause
            zen_bunsetsu    = gold + CTRL_P;        ! Move to previous clause
            tansyuku        = US;                   ! Shrink the clause
            sintyou         = gold + US;            ! Extend the clause
            zen_kouho       = gold + (NULL, CTRL_L);! Previous clause candidate
            kaijo = CTRL_N;                         ! Cancel the conversion
                                                    !   and go into input state
            sakujo          = DEL;                  ! Delete previous character
            hidari          = LEFT;                 ! Move the cursor left
            migi            = RIGHT;                ! Move the cursor right
            space_first     = "¥ ";                 ! Finish by space
                                                    !  (space at initial state)
            space_input     = "¥ ";                 ! Finish by space
                                                    !  (space at other states)
STATE "initial" =
    space_first           : NONE;
    kanji_henkan          : START_SELECTED, CONVERT, GOTO "kk_converting";
    hiragana_henkan       : START_SELECTED, HIRAGANA, GOTO "converting";
    katakana_henkan       : START_SELECTED, KATAKANA, GOTO "converting";
    zenkaku_henkan        : START_SELECTED, ZENKAKU, GOTO "converting";
    hankaku_henkan        : START_SELECTED, HANKAKU, GOTO "converting";
    kigou_henkan          : START_SELECTED, SYMBOL, GOTO "converting";
    oomoji                : START_SELECTED, UPPER, GOTO "converting";
    komoji                : START_SELECTED, LOWER, GOTO "converting";
    TYPING_KEYS           : START, ECHO, GOTO "inputting";
    END;
          ...
```

Figure 5 Portion of KEYBIND Database

attribute for the preediting string, preediting exception handling, and application-specific processing. The PROFILE database stores these additional parameters the same way as the resource file does in the X Window System.

The PROFILE database is a text file. It contains several records that represent each environment. This record format has the style of INDEX : value. The application predefines the INDEX for its purpose; however, IMLIB defines some INDEXes related to Japanese input operation because it requires some common environment definitions. The range or value corresponding to the INDEX is placed in the right-hand side of the record. Figure 6 shows a record from a PROFILE database.

KEYBIND Compiler

The KEYBIND compiler analyzes the KEYBIND text file and creates the KEYBIND binary file. IMLIB services reads the PROFILE database and the KEYBIND binary file and maintains them in memory. As a response to an application's query, IMLIB services sends it the actions in KEYBIND and the data in PROFILE and at this time maintains the KEYBIND states. Figure 7 shows the relationship among the IMLIB components.

IMLIB is available on the OpenVMS VAX, OpenVMS AXP, ULTRIX, and DEC OSF/1 AXP operating systems. The major applications are DECwindows/Motif, DECwrite, the front-end input process, and screen management (SMG).

```
DEC-JAPANESE.KEY.keybind : im_key_jvms_level2
DEC-JAPANESE.KEY.keybind_1 : im_key_jvms
DEC-JAPANESE.DISP.preEditRow : current
DEC-JAPANESE.DISP.preEditColumn : current
DEC-JAPANESE.DISP.inputRendition : bold
DEC-JAPANESE.DISP.kanaRendition : bold
DEC-JAPANESE.DISP.currentClauseRendition : reverse
DEC-JAPANESE.DISP.leadingClauseRendition : none
DEC-JAPANESE.DISP.trailingClauseRendition : none
DEC-JAPANESE.ECHO.ascii : hankaku
DEC-JAPANESE.ECHO.kana : hiragana
DEC-JAPANESE.ECHO.autoRomanKana : off
DEC-JAPANESE.OUTRANGE.clauseSize : none
DEC-JAPANESE.OUTRANGE.clauseNumber : rotate
DEC-JAPANESE.OUTRANGE.cursorPosition : done
```

Figure 6 PROFILE Database Record

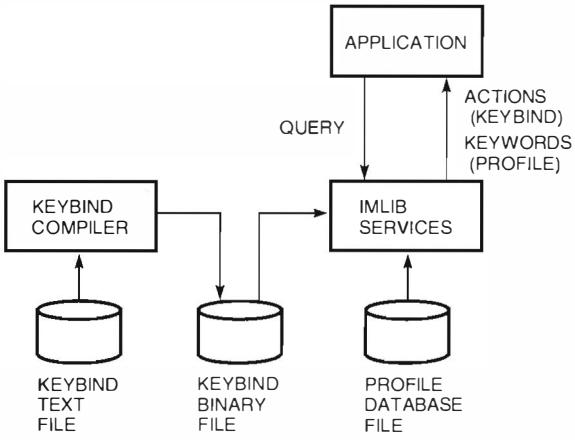


Figure 7 Relationship among IMLIB Components

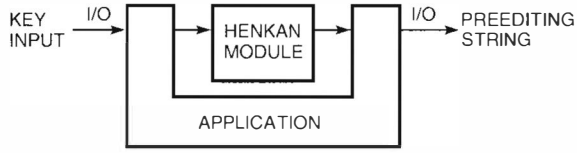


Figure 8 Henkan Module Function

Implementation of the Henkan Module

The second layer is part of the Japanese input manipulator and is called the *henkan* module or HM. (*Henkan* means conversion in Japanese.) It does not handle I/O operation but accepts key input from the application and converts it to a string in preediting.

Figure 8 summarizes the function of HM. An application passes the key input to HM stroke by stroke. HM performs all Japanese preediting operations; the application has no direct manipulation of the input. Then the application gets the preediting string from HM. Because HM does no I/O, it is independent of any specific device. As a result, all applications, including windowing systems, can use HM.

In addition, HM can access the resources of IMLIB. This feature helps the unification of the Japanese input user interface and reduces the number of similar product conversions. HM has another significant capability. We defined the common (minimum) application programming interface to potentially accept all Japanese conversion engines and implemented "PLUGGS" in HM. Therefore HM can use one or more engines for *kana*-to-*kanji* conversion.

HM Mechanism Overview

HM is a tool that any application can use. An application passes key input to HM by a normal procedure call. After HM processes it, HM calls application routines with the processed result. Because HM handles large string buffers, it dynamically allocates/deallocates memory. To ensure that memory is retained, we used a callback technique. (These techniques are described later in the Callback Routines section.)

HM operates by key input as follows:

1. HM gets a keycode from an application with procedure arguments.
2. HM gets the actions assigned to the key from IMLIB.

3. If the key is not assigned to the Japanese input operation, HM tells the application to process it separately.
4. If the key is assigned to the Japanese input operation, HM processes it according to the actions.
5. HM modifies the information to be displayed according to the action and calls a registered callback routine to update the screen.

HM passes the information that should be displayed on the screen in an argument of the callback routines. The callback routines are prepared by the application and registered into HM context at the initialization of HM. This callback method makes the application interface and data flow more easily.

Components

Figure 9 shows the composition of HM. The application interfaces include both the C and the VMS binding interfaces for the OpenVMS operating system.

The Japanese input manipulator performs all Japanese input operation by using IMLIB, the *romaji*-to-*kana* converter, and the *kana*-to-*kanji* converter. After it processes the input key, it calls back the application routines. There are several types of *romaji*-to-*kana* converters. We implemented a submodule *romaji*-to-*kana* converter driven by a conversion table; a user can change this table to another.

The *kana*-to-*kanji* converter module is a generalized Japanese conversion library. Many Japanese conversion engines exist, and each one is used differently. The *kana*-to-*kanji* converter loads the interface routine that absorbs these differences dynamically at the initialization of the HM context. It then processes the conversion request with any engine.

Services

HM provides 17 library entries. In this section, we describe three basic routines: HMInitialize, HMConvert, and HMEndConversion.

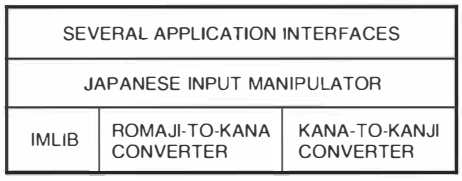


Figure 9 HM Component Structure

- HMInitialize. This routine creates a context for HM. It accepts three callback entries, a user-defined data pointer that would be passed to the callbacks, and an item list for initial information as its arguments.
- HMConvert. This routine sends a key to HM. The key is represented as a 32-bit data (longword) that is generated by a function HMEncodeKey from an escape sequence that the keyboard sends or by a function HMKeysymToKeycode from a keysym of the X Window System. IMLIB interprets the keycode, and HMConvert performs a conversion in accordance with the information. (A summary of what is executed was given in the Mechanism Overview section.)
- HMEndConversion. This routine aborts the conversion and resets an internal status. It is used when the application has to stop the input for a particular reason, for example, if an application issues the cancel request.

Callback Routines

HM requires three callback routines: start_conversion, format_output, and end_conversion. They are used as follows.

- start_conversion. This routine is called when the conversion string input is started. The application memorizes where the cursor is positioned.
- format_output. This routine is called whenever the information to be displayed has been changed. The application updates the screen.
- end_conversion. This routine is called when the input string is determined. As a result, the application takes the string passed in the argument of the last call of format_output into its input buffer.

The user-defined data pointer, one of the arguments for HMInitialize, is always passed to these callbacks. Since HM is not concerned with its contents, the user can put any kind of information into it.

HM is available on the OpenVMS VAX, OpenVMS AXP, ULTRIX, and DEC OSF/1 AXP operating systems. This portability is due to the module's independence from physical I/O. The major client applications working on these operating systems are DECwindows/Motif, Japanese SMG, and the front-end input process.

Implementation of the Front-end Input Processor

The front-end input process (FIP) for a dumb terminal supports full operations for the Japanese string manipulation. FIP is implemented on the following operating systems: OpenVMS/Japanese/VAX version 5.5-2 or later versions and OpenVMS/Japanese/AXP version 1.0 or later versions.

Full Operation Support

The original product can use FIP if the product's mechanism, particularly its I/O operation and preediting function, does not conflict with the FIP implementation. Some applications conflict with the design of FIP due to the limitations of FIP and its environment. For example, FIP does not detect the read request that includes the NOECHO item code, so the application that issues such a read request to the terminal driver (TTdriver) cannot use FIP as a Japanese front-end input process. Also FIP does not step into a process for the termination of a read request simply because a read buffer that is defined by an application has overflowed. FIP continues to communicate with the TTdriver and a conversion engine to get the Japanese string unless the terminate key is explicitly input. To overcome these conflicts, we implemented a pseudo-driver named FIdriver to intercept I/O requests from the application before they are processed by the TTdriver.

FIP Mechanism Overview

FIP processes all Japanese input operations using HM. We supplied the Digital Command Language (DCL) command, INPUT START/STOP for activating/deactivating FIP. Once a user activates FIP from DCL, it is available until the user logs out or the system is deactivated.

Figure 10 shows FIP and its environment for the manipulation of Japanese input. An application issues I/O requests to the TTdriver to get user inputs, but FIP fetches the requests from the TTdriver through the FIdriver. Then FIP starts to communicate with the drivers and the Japanese string conversion engine to pass the resultant string as well as preedits to a screen.

The sequence of the front-end input process follows.

1. An application creates a front-end input process.
2. A front-end input process exchanges packets with an application through its mailbox.
3. An application issues a queued I/O ($QIO) read request to the TTdriver.
4. The FIdriver intercepts the request and passes the information to FIP as a packet.
5. FIP issues a $QIO read request to the TTdriver to get input strings for conversion.

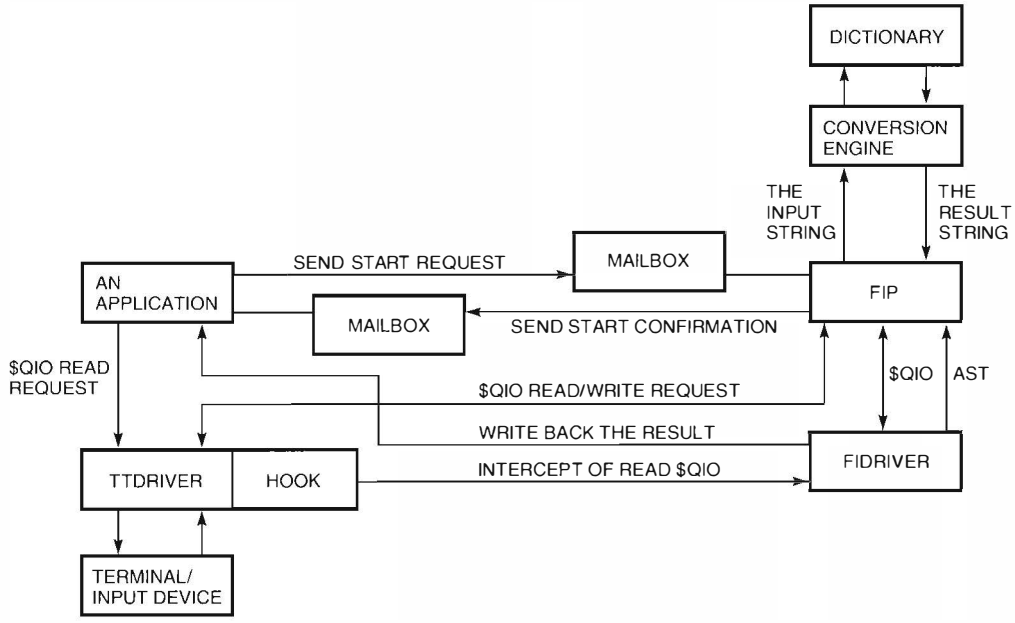


Figure 10 FIP Environment for Manipulation of Japanese Input

6. A user inputs a key from a terminal. FIP receives the input and decides whether or not to call a routine of the conversion engine. If an input key is recognized as one of the conversion keys, FIP calls the routine and passes the input strings. If not, FIP issues a $QIO write request to the TTdriver to echo an input character.
7. A conversion engine receives a string and converts it to the Japanese string.
8. A conversion engine returns the result to FIP.
9. FIP issues a $QIO write request to the TTdriver to display the resultant string from the engine and arranges the current editing line.
10. Steps 5 to 9 are repeated.
11. Once a user inputs the Terminate key of an application's request, FIP recognizes it as a terminator and returns the entire resultant string to the FIdriver as a write packet.
12. The FIdriver sends the result string and I/O status to an application.
13. An application accepts the converted string. After executing its internal process, it issues another $QIO read request to the TTdriver. (Return to step 3.)

FIdriver The FIdriver is a pseudo-driver that intercepts $QIO read requests from an application to the TTdriver. Functioning as a bridge between terminal read requests and FIP, the FIdriver gets a read request, passes its information to FIP, and maintains it. When FIP returns the completion message with its processed Japanese string, the FIdriver validates it and completes a user's read request as if the TTdriver had returned it. Thus the user/application can get the Japanese string without modification for Japanese input method.

The FIdriver has other notification functions for exception handling such as logout, cancel, or abort.

Front-end Input Process Operations All the operations in the front-end input process are driven by the mailbox event, the FIdriver event, and the key event. Figure 11 shows the functional structure of FIP.

The following operations in the front-end input process correspond to these three events.

- Mailbox Event. The mailbox event provides communication with an application. FIP issues a read request to its own mailbox. The mailbox event notifies FIP of the arrival of a message from an application. When an application sends a start request to the FIP mailbox, the mailbox event is set so FIP starts to initialize its environment. Also FIP terminates itself at the time a stop request message is delivered to its mailbox.
- FIdriver Event. The FIdriver event provides communication with the FIdriver. The FIdriver intercepts a request from an application to the TTdriver and creates a packet for FIP. FIP issues a read request to the FIdriver, and this event is set when a packet is delivered. A request is categorized in three types: read request, cancel request, and disconnect request.
- Key Event. The key event provides communication with the TTdriver. FIP issues a $QIO read request to the TTdriver byte by byte. All the input from a keyboard is recognized as a key event in FIP. Once a key event is set in FIP, FIP examines the key sequence in a read buffer.

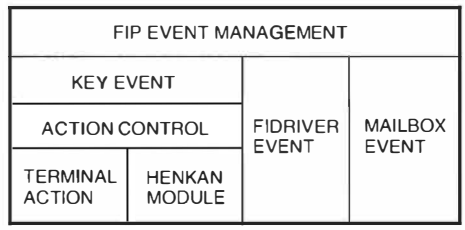


Figure 11 FIP Functional Structure

If the input is in the range of a terminator mask, FIP terminates a read operation from the TTdriver and writes back the resultant string and I/O status block to the FIdriver as a write packet. (A terminator mask is defined in the $QIO read request from an application.)

If the input key is a conversion key, FIP calls a conversion engine and gets the resultant converted string. Then FIP issues a write request to the TTdriver to display the updated string.

If the input key is a printable character, FIP updates the contents of its internal buffers defined in the context and issues a write request to the TTdriver to echo the character.

If the input key is for line editing, for example, to delete a line or a word or to refresh a line, FIP emulates the line-editing function of the TTdriver so its editing function is executed.

FIP stores all user input and read-request information from an application in its internal buffers and database. The buffers contain the codes of user input and corresponding video attributes to display. The database contains item codes in a read request, channel numbers to connect other devices, and so on.

FIP creates a new database when the updated read request from an application is delivered, in other words, when the FIdriver event is set. Also, FIP adds the ASCII code and an attribute of the updated user input into buffers when a user inputs, that is, when the key event is set.

Client/Server Conversion

The use of a client/server conversion has two advantages: (1) It reduces the required resources for language conversion by distributing some components to other systems, and (2) It presents an environment that shares a common dictionary.

All procedures for the Japanese conversion require large system resources such as CPU power. A user can place the conversion information server (CIserver) and a dictionary on a remote node and call some functions of the CIserver client library to get the resultant string. In this way, a local system saves its resources while the remote server processes the conversions.

In addition, many users can access a common dictionary on the specific remote node. It is possible for any local user to access a dictionary on a remote node if the CIserver on the node is active.

CIserver

The object name is "IM$CISERVER". The CIserver initializes itself by finding the name of a transport protocol in a logical table. It then creates corresponding shareable images, maps its required routines, and waits for a connect request from a client. The CIserver communicates with its client via a mailbox at the transport level. The server sets the asynchronous system trap to the mailbox and reads a message in it such as a connect request, a disconnect request, a connect abortion, or a client's image termination. The CIserver can identify the connection to a client and specify a conversion stream in the connection.

CIserver Client Library The client library presents programming interfaces. These are callable routines that execute various string manipulations and operations for the Japanese conversion. The CIserver client library is located between an application and the CIserver body.

Input Method Control Program (IMCP) IMCP is a command line interface to customize the CIserver environment. A user sets proxy to a Japanese system dictionary at a remote node on the network, and IMCP administrates a proxy database. A user can confirm the status of the server at a command line and can shut down the server from the IMCP interface.

Other Servers HM has a conversion engine dispatcher that can dynamically select from several Japanese conversion engines. HM now serves the CIS (CIserver, Digital Japan), the Wnn (Omron Company), the Canna (NEC), and the JSY (Digital Japan) engines. Therefore, an application that uses HM as the Japanese conversion interface can select its preferred engine.

Extension in the Future

In this section, we describe the possibilities for internationalization of FIP, HM, IMLIB, the CIserver, and the FIdriver. Although our approach does not provide a multilingual input method, it does provide an architecture that can be used for any language.

FIP has a multibyte I/O operation that can be applied to other 2-byte languages. In addition, all the read/write communications among FIP, the FIdriver, and the TTdriver proved able to handle one-byte languages such as English. Also, IMLIB can expand its keybinding system for conversion of other languages, and HM can add the interfaces for conversion engines of other languages if such engines are prepared.

Summary

The Japanese input method is a complex procedure involving preediting operations. Various keybinding systems and manipulators accelerate input operations. Our approach for the Japanese input method allows an application three choices: (1) An application can use a front-end input processor to perform all operations including I/O. (2) An application can use the *henkan* module and implement I/O operation itself. (3) An application can execute all operations except keybinding, which is handled by an input method library.

Acknowledgments

We want to express our appreciation to Katsushi Takeuchi of the XTPU development team for his initial designing and prototyping of IMLIB and some implementation of FIP, and Junji Morimitsu on the same team for his initial implementation of IMLIB and its compiler. Also, we wish to thank Makoto Inada on the DECwindows team for his implementation of HM; Hitoshi Izumida, Tsutomu Saito, and Jun Yoshida from the JVMS driver team for their contribution toward creating the FIdriver; and Naoki Okudera for his implementation to the entire CIserver environment. As a final remark, we acknowledge Eiichi Aoki, an engineering manager of ISE Japan, and Hirotaka Yoshioka in the ISA group for their encouragement in writing this paper.

References

1. *Guides to the X Window System Programmer's Supplement for Release 5* (Sebastopol, CA: O'Reilly & Associates, Inc., 1991).
2. *Standard X, Version 11, Release 5* (Cambridge, MA: MIT X Consortium, 1988).
3. K. Yoshimura, T. Hitaka, and S. Yoshida, "Morphological Analysis of Non-marked-off Japanese Sentences by the Least BUNSETSU's Number Method," *Transactions of Information Processing Society of Japan,* vol. 24 (1983).
4. K. Shirai, Y. Hayashi, Y. Hirata, and J. Kubota, "Database Formulation and Learning Procedure for Kakari-Uke Dependency Analysis," *Transactions of Information Processing Society of Japan,* vol. 26 (1985).
5. *IMLIB/OpenVMS Library Reference Manual* (in Japanese) (Tokyo: Digital Equipment Corporation Japan, Order No. AA-PU8TA-TE, 1993).
6. *User's Manual for Defining User Keys in IMLIB* (in Japanese) (Tokyo: Digital Equipment Corporation Japan, Order No. AA-PU8UA-TE, 1993).

Further Readings

The Digital Technical Journal *publishes papers that explore the technological foundations of Digital's major products. Each* Journal *focuses on at least one product area and presents a compilation of refereed papers written by the engineers who developed the products. The content for the* Journal *is selected by the Journal Advisory Board. Digital engineers who would like to contribute a paper to the* Journal *should contact the editor at RDVAX::BLAKE.*

Topics covered in previous issues of the *Digital Technical Journal* are as follows:

Multimedia/Application Control
Vol. 5, No. 2, Spring 1993, EY-P963E-DP

DECnet Open Networking
Vol. 5, No. 1, Winter 1993, EY-M770E-DP

Alpha AXP Architecture and Systems
Vol. 4, No. 4, Special Issue 1992, EY-J886E-DP

NVAX-microprocessor VAX Systems
Vol. 4, No. 3, Summer 1992, EY-J884E-DP

Semiconductor Technologies
Vol. 4, No. 2, Spring 1992, EY-L521E-DP

PATHWORKS: PC Integration Software
Vol. 4, No. 1, Winter 1992, EY-J825E-DP

Image Processing, Video Terminals, and Printer Technologies
Vol. 3, No. 4, Fall 1991, EY-H889E-DP

Availability in VAXcluster Systems/Network Performance and Adapters
Vol. 3, No. 3, Summer 1991, EY-H890E-DP

Fiber Distributed Data Interface
Vol. 3, No. 2, Spring 1991, EY-H876E-DP

Transaction Processing, Databases, and Fault-tolerant Systems
Vol. 3, No. 1, Winter 1991, EY-F588E-DP

VAX 9000 Series
Vol. 2, No. 4, Fall 1990, EY-E762E-DP

DECwindows Program
Vol. 2, No. 3, Summer 1990, EY-E756E-DP

VAX 6000 Model 400 System
Vol. 2, No. 2, Spring 1990, EY-C197E-DP

Compound Document Architecture
Vol. 2, No. 1, Winter 1990, EY-C196E-DP

Distributed Systems
Vol. 1, No. 9, June 1989, EY-C179E-DP

Storage Technology
Vol. 1, No. 8, February 1989, EY-C166E-DP

CVAX-based Systems
Vol. 1, No. 7, August 1988, EY-6742E-DP

Software Productivity Tools
Vol. 1, No. 6, February 1988, EY-8259E-DP

VAXcluster Systems
Vol. 1, No. 5, September 1987, EY-8258E-DP

VAX 8800 Family
Vol. 1, No. 4, February 1987, EY-6711E-DP

Networking Products
Vol. 1, No. 3, September 1986, EY-6715E-DP

MicroVAX II System
Vol. 1, No. 2, March 1986, EY-3474E-DP

VAX 8600 Processor
Vol. 1, No. 1, August 1985, EY-3435E-DP

Subscriptions and Back Issues

Subscriptions to the *Digital Technical Journal* are available on a prepaid basis. The subscription rate is $40.00 (non-U.S. $60.00) for four issues and $75.00 (non-U.S. $115.00) for eight issues. Orders should be sent to Cathy Phillips, Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460, U.S.A., Telephone: (508) 486-2538, FAX: (508) 486-2444. Inquiries can be sent electronically to DTJ@CRL.DEC.COM. Subscriptions must be paid in U.S. dollars, and checks should be made payable to Digital Equipment Corporation.

Single copies and past issues of the *Digital Technical Journal* are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent back issues of the *Journal* are available on the Internet at gatekeeper.dec.com in the directory /pub/DEC/DECinfo/DTJ.

Recommended Reading on Internationalization Topics

B. Comrie, editor, *The World's Major Languages* (New York: Oxford University Press, 1987).

F. Coulmas, *The Writing Systems of the World* (Oxford: Basil Blackwell, 1989).

J. DeFrancis, *The Chinese Language Fact and Fantasy*, Second Paperback Edition (Honolulu: University of Hawaii Press, 1989).

J. DeFrancis, *Visible Speech: The Diverse Oneness of Writing Systems* (Honolulu: University of Hawaii Press, 1989).

Digital Guide to Developing International Software (Burlington, MA: Digital Press, Order No. EY-F577E-DP, 1991).

S. Martin, *Internationalization Explored* (UniForum, 1992).

MultiLingual Computing is a publication of Worldwide Publishing Group, Clark Fork, Idaho, U.S.A. and is available on the Internet : Multilingual@Applelink.Apple.com

A. Nakanishi, *Writing Systems of the World*, third printing (Rutland, VT, and Tokyo: Charles E. Tuttle Company, 1988).

D. Taylor, *Global Software: Developing Applications for the International Market* (New York, Berlin, Heidelburg, London, Paris, Tokyo, Hong Kong, Barcelona, Budapest: Springer-Verlag, 1992).

The Unicode Standard, Version 1.0, Volume 1 (Reading, MA: Addison-Wesley Publishing Company, 1991).

The Unicode Standard, Version 1.0, Volume 2 (Reading, MA: Addison-Wesley Publishing Company, 1992).

Technical Papers by Digital Authors

R. Abugov, "From Trendcharts to Control Charts: Setup Tests for Making the Leap," *IEEE/SEMI International Semiconductor Manufacturing Science Symposium* (June 1992).

R. Al-Jaar, "Performance Evaluation of Real-Time Decision-Making Architectures for Computer-Integrated Manufacturing," *Robotics and Computer-integrated Manufacturing* (January 1992).

P. Anick and S. Artemieff, "A High-Level Morphological Description Language Exploiting Morphological Paradigms," *Proceedings of the 15th International Conference on Computational Linguistics* (August 1992).

P. Anick and R. Flynn, "Versioning a Full-text Information Retrieval System," *Fifteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (June 1992).

B. Archambeault, "A New Standard Radiator for Shielding Effectiveness Measurements," *IEEE International Symposium on Electromagnetic Compatibility* (August 1992).

A. Berti and V. Bolkhovsky, "A Manufacturable Process for the Formation of Self Aligned Cobalt Silicide in a Sub Micrometer CMOS Technology," *Proceedings of the Ninth International VLSI Multilevel Interconnection Conference (VMIC)* (June 1992).

G. Bock and D. Marca, "GROUPWARE: Software for Computer-Supported Cooperative Work," *IEEE Computer Society Press Tutorial* (January 1992).

C. Brench, "A Method to Improve EMI Shielding Predictions," *IEEE International Symposium on Electromagnetic Compatibility* (August 1992).

D. Byrne, "Accurate Simulation of Multifrequency Semiconductor Laser Dynamics Under Gigabits-Per-Second Modulation," *IEEE Journal of Lightwave Technology* (August 1992).

R. Collica, "The Effect of the Number of Defect Mechanisms on Fault Clustering and its Detection Using Yield Model Parameters," *IEEE Transactions on Semiconductor Manufacturing* (August 1992).

D. Davies and J. Pazaris, "Requirements for Optical Interconnects in Future Computer Systems," *SPIE International Symposium on Optical Applied Science and Engineering* (July 1992).

D. Dossa, "Above-Barrier Quasi-Bound Electronic States in Asymmetric Quantum Wells," *Physics Review* (March 1991).

D. Dossa, "Observation of Above-Barrier Quasi-Bound States in Asymmetric Single Quantum Wells by Piezomodulated Reflectivity," *Applied Physics Letters* (November 1991).

B. Doyle, C. Conran, and B. Fishbein, "Thermal Instability in P-channel Transistors with Reoxidized Nitrided Oxide Gate Dielectrics," *IEEE Fiftieth Device Research Conference* (June 1992).

B. Doyle and K. Mistry, "Hot Carrier Stress Damage in the Gate 'Off' State in n-Channel Transistors," *IEEE Transactions on Electron Devices* (July 1992).

R. Dunlop, "Design for Electronic Assembly," *Design for Manufacturability* (vol. 6 of the SME *Tool and Manufacturing Engineers Handbook* series) (January 1992).

M. Good, "Participatory Design of a Portable Torque-Feedback Device," *CHI '92 Conference Proceedings* (ACM Conference on Human Factors Computing Systems) (May 1992).

D. Krakauer and K. Mistry, "ESD Protection in a 3.3V Sub-Micron Silicided CMOS Technology," *IEEE Electrical Over Stress/Electrostatic Discharge Symposium Proceedings* (July 1992).

P. Martino, "Analysis of Complex Geometric Tolerances by Linear Programming," *ASME Computer in Engineering* (August 1992).

J. Oparowski and P. Terranova, "Material and Design Considerations of Flexible Signal Connectors for the VAX 9000 MCU," *ASM International 7th Electronic Materials and Processing Congress* (August 1992).

A. Philipossian and D. Jackson, "Kinetics of Oxide Growth during Reoxidation of Lightly Nitrided Oxides," *Journal of the Electrochemical Society* (September 1992).

K. Ramakrishnan, "Effectiveness of Congestion Avoidance: A Measurement Study," *IEEE Infocom '92* (May 1992).

K. Ramakrishnan, P. Biswas, and R. Karedla, "Analysis of File I/O Traces in Commercial Computing Environments," *ACM Sigmetrics* (June 1992).

Y. Raz, "The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment," *Proceedings of the 18th International Conference on Very Large Databases* (August 1992).

A. Rewari, "AI for Customer Service and Support," *IEEE Expert* (June 1992).

J. Rose, "Fatal Electromigration Voids in Narrow Aluminum-Copper Interconnect," Applied Phsyics Letters (November 1992).

W. Samaras, "Futurebus+ Electrical Behavior for High Performance," *BUSCON '92 West Conference Proceedings* (February 1992).

M. Sayani, "DC-DC Converter Using All Surface-Mount Components and Insulated-Metal Substrate," *IEEE Seventh Annual Applied Power Electronics Conference* (February 1992).

H. Smith and W. Harris, "SIMS Quantification of AsCs+ at CoSi2/Si Interfaces," *Proceedings of the Eighth International Conference on Secondary Ion Mass Spectrometry (SIMS VIII)* (September 1991).

M. Stick, "Matrices and Vectors," *Six Sigma Research Institute* (April 1992).

R. Ulichney, "The Construction and Evaluation of Halftone Patterns with Manipulated Power Spectra," *Raster Imaging and Digital Typography (RIDT)* (October 1991).

G. Wallace, "The JPEG Still Picture Compression Standard," *Communications of the ACM* (April 1991).

G. Wallace, "Overview of the JPEG (ISO/CCITT) Still Image Compression Standard," *SPIE Image Processing Algorithms and Techniques* (February 1990).

Recent Digital U.S. Patents

The following patents were recently issued to Digital Equipment Corporation. Titles and names supplied to us by the U.S. Patent and Trademark Office are reproduced exactly as they appear on the original published patent.

D337,761	M. Hetfield and S. K. Morgan	Electronic Device Module
D338,001	M. J. Falkner, M. R. Wiesenhahn, and M. D. Good	Positioning Device
D338,653	M. Hetfield and S. K. Morgan	Power Supply Module
5,185,877	W. Bruckert	Protocol for Transfer of DMA Data
5,220,661	A. H. Mason, W-M. Hu, C. Kahn, and J. C. R. Wray	System and Method for Reducing Timing Channels Digital Data Processing Systems
5,224,263	W. Hamburgen	Gentle Package Extraction Tool and Method
5,225,790	P. Esling, J. M. Rinaldis, and R. W. Noguchi	Tunable Wideband Active Filter
5,226,092	K. Chen	Method and Apparatus for Learning in a Neutral Network
5,226,170	P. Rubinfeld	Interface between Processor and Special Instruction Processor in Digital Data Processing System
5,227,041	B. Brogden, L. Brown, and S. Husain	Dry Contact Electroplating Apparatus
5,227,582	J. Copeland and D. Robinson	Video Amplifier Assembly Mount
5,227,604	G. M. Freedman	Atmospheric Pressure Gaseous-Flux-Assisted Laser Reflow Soldering
5,227,778	G. Vacon	Service Name to Network Address Translation in Communications Network
5,228,066	C. J. Devane	System and Method for Measuring Computer System Time Intervals
5,229,575	L. Colella, R. Pacheco, and D. Waller	Thermode Structure Having an Elongated, Thermally Stable Blade
5,229,901	M. L. Mallary	Side-by-side Read/Write Heads with Rotary Positioner
5,229,914	D. A. Bailey	Cooling Device that Creates Longitudinal Vortices
5,229,926	D. Donaldson and D. Wissell	Power Supply Interlock for Distributed Power Systems
5,230,044	N. Quaynor and X. Cao	Arbitration Apparatus for Shared Bus
5,231,246	D. Alessandrini, J. M. Benson, and W. Rett	Apparatus for Securing Shielding or the Like
5,232,570	C. Byun, B. Haines, E. Johns, Q. Ng, G. C. Rauch, R. M. Raymond, and D. Ravipati	Nitrogen-Containing Materials for Wear Protection and Friction Reduction
5,233,616	M. Callander	Write-back Cache with ECC Protection
5,233,684	R. Ulichney	Method and Apparatus for Mapping a Digital Color Image from a First Color Space to a Second Color Space
5,235,211	W. Hamburgen	Semiconductor Package Having Wraparound Metallization
5,235,642	M. Abadi, A. Birrell, B. W. Lampson, and E. Wobber	Access Control Subsystem and Method for Distributed Computer System using Locally Cached Authentication Credentials
5,235,644	B. W. Lampson, C. Kaufman, W. Hawe, M. F. Kempf, J. Tardo, and A. Gupta	Probabilistic Cryptographic Processing Method
5,235,693	M. Gagliardo, J. J. Lynch, and P. M. Goodwin	Method and Apparatus for Reducing Buffer Storage in a Read-Modify-Write Operation

5,235,697	S. C. Steely and J. H. Zurawski	Set Prediction Cache Memory System using Bits of the Main Memory Address
5,237,662	T. L. Carruthers, K. Green, and S. Jenness	System and Method with a Procedure Oriented Input/Output Mechanism
5,239,260	D. Ringleb and D. C. Widder	Semiconductor Probe and Alignment System (SPAS)
5,239,493	S. K. Sherman	Method and Apparatus for Interpreting and Organizing Timing Specification Information
5,239,630	R. F. Lary and X. Cao	Shared Bus Arbitration Apparatus Having a Deaf Node
5,239,634	B. Buch and C. MacGregor	Memory Controller for Engineering/Dequeuing Process
5,239,637	D. W. Thiel, W. Goleman, and S. H. Davis	Digital Data Management System for Maintaining Consistency of Data in a Shadow Set
5,240,549	J. E. Fitch and W. Hamburgen	Fixture and Method for Attaching Components
5,240,740	K. A. Frey and M. L. Mallary	Method of Making a Thin Film Head with Minimized Secondary Pulses
5,241,564	J. L. Yang	Low Noise, High Performance Data Bus System and Method
5,241,621	R. Smart	Management Issue Recognition and Resolution Knowledge Processor
5,241,652	W. Barabash and W. Yerazunis	System for Performing Rule Partitioning in a RETE Network

digital™

ISSN 0898-901X