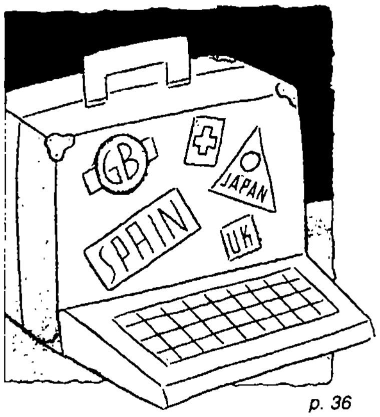
MAY 1992

VOLUME 1 NUMBER 5

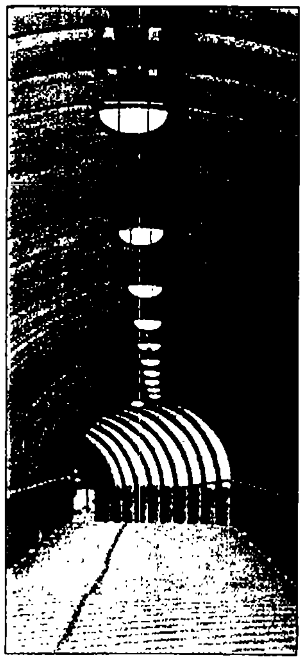
RS/Magazine

The Journal For IBM Workstation Users



p. 36

p. 40



columns illustrated by GREG CLARKE
cover photograph by KORY ADDIS/The Picture Cube

RS/Magazine is not affiliated with IBM Corp.

RS/Magazine is published monthly by Computer Publishing Group Inc., 1330 Beacon St., Brookline, MA 02146. Telephone (617) 739-7001. Third-class Postage Rates paid at Long Prairie, MN, and at additional mailing offices. Authorized as third-class mail by the U. S. Post Office, MN. This publication is free to qualified subscribers as determined by the publisher. Subscription rates are $49.50 per year in the United States, and $70.00 abroad. Subscription requests can be sent to Circulation Department, RS/Magazine, 1330 Beacon St., Brookline, MA 02146 or electronically mailed to circ@expert.com. POSTMASTER, please send all address changes to RS/Magazine, Circulation Department, 1330 Beacon St., Brookline, MA 02146. Please allow 6-8 weeks for change of address. Include your old address as well as new-enclosing, if possible, an address label from a recent issue. All rights reserved. © Copyright 1992, Computer Publishing Group Inc. No part of this publication may be transmitted or reproduced in any form by any means without permission in writing from the publisher. Material for publication should be sent to the attention of Anne Knowles at the above address or electronically mailed to aknowles@expert.com. Letters sent to the publication become the property of the publication and are assumed to be intended for publication and may be used so. All information herein is believed to be accurate to the best of our ability

RS/Magazine
The Journal for IBM Workstation Users

publisher
S. HENRY SACKS

editorial director
DOUGLAS PRYOR

editor
ANNE KNOWLES

art director
JOHN W. KELLEY JR.

senior editor
JANE MAJKIEWICZ

technical editor
BARRY SHEIN

research editor
MAUREEN MCKEON

production editor
MARY ANNE WEEKS MAYO

marketing manager
SUSAN R. SACKS

design director
STEVEN LEE

associate designer
HANNA DYER

production director
RICHARD M. ABAID

production manager
DEBORAH BEDLOW

circulation director
DEBORAH MOORE

circulation assistant
DIANNA DAPKINS

assistant to the publisher
LESLIE GAFFNEY

EDITORIAL OFFICES

1330 BEACON STREET
BROOKLINE, MA 02146
(617) 739-7001
email: aknowles@expert.com

Editorial

If It's Tuesday...

This issue of *RS/Magazine* introduces a new column called I18N. That's shorthand for internationalization, a term used to describe all the work being done "to allow software to handle other cultures, other natural languages and other alphabets." As this introductory column illustrates, transforming an application written in C and used by Americans into the same application to be used by a Japanese in Osaka or a Czech in Prague is daunting, to say the least. In fact, the differences between seemingly identical languages, like French spoken in France and Canada or German spoken in Germany and Switzerland, present enough problems to give any programmer a migraine. (I know just the thought of producing the Cyrillic alphabet or too many cedillas has caused a few headaches for our production department.) But as the article points out, there is a huge software market outside the United States and with a little patience (and the help of our new column) even the most fearsome task is possible. The column is written by Jeffrey Haemer and Jeffrey Copeland, both with Interactive Systems Corp., the original developer of AIX.

Our cover story, "Entering the Glass House," derives its title from the traditional IBM stronghold–the glass room where the mainframe holds court. PCs have been connected to the mainframe since their genesis. Now RS/6000 users are finding reasons to tap into the mainframe in much the same way PCs do. This article covers the various schemes for doing that and the IBM products and third-party software available to help you accomplish it. "Customizing SMIT," by Q&AIX columnist Dinah McNutt and Pencom's Paul Martin, uses a simple exercise to show readers how to tailor the RS/6000 systems managment tool. And Technical Editor Barry Shein reviews some popular products–dBASE IV, for example–for the RS/6000 in "Tried and True."

In "RS/News," check out IBM's new $96,000 high-end server. According to IBM, the system has hit an industry high with 100 SPECmarks.

[illegible]

Anne Knowles
Editor

Internationalization: An Introduction

by Jeffreys Copeland and Haemer

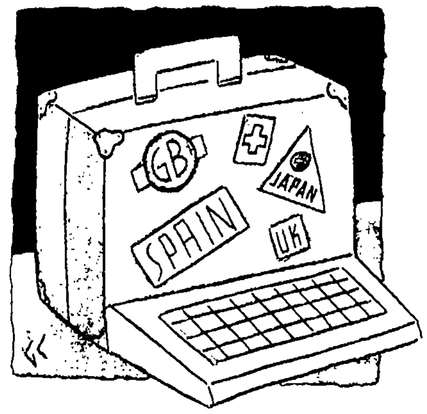
Consider this program:

```
#include <stdio.h>

main()
{
printf("hello, world\n");
}
```

There's not much question that it wouldn't be very interesting to the average non-English-language speaker. But who cares? First, most computer users are American, and of the ones who aren't, most speak English. Second, UNIX and C programmers are even more English-language-oriented, if that's possible. C, after all, has keywords in English, and everyone learns it from Kernighan and Ritchie.

These were good points once, but a bit behind the times now, especially for IBM. By 1993, the projected U.S. software market will be $26.4 billion, according to International Data Corp. IDC forecasts the 1993 Japanese software market will be $7.45 billion–not as large as the U.S. market, but still enough to make vendors sit up and take notice. And the European software market should be $25.9 billion. Moreover, for historical reasons, UNIX is much more important



relative to other operating systems in Europe and Japan than it is in the United States.

What to do, what to do? Must we now confront special compilers that expect programs like this to take advantage of the international UNIX market?

```
#incluye <estenex.h>

principal()
{
imprimef("¡hola! al mundo\n");
}
```

Well, not right away.

Our Purpose

In this column, we'll provide an overview of changes taking place in UNIX and C to allow software to

Jeffrey Copeland (jeff@itx.isc.com) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* troff *on every continent.* Jeffrey S. Haemer (jsh@ico.isc.com) *has worked at ISC's Colorado Technical Office since 1983. He has experience working, writing, and speaking on the interrelated topics of open systems, standards, software portability and porting, and internationalization. This year, Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

handle other cultures, other natural languages and other alphabets. These changes are spread across many parts of the system; accordingly, nonoverlapping changes to C and UNIX are being proposed, designed and pitched by numerous different organizations. Although their efforts influence one another, these organizations act independently, so we'll try to treat their contributions separately to leave you with a clear picture of which changes are due to the C standards committee, which to the POSIX committees, which to X/Open, which to the Unicode consortium and so on. There are a couple of reasons we want to do this:

1. There's no way to avoid hearing the organization names. You might as well get used to them. Besides, partitioning the topics by the group involved with them also provides handy mental organizers that will help subdivide an otherwise enormous topic.
2. Some organizations' proposals are inherently more stable than others. This doesn't just mean that some organizations are more contentious than others, though that's certainly true. It just points out that some of the organizations are official standards bodies, with industrywide participation and support, while others are vendor-sponsored consortia.

Naturally, we will concentrate on solutions either provided by AIX or likely (in our opinion) to appear in AIX in the future, but many of the things we say will also apply to other platforms.

Although the columns will strive to be practical, we'll try to avoid making the columns strict, how-to treatises. These columns are meant to be educational works, not reference works. We'll try to help you develop a sense of the basic problem being addressed, and a feeling for why the current solutions are what they are. Besides, if we make our articles too dry, you won't read them.

Software Localization

Application developers have been trying to sell software in markets outside the United States for a long time. After all, some of those application developers aren't even in the United States. And although programmers often speak English, customers often don't.

The simplest change to make is to provide screens, prompts and output messages in the local language, and to let the programs accept input in that language as well:

```
#include <stdio.h>

main()
{
printf("¡hola! al mundo\n");
}
```

Such an approach assumes that the engineers speak English, but the local customers–ordinary end users–need not.

Localization–tailoring software for new locales–presents a couple of technical problems. The first is accepting and displaying the right characters. Even if a Spaniard recognizes buenos dias as meaning buenos días, what do we do about Greek or Russian? A transliteration? pleos instead of πλεοσ? End users may not know how their alphabet maps into ours. Besides, a customer-billing package that will accept and print names spelled correctly is certain to sell better than one that drops diacriticals or transliterates characters.

The IBM PC provides many characters that support non-English alphabets, and most MS-DOS software accepts these characters, but important parts of UNIX have historically assumed an ASCII character set, and ASCII was built to display English. Even MS-DOS balks at languages that are written quite differently from English, like Japanese, Arabic or Hindi.

The second problem is more subtle: Some application features have cultural or linguistic dependencies that UNIX or C were not designed to handle.

```
printf("%5.3f\n", 5.0/2.0);
```

prints

```
2.500
```

In much of Europe this means two thousand five hundred, and 2,500 means two and a half. What this means is that some support for software localized for non-American markets must be provided by C itself.

Early AIX Localization

IBM responded early to demands for national language support–support of languages other than English–in UNIX, coming out originally (1987) with a version of AIX on the RT that supported a wide variety of European characters, and later with a version of AIX that handled Japanese characters. Both versions had interestingly expanded character sets, together with a miscellany of other changes to support the countries, languages and cultures of their target markets. Significantly, both versions went out of their way to be compatible with English-only versions of UNIX. This same strategy was in play for AIX 3.1 on the RS/6000.

What makes this compatibility interesting? Well, character-set sizes, for one. First, both versions used a much larger character set. To support a wide range of European languages, the European version added 488 characters. The Japanese version added even more; even moderately straightforward Japanese requires something like 50,000 glyphs. C's confusion between characters and bytes–C's keyword for a byte is char–turned this into an instant problem. There are only 256 different eight-bit bytes.

It's easy to think of schemes to represent large character sets: Put more bits in a byte, make all characters two bytes instead of one, provide escape sequences that let the same byte represent multiple characters, and so on. You can probably design two or three in under five minutes; most of the ones you come up with have been used by someone at some time. Many vendors, including IBM, chose to add the constraint that ASCII had to be a proper subset of the code set–the encoding of the characters–to ease data interchange with other, ASCII-only machines. Some characters had to be one byte long; others had to be more than one. Non-ASCII characters were recognized by either having the high bit set or by beginning with a special byte. (These two methods of recognizing non-ASCII differ: AIX chose to use the Japanese default PC standard, SJIS–shifted JIS–which ensures all bytes of a non-ASCII character have their high bits set. The IBM European character set reserves the single-byte characters for common text characters, and uses the roughly 500 two-byte characters for graphics, superscripts and the special-purpose characters that typesetters refer to as pi characters.)

A decision to use these new code and character sets cascaded into a variety of other design changes. Here are some examples:

- Printers and terminals had to be built that would print and display the characters in the character set. Drivers had to be written to drive those devices. Keyboards had to be made that had the right keycaps.
- Library routines had to be constructed to deal with variable-width characters.
- All programs that deal with data as chars needed to be examined to see if they were processing characters or bytes. (Sometimes the intent isn't entirely clear. What should the last column of wc show? Bytes or characters?)
- Programs that dealt with characters had to be recoded. Sometimes, changes were as simple as using new library routines–calling `NCstrlen()`, instead of `strlen()` to determine the length of a string in characters. Other routines called for more dramatic recoding. Quite a few routines keep and traverse arrays indexed by characters. Changing the size of the array from 128 (or 256) to 65,536 (256^2) often calls for new algorithms.
- As a special case, anything that dealt with the alphabet had to be rethought. Examples? The entire ctype library, and anything that uses it, for starters. What's an upper-case "é": "É" or "E"? (It depends, and not even on the language; in French, it's one, in French-Canadian, it's the other.) German scharfe-S, ß, is a lower-case letter with no upper-case equivalent. And what happens if you call `toupper()` on a Japanese character? The complement to that is missing functionality. The Japanese character set has at least four major pieces–Hiragana, Katakana, Kanji and Romaji–that the Japanese programmer will want to distinguish. A useful Japanese ctype library needed to provide new character classification functions.
- Worse than that was anything that had to do with sorting. Far too much code assumes that the order of the ASCII lower-case characters and the order of the alphabet are the same. Stick to English and you're fine, but there is no single character-set ordering that even works for all of Western Europe. This means that an entire mechanism had to be invented and installed for character collation.
- Allied to collation problems, though distinct from them, were problems with regular expressions. Not only were the algorithms hopelessly tied to small character sets (transition tables went from 256-by-256 arrays to 65,536-by-65,536 arrays!), but the shorthands we're all used to were no longer adequate. Should [a-z] span the lower-case letters if "z" isn't the last letter in the alphabet? How do I look for any variety of "a," including "á," "à," "â" and so on? Every program that handled some variety of regular expression, from the editors to awk to the shells to `find`, had to be retooled.

Those, of course, are just character-set related changes. Other changes, alluded to above, were also required: Changes to library functions to allow proper formatting of numbers, currency, dates. (In Europe, you just need different date formats. In Japan, the year depends on when the emperor was enthroned.) Changes to formatting routines that assume that white space delimits words (not true in Japan!). Changes to anything that controls the screen that assumes that all glyphs–the displayed versions of characters–are the same width (Japanese mixes single- and double-width glyphs) or character widths are the same as glyph widths (true in SJIS, but not in the AIX European character set). Changes to the documentation.

And messages. We almost forgot.

Pick a Language, Any Language

To make a localized UNIX backward-compatible with other UNIX versions requires that many program messages appear in English. (Think of shell scripts that recognize specific output from a UNIX command, like the total line from a wc command. Think of the American maintenance programmers.) To make a localized UNIX useful requires that those messages appear in the local language. Thus (except in English-speaking countries), the language in which messages display must be selectable at run time, on a per-user basis.

Well, not just messages really. Actually, this applies to most of the things in the previous section, in fact, almost all the text the program is likely to display. On top of everything else, the behavior of the system was made to depend on the value of an environment variable $LANG, which indicated the language and culture the user wanted.

Most user-selectable behavior can be implemented by having the action

Figure

```
#include <stdio.h>
#include <stdlib.h>
#include <nl_types.h>
#include <locale.h>
#include "hello_msg.h" /* decls of HELLO_MSG, etc. */
#define GRTNG "hello, world\n"" /* default message */
nl_catd catd;

main(int argc, char **argv)
{
  setlocale(LC_ALL, "");
  catd = catopen(MF_HELLO, 0);
  printf("%s\n",
   catgets(catd, HELLO_SET, HELLO_MSG, GRTNG));
  catclose(catd);
}
```

of library routines depend on $LANG, but messages are special. Most messages are hard-coded strings (or, worse still, strings constructed at run time from hard-coded substrings). Nearly all strings had to be replaced by calls to library routines that extract strings to be printed from *message catalogs*–the external files that contain the text strings for each language. The message catalog your program uses is decided at run time, based on $LANG. (Feeling like practicing your Español today? When you pour your first cup of coffee, type LANG=EsES. Or try your Swiss German with LANG=DeCH. Offended by Americanisms in your English? Use LANG=EnGB.)

Under AIX, a "messagized" version of the program we wrote earlier looks like the figure above.

Internationalization

The design alluded to in the last section has some bonuses. First, users in multilingual countries, like Switzerland and Canada, can run in more than one language on one machine. Second, distributors in multilingual markets, like IBM Europe, can stock and ship one set of binaries instead of a dozen. Third, adding a new market–say, Eastern Europe–only requires adding new message catalogs and other data files to describe the new locale–information about how the system should behave for the new market.

Fourth, development organizations, like IBM's AIX developers, can maintain a single European version, instead of having to propagate bug fixes across the English version, the Spanish version, the French version, the Croatian version and so on. With maintenance costs typically half of the cost of software even in proprietary environments, where porting is a smaller proportion of costs, savings from being able to maintain fewer versions can overshadow all else.

What's missing from this design is the ability to switch character sets. Reflecting on some of the character set-related changes described earlier, one would expect that making a truly character set-independent version would add extra wrinkles. Still, maintenance costs are so important that industry focus is turning away from localization and toward internationalization (or, in Britain, internationalisation, hence the internationalized/sed abbreviation, which replaces the 18 characters "nternationalizatio" with two digits, making it easier to write in a dark conference room) creates single-source products that are truly independent of language, country and character set, by taking all those dependencies out of the code and storing them somewhere else.

Contributing to this is the expected emergence and growth of UNIX markets in places like Korea, China, the Middle East and India, each of which introduces one or more new scripts. (The world's largest employer is the Indian national railway system, with six million employees. Payroll is done by hand.)

Standards

As you may imagine, IBM isn't the only vendor interested in international markets. Because portability is a hallmark of open systems, the recent standards mania sweeping the UNIX community includes efforts to standardize facilities that support i18n. IBM's localized variants predated many of these efforts. Two pieces of fallout from that:

- IBM's early efforts have influenced the direction of the standards efforts.
- IBM's efforts aren't identical to the standards, which, given the open systems orientation of the IBM of today, means that you can expect IBM to change what they offer to support the emerging standards.

We'll try, throughout this series, to outline the emerging standard solutions, because we believe these are the keys to i18n for the RS/6000 user.

Still to Come

We've tried to set the stage for the articles that follow by sketching the problems that an English-language UNIX faces in evolving to serve an international market. We'll return to many of the points we raise above in issues to come, treating both the problem and the solutions in more detail. We hope we've raised enough questions, fears and curiosity to get you to read what we write. ▲

Internacionalización: Standard C

by Jeffreys Copeland and Haemer

Sow a Thought, and you reap an Act;
Sow an Act, and you reap a Habit;
Sow a Habit, and you reap a Character;
Sow a Character, and you reap a Destiny.
-Anon.

Last month, we sketched the problems that software confronts in an international world. This month, we launch into a tour of the various pieces of solutions provided by the different parts of AIX. We begin with ANSI/ISO C for several reasons:

1. It's the oldest and most portable standardized piece. Programs that use only the standard C facilities can, with a little care, be guaranteed to work on any machine with a standard C compiler.
2. Most of the other internationalization-related facilities are features added to enhance or complement the standard C facilities. Although there is no unified design for the internationalization facilities, the design of C's features profoundly influences that of the others.
3. We organize our explanation by the standards organizations that specify the partial solutions, rather than by the areas the solutions address, in part because internationalization standards are somewhat disjointed and have grown by accretion rather than by a coherent plan. Also, if you know what standards bodies are responsible for what, and what standards a certain platform conforms to, you can better understand what parts will work together as you move from platform to platform.

Standard C attacks two interna-

Jeffrey Copeland (`jeff@itx.isc.com`) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* `troff` *on every continent.* **Jeffrey S. Haemer** (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. This year, Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

tionalization problems: large character sets and locales. For most programs, large-character-set handling is the more pervasive and more interesting change, so we'll start with it.

All Characters, Great and Small

Consider this program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{
 char s[1024];
 int Len, max=0;
 while (gets(s) != NULL) {
  Len = strlen(s);
  max = (Len>max)?Len:max;
 }
 printf("longest line: ");
 printf("%d characters \n", max);
 exit(0);
}
```

What should it print for this input?

```
hello, world
今日わ, ｼﾞｪﾌﾘｨ
```

Well, what's `strlen (今日わ, ｼﾞｪﾌﾘｨ)`? Here are three arguably rational answers:

1. There are 11 characters in 今日わ, ｼﾞｪﾌﾘｨ, so `strlen (今日わ, ｼﾞｪﾌﾘｨ)` is 11, which means the "longest" line is the English one, and the program prints

```
longest line: 12 characters
```

2. 今日わ, ｼﾞｪﾌﾘｨ is clearly longer than "hello, world," because some of the Japanese characters are double-width; `strlen(今日わ, ｼﾞｪﾌﾘｨ )` is the width of 14 single-width characters, so the program prints

```
longest line: 14 characters
```

3. The binary representation of 今日わ, ｼﾞｪﾌﾘｨ must be more than 12 bytes. A single byte can only represent 256 characters, and Japanese has more than 30,000 Kanji characters in fairly common use. If we use a two-byte internal representation for each character, `strlen (今日わ, ｼﾞｪﾌﾘｨ)` is 22–the number of bytes it takes to represent the string–and the program prints

```
longest line: 22 characters
```

Why the ambiguity? The assumption that a character will fit in a `char` pervades traditional C programming practice–obvious just from C's word for byte: `char`. You may have done ports that stumbled because the code assumed pointers could fit in integers. In an internationalization project, the biggest stumbling block is the byte/character problem. In our experience, at least half the battle of internationalizing an existing suite of applications is teasing apart the two meanings for `char`.

Near the end of the standardization process, the ANSI C committee, X3J11, realized that it had to address internationalization. If it didn't, the International Standards Organization (ISO) would shoehorn something in after it left X3J11's hands. To forestall this, the committee decided to add features to C that would help internationalize applications.

Three proposals were advanced to help programmers distinguish characters from bytes:

- The simplest proposal, Doug Gwyn's, made `char` the keyword for characters and permitted a new type, `short char`, to hold bytes in cases where bytes and characters were different sizes.
- The largest, favored by the Japanese, specified a new type to hold characters and a large library of functions that operated on that type.
- The committee adopted a third, compromise route, which provides a new, wide-character type, `wchar_t`, together with a handful of facilities that permit programmers to express and print data in those types, and to interconvert `char` and `wchar_t` data.

Many versions of AIX before 3.2 offered similar internationalization features. Because they predated the ANSI standard, they differ from the ANSI/ISO standard syntactically, but they have nearly identical semantics. We'll treat the standard features and let you do any necessary conversions.

Oh, the answer is the third proposal. In standard C, `strlen()` deals in bytes. (By the way, the Japanese line says: "kon-nichi wa, jyufurii"–"Good afternoon, Jeffrey.")

How AIX C Programs Handle Data

Here's the basic approach AIX takes to large character sets. First, it's important to handle ASCII input and output. After all, most systems with which RISC System/6000s exchange data can only handle ASCII. Yet one byte can't hold all the needed characters, so some non-ASCII characters must take up more than one byte. The logical conclusion is that characters outside the programs must have variable lengths. (We'll postpone talking about the encodings to another column; AIX supplies several.) These variable-length characters are called multibyte characters.

Inside programs, however, it's critically important to be able to process strings a character at a time, like this:

```
for (p = s; *p; p++) {
       /*
       process next character
       in array s
       */
}
```

To this end, standard C provides a new, constant-width internal representation: the `wchar_t` , `typedef`'d in `stddef.h` (an unsigned short on AIX).

How do we get from multibyte

characters to wide characters and back again? stdlib.h provides two basic functions for this:

```
int mbtowc(wchar_t *pwc,
    const char *s, size_t n)
```

which converts a multibyte character to a wchar_t , and

```
int wctomb(char *s, wchar_t wchar)
```

which converts a single wchar_t to a multibyte character. We also have three functions that are helpful:

```
int mblen(const char *s, size_t n)
```

to provide the length in bytes of a (possibly multibyte) character in a byte array,

```
size_t mbstowcs(wchar_t *pwcs,
  const char *s, size_t n)
```

to convert a string of multibyte characters to a wchar_t string, and

```
size_t wcstombs(char *s,
  const wchar_t *pwcs, size_t n)
```

to convert a wchar_t string to a multibyte string.

The arguments are self-explanatory. For example, in mbtowc(), the third argument says "convert no more than the next n bytes." All the functions follow this basic model.

Sort of. Input values are declared to be constants, so calls can't change them. Except for wctomb(). And notice that the input to wctomb() is the wchar_t being converted, but the input to mbtowc() is a pointer. And wctomb() doesn't have a third argument, because the size of the input is fixed, yet wcstombs() has a third argument, which is the maximum number of bytes in array s that will be modified.

Return values are just as bad. All of them return -1 on error. It's when they succeed that things get confusing. For example, mbtowc() returns the number of bytes contained in the converted multibyte character, unless *s is '/0', in which case it returns zero. Unless, of course, s itself is NULL, in which case mbtowc() either returns zero or it doesn't, depending on the details of the character-encoding scheme. Unless it's a leap year, in which case. ...

Mind you, much of this isn't even wrong. A little thought reveals that mbtowc() has to take a pointer, since what's being converted is potentially a multibyte character–that is, a string. Still, the lack of uniformity bears all the marks of design by committee and makes the details hard to remember and easy to get wrong.

Which, in turn, means that you have to pay attention to what you're writing, and how you use the wide character features. You can trip yourself up–see Pragmatics below. The good news is that when written properly, your software will be portable not only from platform to platform, but from language to language and country to country. (How is *Izvestia*–**Известия** –typeset today? How do you think it will be typeset once the Russians have hard currency to buy computers? And how many desktop-publishing packages support Cyrillic keyboards and produce their error messages in Russian?)

Other wchar_t Odds and Ends

Before leaving the wide, wide world of wide, wide characters, we'll touch on a pair of macros, a pair of constants and printf().

Macros

Whenever we've used mbtowc() or mbstowcs(), we've set n, the maximum number of bytes to examine, to the maximum length we expect a multibyte character to be. stdlib.h provides a convenient macro to use for this: MB_CUR_MAX. Don't confuse this with MB_LEN_MAX, in limits.h, which is the maximum length we're allowed to make a multibyte character, or with the value given in section 2.2.4.2.1 (really) of the ANSI standard for the minimum value that MB_LEN_MAX can have: 1. Aren't standards nifty?

Here's what's going on. As we'll explain in a bit, it's possible to run the same executable with different multibyte character sets–different encodings for the individual characters. In theory, it's even possible to change character sets on the fly.

MB_LEN_MAX is the largest possible character from all the possible character sets on your system. (Section 2.2.4.2.1 helpfully requires that you have at least some characters of non-zero width.) MB_CUR_MAX, in contrast, is the length of the biggest character in your current character set (on AIX, typically one, two or three). Because character sets can change at invocation time, or even at runtime, this is re-evaluated each time it's encountered.

Thus, MB_LEN_MAX is really a #define'd constant, while MB_CUR_MAX is really a parameterless function, implemented as a macro, evaluated at runtime. Someday you, or someone maintaining your code, will wish the committee had written it MB_CUR_MAX().

Constants

It's convenient to be able to write "a" instead of (char) 0141. It doesn't take much imagination to realize that generating wide-character equivalents of character constants using the stdlib.h functions would quickly lose its appeal.

```
#include <stddef.h>
#include <stdlib.h>

wchar_t wc;
char c = 'a';
 ...
mbtowc(&wc, &c, MB_CUR_MAX);
```

Helpfully, standard C provides the syntax L'a', which means "the wide character constant that corresponds to 'a'." (Usefully, the ASCII characters needed to print C programs–upper- and lower-case letters, underscores, digits plus punctuation like

"{," "&" and so on–are guaranteed to be numerically identical (when appropriately cast) to their wchar_t counterparts. AIX goes farther, providing this identity for all ASCII. You might even guess that a multibyte character, like L'[illegible]', would be (wchar_t) '[illegible]', which would allow us to replace mbtowc() by sprintf(). You might, but you'd be wrong.) The conversion is guaranteed to be the same as that provided by mbtowc(). (That is, if I say L'a', I've done the equivalent of execute the program fragment above.) Similarly, L"ｼﾞｪﾌﾘｲ" is a wide-character string constant–the wchar_t array you'd get from handing "ｼﾞｪﾌﾘｲ" to mbstowcs(). Incidentally, wide-character strings, like L"ｼﾞｪﾌﾘｲ," end with L' 0', which is guaranteed to be zero.

The syntax is a little odd–it adopts neither the suffix model used for long, unsigned and floating-point constants, like 3034940924L, or the backslash-style prefixes of hex and octal character constants, like \xa5. Moreover, why L for "wide character?" Still, you get used to it quickly. Warning: Remember a moment ago, when we said that there might be more than one multibyte character set at runtime? These are compile-time constants. Which character set does the system use for the encoding? The standard says the answer is system-dependent. Don't use wide character strings and constants unless you have some strong reason to believe you know the runtime encoding.

Output

It would be great if standard C provided printf conversion characters for wide characters and wide-character strings, so you could say

```
printf("%s, %S n", "hello",
   L"world");
```

It doesn't.

AIX provides such a printf conversion, but be aware that programs that use it may not be portable.

Pragmatics

Fine, so the wide character additions are confusing and incomplete. How do you use them? The answer depends on the application, but here are some guidelines:

1. Keep a copy of ANSI X3.159-1989 or ISO IS9989:1990 handy. (Contact: Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018 (212) 642-4900.) Alternatively, get a book that reprints enough of the standard to be useful. We can highly recommend P.J. Plauger's *The Standard C Library*. (Plauger, P.J., *The Standard C Library*, Englewood Cliffs, NJ: Prentice-Hall, 1992, ISBN 0-13-131509-9.)

2. Examine each application to see if it's handling bytes or characters. If you're internationalizing old applications, just searching for char will find many of the trouble spots: variables, parameters and functions of type char and char *, plus invocations of getchar() and putchar().

Don't just blindly change to wide characters. Surprisingly, many applications will require no wchar_t changes. Anything that's just handling bytes (e.g., cat) shouldn't need any changes. Ditto for spots where char is just a convenient size for a small integer.

3. mbtowc() and friends can be slow. Even if you're handling characters, try to avoid converting to wide characters if you can. Even with variable-width characters, this loop still finds the string-terminating NUL:

```
for (cp = s; *cp; cp++)
    ;
```

You do not need to use the following, more complicated, fragment:

```
mbstowcs(wcs, s, MB_CUR_MAX);
for (wcp = wcs; wcp != L' 0'; wcp++)
    ;
```

In fact, on AIX, any byte less than 0x3f in a multibyte string is guaranteed to be the corresponding ASCII character, so this loop finds a new line:

```
for (cp = s; *cp != ' \n'; cp++)
    ;
```

4. If you convert to wide characters, don't convert back unless you want to do output.

5. Where you can process multibyte strings one (wide) character at a time, do so. Use loops like this:

```
for (cp = s; *cp ; cp += n) {
n = mbtowc(&wc, cp, MB_CUR_MAX);
if (wc = = 'a')
   /* but remember, 'a' = = L'a' is
   only true for ASCII */
   break;
}
```

(By the way, you'll usually use mbtowc() instead of mblen(). They're often similar in cost, and both give you the size, but mbtowc() gives you the conversion as a side effect.)

6. Keep alert to the possibility of rethinking algorithms. A program that keeps an array indexed by letters of the alphabet will still work if you change the array to have a wchar_t subscript, but the array now jumps from having 128 (or 256) elements, to having 65,536.

7. Don't be too clever. Avoid the temptation to save a tiny bit of execution speed by using a lot of complex code. If you can make the code simple by converting to wide characters once on input and once on output without losing too much performance, do so.

Don't forget that maintenance will be a major cost. We have had to reinternationalize code from time to time, to change internationalization schemes. It's been our experience that clever and complex internationalization is often also wrong.

Locales

Most of standard C's new internationalization features, besides the ability to deal with new character sets, are wrapped around the idea of locales, which are sort of a solution to sort of a problem. (The trigraphs (e.g., ??/) are related to internationalization, too. We'll skip them, since they are intended for input from terminals without the full ASCII character set. So should you.) Locales are intended to encapsulate the information needed to change program behavior for different...well...locales. For example, many countries use a decimal comma, "," instead of a decimal point. Standard C retains its traditional, American conventions inside programs, but lets `scanf()`, `printf()` and their relatives recognize and generate decimal commas in certain locales.

Depending on the locale,

```
printf("%f \n", 0.6);
```

can print `0.6` or `0,6`. So how do you set a locale? Standard C provides a function:

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

Let's explore this call a little.

The LC_ Categories

Six categories are defined by the standard: `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC` and `LC_TIME`. These partition the locale into several components, so C programmers can actually make their programs format numeric data to suit a Spaniard, while formatting time the way a Japanese might and printing all monetary quantities in Kuwaiti dinars.

Do real people do this? Hardly ever. Almost all invocations of `setlocale()` are at the beginnings of programs, and look like this:

```
main()
{
    setlocale(LC_ALL, "");
```

This sets all components of the locale to an "implementation-defined native environment." (That is,, `LC_ALL` sets the other five categories.) On AIX, this means the locale specified by the `LANG` environment variable.

Don't explicitly name locales in your program if you can help it, because the names probably won't be portable.

In practice, every internationalized program starts like this, because the committee decided to require that programs without a `setlocale()` call behave in a fixed, predictable way–roughly, in traditional C fashion. (Never mind that traditional C and UNIX never addressed money, collation or externalized messages.)

The only other way you're likely to use setlocale is with a `NULL` second argument, like this:

```
char oldlocale[100];
strcpy(oldlocale,
    setlocale(LC_ALL, NULL));
```

This saves the name of the current locale for future use. Having saved it, you can switch to a new locale, with another `setlocale()` call, then later restore the original by calling

```
setlocale(LC_ALL, oldlocale);
```

Locale Names

You may have noticed that we've mentioned a C locale, and suggested that there are other locales, but avoided naming any. There is no standard form for locale names.

In AIX Version 3.2, the names follow the formula "ll_CC.codeset." "ll" is a language, like "fr" or "Fr" for French, or "jp" or "Jp" for Japanese. "CC" is a country code, like "CH" for Switzerland or "BE" for Belgium. "codeset" is the character set encoding, like "IBM-850," "ISO8859-1" or "IBM-eucJP."

On the other hand, AIX versions before 3.2 used a different scheme. In other words, don't explicitly name locales in your program if you can help it, because the names probably won't be portable.

What Does the Locale Affect?

Earlier, we mentioned the effect of `LC_NUMERIC` on `printf()`. What else can a `setlocale()` call change? Here's a breakdown by category:

- `LC_COLLATE`: `strcoll()` and `strxfrm()` depend on this: `strcoll()` is a locale-specific `strcmp()`. `strcmp()` compares two byte strings numerically. `strcoll()` lets you specify what the ordering of the characters should be, though the standard doesn't specify how, except that it will depend on `LC_COLLATE`. `strxfrm()` is a half-way `strcoll()`, which uses the user-specified collating sequence to transform an input string into a form suitable for later numerical comparison via `strcmp()`. You can think of `strcoll()` as performing a `strxfrm()` on each of its arguments followed by a `strcmp()`. Because the transformation is likely to be expensive, a large set of intercomparisons (e.g., for a quicksort library routine) might be done at less expense by doing all the transformations first, then doing simple `strcmp()` calls.
- `LC_CTYPE`: This category can affect all the ctype functions except `isdigit()` and `isxdigit()`, and the multibyte functions. We confess we're not sure why `LC_CTYPE` would affect the multibyte func-

tions, but the standard says so. Our only thought is that there is no component of the environment that specifically points at the code set, and the intent may have been for this to serve that purpose in addition to the more clear-cut purpose of indicating locale-specific rules for character classification.

- `LC_NUMERIC`: This determines the rules for formatting numbers. It affects formatted I/O functions, like `printf()` and `scanf()`, and string conversion functions, like `atof()` and `strtod()`.
- `LC_TIME`: This determines the rules for printing times, used by `strftime()`. `strftime()`, declared in `time.h`, is what you'd get if you replaced `asctime()` by a cross between `printf()` and `date`. Like `asctime()`, it takes a tm structure–a UNIX broken-down time–and converts it into a printable string. Like `printf()`, it does the conversion based on a format string, which can contain a mix of ordinary characters and any of 22 (!) different %-prefixed conversion specifiers. The good news is that, the conversion specifiers for `strftime()` and date mostly overlap, and where they do so they are identical.
- Last, and arguably least, both `LC_MONETARY` and `LC_NUMERIC` can affect the information returned by a call to `localeconv()`–another committee invention. This function returns a `struct`, defined in `locale.h`, containing a hodgepodge of information about formatting numbers and money. You cannot use this structure to feed to `setlocale()`, nor can you use it to ask for any of the information determined by the other locale categories.

These six categories, including `LC_ALL`, are a minimum set. Implementations may define additional `LC_` categories, but the value of any such new categories can't alter the behavior of standard library functions. (AIX provides a seventh, `LC_MESSAGES`, which is required by χ/Open. See? There's a reason you want to read the column on χ/Open internationalization features, a few issues hence.)

Summary

We've talked about multibyte characters, wide characters and locales–things ANSI C added to traditional C to support internationalization. What else does AIX supply to help you write internationalized programs? Plenty. You'll see some more next time, when we discuss the internationalization facilities from another key standard: ANSI/IEEE 1003.1-1990 (ISO IS9945-1:1990). For those of you who don't memorize standard numbers, that's POSIX.1. ▲

WE BOUGHT IBM® By the Thousands

Computer Marketplace has bought and sold thousands of IBM systems, tape drives, disk drives, printers and peripherals. We are interested in offering top dollar for your RISC System/6000, AS/400 or System/36.

SYSTEMS & RENTALS	PERIPHERALS
■ RISC System/6000™ Systems	■ Controllers
■ Memory & Features	■ Disk Drives
■ AS/400™	■ Tape Drives
■ Network Solutions	■ Memory Options
■ System/36	■ Modems
■ PS/2	■ Displays
	■ Printers

BUY • SELL • RENT
5 Offices To Serve You!

COMPUTER™ MARKETPLACE
(800) 858-1144

(714) 735-2102 FAX (714) 735-5717
205 East 5th Street, Corona, CA 91719
RISC System/6000, AS/400 & PS/2, System/36 are trademarks of IBM.

Circle No. 8 on Inquiry Card

WE SUPPORT RISC TAKERS

- Systems Integration
- Software Development
- Consulting
- Staffing
- Training

Pencom (512) 343-1111

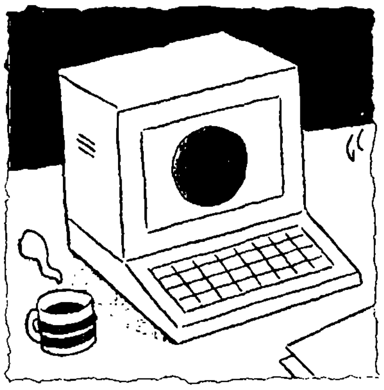
NEW YORK • AUSTIN • BOSTON • SAN JOSE • WASHINGTON DC

Circle No. 24 on Inquiry Card

国際化 POSIX.1

by Jeffreys Copeland and Haemer

Last month we talked about the features Standard C provides for internationalization. This month, we peel back another layer of the onion and discuss internationalization in the POSIX.1 specification.



What's POSIX Anyway?

Our topic is POSIX.1, or, more formally, IEEE Std 1003.1, *Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language]*. It's also a standard of Joint Technical Committee 1 (JTC1) of the International Standards Organization and the International Electrotechnical Commission, as International Standard ISO/IEC 9945-1. The current edition was published in December 1990, and if you keep track of these things, the volume is an international-standard-A4-sized paperback, with a gaudy purple, yellow and red cover.

POSIX, as a whole, describes a portable computer environment that looks more or less like UNIX. It currently has about 20 parts, omitting the kitchen sink only because household fixtures are not within the POSIX charter. They range from POSIX.0, which is the POSIX guide, through POSIX.4 (real-time extensions) and POSIX.6 (security extensions) to POSIX.20 (Ada binding to POSIX.4). POSIX.1 and POSIX.3 (which gives general rules for developing test assertions and related test methods to ensure POSIX conformance) are the only published interfaces. However, many others are nearly complete.

Because POSIX.1 describes an API–a set of subroutine interfaces–you may wonder how it differs from the Standard C library. There is

Jeffrey Copeland (jeff@itx.isc.com) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* troff *on every continent.* Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. This year, Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

deliberately little overlap between the C standard and the POSIX.1 standard. Standard C specifies about 100 library functions. POSIX.1 specifies about 100 *more* functions that aren't required in a Standard C implementation but are necessary for a UNIX-like operating system. For example, POSIX.1 specifies `fork()`. A typical MS-DOS C environment lacks this, but you can't be POSIX without it. To get a quick feeling for what other kinds of functions POSIX.1 provides, look at `<unistd.h>`, which declares a large, representative sample.

Besides adding functions, POSIX.1 also sometimes extends the Standard C functions; i18n (that's the common abbreviation for "internationalization," if you're jumping in late) is a major source of such extensions. For example, Standard C specifies `getenv()`, to get values from the environment, but is silent about the names of any specific environment variables. POSIX.1 defines a dozen: HOME, LOGNAME, PATH, TERM, TZ, LANG and a half-dozen LC_ variables. Of these, the last eight have to do with i18n.

The LC_ Categories

Let's start with locales. Standard C provides the function `setlocale()` to let programs set part or all of the locale. But where the C standard is vague about "implementation-defined" behaviors, POSIX.1 lets `setlocale()` get information from its environment. For example, we told you last month that the typical call (`setlocale(LC_ALL, "");`) normally causes AIX to set these categories based on the environment variable LANG. Though that's true, it's because of POSIX.1, not because of Standard C. It's also not the whole story. You can also set an environment variable corresponding to any LC_ category. There's a strict hierarchy. When you call `setlocale(LC_ALL, "");` the value of the environment variable LC_ALL is used for all components of the locale. If LC_ALL isn't set, the environment variables for the categories are examined. For any category that doesn't have a corresponding LC_ environment variable set, the value of $LANG is used. Usually, you just set LANG and have done with it, but you really can have your program print its numbers to suit a Spaniard, its time as a Japanese imperial date and its money in Kuwaiti dinars just by setting environment variables.

All this works. If you have all the locales installed on your RS/6000, you can type:

```
LANG=Ja_JP
export LANG
xinit
```

and the X Window System will start up with Japanese labels, instead of English ones. This is how we composed the title of this month's column.

In the absence of a `setlocale()` call, the default is still the "C" locale (roughly American English with the ASCII character set) but POSIX.1 also defines a "POSIX" locale, which is currently identical to the "C" locale, but will probably add extensions as POSIX evolves. Watch this space.

Time Zones

Traditional UNIX has a simple model of time zones that works well in the United States, but not internationally. Time zones, specified by the environment variable TZ, originally looked like EST5EDT. This named the time zone EST, and set it five hours west of Greenwich, or Coordinated Universal Time (UTC); during the summer, the time-zone name changed to EDT, set one hour later than standard time.

But when does summer start and end? And what about India and central Australia, which have time zones on half-hour differences from UTC? Or even worse, Singapore, which once had a seven-hour, 45-minute offset? POSIX.1 addresses all these problems.

The time-zone difference from UTC is still given in hours but can also take optional minutes and seconds. This means Delhi has a time zone offset `-5:30`, and Nova Scotia is `+3:30`.

The most complicated new feature allows a rule to define the times to shift between standard and daylight-saving time, or "summer time" as the British call it. Traditional UNIX hard-wired these shift times into the time routines, using the rules in place in the United States at the time. Now, you can provide the information, encoded as text, through the TZ environment variable.

The general rule is to provide month, week, day and time of the change: So 10:30 a.m. on the first Monday in October is `M10.1.1/10:30`, and 5 p.m. the last Tuesday in July is `M7.5.2/17`. (There are a couple of alternate forms that allow you to specify a fixed date in the year.) For example, in the United States, daylight-saving time extends from 2 a.m. on the first Sunday in April to 2 a.m. on the last Sunday in October, so the full time zone specification for Austin, TX, is:

```
CST6:00CDT7:00,M4.1.0/
  2:00,M10.5.0/2:00
```

In Denmark, however, daylight-saving time switches on and off the last Sunday of March and September respectively, so the TZ specification–taken directly from page 320 of the POSIX.1 document–is:

```
CET-1CET DST,M3.5.0/M9.5.0
```

Limits.h–How Many? How Big?

POSIX.1 takes some of the guesswork out of system limitations by encapsulating them in the header file `<limits.h>`.

There's more to these limits than you might imagine at first. Let's go through an example. How big can the name of the time-zone name be? Notice that the name for daylight-saving time in Denmark, given a moment ago is CET DST. (Time-zone names can have embedded blanks!) POSIX.1 says that the time-zone names will be no larger than TZNAME_MAX bytes (that's not characters!), where TZNAME_MAX is defined in <limits.h>. But that maximum is specific to your system, and POSIX.1 is concerned with writing portable applications, so POSIX.1 also specifies portable maxima in <limits.h>.

For example, how big can we make an AIX filename? <limits.h> gives this value, called NAME_MAX, as 255. However, other systems may not allow filenames this large. How many bytes in the largest portable filename? In other words, how small can NAME_MAX be on any POSIX.1-conforming system? Section 2.8.3 of the standard calls this value _POSIX_NAME_MAX and says it must be 14. In general, the smallest allowable value of FOO_MAX is named _POSIX_FOO_MAX, and both values can be found in <limits.h>. _POSIX_TZNAME_MAX is 3.

Unfortunately, on AIX, TZNAME_MAX is _POSIX_TZNAME_MAX. This means that the Danish time-zone example we extracted from the POSIX spec won't work on AIX.

Now, if we told you that POSIX.1 had ALL THIS, how much WOULD YOU PAY? But WAIT...there's MORE!!

Suppose you're writing a portable application and you want to get the value of TZNAME_MAX at run time. POSIX.1 supplies the function long sysconf(int name), defined in <unistd.h>, which can be used to retrieve the values of system limits. (Some system limits, like filename lengths, can depend on where you are on the system, and are retrieved instead with the POSIX.1 functions pathconf() and fpathconf().)

There's a subtlety here that requires a moment's digression. Because the system on which your application is written may have different limits from the system on which it's run, the call cannot be

```
sysconf(TZNAME_MAX)
```

since TZNAME_MAX will be expanded at compile time, and we want a parameter that is evaluated at run time. One approach would be to have the parameter to sysconf() be a string: sysconf("TZNAME_MAX"). Instead, POSIX.1 provides yet another macro, _SC_TZNAME_MAX, which is guaranteed to return the correct answer.

Versions

One useful sysconf() argument is _SC_VERSION, which returns the value of _POSIX_VERSION, the version of the POSIX.1 standard you're running under. Why? Because some of the features we've mentioned, like TZNAME_MAX, are new to the 1990 version of the standard and won't be found in versions of AIX written to conform to the older, 1988 standard.

If you were wondering what POSIX your system conforms to, you can write a program like:

```
#include <stdio.h>
#include <unistd.h>
main()
{
 printf( "POSIX.1 version is:
   %ld n", sysconf(_SC_VERSION) );
}
```

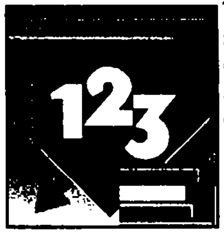
which, on AIX 3.2, gives the helpful information:

```
POSIX.1 version is: 199009
```

Bytes Versus Characters, Revisited

We've discussed how in the larger world, where ASCII isn't sufficient to encode words like "münster," and "façade," bytes don't always hold characters. We've also talked about the new Standard-C data type

Lotus 1-2-3 for RS/6000



The world's best selling spreadsheet is coming to AIX on July 1st. Fully file compatible with your DOS software. Same as Release 3 on DOS. Reserve your copy today!

	Reg.	UniD
Lotus 1-2-3 First User	$695	569
5 add-on character nodes	$1,250	1099
25 add-on character nodes	$4,525	3988

Menus for One and All

Do you want to create menus quickly for all your users? As a system administrator, wouldn't it be nice if you had all the admin functions already set up for you on a menu? XMENU (for character terminals) takes care of it all. *The Guru* says: "My recommendation: It's a BUY."

	Reg.	UniD
XMENU for RS/6000	$925	729

New for AIX.

UniDirectSM

USA & Canada

800-755-UNIX

International FAX 714-707-3095

45 DAY MONEY BACK GUARANTEE

With our downsized prices, your downsized budget has tremendous buying power. Try us today. Visa & AmEx.

WordPerfect & FAX

The world's best supported, best selling word processor. File compatible with your DOS software. Fax directly from WP or any application with VSI*FAX.

	Reg.	UniD
WordPerfect 5.0 10 user	$1,500	1194
VSI•FAX s/w & modem	$2,495	2199

Faster Performance!

REVEAL gives a graphic picture of how critical system resources are being used, finds bottlenecks and problems and allows you to fine tune your system for maximum performance. "Nothing could be quicker or easier," says *The Guru* Magazine. Call for details.

	Reg.	UniD
REVEAL RS/6000	$999	Call

More Great Values!

Call today for a FREE color catalog featuring special savings on FacetTERM, BLAST, CTAR, ICE.TEN.PLUS, cymail, dBase IV, TinyTerm, CorelDRAW, BRU, FrameMaker, and many more.

One Venture #230, Irvine, CA 92718

wchar_t.

With that in mind, you might be surprised to find out that POSIX.1 mentions wchar_t only once, in an informative (nonbinding) Annex, which lists the symbols that will appear in header files for Standard-C-based POSIX.1 implementations. All the routines in the API handle their character arguments as chars or char * s–that is, bytes or byte strings.

In practical, international terms, this means that if, for example, the filename size NAME_MAX were set to its minimum value of 14, an eight-character Japanese filename might be too long. (This assumes a SJIS character set, in which all characters are one or two bytes. If we're using EUC, which has three-byte Japanese characters, a five-character filename might go over the limit.)

Environment variables returned by getenv() are also byte strings, as are directory entries parsed by the routines in <dirent.h> and filenames passed to routines like open() and unlink(). A little reflection will show that this was the right choice. If we changed the file and directory functions to operate on wide characters instead of bytes, too much existing code would break. On the other hand, if you're parsing environment values or paths and need to search for specific characters, dust off your notes on mbtowc() and mbstowcs() and don't be afraid to use them.

A Red Herring: Language Independence

What version of C are we talking about? POSIX.1 is of two minds on this. Currently, POSIX-conforming implementations can have either Standard C, or "Common-Usage C" APIs. Common-Usage C is roughly the C specified in the first edition of K&R, so POSIX.1 enumerates a handful of Standard C functions, such as setlocale() and strftime(), that must be added to Common-Usage C implementations to give full POSIX.1 functionality.

This brings up the important point that POSIX.1 is really two separate, yet intermingled, kinds of specifications:

- Functional specifications of system facilities and
- calling sequences to invoke those facilities.

In theory, it should be possible to distill out the functional specifications and relegate language bindings to a separate document. Right now, the POSIX.1 committee is working to create just such a functional specification, and a separate but closely coordinated, parallel effort, POSIX.17, is writing a Standard C language binding to it. Close at their heels you can expect Ada-language and FORTRAN-language bindings, which will allow programmers full access to the POSIX.1 facilities from other programming languages. There are hints that still more bindings are in the works (Modula-2 and perhaps C++).

This is such a powerful idea that the IEEE has promised ISO it will provide all future base POSIX standards as functional specifications without calling sequences. Such specifications are called "Language Independent Specifications," but have nothing to do with natural languages, except that they will provide programmers in other programming languages access to the POSIX i18n facilities.

Stay Tuned for POSIX.2

We've now surveyed the major internationalization features provided by Standard C and POSIX.1. Some important things that aren't covered by either API are features you don't get without a command interpreter. Such features are described in POSIX.2–Shell and Utilities. The final version of POSIX.2 will almost certainly be published this year, so we'll discuss them next month. ▲

IBM

For the maximum in IBM **RS/6000 2xx, 3xx, 5xx, 9xx system** and **Xstation** performance, Kingston's memory and mass storage upgrades are second to none.

Memory Upgrades
- 64MB upgrade module kits
- 32MB upgrade module kits
- 16MB upgrade module kits
- Video RAM

Mass Storage Upgrades
- Internal and External Fixed Disks
- Removable Disk/DAT Storage Subsystems
- Peripheral Expansion Kits

With Kingston memory and storage upgrades, no job is too big. You can expand your 5xx or higher RS/6000 to 512MB of main memory and 11.2GB of SCSI disk storage. All Kingston enhancements are individually tested, fully warranted, and 100% compatible.

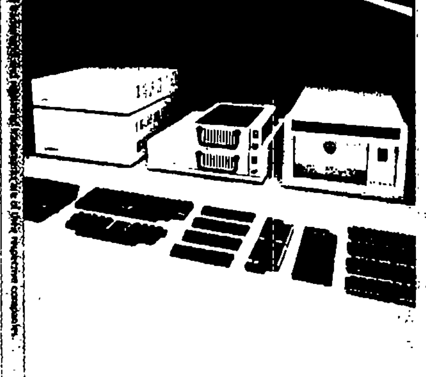
For more information call Kingston today at:

1-800-835-2545
Ph. 714-435-2667
Fax 714-435-2618
East Coast 516-673-6300

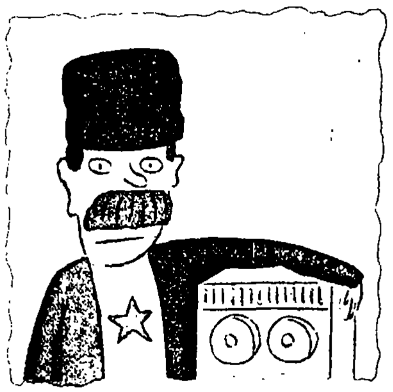
Manufactured by

Kingston TECHNOLOGY CORPORATION
17600 Newhope Street
Fountain Valley, CA 92708

Memory & Disk Storage



Circle No. 15 on Inquiry Card



Интернационализация POSIX.2

by Jeffreys Copeland and Haemer

So far, in our tour of AIX's internationalization, or i18n, facilities we've confined ourselves to C programs. This time, we're going to look at the facilities available at the shell level, by surveying the i18n facilities provided by the upcoming POSIX.2 Shell and Utilities standard.

Draft 11.2 of the POSIX.2 standard is nearly 1,000 pages–three times the size of the POSIX.1 standard. That's not even counting the POSIX.2a work, which will be inserted as Section 5, User Portability Utilities Option, by the time it becomes an ISO standard early next year (9945-2). POSIX.2a is being worked on separately. We won't treat it here explicitly, but it's another 324 pages.

This is more than we can sensibly cover in one column (or even five), so our plan is to do a pair of columns: this month's to give a feeling for what sorts of i18n topics the standard addresses, and next month's to discuss a few major topics in more detail. That confessed, we begin.

Standard? What Standard?

POSIX.2 probably won't become a final standard until the fall of 1992. (The draft we're writing this column from is 11.2.) Why bother with a standard that isn't even out?

AIX tracks the POSIX standards very closely. How closely? Not only can you bet that every AIX release following that standardization will be fully POSIX.2-conforming, but many of the command-level facilities already are and others are just pre-

Jeffrey Copeland (`jeff@itx.isc.com`) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* `troff` *on every continent. Jeffrey S. Haemer* (`jsh@ico.isc.com`) *has worked at ISC's Colorado Technical Office since 1983. He has experience working, writing and speaking on the interrelated topics of open systems, standards, software portability and porting, and internationalization. This year, Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

mature guesses at what would be adopted, soon to be updated.

Actually, everyone tracks the POSIX standards very closely, which is another reason to learn about it. If you're interested in i18n because you're going to be moving applications from one country to another or one language to another, you may also find yourself moving them from one platform to another, so today's RISC System/6000 application may be yesterday's MS-DOS or Sun application and tomorrow's...well, who knows? But whatever it is, it's likely to be POSIX.2-conforming.

Ho Hum?

Why is a command-level standard even interesting? After all, most of your programs are probably in C, and the C-level internationalization facilities we've already covered look fairly well-defined, even if occasionally a little tortuous.

The first reason is perspective. After all, on AIX commands are applications–most of them C programs, just like yours. The influence of internationalization on a shell and tools standard should give you some feel for how much you'll have to pay attention to internationalization in your applications. How much is that?

Let's begin at the end. The index tells us that of the 80 standardized commands, neither `true` nor `false` is affected by LC_ALL. That's it. The rest change their behavior when the locale changes, so chances are your application will have to, too. Not every command cares about every LC_ environment variable, of course. LC_CTYPE hits a lot, but LC_NUMERIC only affects awk, od, `printf` and `sort`–and LC_MONETARY doesn't affect any.

Working backward from there, a pair of the Annexes (that's POSIX for "appendices") reinforce this theme. Annex F is a "Danish National Profile," which specifies how the POSIX.2 internationalization features should be used to create a localized, Danish version. Annex E of the POSIX.1 standard is also a Danish national profile, but it's four pages long; this one's 70. And first but not least is Annex D, the bibliography. Of the 31 references, more than a third are on some aspect of internationalization.

In short, the internationalization you've seen so far isn't hidden by going up a level; it's magnified and pops up everywhere.

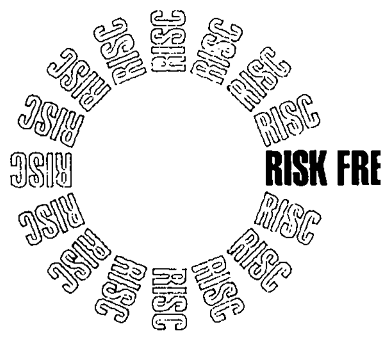
The second reason is that you may write more shell script than you realize. Even on the earliest UNIX installations, shell scripts outnumbered executable C programs by about 4½ to one. (See, for example, Kernighan & Mashey: "The Unix Programming Environment," *Computer*, April 1981, Pages 25-34.) Give users a really adequate set of filters and commands plus a programmable, genuinely useful command language and they'll write shell scripts. But without any guarantee of which filters or shells will be available, or what their flags and options might be, the scripts aren't portable. POSIX.2 solves that problem. What's more, it does it in an internationalized way.

What's There?

So how do we approach this behemoth? First, don't think of the size as unreasonable. The system calls and library functions standardized by POSIX.1 and the C standard come from Sections 2 and 3 in the traditional UNIX manuals. (In the AIX info hypertext, these have become the "Subroutines" section.) The POSIX.2 standard covers Section 1 (or the AIX info "Commands" section) which has always been much larger. (Despite this, the traditional complaint has always been that Section 1 is too terse.)

Most of the POSIX.1 features resurface at the shell level, though sometimes in slightly different guises, and this is also true for its i18n features.

One example of this is `getconf`, which lets you get at the configuration. It's just thinly disguised calls to



When you need more disks or backup alternatives, uncover the secret already enjoyed by many Fortune 500 companies and the US Government: Dynamic Computer Products.

Thorough knowledge plus years of experience translate to an uncanny ability to understand your specific needs. And we have the resources to match these needs to the right products from a wide variety of elegantly simple subsystems. Easy to install — Just open the carton and plug in.

DISK AND TAP STORAG SOLUTIONS

- 100% RS compatible
- Fully supported
- 1-year warranty
- On-site maintenance available

That's RISK FREE!

Compare — $2695 for a 1.6Gb disk in a complete desktop subsystem! Risk free and low cost!

For all your storage requirements call and let our expertise work for you.

Dynamic Computer Products
63 Commercial Ave.
Garden City, NY 11530

office 516-877-2777
fax 516-877-2780

Circle No. 9 on Inquiry Card

sysconf() and pathconf().

Thus,

```
tlen=$(getconf TZNAME_MAX)
```

puts the value of TZNAME_MAX into the variable tlen. (As you can see, the POSIX.2 shell looks a lot like the Korn shell. That's just a welcome upgrade for a typical Bourne shell user, but if you've been a C shell user, you might want to start boning up now.)

The three interesting differences are:

1. getconf takes the symbolic name, not the integer sysconf() requires. (sysconf() makes you remember to prefix symbolic constants you want to ask about with _SC_. The equivalent call is sysconf(_SC_TZNAME_MAX);.)
2. getconf returns a string, not sysconf()'s long;
3. getconf guarantees that the evaluation is done at runtime. (The POSIX.2 rationale also points out that on some implementations, sysconf() is permitted to return a value determined at compile time–probably not what you want at all.)

We like this.

Allowing return values to be strings also opens the door to more functionality. Originally, getconf was supposed to provide shell scripts with a trusted default path: getconf PATH returns the value of the system path to the standard locations of the POSIX.2 utilities, so that putting

```
PATH=$(getconf PATH)
```

at the beginning of shell scripts helps to guard against Trojan horse security attacks. (Unfortunately, on AIX 3.2, getconf doesn't recognize PATH.)

Locales

As we mentioned, the standard locale-related variables resurface in POSIX.2. There are also a new pair of variables: LC_ALL and LC_MESSAGES.

LC_MESSAGES responds to the need to have internationalized utilities emit error and warning messages in a user-selectable language. In a POSIX.2 environment, LC_MESSAGES selects that language for all utilities (except, it seems, true and false). Because dot two defines it as an environment variable, but it's not defined in dot one, there's no guarantee that setlocale(LC_MESSAGES, ""); will do anything within a C program. But remember the POSIX.1 standard permits new LC_ categories, and you can bet that any environment that supports POSIX.2 will add LC_MESSAGES and that setlocale() call will have the anticipated effect.

Note, though, that POSIX.2 standardizes how to specify the language in which messages are shown, but it says nothing about how these messages are stored or retrieved, nor does it say how to do those operations from either the shell or any other programming language.

LC_ALL is a third-class LC_ category. POSIX.1 establishes a hierarchy in which the LC_ALL setting overrides the settings of each of the other individual LC_ categories. With LC_ALL, POSIX.2 provides a default that each of the individual categories overrides. For example:

- If nothing is set, your application uses the POSIX (or C) locale by default.
- If LC_ALL is set, but nothing else is, then the locale is $LANG.
- If only LC_ALL and LC_MESSAGES are set, then LC_MESSAGES tells where to get warnings and error messages from, and everything else is determined by the value of LC_ALL.
- If LC_TIME, LC_MESSAGES and LC_ALL are set, then only LC_ALL counts.

(All this assumes that your application is even paying attention, which it can only do by calling setlocale(LC_ALL, ""); .)

Where Am I?

If you forget where you are, you can use the locale command. This helpful tool will tell you how LANG and the LC_ environment variables are set. For example, see the first invocation of locale in the figure below.

You can also get an exhaustive list of all the available locales by using the -a flag. On AIX 3.2, this gives a long list beginning

```
da_DK.ISO8859-1
Da_DK.IBM-850
de_CH.ISO8859-1
```

and ending

```
sv_SE.ISO8859-1
Sv_SE.IBM-850
tr_TR.ISO8859-9
```

How can I tell what the locale categories I've chosen really mean? Many of the parameters set in the locale can be examined with the -ck flag (which writes both category names and keyword values) of the locale command. For example, continuing from the above example, I can ask for the days of the week, the name of the currency and the way my numbers are formatted. (See the second and third invocations of locale in the figure.) What does all that output mean? My bank uses "L." (for Italian lire) as its currency symbol and separates the thousands in my balance with periods; my Swiss calculator uses a period for a decimal point but separates thousands with apostrophes (instead of the American comma); my Canadian calendar uses the French names for days.

The complete list of these keywords is given in Section 2.5 of POSIX.2, which describes how the locale itself is set up.

While you can set a locale in a shell script, there's no guarantee that the system you port it to will use the same name for the same locale. (AIX standardizes both–or, to be precise, adopts a standard for them: the one proposed by X/Open.)

New I18n Topics in POSIX.2

That was extensions to old stuff, but some brand-new things also appear at the command level.

`{POSIX2_LOCALEDEF}` and `localedef`: What happens if you don't like the locales available on your system? You can define a new one, which will support French time format, German monetary practice and Spanish punctuation of numbers. POSIX.2 defines a symbolic constant, `POSIX2_LOCALEDEF`, which tells if your system supports building locales and a utility `localedef` to compile the text version of the locale into a version the system and utilities can use. For now, it's enough that you know these exist. Next month, when we talk about the structure of locale sources in more detail, we'll explain how `localedef` works.

Characters and charmap files: Characters and their codes are defined in charmap files. These are mostly long lists of lines such as:

```
<semicolon>    \x3b
```

Again, we'll discuss these in detail next month along with locales.

Reformed regular expressions: Here you'll find big changes, mostly for the better. Anyone who's tripped over the dozens of little, incompatible differences in regular expression rules in the dozens of utilities that use them, from `find` through `vi` to `grep`, will be grateful to hear that POSIX.2 has cut this down to two sets: Basic Regular Expressions (BREs) and Extended Regular Expressions (EREs). Soon you can learn this pair and forget the rest. What's more, they tackle i18n head-on. Instead of making us change `grep "[a-z]"` to `grep "[a-záéíóún]"` for Spanish-speaking locales, POSIX.2 gives us `grep "[[:lower:]]"`, which works everywhere.

We'll defer discussing these in detail to next time.

Shell Pattern Matching: The one place POSIX.2 regular expressions don't take over is in the shell. Who wants `cd ..` to mean "change to some subdirectory whose name is two characters long"? At the same time, some extensions are again necessary to help the shell handle international characters. Bracket expressions in file names, like `vi [Mm]akefile`, can now take the new character-class expressions:

```
nroff -mm R[[=e=]]sum[[=e=].mm]
```

Figure. Invoking Locale

```
$ export LANG=POSIX
$ export LC_MONETARY=It_IT     # my bank is in Rome
$ export LC_NUMERIC=De_CH      # my calculator is from Zurich
$ export LC_TIME=Fr_CA         # my calendar is from Montreal
$ locale
LANG=POSIX
LC_COLLATE="POSIX"
LC_CTYPE="POSIX"
LC_MONETARY=It_IT
LC_NUMERIC=De_CH
LC_TIME=Fr_CA
LC_MESSAGES="POSIX"
LC_ALL=
$ date
Lun Jun 8 14:21:00 1992\" this is Caroline Megan Tulloh's birthday
$ locale -ck day currency_symbol mon_thousands_sep
LC_MONETARY
currency_symbol="L."
mon_thousands_sep="."
LC_NUMERIC
decimal_point="."
thousands_sep="'"
LC_TIME
day="Dimanche";"Lundi";"Mardi";"Mercredi";"Jeudi";"Vendredi";"Samedi"
```

C Functions?

It usually comes as a surprise to people that there are also C-level interfaces in POSIX.2. In hindsight, it makes sense. You can't call `popen()`, `pclose()` and `system()` until the shell has been standardized.

As a convenience to the programmer, POSIX.2 also provides programming-language interfaces to many of the internationalization extensions that it defines. Locale control, system configuration, regular expression matching and shell pattern matching are all examples.

We've begun discussing POSIX.2, which rivals the New York City phone book in size. We've touched briefly on several commands, and, we hope, whetted your appetite for a more detailed discussion next month. We still need to talk about locales and how they are defined, regular expressions in more detail, and touch on some more specific commands in the canonical POSIX.2 set.

Art Notes

The Cyrillic font used in the headline this month is based on a beautiful Metafont by Nana Glonti and Alexander Samarin of the Institute for High Energy Physics in Protvino, Russia. We admit with regret than an aesthetic loss was caused by our modifications.

Similarly, the kanji characters in last month's headline were built using `pic` and `troff` beginning with the 16-bit JIS bitmaps. Again, blame us for any mistakes, particularly the rough edges in the kanji for *sai*. ▲

국제화

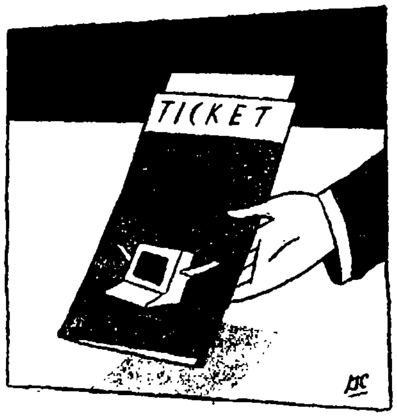
POSIX.2 Continued

by Jeffreys Copeland & Haemer

Last month, we discussed the general features of POSIX.2–the Portable Operating System Interface Shell and Utility Standard, or IEEE 1003.2, which will soon also be ISO 9945-2. This month, we will discuss two of those features in detail. A quick reminder: 1003.2 still isn't a final standard. It's expected to be ratified this fall. We're writing this article based on Draft 11.2 of the proposed standard.

Locales

In our ongoing discussion of internationalization features of various standards, we've talked a bit about locales. We discussed what a POSIX.1 locale contains. However, because POSIX.2 has more i18n implications than POSIX.1 (as we discussed last month), the POSIX.2 locales are a little more complicated than the POSIX.1 locales. In fact, the complete description of how to construct a locale in the standard is roughly 60 pages, densely packed with text and tables. And, as we've mentioned, the sample Danish locale in annex F is 70 pages.



What use is this information? To begin with, you need it if you're planning to build or modify a locale. More importantly, once you begin to experiment with the internationalization features of AIX, you will likely find yourself referring to the text of the locale sources to understand why things work the way they do.

Character Sets

Locales begin with a character set description, or charmap file. This is a list of the names and encodings for all the characters to be used by the system. (On AIX, the charmap files

Jeffrey Copeland (`jeff@itx.isc.com`) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* `troff` *on every continent. Jeffrey S. Haemer* (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. This year, Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

live in /usr/lib/nls/charmap/.) Each charmap file starts with some descriptive lines:

```
<code_set_name>  "IBM-932"
<mb_cur_max>     2
<mb_cur_min>     1
```

These name the code set and tell us how big the characters can be. In this example, we have the IBM-932 code page, which is a multibyte character set, corresponding roughly to the Japanese Shift-JIS set. The largest character (<mb_cur_max>) is two bytes.

This is followed by the character map itself, which is a relatively dull list, consisting of symbolic character names and values. In this list, the "portable character set" must appear. The portable characters are the characters in ASCII, which are used (among other things) to compose file and command names we need to be portable between the different character sets on the system. In general, we assign the portable characters their ASCII values. More importantly, the portable characters must have the same encoding across all the character sets in your system. This last restriction makes sense, if you think about it: If a "J" is 0x4A in one character set, but 0xD1 in a different character set, how can you encode the file name "JeffDraft"?

So, selecting some lines from the character map itself, we have:

```
CHARMAP
<exclamation-mark>    \41
<quotation-mark>      \42
<number-sign>         \43
<kana_a>              \d167
<kana_i>              \d168
<j1601>...<j1618>     \x88\x9f
<j1619>               \xe9\xcb
<j1620>...<j1694>     \x88\xb2
END CHARMAP
```

There are four things to notice here: First, the characters in the portable set have preset symbolic names, which are listed in Table 2-3 of the standard. Next, we can add new symbolic names as needed for the nonportable characters, as long as the symbolic names contain only the portable characters. Third, we can provide the character encoding as octal, decimal or hex. Lastly, notice that we have a special syntax for ranges of characters. In this case, the multibyte characters have been given the JIS "kuten" numbers–more or less the row and column in which they appear on the JIS character chart. The encoding is given for the first character in the range, and the remaining characters in the range are formed by incrementing the previous one. This trick works because the symbolic names contain a numeric string, and we increment both the value in the symbolic name and in the character encoding.

The Locale Source File

Varying the character set is only one of the things we need to be able to do. We also need to have a way to describe the language- and location-dependent items, such as the currency and date formats and collation order of the characters. This information is provided in the locale file itself, which along with the charmap file, is compiled by the localedef program into a form readable by the setlocale() function.

A warning before we proceed: In the following discussion, we've included excerpts from the AIX Fr_FR locale source as examples. You'll find this text in /usr/lib/nls/loc/Fr_FR.IBM-850.src on your RS/6000. However, we've dissected the original to illustrate points under discussion. In some places, we've even modified the AIX version to illustrate something a little more clearly. As a result, don't assume the examples here represent a correct implementation of the French locale.

As we've discussed previously, there are six locale categories: LC_COLLATE, which controls collation order; LC_CTYPE, for character types (such as the familiar UNIX ctype functions); LC_MONETARY and LC_NUMERIC, controlling formatting of monetary and numeric information; LC_TIME, to manage time and date formats; and LC_MESSAGES, which controls formats of messages. Not surprisingly, there is a section concerning each of these in the locale.

Let's begin with some excerpts from the LC_CTYPE section of the AIX French locale (see below).

```
LC_CTYPE
upper    <A>;...;<Z>;\
         <A-grave>;<A-circumflex>;<AE>;<C-cedilla>;\
         <E-grave>;<E-acute>;<E-circumflex>;<E-diaresis>;\
         <I-circumflex>;<I-diaresis>;<O-circumflex>;\
         <U-grave>;<U-circumflex>;<U-diaresis>
digit    <zero>;...;<nine>
toupper  (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
         (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
         (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
         (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
         (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);\
         (<z>,<Z>);\
         <a-grave>,<A-grave>);\
         <a-circumflex>,<A-circumflex>);\
         (<ae>,<AE>);\
         <c-cedilla>,<C-cedilla>)
END LC_CTYPE
```

The first and second sections, upper and digit, define the corresponding character classes. There are similar lists for lower, alpha (which defaults to the upper set plus the lower set), alnum (which defaults to alpha plus digit), space, cntrl, punct, print (alnum plus punct plus <space>), graph (alnum plus punct) xdigit and blank. The toupper section defines the mapping for the toupper() function. There can be a corresponding tolower section; if it's missing, the inverse of the toupper mapping is used. Notice that we don't use the character codes, but the symbolic names we've defined in the charmap file. This makes this character classification independent of the character encoding actually used. What do the character classifications affect? First and foremost, the familiar character classification routines from the ctype.h header file. But they also change the way regular expressions work, as we'll see below.

Next, we need to define the collating order of the characters. Obviously, this is important for programs such as sort, but it is also necessary for regular expression ranges, such as [a-æ]. Again, let's look at an excerpt from the French locale:

```
LC_COLLATE
order_start forward

<a>
<a-grave>
<a-circumflex>
<ae>            <a><e>
<b>
<c>
<c-cedilla>
<e>             <e>
<e-grave>       <e>
<e-acute>       <e>
<e-circumflex>  <e>

order_end

END LC_COLLATE
```

First of all, this tells us we have only one sort pass, and it is forward. It tells us to sort the characters in the given order: a, à, â, æ, b, c, ç, followed by e, è, é and ê. However, the last four characters act as an *equivalence class*, and all sort as though they were equal. The character "æ" is sorted as though it were two characters, "a" and "e." (We can add multiple sort passes, with additional equivalence classes, for really strange sorting requirements.) We can also supply some additional rules, for special features, such as:

```
collation-element <M-pref> from <M><c>
order_start
 ...
<M-pref>   <M><a><c>;
 ...
order_end
```

This substitutes the virtual character <M-pref> (which we haven't yet defined in the charmap file) any place the two characters "Mc" appear and sorts this virtual character as though it were the three characters "Mac."

This brings us to numeric formatting. The corresponding fragment of the locale source file is:

```
LC_NUMERIC
decimal_point     <comma>
thousands_sep     <period>
grouping          4;0
END LC_NUMERIC
```

This causes numbers to be formatted with a decimal comma, and period as the grouping separator, used every three digits. For example, "123.4567,89." (Note that printf() will print the correct decimal separator but won't automatically handle the thousands grouping; that's managed with the localeconv() routine.)

LC_TIME is pretty simple: It tells us how the strftime() function and date command format time and date information. It provides the text for substitution into the format strings of those commands, giving us the abbreviated day names (abday), full day names (day), and the abbreviated and full month names (abmon , mon). The LC_TIME specification also gives us the default date formats. For example, again in French:

```
LC_TIME
abday     "<D><i><m>";"<L><u><n>";"<M><a><r>";\
          ...
day       "<D><i><m><a><n><c><h><e>";\
          "<L><u><n><d><i>";\
          "<M><a><r><d><i>";\
          ...
abmon     "<J><a><n>";"<F><e-acute><v>";"<M><a><r>";\
          ...
mon       "<J><a><n><v><i><e><r>";\
          "<F><e-acute><v><r><i><e><r>";\
          "<M><a><r><s>";\
          ...
d_t_fmt   "%a %e %b %H:%M:%S %Z %Y"
d_fmt     "%d.%m.%y"
t_fmt     "%H:%M:%S"
END LC_TIME
```

This also tells us the default date and time format used by date (d_t_fmt) is the familiar one: weekday, followed by day, month, time, time zone and full year, for example, Lun 22 Jun 16:41:23 CDT 1992. Similarly, the date and time formats for the date %x and %X field descriptors are provided by d_fmt and t_fmt.

Lastly, the LC_MESSAGES category provides us with a means to handle messages. However, the messages themselves don't appear in the locale; they appear in message catalogs (which we discussed in our second column, on standard C). Instead, what appears in the locale are the two definitions yesexpr and noexpr, which are used to compare your response to a query like:

```
rm: Do you wish to override protection setting
444 for lifes-work/only-copy?
```

To answer "yes," you'd enter something matching the yesexpr, such as y; similarly, "no" matches the noexpr. For English, these lines are:

```
yesexpr  "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
noexpr   "<circumflex><left-square-bracket><n><N><right-square-bracket>"
```

Notice that we're still using the symbolic characters we defined in the charmap file.

So, except for LC_MONETARY, which has a lot in common with LC_NUMERIC, that's what's in a locale source file.

Now what?

Localedef

The program localedef compiles our charmap and locale files into a form useful to programs upstream, such as the setlocale() routine.

Simply put, we say:

```
localedef -f IBM-850 -i Fr_FR.src Fr_FR
```

which, if we're lucky, and haven't made any errors, generates the French locale for us.

Following the AIX naming conventions, we can create a different version of this locale, using an alternate character set, with the command:

```
localedef -f ISO-8859-1 -i Fr_FR.src fr_FR
```

Now you see why the character mapping is separate, and all the locale categories are defined in terms of symbolic names.

(A warning: This actually won't work for French, since there is not perfect overlap between the characters in the IBM-850 and ISO 8859-1 code sets, and you will end up with unrecognized symbolic character names in your locale and loads of error messages. However, on AIX there *is* a one-to-one mapping between the two different Japanese code sets available. This means AIX uses the same locale source for both versions of the Japanese locale, just changing the charmap file in the localedef invocation.)

Regular Expressions

The other topic we promised to treat in detail is regular expressions, which is probably a bit more interesting than the minutia of locale definitions.

As we explained in the last column, instead of a slightly different regular expression syntax for each of sed, awk, ed, grep and egrep (to name but the tip of the iceberg), there are now simply two kinds of regular expressions defined in POSIX.2: Basic Regular Expressions (BREs)–sort of like what used to be used by grep–and Extended Regular Expressions (EREs)–sort of like what was used by egrep. For those of us who've spent hours reading the regular expression discussion in the ed(1) man page, the following will be mostly familiar.

Regular expressions are made up of strings of characters and metacharacters. Let's use some examples to show some simple REs:

• Single characters match themselves. For example, ABC matches the substring in "Now I've sung my ABCs..."

• Circumflexes and dollar signs are special at the beginning and end of expressions, respectively. ^Allie only matches when the string "Allie" is at the beginning of a line. Similarly, Zoe$ only matches when "Zoe" is at the end of line.

• Also, the dot matches any non-null character, so J.. matches a "J" followed by any two additional characters.

• Asterisk means zero or more of something, except at the beginning of an expression, so *foo matches "*foo," but foo* matches "fo" or "foo" or "fooo" or...Well, you get the idea.

So far, we've just described the rules in the old, familiar AT&T or BSD universe. Once we start adding square brackets, things start to get more interesting.

• The first use of brackets is familiar. Characters grouped between square brackets represent a choice from among a group of characters. So, [XYZ] matches any of the characters X, Y or Z.

• Inside of brackets, we can now use more brackets. We can use collating symbols we defined in the locale with the open/close pair [. .]. So, in a Spanish locale, where the two characters ch act as a single character we can use the expression [a[.ch.]b] to match the single characters a or b or the pair of characters ch.

• We can also find any character in an equivalence class, as defined in the LC_COLLATE category, with open/close pair [= =]. For example, in the locale we defined above, [[=e=]] will match any of e, é, è or ê. However, for most of the locales defined on AIX, these equivalence classes are not defined.

• Lastly, we can find a character class (such as upper or lower) with the [: :] brackets. A good expression to find odd combinations of upper- and lowercase characters in a word would be:

```
[[:upper:]][[:upper:][:lower:]][[:upper:][:lower:]]*
```

That is: an uppercase character, followed by an upper- or lowercase character, followed by zero or more upper- or lowercase characters.

• We can still use the old-style ranges, also, so that the last example could also be rendered (for English) as:

```
[A-Z][A-Za-z][A-Za-z]*
```

• Substitutions are still possible: We can "grab" a subexpression with `\(  \)`, as in:

```
\ ([A-Z][a-z]*\ ) Mc\ ([[:upper:]].* \)
```

and then "redeposit" these subexpressions by back reference:

```
s/ \ ([A-Z][a-z]*\ ) Mc \([[:upper:]].* \)/Mac\ 2, \1/
```

These are useful for transforming lines like "Angus McDonald" into "MacDonald, Angus."

Similarly, we can use the back reference inside the search expression, so `^\(.*\)\1$` finds lines containing two adjacent appearances of a string.

We add several extra wrinkles for EREs, as we'll now explain:

• In addition, to the repeat operator `*`, we add `+` as a repeat character, meaning one or more, and ? meaning zero or one.

• We can enclose EREs in parentheses, so (foo+) is equivalent to foo+.

• We can separate EREs in parentheses by vertical bars, meaning "any." Thus, (foo|bar) will match either "foo" or "bar."

EREs will sound familiar to any of you who regularly use egrep. Added to the powerful new features of collating symbols, equivalence classes and character classes, EREs can be very useful. Expanding our name substitution ex-ample above, we can add some Italian and German variants:

```
s/ \([[:alpha:]]\ ) \ ([dD]' *[[:alpha:]]* )/ \2, \1
s/ \([[:alpha:]]\ ) \ (von [[:alpha:]]* )/ \2, \1/
```

This ensures that names like "d'Andrea," and "von Beethoven" are sorted with their connectors intact.

We've completed our discussion of POSIX.2 with a detailed discussion of the construction of locales and regular expressions, two of the features of the standard that permeate a large number of commands. Next month, we'll move on to the third edition of the X/Open Portability Guide and discuss the X/Open model of internationalization. ▲

Reader Feedback

To help *RS/Magazine* serve you better, please take a few minutes to close the feedback loop by circling the appropriate numbers on the Reader Service card located in the back of this magazine. Rate the following column and feature topics in this issue.

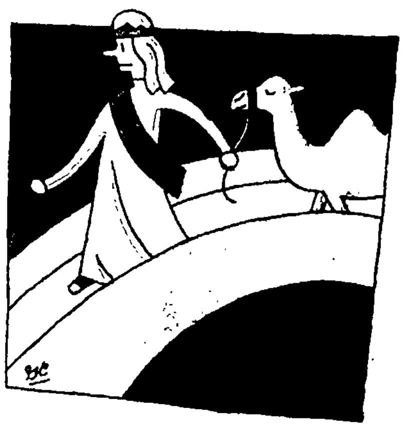
	High	Medium	Low
Features:			
Manufacturing Maze	150	151	152
Multiplatform Multimedia in the Making	153	154	155
Beefing Up RS/6000 Performance	156	157	158
RAID and Reliability	159	160	161
Cyberspace Meets the RS/6000	162	163	164
Columns:			
Q&AIX–Customizing Mwm	165	166	167
Datagrams–Routing Decisions	168	169	170
AIXtensions–Highways and Byways	171	172	173
I18N–Internationalization: POSIX.2 Continued	174	175	176

تدويـــل

Overview and X/Open

by Jeffreys Copeland & Haemer

This is the sixth in our series of columns on i18n–internationalization. Now is a good time to pause and briefly review the material we've covered so far. Then we'll continue with an overview of the X/Open family of standards and specifications.



Our Story So Far

UNIX users are no longer guaranteed to speak American English. As a result, we need to either localize or internationalize software. Localization assumes a specific environment. For example, if the program will always be run in Iceland, we would translate all the text printed by the program to Icelandic, change all the dollar signs printed in numerical amounts to "kr," adjust the date printing routines to say "laugardagur" instead of "Saturday," and make similar hard-wired changes to account for local practice. Among those other changes, we'd need to account for characters peculiar to the local language and how they are sorted into the English alphabet in the ASCII code. Most of the programs you've written started out localized for American English.

Internationalization is more general. We externalize all the text printed by the program and select the correct set of text based on our local language. Similarly, we use external routines to format dates, times and monetary and numerical information. We accept that the collation order of characters doesn't necessarily match their numeric order in the code set we use. We are very careful about assumptions on the order and

Jeffrey Copeland (jeff@itx.isc.com) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* troff *on every continent. Jeffrey S. Haemer* (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. This year, Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

character codes available in our code set.

All the information we need to describe the environment–language, cultural peculiarities of numeric representation, names of the days of the week–is encapsulated in a locale. Locales encapsulate six aspects of the local culture and language:

- Collation sequence–In what order are the characters sorted?
- Character type–What character codes represent numerals, uppercase characters and lower case? How do I convert between them?
- Money–What's the local currency symbol?
- Numbers–Do I use a point or a comma for separating the decimal part of the number? What about a thousands separator?
- Time–What are the names of the weekdays?
- Messages–How do I recognize a yes or no answer? What language do I use for message text?

If we are localizing our programs, we effectively need to port them once for each language/country combination–once for France, twice for Canada, four times for Switzerland–the numbers get large with combined speed. In contrast, if we are internationalizing a program, we write it once and sequester all the environment-specific information outside the program. To port our program for a new country, we add a new locale and translate the external text-and-messages files. The locale is not program-specific–we use standard routines for extracting information from it that can be used by every program to be run in a particular language environment.

In addition to locales, standard C provides other features for internationalization. The most important is a new type, `wchar_t`, which holds a "wide character." Some languages, such as Japanese, have too many characters for all of them to be represented in an 8-bit `char`. A wide character is large enough to hold them all. Standard C also provides us with a handful of routines for converting between bytes and `wchar_t`'s.

Besides supplying Standard C, AIX is also POSIX-conforming. The IEEE POSIX.1 standard is also a C application programming interface, but it ensures a UNIX-like environment by specifying the behavior of the operating-system-level routines absent from Standard C, like `fork()`. This system-call standard connects locales to the operating system environment and specifies how the operating system handles time zones. The standard also gives us a guaranteed way to determine system-dependent information, such as whether the system supports job control or the maximum number of bytes in a time-zone name.

The POSIX.2 standard for UNIX commands and utilities describes the user interface for a UNIX-like system; that is, this Denver-phone-book-sized volume explains how familiar commands like `find` and `ls` work. It also explains, in detail, POSIX's view of locales, and the extensions to regular expressions and the shell's syntax for internationalization and non-ASCII character sets. At the moment, anyone talking about POSIX.2 is talking about a draft–the standard won't be ratified until late in 1992, but AIX is careful to track standards development, and all these features can be found in the current releases.

Message Catalogs

On top of being proactively standards-conforming, AIX is X/Open-branded, which means it conforms to a set of standards and specifications set out in the XPG–the X/Open Portability Guide–and will run any application that sticks to these specifications and interfaces.

What more than POSIX and standard C does the XPG provide? In its current incarnation, XPG3, not much. This isn't because X/Open is conservative, but because many of X/Open's internationalization-related inventions have now been subsumed into the C and POSIX standards.

A few still haven't. The biggest is message catalogs, which we've already grazed up against several times. XPG3 specifies a detailed format for message-text source files, a new message-catalog type, `nl_catd`, and a handful of routines that handle message catalogs, together with an include file, for all the necessary declarations. All are derived from a scheme used a few years back by Hewlett-Packard.

Message-text sources are line-oriented text files, with keywords and comments marked by "`$`"in the first column. "Why not '`#`'?" you're asking. "Why not let comments start anywhere on the line?" Unfortunately, the design is not consistent with other pieces of C and UNIX, though much is similar enough to be confusing. We can only be thankful it isn't "C in column 1."

Messages are grouped into sets, and message lines are a message number (which must be positive –there is no message 0–and in ascending order within message sets), followed by a tab or blank, and a (possibly quoted) line of text. Below is a sample message catalog:

```
$ this is a comment because the dollar-sign is followed by a space (!)
$delset 1
$set 1
$ the next line sets the quote character, which has no default
$quote "
1   "hello, world\n"
```

The XPG specifies two types of routines to manipulate catalogs: C functions and shell-level commands.

The C functions let an application open, close and read specific messages from a message catalog. They have these prototypes:

```
#include <nl_types.h>
nl_catd catopen(const char *name, int oflag);
int catclose(nl_catd catd);
char *catgets(nl_catd catd, int set_id, int msg_id, const char *default_msg);
```

Why something with only a read access routine should need an oflag when it's opened is undisclosed, but rather than requiring either O_RDONLY or "r" as a second parameter, programmers are instructed to set oflag to zero. Thus,

```
printf("hello, world\n");

becomes

catd = catopen("hello.cat", 0);
printf("%s\n", catgets(catd, HELLO_SET, HELLO_MSG, "hello, world"));
catclose(catd);
```

The fourth argument to catgets() is a default message, displayed if no external message is available. During maintenance, be sure to make any changes to the text of messages to both the message catalogs and the default message in the source.

Programmers should note that the message text source files, whose formats are specified, are not the message catalogs, whose formats are not. On AIX, for example, message catalogs are binary files, for efficiency of access. The XPG specifies a shell-level program, gencat, to compile message-text source files into message catalogs or merge them into an existing, already-compiled message catalog.

Normally, AIX does not distribute the message-text files, only the message catalogs, which are located in /usr/lib/nls/msg/$LANG/*program-name*.cat. Actually, message catalogs can be located anywhere. catopen() finds message catalogs by searching NLSPATH, another XPG3 contribution. Although NLSPATH holds a colon-separated list that is searched in order, like PATH the conventions are, again, new and different. The location shown above, for example, is specified by /usr/lib/nls/msg/%L/%N. Take note of both the special syntax–%L for $LC_MESSAGES and %N for *program-name.cat* (or whatever was passed as an argument to catopen()) and of the specification of a file, instead of a directory. As an extra wrinkle, if LC_MESSAGES is set to C, AIX ignores NLSPATH and uses the default message.

With the proliferation of PATH variables–PATH, CDPATH, NLSPATH, and others–we hope that some group will step up to standardizing paths, providing a uniform interface that takes a path and a filename and finds the file, and standardizing shell-level semantics for special cases like :: and %L.

Quo Vadis?

If you've read this far, you're now hip deep in internationalization. We hope you've found our columns useful in threading your way through the available facilities. So where will we go from here?

We still lack a few topics necessary for a good grounding in internationalization under AIX, and nearly all have to do with character handling. The functions provided by Standard C just aren't enough to write really character-set-independent code. So what else do we need and where do we get it?

First, there's the handling of characters inside programs. Standard C provides wide characters, but no routines that manipulate them. To fill this gap, AIX provides an entire suite of routines, part of the upcoming XPG4 specifications, which we'll survey in an upcoming column. In the process, we'll also introduce wint_t and WEOF.

We also think it will prove useful to spend a column discussing the character sets available on AIX. We'll try to provide a feel for the sets that are currently available, how they're laid out, and how to interconvert them.

Careful readers will notice that in all of this we've managed to avoid talking about both how to enter non-ASCII characters in the first place, and how to display and print them. This is not because the subject's trivial. Quite the reverse, and we'll spend a couple of columns discussing these topics.

Finally, we intend to spend a column on some unsolved problems, like bidirectionality, because part of knowing a subject is knowing what's not known.

Final Note

Our thanks to Rocky Mountain Translators of Boulder, CO,who have been kind enough to tell us how to say "i18n" in a variety of languages. ▲

בינאום

XPG4–Wide Characters

by Jeffreys Copeland & Haemer

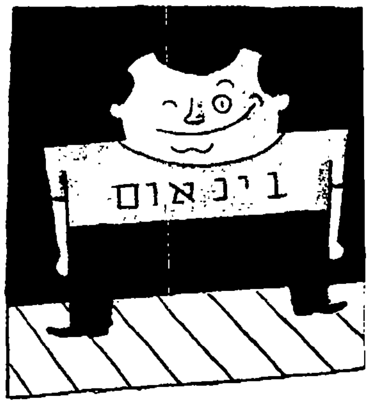
In this month's column, we spend a bit of time constructing a set of fairly serious problems and then quickly demolish them using a suite of utilities designed by X/Open and supplied by AIX.

I Have Good News, and I Have Bad News...

If you've stuck with us this far, you understand that internationalization, or i18n, is an array of sometimes interlocking but often independent problems.

One of those problems is handling character sets so large that individual characters won't fit in a `char`. In theory, one could imagine just increasing the size of a `char` to, say, 32 bits, but so much code depends on having a convenient data type for eight-bit data that C's solution to the problem was to provide a new data type: the wide character, or `wchar_t`.

Two other conditions constrain the design of large character sets in an interesting way. First is the volume of existing, eight-bit ASCII data files. Second is the lack of file types in UNIX. Taken together, these imply that applications must be prepared to handle either multi-byte or single-byte data, and that multi-byte characters must be self-identifying, so that applications that need to break input into individual characters or character strings can find the beginnings and ends of individual characters.



The good news is that there are lots of large character sets to fill this bill. We'll talk more about character sets and character-set design in the next installment of this series, but what makes designing large character sets easy is that the 128 ASCII characters only require seven bits of a byte; all ASCII characters have a high bit of zero. This leaves the eighth bit free to provide an escape mechanism. Most large character sets use the

Jeffrey Copeland (`jeff@itx.isc.com`) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* `troff` *on every continent. Jeffrey S. Haemer* (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. This year, Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

eighth bit of a char as a flag to say "here comes a multibyte character."

The bad news, like most bad news, comes in threes. First, some applications already use the high bit of each character. Others mask it. Applications must be screened for code like this:

```
while (c = getchar())
    c &= 0x7f;
```

and

```
while (c = getc(fp))
    c = c & 0177;
```

and anything else that masks or manipulates the high bit of bytes. Code that leaves the eighth bit of its data untouched is called eight-bit clean. Achieving eight-bit cleanliness is often trivial, but occasionally, as in the case of vi or the Bourne shell, it's an enormous amount of work.

Second, individual applications must now distinguish between large characters and small, since the input stream may contain either, or even a mix of both. Character-handling routines often have to be recoded to examine individual characters and then do different things depending on the size of the character. Routines that manipulate arrays of characters, equating them with arrays of char, fare even worse. No longer is it possible to find the fifth character in array char foo[] by asking for foo[5], and it certainly isn't safe to assume that if cp points to a char, *cp++ advances to the next character.

Third, there are lots of large character sets, each with a different scheme for delimiting characters. This means that when internationalizing code that manipulates characters, the most straightforward approaches often only work for one character set or character-set family. The costs of supporting and maintaining different code for different code sets can become ruinous.

Straw Solutions

Luckily, wchar_ts provide a way around the second problem. Good coding provides a way around the third.

Consider, for example, the following code, which replaces backslashes by colons:

```
char *cp, s[MAXLINE];
FILE *fp;
while (fgets(s, MAXLINE, fp) != NULL) {
    for (cp = s; *cp; cp++)
        if (*cp == '\ ')
            *cp = ':';
    fputs(s, fp);
}
```

Although this works fine for ASCII data, it fails completely for Shifted-JIS, the code set typically used for Japanese on AIX (and the de facto standard on IBM personal computers and their clones), because '\' can be the second byte of a two-byte character.

This, on the other hand, works fine:

```
char s[MAXLINE];
wchar_t *wcp, ws[MAXLINE];
FILE *fp;

while (fgets(s, MAXLINE, fp) != NULL) {
    mbstowcs(ws, s, MAXLINE);
    for (wcp = ws; *wcp; wcp++)
        if (*wcp == L'\')
            *wcp = L':';
    wcstombs(s, ws, MAXLINE);
    fputs(s, fp);
}
```

This, basically, is the whole point of wide characters. Once you convert to wchar_t s internally, you can manipulate characters as units without worrying about the size of their external representations.

Iatrogenic Diseases of Software

George Washington was bled to death by his physicians. Sometimes the cure really is worse than the disease. The simple-minded approach outlined above has two serious problems: convenience and performance. Code internationalized this way is often awkward and, even more often, slow. Contrast, for example, the two programs shown above: On a file of about one megabyte, the second program takes nearly twice as long to run. The solution for this problem is not to convert to wide characters unless you have to. (Patient: "Doctor, it hurts when I do this. What do you recommend?" Doctor: "Don't do that.") In some surprisingly large fraction of cases, like cat, char really *does* mean byte, and the code doesn't care about character boundaries. In others, the characters being sought are guaranteed not to be part of a multibyte character, whatever the character set. (This property is called ASCII transparency. AIX guarantees that all numbers below 0x3f are ASCII transparent; you can still search for the end of an input line, for example, by scanning for the byte '\n'.) Still, sometimes wide characters are the way to go. After all, they're there for a reason. When they are, Standard C's limited array of wide-character-handling routines become a serious limitation. This code fragment illustrates a couple of the reasons:

```
cp = s;
while ((*cp = getchar()) != -1) {
    /* ... */
    if (isupper(*cp))
```

```
        *cp = tolower(*cp);
    /* ... */
    n = strlen(s);
    /* ... */
}
```

First, character-at-a-time I/O is frequent in C programs. Standard C provides the ability to interconvert byte sequences and wide characters, but to do so, you have an entire character in your hands to convert. To convert characters as they appear, an internationalized version must analyze every byte to see when each character begins and ends, and then call `mbtowc()` to perform the conversion. But even once the characters are converted, the code has a problem. C's useful array of character- and string-manipulation functions won't work on wide characters. For functions like `strlen()`, this just means losing the convenience of a standard interface and having to write your own. Based on what we've already said, we could do this instead:

```
for (wcp = wcs; *wcp; wcp++)
    return(wcp - wcs);
```

Functions like `isalpha()` have a far more serious problem because determining character type information depends on the character set. "Well," you think at first, "individual wide characters could be reconverted to their multibyte representations, and...and..." And what? Functions like `isupper()` and `tolower()` have been internationalized–their behavior depends on `LC_CTYPE`–but they take integer arguments. Indeed, the C standard specifies that the value of those arguments must fit into an `unsigned char`. Where's that leave character sets that can't fit into an `unsigned char`?

XPG4

To the rescue comes XPG4, the new, indeed still-unpublished, version of the XPG. In its internationalization facilities, XPG4 adds a suite of facilities for handling wide characters that parallel those traditional C supplies for chars. Fifty-four new interfaces. Name it, it's there. You wanted to do `isupper()`? Now try `iswupper()`. `getchar()`? `getwchar()`. `strstr()`? `wcswcs()`. `vsprintf()`? `vwsprintf()`. And AIX has all of them, including a favorite of ours: `strptime()`.

We've all spent a lot of time writing little parsers to convert input like "25 Dec 92" into a UNIX `tm` structure. `strptime()` takes a string, a format specifier and a pointer to a `tm` structure and parses the string based on the format, *and the current locale!* For example,

```
strptime( s, "%a %d %b", tp );
```

will parse "Fri 25 Dec" or "Friday 25 December" in English, or "fimmtudagur 10 október" in Icelandic. Basically, it's a `scanf()` for times. In practice, it's a backwards `strftime()`, since it uses the same format specifiers; if `strftime()` wrote it, `strptime()` can read it.

Indeed, this panoply of wide-character functions lets us effectively address the third of the three problems we started with. The repertoire of functions is rich enough that nearly all code, with some care, can be written to be *code-set-independent*. `$LANG` can be set at runtime to any of a number of character sets without requiring any special knowledge on the part of the application.

Of course, the library functions may be quite complex, but all of that complexity is hidden from the applications programmer, and the maintenance costs fall to the vendors. The plural is important. Because this specification is an X/Open specification, the character-set-independent code you write for AIX is also nearly vendor-independent. It's guaranteed to be portable to any machine that supplies an X/Open-branded operating system. .

A Small Wrinkle

Some hardware (like the RS/6000) has unsigned characters. Since `EOF` is -1, this means that the following loop is guaranteed to never complete:

```
char c;
while ( (c = getchar()) != EOF )
    putchar( c );
```

Of course, `getchar()` returns an `int`, and `putchar()` takes an `int` as its argument, so the correct loop is:

```
int ci;
while ( (ci = getchar()) != EOF )
    putchar( ci );
```

As you might suspect, just like there's a `getwc()` to match `getc()`, there's a way around this, too. XPG defines a data type `wint_t`, which, on AIX, is defined in `<ctype.h>` with `typedef int wint_t`, and `getwc()` and `putwc()` deal with `wint_ts` rather than `wchar_ts`. Similarly, there's a `WEOF`, which is analogous to `EOF`. So, the internationalized version of the program above is:

```
wint_t ci;
while ( (ci = getwchar()) != WEOF )
    putwchar( ci );
```

Summary

We've talked about XPG4, and how it solves problems with wide characters and provides some other useful utilities. Next time, we'll talk about character sets and character encoding, and how the character represented with six bits in the 1960s now requires 32 bits, and why. ▲

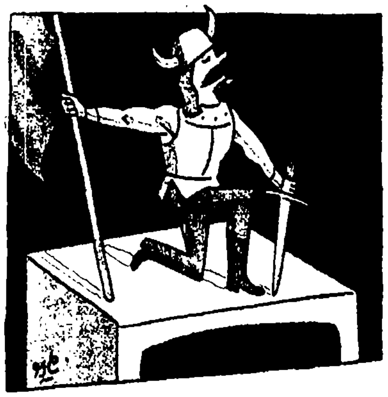
Internationalisering: Code Sets

by Jeffreys Copeland and Haemer

Back in the days when men were men, giants walked the earth, and computers had tubes and read their input from cards, we had ad hoc character codes. On the IBM 1620 (a machine with a maximum memory of 60,000 decimal digits), for example, characters were represented by two decimal digits. An "A" was represented by a 6 followed by 1, "B" was 62. There were 100 possible character codes, but fewer than half were used.

Of course, the character codes on any other machine were different. CDC, for example, had a tightly packed 64-character set, which was the internal character set for the Wirth's first Pascal compiler.

Outside of the computer, things were a little bit clearer. For example, an "A" was represented by a 12 punch and a one punch, a "B" was a 12 punch and a two punch, and so forth, on the old standard 80-column keypunch card. We had numbers, upper-case letters and some punctuation. There was no lower case. There were no accents. But, because we had a set of codes that were the same (more or less) on our cards, we had a way of moving data from one



computer to another. Externally, an "A" was always encoded the same way. Because we didn't have any networking, and because most programs were written in nonportable assembly language or in quasi-portable, numeric FORTRAN anyway, it didn't matter that the internal codes on each machine were different.

In a universe where we have networking, and we must exchange data, character codes are like the Force: They flow through our software and around it, and bind it together. But even in a strictly Roman alphabet, strictly English language environment, we have, if not

Jeffrey Copeland (`jeff@itx.isc.com`) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* `troff` *on every continent. Jeffrey S. Haemer* (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. This year, Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

the tower of Babel, at least the two separate houses of Babel.

ANSI tried to bring order out of chaos by developing the American Standard Code for Information Interchange, a 7-bit code containing upper- and lower-case letters, digits and copious punctuation marks, for a total of 95 graphics characters, along with 33 control characters. You probably have the chart of this character code tacked above your terminal, labeled ASCII.

This wasn't the only solution, because meanwhile, on another planet, IBM was building the System/360 and inventing the Extended Binary Coded Decimal Interchange Code (or EBCDIC) to go with it. It was an 8-bit code, though many of the 256 possible character positions were empty. But it still only handled the same basic characters as ASCII. There still weren't accents, or accented characters. So there was no way to print "façade" or "münster."

Voyages of Discovery 1: Leif Erikson

This month's title is in Swedish, but you may notice it doesn't have any accented characters. So it's possible to do a lot of European text without the "funny characters," but not everything.

The Europeans came up with a solution of their own, in the form of ISO 646, a version of ASCII supporting national variations. They replaced things such as the curly braces and vertical bars with other useful characters so they could produce names such as "Bjørn." Other national versions replaced "$" with "£." There was even an international reference version (or IRV) of ISO 646, which was identical to U.S. ASCII. But to show you how political the standards process can be, it was blocked from adoption by the Soviet Union for a number of years because it contained a dollar sign.

But these national ASCII variations still didn't do the trick because the character codes that produced "Bjørn" in Norway might give us "Bjçrn" in France.

How could software cope? By using a single character code that had all the useful European characters in it. Thus, we have ISO 8859-1, (often called ISO Latin-1) which, the standard tells us, supports "typical office applications" in at least Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish and Swedish. It does this by giving us ASCII in positions 0 through 127, and then, in positions 140 through 255, all the characters we need to write the roman alphabet-based European languages: å, ç, ü, the French quotation marks « and », and so on.

In fact, we have a whole family of ISO 8859 character sets: 8859-2 gives us characters for Albania, Czechoslovakia and other eastern European countries. 8859-3 and -4 provide other Latin languages, such as Turkish and Latvian. 8859-5, -6, -7 and -8 provide ASCII in their lower halves, and Cyrillic, Arabic, Greek and Hebrew, respectively, in their upper halves.

The useful feature that ties these standards together is that if a character exists in multiple members of the 8859 family, it appears in the same place in all of them. For example, "é" appears as 0xE9 hex in all four of 8859-1, -2, -3 and -4, and "Ç" appears as 0xC7 hex in 8859-1, -2 and -3. A second useful feature is that, for the most part, the conversion from upper-case to lower-case letters is accomplished by adding 040 octal or 0x20 hex, just as with ASCII. (The exceptions to this rule are Arabic and Hebrew, where there are not upper- and lower-case letters, and Cyrillic, which has too many letters, so for some Cyrillic characters, we add 0x50 hex to convert them to lower case.)

Meanwhile, IBM had an independent solution. The IBM PC-850 code set uses positions above 0177 octal for the characters used in western European languages. But it does so in no particular pattern: "C" is 0200, and "c" is 0207; "u" is 0201, but "U" is 0232. PC-850 also gives us some of the graphics characters for drawing box outlines that the original PC graphics characters provided.

Locales using both ISO 8859 and IBM-850 are available on AIX. This raises the first problem: If I've entered a file using a command like:

```
LANG=Fr_FR.IBM-850 vi foo
```

how can I print it on a printer or send it to a colleague's computer that only understands ISO 8859-1? Enter iconv, which is provided by the standard and converts between code sets. By using the simple command:

```
iconv -f IBM-850 -t ISO8859-1 foo | lp
```

I can print that document on a printer that only knows about the ISO code set.

Voyages of Discovery 2: Marco Polo

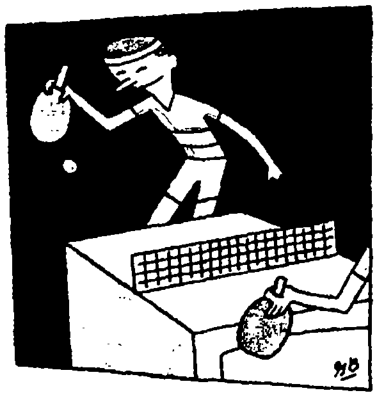
Our friends in the Orient have a tougher problem. They have large alphabets of complicated characters. How large? Traditional Chinese, like they use in Taiwan, has roughly 13,000 Hanzi. China uses a simplified alphabet of Hanzi but still has upwards of 6,700 characters. Japanese has 6,400 Kanji in daily use and needs to allow room for specialized local characters. Korea uses slightly fewer than 6,400 characters.

Obviously, this number of characters can't be expressed in a single byte. So, for example, the Japanese character standard, JIS X0208, uses two bytes to express each character. To shift into and out of ISO 646, that is, ASCII, an escape sequence is used. A variant of JIS, called Shifted-JIS, or SJIS, skips the escape sequence but guarantees that the first byte of each

國際化 Input and Output

by Jeffreys Copeland and Haemer

Now that we've talked about manipulating non-English characters inside your programs, and the code sets that are used to represent them, we need to talk about how to get those characters into the computer. Once we've discussed that, we can talk about how to get them back out of the computer.



Moving the Cursor

At the command prompt, type the characters `pwd`. Anything happen? (If it did, you also typed a carriage return, which we didn't say to type, so start again.) Go get some coffee. Come back. Anything happen yet? (If it did, you needed the coffee. Start again and go get another.) Now type a carriage return. Clearly, someone is waiting until the carriage return to process lines.

Or are they? Type `pwlb^H^Hd<carriage return>`. Someone knows to process "`^H`" (control-H) without waiting for a carriage return. To reinforce this, type in `foobarmumble^U` (if your erase and kill characters are set to something other than "^H" and "^U," use them). Is the shell doing this? Let's explore further.

Type `cat<carriage return>`. Now type in anything you want, but be sure to throw in some "^U" and "^H" characters. Experimenting with some other filters, like `awk` and `nl`, should convince you that the erase and kill processing is widespread. Each application could certainly be performing its own erase and kill processing, but it isn't: This job is being handled by the tty driver, which sits between the

Jeffrey Copeland (`jeff@itx.isc.com`) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* `troff` *on every continent. Jeffrey S. Haemer* (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

terminal and the application.

As one more experiment, try creating a file with a backspace, like this.

```
$ echo "1234x \0105" > foo
$ cat foo
$ cat -v foo # to make backspaces visible
$ wc < foo
```

Now invoke wc again, but with input from the terminal, and type the characters 1234x^H5, followed by a carriage return and a pair of control-Ds to terminate input. This time, wc thinks this input is two characters shorter; this time, wc never saw the "x" and the "^H."

What the heck does this have to do with internationalization? Just this: Until now in this series, we've talked about application-level character processing, but the kernel also processes characters. We know kernel hackers who would object to calling drivers part of the kernel, and driver writers who would object to calling line disciplines part of the driver. We're applications programmers and use "kernel" to mean anything not in user space. On an internationalized system, that character processing can suffer dramatic changes. For example, erase processing must move the cursor backwards, erase the character on the screen, and remove the erased character from the buffer inside the driver that's storing a line of characters to send on to the application. For characters that are more than one byte long, the driver must remove all the bytes. For Kanji characters, which have double-width glyphs on the screen, the driver must know how much to erase from the entire on-screen representation.

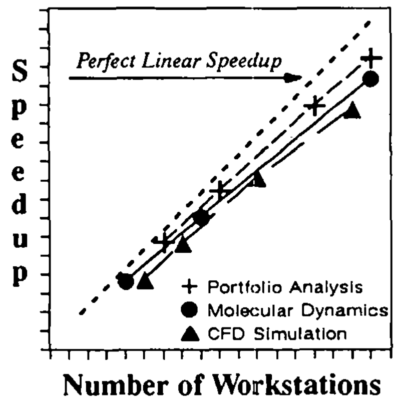
You're thinking, "That's nice. Luckily, I don't write drivers." Time for another experiment. Start vi (or any other screen editor) on an existing file. Now, in command mode, type "^H." The cursor moves without your having to type a carriage return. Indeed, all of the commands are instantly processed by the application without carriage returns, which means that the application, not the kernel, has to manage the input, the cursor and the text displayed on the screen. Raw-mode applications–applications that see all characters as they're typed in, with no intermediate processing by the driver–actually have to worry about both internal character boundaries and the boundaries of glyphs used to represent them on screen. (One selling point of SJIS, the de facto standard code set for Japanese personal computers, is that the number of bytes used to represent each character is the same as its width on the screen.) Raw mode applications aren't common, but they're important: cu, vi, ksh, rn and many games are familiar examples. The curses library, too, needs to pay close attention to glyphs. If you will write or maintain applications that take charge of displaying characters and managing the screen, you, too, will have to worry about such things. Become familiar with wcwidth() and wcswidth(), which give display widths of their wchar_t and wchar_t* arguments.

Input Methods

There's one more question to ask about input processing before we leave the topic: What do you do when you have 50,000 possible characters to enter and a 101-key keyboard? In general, processing new character sets has to go hand in hand with being able to type them, and the basic assumptions of American English in software that we've discussed in earlier parts of this series are echoed by basic assumptions of American English in the design of the keyboard.

The solution varies with the problem. Let's take Japanese as an example. First, the Japanese keyboard is almost unchanged. The standard-issue IBM RISC System/6000 terminal for Japan has familiar-looking

NETWORK SUPER COMPUTER



SCIENTIFIC's Network Linda® lets you combine your IBM RS/6000 workstations into a powerful parallel processor and puts idle network cycles to work on your most difficult jobs.

Linda's six simple commands complement existing programs and standard languages to deliver supercomputer speedups on a wide range of applications, from molecular modeling and seismic analysis to financial simulations, flow codes, and distributed databases.

SCIENTIFIC also offers Linda systems for Sun, DEC, SGI, and HP workstations--not to mention most multiprocessors, including IBM's own Power Visualization System--so your programs can run in parallel on the high performance platform of your choice today <u>and</u> tomorrow.

If you'd like to learn more about Network Linda, please contact us at any time:

SCIENTIFIC
COMPUTING ASSOCIATES, INC.
One Century Tower
265 Church Street, 10th Floor
New Haven CT 06510-7010
203-777-7442
email: software@sca.com

Circle No. 26 on Inquiry Card

ASCII keycaps. There are a few extra shift keys, and each keycap is labeled with an additional Hiragana character in the lower-right-hand corner. Some keys have a second Hiragana character, accessible through a shift key.

To type Hiragana, you toggle one of the extra "Hiragana lock" keys, and each key sends the Hiragana character represented on its keycap. Hiragana stands for Japanese syllables; all Japanese can be written in Hiragana (beginning childrens' readers often are), but it isn't. Normal Japanese is written in a mix of Hiragana, Katakana and Kanji. Katakana characters map one-to-one onto the Hiragana set. (Pierre MacKay at the University of Washington points out that the Japanese had a perfectly good alphabet, and then they hopelessly complicated their lives by stealing ideographs from the Chinese.)

To type a Katakana character, you press another key–it's about where your "Caps Lock" key is, and it's labeled "Katakana," in Katakana characters, which makes it easy to find. Each key now sends the Katakana character that corresponds to the Hiragana on the keycap.

Kanji is the difficult problem. Individual Kanji characters, borrowed from Chinese, are ideographs that stand for whole words. Although the ideographs can be combined to make new words, the combination is decidedly not phonetic. "Tokyo," for example, is made up of the Kanji for "East" and "capital." When used by itself, the first character still means "East," but it's pronounced "higashi." (Well, sometimes it's pronounced "Azuma," and it's a proper name, but you get the idea.) There are tens of thousands of Kanji characters. How do you type them in?

The answer is ingenious: phonetically. To type in "Fujiyama," first type in the four syllables (fu-ji-ya-ma), then hit a special conversion key and the four Hiragana are transformed into the two Kanji normally used to write the word. Syllables can be typed either in Kana (Hiragana and Katakana are collectively called "Kana") or in English letters–Romaji. When typed in English letters ("fujiyama"), individual syllables appear in Kana on the screen as they are completed. This is not just a concession to Westerners who can't touch-type in Hiragana. The Japanese had off-the-shelf IBM PCs and IBM Selectrics long before they had RS/6000s.

This phonetic approach has one complication: Japanese is a language full of homonyms. It's common to have a typical phonetic spelling correspond to half a dozen or more Kanji. By default, the RS/6000's Kana-to-Kanji Conversion (KKC) system replaces the phonetic spelling by its most frequent Kanji, but then lets you use the space-bar to toggle through alternatives until you find the one you want.

Alternatively, you can put up a menu and choose your selection with the mouse. This most recent choice then becomes the default. For a real-life example, we can sit down at our RS/6000 with a Japanese dictionary and try this out. We begin opening a Kanji terminal window. By default, this starts up in Romaji mode.

英数　半角　R

But if we enter a mixed Hiragana and Romaji mode–that is, we toggle into Hiragana mode, so that characters we enter are displayed as Hiragana, but we are entering them as their Romaji equivalents–we can enter the Kana for "dai ichi" by typing "daiichi" in regular Latin characters. (Dai Ichi Kanyo is a major bank in Japan. The name means "Big One," and the Kanji are simple enough to show up in the following illustrations.)

かな　半角　R

Why are the Kana in reverse video? Because we have also entered Kana-to-Kanji translate mode. Next we hit the "Kanji" key to get the Kanji translation.

かな　半角　R

But these aren't the Kanji for "Dai Ichi." Instead it says "primary" or "first one." So we hit the Kanji key again, and get

かな　半角　R

This actually says what we want. At this point, we hit Enter to select this as the correct Kanji. As you can imagine, this requires both a lot of processing and a lot of memory. In Japan, this front-end processing is often done by a PC running a terminal emulator coupled to an RKC (Romaji-to-Kana Conversion) and a KKC system. To the computer, the PC looks like a very smart terminal generating a stream of mixed Kana, Romaji and fully formed Kanji characters. Such programs are widely available.

Workstations, like the RS/6000, typically also offer a host-based system that is built as an X Window application. In the case of AIX, you invoke X with `LANG=Ja_JP`, and all invocations of `aixterm` are Kanji terminals. (You must start X in the Japanese locale, rather than just starting up your `aixterm` in that locale, otherwise the Kanji keyboard isn't recognized.)

Not all input methods look like this. For another perspective, consider Arabic. Arabic is alphabetic, and its alphabet is only about the size of ours. (Jeff and Jeff's, that is. No slight is intended to readers of this column whose native language isn't English.)

But that's where the similarity ends. First, it's obligatorily cursive–there's usually no space between letters–and the form that a letter is displayed in depends on the letters on either side. A typical letter has four forms–initial, medial, terminal and isolated–that can look as unrelated as "a" and "A" do in English.

But that's an oversimplification. Arabic script also has letters that join on one side but aren't allowed to join on the other and a few special forms for some letter pairs ("la" is a combination that looks like neither "l" nor "a"). All this means that as you type in letters, each letter you type can change the form of letters already on the screen.

Oh, and we forgot to mention that Arabic is written from right to left, except for the numbers, which are written left-to-right, which means that the cursor can be moving merrily rightwards, then suddenly stop and begin pushing the text to the right as it stays stationary.

Surprisingly, all of this can be implemented in the driver, but only if you stick to consonants, which Arabic usually does. If you need to add vowels, you have to put them in over and under the consonants, not in line with them, and the problem becomes far more complex.

AIX provides us with the tools to build a custom input method. We don't have space to deal with the details, but it involves a set of library routines that provide an X Window System interface to an IBM High Function Terminal emulation. It's all neatly documented in the on-line Info Explorer hypertext database.

Output

This leaves us with the problem of how to get those odd characters out of the computer and onto paper.

Briefly, you generally use a printer that knows about the character set you need to print. The `iconv` program we discussed in the last column is helpful for this.

But how does the printer know about the characters? Where do the glyphs come from? Often, they're in ROM as part of the firmware. In the case of Kanji, frequently the characters–the glyphs–come from bit maps standardized by the Japanese Institute for Standards. Vendors such as Adobe provide typographic fonts. Further, because there are so many Kanji, and the fonts are so large, printers supporting Japanese, such as the Fujitsu F6788A, often have built-in disks. Similarly, the IBM 3812 and 3816 both have floppy disks containing their firmware and European fonts.

In our next column, we'll discuss the output question in nauseating detail, by exploring AIX's implementation of `troff`.

Amplification

Vassili Leonov wrote to us to answer the rhetorical question in our very first column: "How is *Izvestia* typeset today?"

He says that *Izvestia* has been typeset by computer for at least 10 years. He can't tell us how exactly, but thinks that a large UNIX-based system is used–the system, he says, is named on the masthead on the back page of each edition. If true, this means attempts by the U.S. Department of Commerce and AT&T to keep UNIX out of the Warsaw Pact countries in the '70s and '80s were not successful.

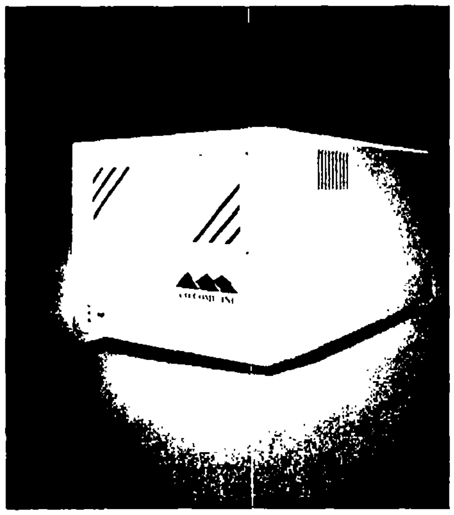
Mr. Leonov also points out that the extensive hacker culture in the former Soviet Union is already filling the market gap for Cyrillic-capable software. As a point of information appropriate to the current article, Vassili reminds us that supporting the Russian keyboard is not the problem–providing a Russian screen font for a particular computer is much more difficult.

Correction

An astute reader pointed out that the header quote in our column on Standard C ("Sow an act, you reap a habit. Sow a habit, and you reap a character. Sow a character and you reap a destiny.") was in fact originally penned by Charles Reade, a British playwright of the last century.

Keep those cards and letters coming, folks. ▲

Smart Idea!



Actual Size: 4" high X 5" wide X 8" Deep

ADD A GIGABYTE PLUS TO YOUR IBM RS/6000

CO COMP'S 3.5 INCH DISK GIGABYTE PLUS SUBSYSTEMS

HUGE! *OVER 1.2 GB PER DISK*
FAST! *AVG. SEEK OF 9.8 ms*
RELIABLE! *500,000 HOUR MTBF*

CO COMP's new Gigabyte Plus storage line offers performance and affordability second to none.

ADD up to 5 years of warranty protection and 24 hour cross ship replacement with CO COMP's Gold Level Service option.

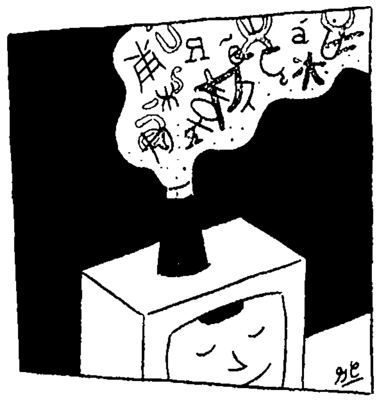
CO COMP, INC.
111 EAST DRAKE ROAD, FORT COLLINS, CO 80525
CALL 1-800-873-0247

Circle No. 7 on Inquiry Card

Διεθνοποίησις The Output Problem

by Jeffreys Copeland and Haemer

Last month, in our ongoing discussion of i18n–internationalization–we talked about input of non-English characters and touched very briefly on how to output them. This month, we will discuss the output problem in more detail. In particular, we'll discuss `troff` in depth.



Why `troff`? Because it's one of our favorite programs; we both use it a lot, including to write these columns. We're very familiar with it: One of us (JLC) has been porting the code to various platforms for so long, he's finally stopped having nightmares about the input routines. And because writing about `troff` gives us a chance to twit our friend and colleague Peter Salus (whose column in this magazine's sister publication, *SunExpert,* we highly recommend) about his preference for the macro package.

Besides those, the serious reasons are that along with its sibling, `nroff`, it's the lingua franca of text processing on UNIX–until recently, it was a standard part of every UNIX system. It's been around for a long time, and many of you are already familiar with it. Most importantly, aside from providing a useful way to get international characters out of the computer, `troff` gives us a good case study of how the interface to an application program changes as a result of internationalization.

We're going to discuss `troff` here, but most of what we're going to say (with the exception of font files) applies just as well to `nroff`. So, if

Jeffrey Copeland (`jeff@itx.isc.com`) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* `troff` *on every continent. Jeffrey S. Haemer* (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

you're a completist, read "nroff and troff" for "troff"; the AIX documentation already has done that transformation for you. Also, a quick aside for the troff-literate before we proceed: In the following, we're going to discuss device-independent troff and its postprocessors as though they were a single program. Ignoring the pipe between them simplifies a great deal of what we plan to discuss. (Remember Donald Knuth's warning in the preface to *The TEXbook*: "Another noteworthy characteristic of this manual is that it doesn't always tell the truth....Once you understand a simple but false rule, it will not be hard to supplement that rule with its exceptions." TEX is the typesetting system Knuth built to replace the hot-lead composition for *The Art of Computer Programming*. It has an interesting and different approach to i18n, which we don't have room to discuss here.) Last warning: What we're discussing here is IBM's approach to internationalizing troff. As far as we know, nobody else has taken exactly this approach, and the i18n'd version of troff we're going to discuss is only available on AIX.

troff Input

Those of you already familiar with troff know that its input consists of text interspersed with escape sequences and directives and macros. Escape sequences almost always start with a backslash. Directives and macros are one or two text characters appearing on a separate line beginning with a dot, possibly followed by arguments. So, for example:

```
This draws a line l'li'
.br
Starting a new line, we have
.BI bold italics
intermixed.
```

The first feature of the new troff in AIX is that the text can be any language. In the example above, the words could just as easily have been kanji, or French. However, the language elements–that is, the directives and escape sequences–remain in the portable character set, ASCII. In particular, names of directives, macros and registers must still be ASCII characters.

This is similar to the principle we discussed in our first column: Like the C programming language, the troff language itself has extensions so that it can deal with non-English circumstances, but the language itself doesn't change. This means that the language is portable across language environments and code sets. In the case of troff, as we'll see, this has the important side effect that the macro packages can be language-independent. (For the non-troff-literate: Macro packages provide additional troff commands built out of primitives. These are essentially libraries of useful commands that native troff doesn't have. The important standard ones are -man, which is used to format UNIX manual pages, -mm, the Bell Labs memorandum macros, which we use to write these articles, -ms, a simplified macro package for memoranda and technical reports, and -me, which was written by Eric Allman as part of the Berkeley system.)

Font Files and How Those Characters Get to the Printer

Normally, troff needs to know some things about each character it puts on the page: It needs to know how big the character is, and what character code produces it on the printer. (The amount of information troff needs is fairly sparse. TEX, on the other hand, also needs to know the exact dimensions of each character's bounding box.) Unfortunately, troff has a fairly baroque way of getting this information.

For each printer there is a device description file, DESC, which contains information about the printer's resolution and special characters available in its font set. Then, for each font available on the printer, there is a file containing the character, its size and the code sent to the printer to produce it. Go ahead and take a look, if you want: The font files are in /usr/lib/font/devibm3816 for the IBM 3816 printer. But what code set is the font file built in? Normally, the native code set of the printer, but in any event, the code set of the font files is specified in the DESC file. This information gets compiled by a supplementary program into a form suitable for troff to read. (AT&T's third release of the Documenters' Workbench software directly reads the text version of these files–this makes life considerably easier.)

That's how troff knows about the characters, but how does the printer figure them out? As we discussed above, characters get sent to the printer in the printer's native character set. For most printers, this means that the characters are already in the printer firmware. PostScript printers with firmware later than Version 23, for example, already have all of the ISO Latin-1 characters in their fonts–basically, you're set unless you've got one of the original Apple LaserWriters.

On the other hand, AIX supports a complete set of downloadable fonts for the IBM 3816 and Hewlett-Packard LaserJet printers. These fonts contain all the characters in the IBM-850 code set. (For the curious, the fonts are based on the Computer Modern fonts Knuth developed for use with TEX, according to the AIX Info hypertext.) This has the advantage (particularly for the IBM printers) of giving a complete set of typographically matched characters across the code set. This wouldn't have been possible in the normal case where the complete set of European characters had to be composed from multiple fonts.

I18n-Specific Features

It remains for us to survey some of the language extensions that allow `troff` to speak multiple languages. A handful of these are examples of localization, rather than internationalization, and we will discuss these first.

Three additional features are needed for `troff` to handle Japanese typesetting.

• `troff` provides *number registers*, for storing numeric information, such as chapter and page numbers. If we have a number register x, containing 142, inserting escape sequence \nx into our text will insert "142". If we've set the format to roman numerals with the alternate format directive

```
.af x i
```

the same escape sequence will insert "cxlii". Similarly, a format of "I" results in "CXLII". However, now on AIX, setting the format to "k" when we are preparing output for a printer with a kanji character set results in

一百四十二

• Normally, `troff` allows you to measure things in terms of em-widths–that is, the width of an "m" in the current font. So, I can draw a line as wide as "mmm" with `\l'3m'`. Similarly, I can draw a 1-inch line with `\l'1i'`. The AIX extensions to `troff` allow me to also say `\l'3K'` to draw a line as wide as three double-width kanji.

• `troff` has always allowed you to set the adjustment mode. I could justify both the left and right margins, for example, by using the command

```
.ad b
```

For kanji typesetting, the Japanese characters are set next to each other without regard for word boundaries. However, in Japanese, we normally invoke something called *kinsoku shori* (roughly "end of line processing") which prevents lines from ending with an open bracket, or beginning with a punctuation mark. To cause this to happen, we invoke the new justification mode

```
.ad k
```

`troff` has string registers, which allow you to store text information to be used later, such as the name of the author of a memo. Some of these registers are preset, when `troff` starts up. For example, traditionally, the name of the typesetter is stored in `\*(.T`. Now, some locale information is available in string registers: `\*(.m` contains the value of LC_MESSAGES and `\*(.t` contains the value of LC_TIME.

This allows us to build text conditionally, based on information in the locale. But it would lead to very complicated macro structures if we had only this way of internationalizing the action of our `troff` input. Getting today's date on a document in a general, locale-independent way, for example, would be a real mess. The macros themselves would have to construct the date based on the value of LC_TIME and the value of the month, day and year number registers `troff` keeps. But there's a better way.

Most macro packages use the number registers containing the date to compose a string containing the text form of the month name. In the -mm macros, for example, the string DT contains something like: "March 13, 1993". But what if we're speaking Swedish? The AIX version provides a new directive, `.Dt`, which takes as its argument a format string suitable for the `strftime()` library routine, which we've discussed before. For example, if we said

```
.Dt %A %d %B %Y
```

`troff` would print "Thursday 20 August 1752" or "fimmtudagur 20 ágúst 1752" or "Jeudi 20 Août 1752", depending on locale.

The last important extension to `troff` for i18n is that `troff` input files can now read message catalogs. The message catalog contains the messages for the macro packages. This is not a feature you're likely to use in casual `troff` use, but if you maintain a macro package, and have access to the message catalog source, and can generate a new catalog, you can add messages to it, too.

Summary

Let's quickly review: `troff` has been internationalized on AIX. The language itself has not changed. This means that the `troff` macros that you wrote in 1985 will still work. The necessary localization features are all nondestructive extensions: They don't stop the existing language features from working. All of the other changes have been extensions, which allow you to produce dates in any locale, for example.

Further Reading

One of us (JLC) still considers the ultimate `troff` reference his dog-eared photocopy of the *Nroff/Troff Users Guide* from the UNIX System III documentation, but he admits that this is probably the wrong documentation for most users. While it is complete in its coverage of base `troff`, it doesn't include information about any i18n features.

The ultimate reference for `troff` on AIX is, of course, the on-line Info hypertext database. Search on `troff`, or on the specific directive name. The "nroff and troff requests for the nroff and troff Commands" manual page is as useful as dogeared hard copy.

Two books we have found to be useful are Narain Gehani's *Document Formatting and Typesetting on the Unix System,* Silicon Press, 1986, ISBN 0-9615336-0, and Dale Dougherty and Tim O'Reilly's *Unix Text Processing*, (Hayden Books, 1987, ISBN 0-672-46291-5). A discussion of the approach Knuth used to allow $\TeX$ to use non-ASCII characters (an interesting related problem) can be found in, for example, *TUGboat*, 10: 3 (November 1989), p325*ff*. ▲

Internacionalizacija Some Problems

by Jeffreys Copeland and Haemer

In the 10 preceding articles, we've talked about what you can do to internationalize software under AIX. In this one, we'll talk about what you can't do, or can't do right. Why? Well, mostly because it may save you some time hunting through the plethora of information we've presented or the even more voluminous IBM documentation. We've all had the frustrating experience of slogging through manuals and not knowing whether there's just no way to do something or there's a way to do it but we just haven't found it. But also because we think that if you're reading this column you may want to know some of the unsolved problems so you can try to solve them. Somebody will have to.

The problems fall into roughly three categories: Clean but hard problems that have been deferred, things that are so messy that there probably isn't a single solution, and problems that have been "solved" already but in messy ways. The good, the bad and the ugly. We'll begin by dismissing the ugly.

Ring Out the Old, Ring in the New

We're writing this on New Year's Eve. This provides a good picture of both the magazine's lead time and our social lives. Vowing to internationalize, like so many New Year's resolutions, shows good intentions but faces the uphill battle

Jeffrey Copeland (jeff@itx.isc.com) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* troff *on every continent.*

Jeffrey S. Haemer (jsh@canary.com) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

of having to contend with everything in your life up until now. We can plan to spend more time with our children and more time exercising, but it isn't as though we weren't using all our time before. We can vow to diet, but what shall we do with that chocolate cake in the refrigerator? (Come to think of it, what is in the fridge? Wait a minute...Ah. Much better. Where were we?) Similarly, we're already up to our ears in work; how in the world will we find the resources to undertake a major internationalization project? We already have enough trouble interchanging our data, maintaining our existing ports and configurations, and administering our existing systems. Throwing in new internationalized platforms, character sets, locales and corresponding hardware will make things messier, not easier–at least for a while.

You often can't usefully exchange data containing Kanji characters with someone who has 7-bit, uninternationalized machines. So what? You can't run RS/6000 binaries on your IBM PS/2 or your IBM mainframe, even if they run AIX. You can't get blood from a turnip. Unfortunately, some folks think that internationalization solves problems like this. It doesn't. Perhaps your boss is such a person. Sometimes appealing to authority works. Tell your boss we said internationalization doesn't solve these problems. Ultimately, change is the price of progress. For new systems to replace the old, the internationalized systems must either bring more benefit than the cost of discarding systems they replace, or internationalization must be a free benefit of another upgrade. A large international market will provide the incentive to replace some older systems. Improved machines like the RS/6000 that happen also to be internationalized will provide the rest.

Our guess is that internationalization will be the typical state of affairs in five years. Until then, expect to deal with machines that aren't internationalized, or are internationalized in ways that your code doesn't deal with properly.

Doctor, It Hurts When I Do This

...the classic response being, "Don't do that." There are a number of problems that AIX simply doesn't try to solve. Let's list a few.

Bidirectional Input: The classic examples of bidirectional input are Arabic and Hebrew. (The buzzword you'll hear is "bi-di.") Here's the problem. Most Arabic text runs right-to-left, instead of left-to-right. Suppose I'm in the middle of a line and want to insert some English, like "antidisestablishmentarianism" or "Happy New Year." Unless I want to type the phrase in backwards, the cursor must switch directions. Because it doesn't make sense to type over the text just entered, or to mess up the right margin (which is where everything is justified in right-to-left text), what's usually done is to have the cursor stand still while the text being entered pushes out to the left. You think you won't want to intermix Arabic or Hebrew and English? Most countries in the Gulf are former British colonies where English is used frequently. Many Levantine and North African countries also use English frequently, and those that don't often want French. Even without that, though, the numbers go right-to-left, so the phone number 512-219-9019 is written 9019-219-512–that is, the segments are "reversed," but not the digits in them–and 7 Rajab 1413–December 31, 1992–is rendered as "1413 bajaR 7" (but in Arabic characters, of course). (Oh, and Arabic has different characters for the numerals. You've been thinking that when you go to Oman you'll at least be able to read the numbers. You've been wrong. Arabs don't use "Arabic numerals," they use "Hindu numerals." Next column, we'll tell you that the Romans didn't speak Pig Latin.)

OK, the cursor behavior should depend on the text you're typing. Next think about a full line, with the cursor in the middle, English (or numbers) to the left and Arabic to the right. Enter another character. How does word-wrap work?

Once you figure that out, let's move on to how the text is stored. Typically, the date above would be stored in entry order, as "7 Rajab 1413." So what sort of regular expression would you use to search for "r" followed physically by "3". And should the field numbering for "sort" go right-to-left or left-to-right? What about in mixed text? Quick: Does the blank between "b"and "3" in "1413 bajaR 7" follow the "b"or the "3"? Same question for the dashes in the phone number. How can you tell if you're a user constructing a shell script?

Not complex enough yet? How about Japanese, which can either have left-to-right horizontal lines, with the first line at the top of the page, English-style, or top-to-bottom vertical lines, with the first line at the right of the page? Right now, computers stick with the English-style input, but it wouldn't be that hard to provide the other; just swap origins and axes. But what if you want to mix the two, the way some newspapers do?

Oh, and Tibetan is classically written in a spiral, which I suppose is either multidirectional or unidirectional, depending on your viewpoint. Luckily, they're not a big market.

Two-Dimensional Composition: At least the bi-di problem is one-dimensional. Arabic and Hebrew share another peculiarity, however, which is the use of vowel points that lie on top of or under the consonants. Usually, only the consonants are written (nly th cnsnnts r wrttn), but some texts (particularly religious texts) are fully pointed, with all the vowels written out, which requires the ability to place characters in two dimensions on the page.

Early in this series, we pointed out that many internationalization problems arise because C and UNIX were designed by people who assumed the use of American English and ASCII. In past columns, we've noted some ways in which the language and its libraries need to be modified to liberate it from these assumptions. Vowel points, like other problems in this section, show how deeply misguided such assumptions can be.

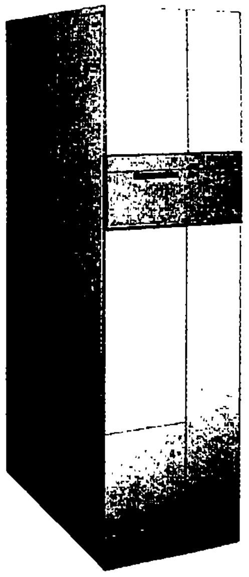
Context Analysis: In English, each letter has two forms, lower and upper case, and the choice of form depends on a variety of complex rules.

In some languages the rules are much simpler; the form depends very explicitly on the context, that is, the surrounding characters. Greek "s" is "ς" at the end of a word, "Â" otherwise. (We just escaped from this recently. In eighteenth-century printed English text, you'll find the nonterminal lower-case "s" looks a bit like "f". Remember your confusion in elementary school when you read Thomas Jefferson's immortal words "When in the courfe of human events..."?)

If you add characters to a word that ends in ς, should you have to change the ς to Â by hand? If you do a `grep`, do you need to use [[=Â=]]? (Or is it [[=ς=]]?)

Before you decide, Hebrew has five such characters. Oh, and Arabic typically has four forms of every letter: initial, medial, terminal and isolated (blanks on each side). (Well, this is oversimplifying. The typical Arabic letter is joined to the letters on either side. However, half a dozen–for example, *alif*–aren't joined to the following letter. This means that an *alif* only has isolated and terminal forms, even though it can come in the middle of a word. A letter following *alif* must, consequently, be either an initial form or an isolated form, depending on what comes after it.)

This problem isn't confined to the Middle East. A quarter of the way around the globe, in Korea, basic forms of individual letters are composed into di- and trisyllabic blocks. The syllable blocks then combine to make words. The ways



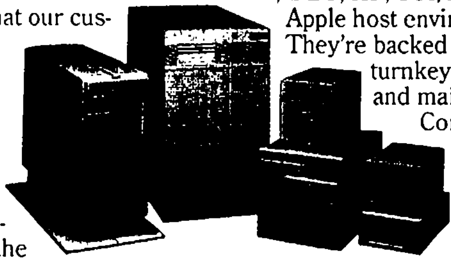
Mass storage.

Mass storage solutions.

Computer Upgrade has always had the hottest optical storage solutions. Like the new multi-functional 160GB CU5160 optical disk autochanger that delivers data access up to two times faster than the competition and has the smallest footprint in its class.

But we think that our customer list says the most about Computer Upgrade's range of products and capabilities. Because we've put mass storage systems in some of the toughest and most sophisticated installations around. Since 1986, we've worked with knowledgeable and demanding customers like Cedar Sinai Medical Center, Chevron USA, DeLoitte & Touche, Los Alamos National Laboratory, NASA, and the U.S. Geological Service.

Computer Upgrade offers OEMs, VARs and end users a diversified variety of stand-alone and automated library subsystem solutions from 128MB to 1TeraByte for Sun, IBM, DEC, HP, SGI, AT&T and Apple host environments. They're backed by nationwide turnkey installation and maintenance.



Computer Upgrade. Our list of satisfied customers says we offer successful mass storage solutions. Can we provide one for you?

FREE! *Evaluating Performance Criteria for Automated Library Systems.* Call for your copy today! **(800) 874-8807**

Computer Upgrade Corporation
(909) 278-5626 • Fax (909) 278-9093

© 1992 Computer Upgrade Corporation All trademarks and copyrights acknowledged.

Circle No. 9 on Inquiry Card

the basic letter forms are written to make blocks depends on the vowel.

Sorting, Again: Earlier, we noted the difficulties of sorting and searching in languages where the order of characters on-screen isn't the same as the order of the characters in the file. Arabic is an easy example of this. It gets worse. There may not be much market for software in Tibet, but Bangkok's a big city, folks. In Thai, vowels sometimes surround the consonants. (For English speakers, this isn't as odd an idea as it sounds. Contrast the words *mad* and *made*, *stat* and *state*, *creat* and *create*. In some odd sense, the long-a is written *a*e*. Ditto for other long vowels.) In some Indian languages, the order of the characters on the page doesn't always correspond to the order in which they're pronounced. (As we pointed out earlier, Ray Swartz says that the world's largest employer is the Indian national railway system, with six million employees. Now there's a market. Payrolls are done by hand.)

If English is your native language, this may leave you unfazed. "Who cares how things are pronounced?" you may be saying. "Sorting and searching is all done on spelling." Wrong again. In Japan, sorting is often done on pronunciation. If you want to look someone up in the phone book, you have to know how their name is pronounced and how everyone else's is, too. On the other hand, the Japanese dictionary is arranged by number of strokes and principal components (or "radicals") in the character. This is not quite as nutty as having to know how to spell something in order to be able to look up how to spell it in the dictionary, the way we do in English, but almost. It certainly makes sort harder to write.

Mixed Character Sets: The RS/6000 provides support for a number of character sets and locales. We've even touched on how to make your code character-set independent, so that single executables can run in any of a number of languages. But what if you want to run in several? If I run in German on Monday, Hebrew on Tuesday, Turkish on Wednesday, and Japanese on Thursday, how do I tell on Friday which files are from which locales?

One simple-minded solution would be to let the operating system give each file a file-type. That's not good enough. Suppose, for example, that we want a single document with Chinese, Korean, Arabic and Armenian? Even 8859-6, which is the standard Arabic character set in the ISO 8859 family, has ASCII in the lower half and Arabic in the upper. That's swell for Kuwait, but little good in Algeria, where users want to intermix Arabic (8859-6) and French (8859-1).

The shining stars on the horizon for this problem are the universal character set, ISO 10646 (easy to remember if you know that ASCII is the American version of ISO 646) and its subset, Unicode. These provide a way of encoding every character in a single character set but bring along their own unsolved problems, which we don't want to go into.

The X Window System: Internationalization of character-based programs has problems. Internationalization of X-based applications has disasters. Most of the solutions we've discussed don't even begin to address the problems in a windowed environment. What happens when you have a form or a menu or graph or picture carefully formatted to look nice and suddenly all the word, phrase and sentence sizes change, perhaps in two dimensions? "Internationalization" in the X Window world often means "8-bit cleanliness" and "Hey, we can display ISO Latin-1!"

Standardized Problems

We've listed several unsolved problems. This is an exaggeration. There are vendors who will sell you solutions for many of these problems. They just aren't standardized solutions. What that means is that you don't have a way to solve them for your application that's guaranteed to work like everyone else's solution. (Technically, we suppose that should be "like anyone else's solution.")

We now hasten to add that this isn't necessarily bad. There isn't a single, standard solution for how to build RISC machines or mainframes. Even in the PC world, there are DOS adherents and Macintosh fans. Sometimes, the marketplace is the right place to thrash out a solution. This brings us to our third category of "unsolved" problems: ones in which the solution design is bad, but prematurely standardized. We've alluded to these problems in earlier columns: X/Open messaging, locales, wide character handling. There are others. These problems have been solved in ways that we suspect will end up costing more money in maintenance than it would have cost to leave the design unstandardized until a genuinely good candidate emerged. The question we keep asking ourselves is, can anything be done to fix the situation? We don't know.

Summary

This is our next-to-last I18N column. We won't pretend that we've covered everything there is about internationalization. Come to think of it, in this column, we've tried to make the point that nobody has. Still, we think we've given you a good start, and we've tried to stay entertaining. We've enjoyed hearing from you

Next month, we'll recap and tell you about our new column starting in August. Tonight, we're off to drink champagne. We hope your New Year's was as happy as ours is about to be. ▲

Internationalization: Wrapping it Up

by Jeffreys Copeland and Haemer

We're now done with our series on internationalization, or i18n. We don't pretend to have discussed everything, but we tried to provide what we promised in our first article–a useful, somewhat pragmatic survey.

We've tried to condition your intuition about what the problems are and what linguistic and cultural problems they stem from, and tried to provide enough feel for the pieces of the solution to let you know where to look when you get stuck.

While we've focused on AIX and the RS/6000, we've tried to make our presentations as general as possible; it would be a pity to wind up with software that can run anywhere in the world except on the Sun on the desk next to you. To that end, we've tried to hammer on a few points: the real problems being addressed and the organizations and standards providing pieces of the solution.

Standard Problems, Standard Solutions

The problems can be pigeonholed into three categories that correspond to the three pieces of an X/Open "ll_CC.codeset" locale name: language, country and character encoding. The most prominent language-related issue is embedded text–error messages, prompt strings and other natural language text inside programs.

```
#include <stdio.h>

main()
{
   printf("hello, world n");
}
```

is a perfect example. Early, localized solutions required making a new version of each program for every language supported, with the embedded string translated. Most internationalized systems today

Jeffrey Copeland (`jeff@itx.isc.com`) *has been with Interactive Systems Corp. in California and Texas since 1981. His specialties include text processing, internationalization and software testing. He is only two-sevenths of the way to his goal of porting* `troff` *on every continent.*

Jeffrey S. Haemer (`jsh@canary.com`) *is an independent consultant based in Boulder, CO. He works, writes and speaks on the interrelated topics of open systems, standards, software portability and porting, and internationalization. Dr. Haemer has been a featured speaker at Usenix, UniForum and Expo Kuwait.*

provide a more flexible approach, storing all text in external data files, with one file per language per program. Users can select what language to print messages in at run time.

This sort of thing isn't novel. We all know that it isn't usually good to embed fixed data values in programs. It's just that hard-wiring "hello, world" always seemed more analogous to "#define PI 3.14159" than "#define MAXSTR 40". It isn't anymore.

As a side effect of this separation of natural-language text and program, it's possible to have more than one version of English text. AIX has two sets of English-language messages, one with traditional UNIX messages, for programmers, and a more verbose version for people who find "*awk: bailing out near line 27*" intimidating, and prefer messages like "*The program you tried to execute has errors, please consult your system administrator.*"

This solution is an example of the general approach. Internationalization today provides run-time selectable, locale-specific behavior by sequestering locale-specific information in data files that are external to the programs themselves. The goal is to write and maintain one binary that works anywhere, and to let vendors and users add support for new locales by changing human-readable data files.

To keep programmers from having to write code that reads and interprets these data files, most behavior changes are hidden inside APIs. For example, the collating order for a locale is specified in an external file, but the function

```
extern int strcoll(char const *, char const *);
```

orders a pair of strings in the current locale, just as `strcmp()` orders ASCII strings. A miscellany of standards organizations and industry consortia that govern the formal definitions of C, operating system interfaces and commands have specified which aspects of the locale affect the behavior of each interface in their domain.

Cultcha

Some aspects of locales are independent of language. For example, Switzerland has four official languages–French, German, Italian and Romansch–but the Swiss all use Swiss francs, rather than the currencies used by the French-speaking French, the German-speaking Germans, the Italian-speaking Italians, and, um, well, you know what we mean. Ditto for date and time formats. Like message text, this formatting information is stored external to the program and is both selectable and accessible at run time, in this case, through the Standard C call

```
extern struct lconv *localeconv();
```

defined in the include file `<locale.h>`. Monetary formats are an example of a grab bag of "cultural" issues touched

PERIPHERAL SOLUTIONS

Only **EXABYTE** could put 2.5GB, 5GB, 12.5GB, & 25GB of tape drive in <u>our</u> box.

Only **Peripheral Solutions** can deliver it to you!

Call 1-800-255-3475

Peripheral Solutions, 151-B Harvey West Blvd, Santa Cruz, CA 95060

EXABYTE®

on by internationalization. Learning which ones can be handled in standardized ways is mostly an exercise in memorization, but an array of environment variables dictated by the POSIX "Shell and Utilities" standard, P1003.2, provide clues. The variables–LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, LC_TIME, LC_MESSAGES–dictate more or less what they sound like. For example, setting LC_TIME determines whether February 4, 1993, is printed as 2/4/93, 4/2/93, 1993-02-04, or any of a number of other popular ways.

The Standards Layer Cake

We've alluded to standards several times, and, indeed, our series has been organized by standards. Let's look back for a moment and review which ones do what.

• **Standard C** standardizes the syntax of C and about a hundred library functions that are operating-system independent. You can find standard C implementations on anything from DOS to IBM mainframes running VM, and every one will let you do `for` loops and `printf()` calls. During standardization, the C standards committee added a new wide-character data type, `wchar_t`; a handful of unpronounceable function calls, like `mbtowc()`, to interconvert wide characters and character strings; interfaces and definitions, like `setlocale()` and `strcoll`, to provide locale-specific behavior; and some specifications of new behaviors for old functions, such as the change in output format of `printf()` based on LC_NUMERIC.

• **POSIX.1** standardizes about a hundred operating-system interfaces– basically the C-programmer's-eye's-view of UNIX. For internationalization, this means things like time-zone handling, tying environment variables to locales (you have to have an operating system to have environment variables), and specifying a few minimum values, like _POSIX_TZNAME_MAX (a minimum, honest), for portability. POSIX.1 also defines the idea of national profiles (the values and options appropriate for a specific country) and provides a sample Danish profile.

Finally, POSIX.5 and POSIX.9, to which we alluded in our POSIX.1 column, make the point that different standards can provide different interfaces to the same functionality. For example, POSIX.9, the FORTRAN binding to POSIX.1, lets FORTRAN programmers find out the value of the environment variable, LANG, with the statement

```
CALL PXFGETENV("LANG", LENNAME, VALUE, LENVAL, IERROR)
```

and POSIX.5, the Ada binding, lets Ada programmers use the environment variable `TZ` to override the default time zone used by the package `POSIX_Calendar`. These two standards let Ada and FORTRAN programmers have access to the full array of POSIX functionality, in FORTRAN and Ada, including internationalization, without having to resort to inserting C code or semantics into their programs.

• **POSIX.2** standardizes the command level, specifying names, options and behaviors for about 100 shell-level commands, like `awk`, `ls` and `c89`. (The regular "about 100" that keeps cropping up is coincidental but mnemonic.) Most POSIX.2 commands vary their behaviors with the values of locale variables; the standard says which ones and how.

POSIX.2 also adds new, international metacharacters to regular expressions, such as [[.ch.]] for the Spanish collating element "ch," and provides a concrete mechanism for specifying locale-specific information about character sets–the `locale` and `charmap` files.

One of the most interesting observations for us about the role of internationalization in POSIX.2 is that internationalization pervades the standard. This shows that internationalization is hidden neither from the user nor from the application programmer. Chances are good that any application you write and any application you run will have visible internationalization changes.

• **X/Open** isn't really a standards organization, but it tries to pick up where formal standards organizations leave off, filling in holes that its members want filled in a canonical way. It offers up both a candidate for messaging interfaces and a set of 54 new interfaces (well, it isn't really a standard) for dealing with wide characters. Because IBM is a member of X/Open, AIX supplies all of these interfaces.

• At the other end of the spectrum, **ISO10646 and its almost-subset Unicode** provide standards for representing all characters in a single character set. This is, in our opinion, a good thing; however, it isn't yet supported by AIX, so we spent a little time discussing the character sets that AIX does currently support, including the ISO Latin-*n* sets, SJIS and EUC.

General Rules

So, what do you do when faced with a program that needs to be internationalized? We dropped some rules of thumb into an earlier column. This seems like a good place to review them.

• Keep a copy of the ANSI C standard or equivalent (such as Plauger's *Standard C Library*) handy.

• Don't worry about internationalizing programs that only deal with byte streams. For example, `cat` can operate on bytes and doesn't need to convert all its data to characters.

• `mbtowc()` can be maddeningly slow. Don't convert to wide characters if you can avoid it. For example, you can still find the end of a string by searching for the NUL byte.

• Remember that on AIX, bytes less than 0x3F are guaranteed to be the ASCII character, so a similar loop works to find a newline.

• If you have to convert to wide characters, don't convert back until you're ready to output. Converting back and forth between bytes and wide characters is expensive.

• Use `mbtowc()` instead of `mblen()`: Both give you the length of the character in bytes, but the first does the conversion as a side effect.

• Where you can process multibyte characters one at a time, do so, using loops like this:

```
for (cp = s; *cp; cp += n) {
 n = mbtowc(&wc,cp,MB_CUR_MAX);
 if( wc = = 'a') /* but 'a' = = L'a' only for ASCII */
 break;
```

• Rethink algorithms when you can. If your program keeps an array of 128 items indexed by character, it's probably impractical to increase the array size to 65,536.

• Don't be too clever. If you can make the code simple by converting to wide characters once on input and on output, without losing performance, do so.

• Don't forget maintenance is a major cost. Clever and complex code is usually wrong.

Corrections

In reviewing our previous columns to write this one, we found a few problems.

In our column covering regular expressions (September 1992), we give an example of an extended regular expression that doesn't demonstrate any of the features of an ERE. A better example would have been the classic regular expression to find strings like *abc* or *aaabbbbc* or *aabbb*; that is, one or more *a*'s followed by one or more *b*'s followed by zero or one *c*. The regular expression

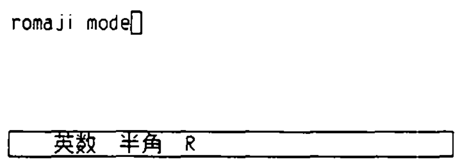
```
a+b+c?
```

will work for finding those strings.

In the same column, we neglected to differentiate between the portable character set and the portable file-name character set. The first is roughly ASCII. The second is the subset of ASCII characters we can use to compose file names and be sure they are portable.

In our discussion of input methods in January, we managed to send incomplete artwork and paste-up directions off to the magazine for our example of kanji input. As you may remember, we used the kana-to-kanji translation function to give us the kanji for dai ichi from the name of a large Japanese bank, Dai Ichi Kangyo. As a result of our goof, only the status line of the window appeared, and those of you who can read Japanese or tried our experiment at home were probably hopelessly confused. (Indeed, we've already had a fax from Paul Contreras at A&I System Co. Ltd., in Tokyo, who asked "Where's the kanji?" By supplying the kanji in his note, Paul also corrected our translation: In this case, "Dai Ichi" is not "big one," as we had said, but is really closer to "the first one.")

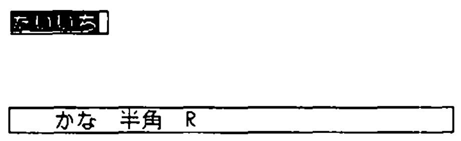
The correct four pictures follow. First, the input window in romaji mode.



```
romaji mode

英数 半角 R
```

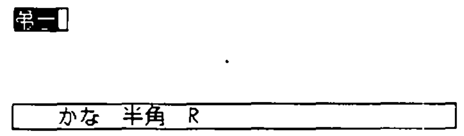
Next, the window containing the kana for dai ichi.



```
だいいち

かな 半角 R
```

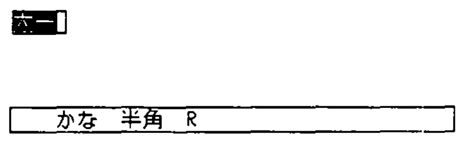
Then, the first attempt at translation to kanji, which gives us the correct kanji pair, for "ordinal number" and "one," respectively; the literal translation is "primary," or "number one."



```
第一

かな 半角 R
```

Lastly, after hitting the space bar again, we get the incorrect kanji, which translates to "big one."



```
大一

かな 半角 R
```

By hitting the space bar twice more, we cycle through the kana and back to the correct kanji.

In December 1992, we discussed the ISO 8859-1 character set. We told you about the characters in positions 0 through 127 and 140 through 255. But what's in positions 128 through 139? Nothing.

Th-th-th-th-th-that's All, Folks!

There you have it. We've covered internationalization in some detail, with a set of references to allow you to explore more on your own. We'll be back in three months with a series on POSIX. In the meantime, columnists Jim DeRoest and Jim Fox will do double duty: While continuing their regular columns, the Jims will jointly write a three-month column delineating the differences between AIX and other flavors of UNIX. ▲