Internationalization in the GNU project

Ulrich Drepper
University of Karlsruhe
Germany

This paper describes the current status of internationalization in the GNU project. It will be shown what is already used in most programs and what programmers could add. The reader will find hints and descriptions of possible pitfalls in internationalizing a program. Beside the known methods we will see descriptions of newly implemented or newly invented functionalities. This includes a short tutorial on how to use GNU `gettext` *and a presentation of the latest changes in the ISO C standard.*

1 Introduction

The form in which a program produces its output reflects a set of cultural conventions, and it is assumed users will know these conventions. Such conventions include both low-level details of how a program interacts with users and their operating environment, such as character set and collation conventions, as well as purely human aesthetic details: the formatting styles used for numbers, monetary amounts, dates and times, and the language used in messages. We call the set of these conventions used in a given country or local area the *locale* describing that place.

Traditionally programs have been written with the local conventions of their authors or intended users encoded directly in the output routines and message strings of the program. This limits the program to being used by people who already know, or are willing to learn, the set of conventions its authors assumed. Most often this means that the program gets used only in the country where it was written. Conversely, such a large body of popular software has been written in the United States that many computer users have been forced to learn English.

With the increasingly multinational and multicultural population of computer users in the modern world, there has been great interest in making programs accessible to a wider community of users by supporting the cultural conventions of many different localities. To do this modularly is a two-step process:

1. Parameterize the program so that it does not encode specific cultural conventions in its output code or a specific language in its message strings. This is called *internationalization*, or *i18n* for short. We will describe the GNU facilities for programmers to internationalize their programs.
2. For each locality where the program will be used, specify the set of formatting conventions to be used by the program's output code, and the catalog of message strings in the appropriate language. This is called *localization*, or *l10n* for short. We will describe the GNU facilities for translators to create message catalogs for internationalized programs, and to create locale specifications describing the conventions of particular localities.

2 The ISO C/POSIX.2 internationalization model

The prevailing programming interface for internationalization in POSIX-compatible systems is that specified in ISO C ([5, 6]), and extended in POSIX.2 ([1, 2]). This model defines *locales* which are specified by a name string, and six categories of localization information for which users may specify their locale of choice.

`LC_CTYPE`
: specifies the character set.

`LC_COLLATE`
: specifies the conventions for sorting order.

`LC_TIME`
: specifies the formatting of dates and times, including the language used for day and month names.

`LC_NUMERIC`
: specifies the formatting of numbers.

`LC_MONETARY`
: specifies the formatting of monetary quantities.

`LC_MESSAGES`
: specifies the language for messages.

Programs that want to use local conventions for their output use the `setlocale` function to specify the locale used for each category of local conventions. The specified locale is then used by the various C library functions which format dates and times, and ISO C specifies the `localeconv` function to return information describing the monetary and numeric formatting conventions of the current locale. The GNU facilities also provide the `nl_langinfo` function to retrieve all items describing local conventions from the current locale.

POSIX.2 specifies a language for writing locale descriptions which are compiled by the `localedef` program into locale specification files. These files can then be chosen by users according to their own locale.

Now some readers will notice that a similar model is described in the X/Open Portability Guides (XPG, [3, 4]). Even if the POSIX definition is clearer and more complete, some systems follow this definition because it was available when the system was defined. The POSIX definition is younger.

3 The `setlocale` function

In <`locale.h`> the following prototype can be found:

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

The first argument `category` specifies one of the available categories or is the special value `LC_ALL`.

The second argument `locale` is a string which tells `setlocale` how to find the locale data which is afterwards used for the specified locale category. The string `locale` can have the following values:

`NULL`
: Query the currently used locale for the specified category. The string which is returned can be used in another `setlocale` call to restore the current program's locale. Please note that it is explicitly allowed to use `LC_ALL` for the `category` parameter.

`""`
: Determine the locale by examining the environment variables `LC_ALL`, `category`, and `LANG` in this order.

`"C"` or `"POSIX"`
: These two values restore the program's locale to the initial value. This is the default value which would be active if no call of `setlocale` were given at all. The call will always succeed.

"identifier"
: Try to use the locale with the name "identifier" for the specified category. If the locale data is available the category will be changed to this locale.

When `category` is `LC_ALL` it is required that *all* categories can be changed. If only one category fails, the whole call fails and nothing is changed. Since the `localedef` tool is used for producing the locale data files this is unusual because files for all categories are created.

With the exception of the first case, the `setlocale` function then returns a pointer to a string describing the newly selected category setting, or it returns `NULL` if the call failed.

3.1 Environment variables for user locale choice

The environment variables which are responsible for determining the locale when `setlocale` is called with the empty string as the argument are worth a closer look. An explanation of the execution of

```
setlocale (LC_CTYPE, "");
```

will make this clear. This kind of call to `setlocale` is especially important because nearly all programs use it. It provides the only way to let the *user* specify the locale.

1. If the environment variable LC_ALL is set for the current process, we use this value. If the setlocale call for this value fails, no attempts with other locale selection strings is done and the locale setting will not change.
2. Same as above, but now for the environment variable named LC_CTYPE.
3. And again the same, but now for the environment variable LANG.
4. The default locale "C" is used.

This hierarchy makes it possible to force all categories to a specific value (setting LC_ALL), provide a generic default value (setting LANG) or specify individual values for the individual categories (setting LC_xxx).

3.2 setlocale in GNU programs

Users of GNU programs should always be able to select the locale the program shall run in. I.e., at the very beginning of every GNU program one should find the line

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

There are only a few exceptions to this: perhaps a program is expected to print the time and date always in the standard form described in the ISO C standard and so the program must use "C" or "POSIX" for the LC_TIME category. In situations like this we have at least two possibilities:

- Only the categories beside LC_TIME are set depending on the users selection.

```
int
main ()
{
  ..
  setlocale (LC_CTYPE, "");
  setlocale (LC_COLLATE, "");
  setlocale (LC_MONETARY, "");
  setlocale (LC_NUMERIC, "");
  setlocale (LC_MESSAGES, "");
  /* LC_TIME is *not* set.  */
  ...
  print_the_date ();
  ...
}
```

  Please note that this is may not be what the user expects. It might be that only a selected number of dates have to be written in the "C" locale format. But changing LC_TIME not at all prevents this.

- setlocale (LC_ALL, "") is still executed at the beginning of the program but the locale for the LC_TIME category is changed before the critical function is called. When the function call is finished the old value can be restored. The code could look like this:

```
int
main ()
{
  ...
  setlocale (LC_ALL, "");
  used_time_locale =
    setlocale (LC_TIME, NULL);
  ...
  setlocale (LC_TIME, "C");
  print_the_date ();
  setlocale (LC_TIME, used_time_locale);
  ...
}
```

4 Access to locale entries: localeconv and nl_langinfo

If a program needs direct access to the individual values of the locale it can use either of the functions localeconv and nl_langinfo. localeconv has a prototype in <locale.h>:

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

The value it returns is a pointer to a structure containing most of the locale entries. This structure is also defined in <locale.h>. localeconv is defined in ISO C and POSIX.2 and so must be available on every ISO C/POSIX compliant system.

The second function's prototype in <langinfo.h> is

```
char *nl_langinfo (nl_item item);
```

This function is *not* defined in POSIX but has been in XPG-compliant systems for a long time. The argument this function takes names a single element of the locale data. Symbolic names for the elements are available in <langinfo.h>. The return value is a string describing the current value. With only a few exceptions all locale elements have a representation as a string.

The decision which function to use should be made by the answer to the question: 'Do I need access to more than one locale element?' Many calls to `nl_langinfo` are certainly slower than one call to `localeconv`, but setting up the returned `lconv` structure will also take some time.

5 What is currently used?

The `setlocale` function is the means for specifying the used locales for the different categories, but it does not directly provide any functionality which could be used for internationalizing an application. Rather, the selected locale information must be used by other functions in the C library. A few such functions existed before the idea of locales arose. When the `"C"` locale is selected these functions behave as in the old days. But some of the locale dependent information need new methods.

For each of the locales we will take a look at the information it provides and how this is intended to be used.

5.1 The LC_CTYPE locale

The `LC_CTYPE` category provides information about the character set used in the current locale. Functions to provide information about the character sets were available since the beginning of the C language. They are defined in the C header file <ctype.h> and many programs use them:

the character class testing functions

`isalnum`, `isalpha`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, and `isxdigit`.

the character mapping functions

`tolower` and `toupper`

5.1.1 It is easy to use

The existence of a character other than the default ASCII character set is completely transparent to the program. Depending on the currently selected locale for the category `LC_CYTPE` the function can returned different values. Example:

ASCII character set

`isalpha('ü') == 0 && toupper('ü') == 'ü'`

ISO-8859-1 character set

`isalpha('ü') != 0 && toupper('ü') == 'Ü'`

It is easy to see that nothing has to be changed to make a program aware of different character sets. A function to capitalize a word, if it only consists of letters, could be written like:

```
int
capitalize (char *word)
{
  char *cp = word;
  while (*cp != '\0' && !isspace (*cp)) {
    if (!isalpha (*cp))
      break;
    ++cp;
  }
  if (cp == word ||
      (*cp != '\0' && !isspace (*cp)))
    return 1;
  *word = toupper (*word);
  return 0;
}
```

Please verify that this function really provides different results depending on the selected locale.

Another use of the character set information is found in the regular expression matching functions (e.g., `re_comp` and `re_exec`). An expression like `[A-Za-z]+` is not portable, and so recent standards extend the syntax of regular expression to allow `[[:isalpha:]]+`. With this expression it would be possible to recognize the word 'regulär', if 'ä' is a letter in the current locale.

5.1.2 ...but it has some drawbacks

Having the character set information accessible this easily has some drawbacks, though. Especially old programs can get into trouble when they are confronted with a non-ASCII character set. E.g., some old programs assume that

'<uppercase character code> + 0x20 = <lowercase character code>'

but this is not always true. Another problem is 8-bit cleanness: some old programs assume the highest bit of the characters passing the `isalpha` test to be cleared. They store additional information in this bit or transmit the character over a not 8-bit-clean channel.

But these problems should not affect any GNU program. GNU programs are expected to be 8-bit clean and ready to be used in internationalized environments. Dependencies like the ones described above should be regarded as bugs.

5.2 The LC_COLLATE locale

While the ASCII character set is organized in a way that the character 'a' has a lower value than the character 'b', the letters in other alphabets do not fit into this scheme. E.g., the encoding value for the letter 'ä' is higher than the value for 'b', but in the lexical order in German the 'ä' is placed in between 'a' and 'b'. For a Swedish locale this would be again a bit different: here the lexical order of the 'ä' character is found after the 'z'.

It gets even more complicated when we follow another aspect of lexical order: sorting is done without paying attention to the case of the character. In a dictionary one will find the words 'fantastic', 'French', and 'gnu' (the animal :-) is this order, although the character 'F' has the lowest encoding value.

Of course a programmer can overcome these problems by writing adequate functions, but making these functions work in *every* locale the user might select is at least very difficult.

The ISO C standard defines two functions for this purpose: `strcoll` and `strxfrm`. The former can be used like a replacement for `strcmp` and in fact it gives exactly the same results in the default locale ("C" or "POSIX"). When a different locale is selected for the LC_COLLATE category and this locale really defines appropriate data, the behaviour of `strcoll` changes. The concrete result will depend on the locale and on the character set this locale is depending on.

The `strcoll` function works by comparing entries of internal tables which are indexed by the characters of both strings. As with `strcmp` the first difference in the value terminates the comparison and the return value (> 0, == 0, or < 0) shows the result. It would require a long description to explain this process completely, but from this vague description one could perhaps gather that it is a rather expensive process. Especially if one string has to be compared several times.

Of course this problem was known to the creators of the `strcoll` functions and so it is not surprising that they also provide a solution. The `strxfrm` function helps to reduce the duplicated works by transforming the input string according the collation tables into another string, which later can be compared against another transformed string by using `strcmp`. I.e., the expensive table processing only has to be done once. After this only simple (and fast) `strcmp` calls have to be done. An example could be this sorting program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSIZE 100

int
xstrcmp (void **p1, void **p2)
{
  return strcoll (*(char**)p1, *(char**)p2);
}

int
main()
{
  char *tab[MAXSIZE];
  size_t cnt = 0, idx;
  while (cnt < MAXSIZE && !feof (stdin))
    {
      size_t len = 0;
      tab[cnt] = NULL;
      getline (&tab[cnt], &len, stdin);
      ++cnt;
    }
  qsort (tab, cnt, sizeof (char *),
         (__compar_fn_t) xstrcmp);
  for (idx = 0; idx < cnt; ++idx)
    fputs (tab[idx], stdout);
  return 0;
}
```

Every string has to be compared more than once.

Each single comparison would required to use the internal collation tables. In the best case the number of collations proportional to $n \times \log n$ where n is the number of strings. A better solution with a linear number of uses of the collation table is:

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSIZE 100

struct str_pair { char *str1, *str2; };

int
xstrcmp (struct str_pair *p1,
         struct str_pair *p2)
{
  return strcmp (p1->str2, p2->str2);
}

int
main ()
{
  struct str_pair tab[MAXSIZE];
  size_t cnt = 0, idx;
  while (cnt < MAXSIZE && !feof (stdin))
    {
      size_t len = 0;
      tab[cnt].str1 = NULL;
      getline (&tab[cnt].str1, &len, stdin);
      tab[cnt].str2 = (char *)
        malloc (1+strxfrm (NULL,
                           tab[cnt].str1,0));
      strxfrm (tab[cnt].str2, tab[cnt].str1,
               INT_MAX);
      ++cnt;
    }
  qsort (tab, cnt, sizeof (struct str_pair),
         (__compar_fn_t) xstrcmp);
  for (idx = 0; idx < cnt; ++idx)
    fputs (tab[idx].str1, stdout);
  return 0;
}
```

Please note that the function `xstrcmp` now calls simply `strcmp`. In the loop for reading the input lines you see a way to deal with non-constant space needed for the transformed string. Calling `strxfrm` with `0` for the third argument does not actually copy anything; instead the number of bytes which would be copied is returned.

Whether transforming the strings before comparing them speeds the program up depends on the actual situation. If one has a database with permanent entries it will pay off. But one thing has to be taken into account here: it is *very* unlikely that the implementations of two different systems' `strxfrm` function use the same encoding of the transformed value. And even more dangerous: changing the locale for category `LC_COLLATE` will most probably invalidate all computed transformed values. Either one has to recompute the value for each architecture/locale or one has to store more than one transformation for each string. When the major concern is the data size the first solution has to be used. Because the `strxfrm` processed version of the string cannot be used for anything but comparing a program normally has to use strings as can be seen in the above example.

5.3 The `LC_TIME` locale

The information collected in the `LC_TIME` category can be differentiated into two groups: concrete information and format descriptions.

5.3.1 Names to print

The former group provides the information to print date and time in text form. It is possible to solve this problem for all cases because there are only a limited number of day and month names.

But date and time formats are too important to be changed by each individual programmer, and so there is no easy way to build own functions for using this information. One could use the `nl_langinfo` function to access the single values, but tasks like computing the day of the week have to be done, too. The organization of the ISO C library suggests using the functions provided in the library itself. For the conversion of a date into a string the main functions are `asctime` and `strftime`. The first function prints a date in the form

```
Sun Sep 16 01:03:52 1973
```

when the default locale is selected. The strings `Sun` and `Sep` are replaced by abbreviated names of the day and month according to the selected locale for category `LC_TIME`. For a French locale this would read

```
Dim Sep 16 01:03:52 1973
```

5.3.2 Formats to use

At this point a French user would of course remark that this is not the form a date is expected to be seen. Indeed, date and time formatting varies widely across the world. To overcome this problem the second group of information in the `LC_TIME` category provides the needed formats. The correct form of the French date string would be

```
Dimanche, 16 septembre 1973, à 01:03:52
```

The `asctime` function itself does not help here, it will not change the order and of the full names of day and month in the French locale is selected (at least this is my information: no standard states this explicitly). It is however always possible to get the date and time printed in the correct format using the `strftime` function after getting the appropriate locale value. The following code would be the internationalized version:

```
char *
i18n_asctime (const struct tm *tp)
{
  static char result[128];
  strftime (result, 128,
            nl_langinfo (D_T_FMT), tp);
  return result;
}
```

The only new thing here is the call to `nl_langinfo`. The argument `D_T_FMT` makes the function to return the locale dependent string for 'date and time formatting'. Similarly there are the symbolic names `T_FMT` and `D_FMT` which allow one to retrieve the format for representing solely the time and date respectively.

Beside these three formats some other non-standardized locale entries are present. But GNU software is expected to be portable and so these values should not be used.

5.3.3 Conclusion

Time and date informations are internally represented in binary form. But this representation is hardly usable for transporting the information from one machine to another. To do this, times and dates must be printed and read. Printing can happen using one of the standardized formats which do not used any texts such as number of seconds after 1970-01-01 or the string "1970-01-01" itself. But if the information also has to be used by the user, it is often convenient to print format as found in GNU Change Logs: `"Mon Dec 25 20:56:39 1995"`.

Now consider this time printed using information from different locales. The function reading these dates will have severe problems because it has to guess the locale the date is printed in. To avoid this all programs printing times and dates which have to be processed by other programs have two possibilities: avoiding any words for weekdays and months by using the ISO 8601 format or switch to the `"C"` locale before printing. These are the only portable ways.

5.4 The `LC_NUMERIC` and `LC_MONETARY` locale

Another area where general support for internationalization can be used in the representation of numbers. We have to distinguish two kinds of numbers here: numbers representing amounts of money and others. The first kind is different from the second because monetary numbers

- are always printed together with a string representing the currency
- (almost) always have a fixed amount of fractional digits printed
- have a fixed format for each locale

5.4.1 Common information of `LC_NUMERIC` and `LC_MONETARY`

The first group of information, which is available in both categories, describe the format of a printed number. One information is the character used as decimal point. Not all countries use the `"."` for this purpose. Many European countries use the comma `","`. The information about this character is automatically used by the families of `printf` and `scanf` functions. For a German locale the output of `printf("%g", 123.456)` would be `123,456`. You cannot avoid this if

LC_NUMERIC is allowed to have an other value but "C". Obviously this can lead to problems. E.g., assume a program which prints some floating point numbers to transport them by email. The receiver will read the numbers using the appropriate program. But what will happen if the producer is American and the receiver German, both using native locales? The number 123.456 will be printed as "123.456", but the reading program only read "123" because the "." is an invalid character.

Another common information in both locales is about grouping the integer parts of numbers. Most (all?) Western countries have groups of 3 digits. The character used as the separator between the groups (called "thousands separator") differs from country to country. In Germany the point is used and one can easily see that this also calls for trouble. In the following code the numbers are written and read using number grouping, which is selected using the flag ' (single quote) in the specifier for the number.

```
printf ("%'g %'g", 1234.56, 42);
```

would produce the output "1,234.56 42". The appropriate call of scanf would be

```
scanf ("%'g %'g", &a, &b);
```

But the result would be wrong for the German receiver: a == 1.234, b is unset and the next character to read is the dot.

The printf and scanf functions use the information from the LC_NUMERIC category. Implementations with this extended functionality are found in most of today's systems.

5.4.2 Additional information in LC_MONETARY

Now for the information only available in the LC_MONETARY category. We don't have to say much about this because this is not used in any known implementation of a C library yet. We need say here only that the available information permits printing and reading monetary number depending on

- character and position of the sign of the number
- string and position of the currency symbol
- additional format styles like parentheses around the number

As the authors of the standard investigated the needs of the different countries in the world it can be assumed that an implementation using the available information can handle all cases ever happening.

We have not seen any program code that uses the information from LC_MONETARY category. And in fact in a standard C library implementation no such function is found.

5.4.3 Conclusion

The programmer has to decide whether it is useful and possible to use the information from the LC_NUMERIC category. If the output is only looked at from the user it is a good idea to use the local habits of writing numbers. But if the output data has to be processed by other programs the fancy style must be avoided. Because the used decimal point character always depends on the current locale, one has to switch to the locale "C" before printing or reading.

The LC_MONETARY category does not raise this problem because this information cannot be used in portable programs.

5.5 The LC_MESSAGES locale

The last standard category described in POSIX.2 is LC_MESSAGES. Here we can make a long story short: the small amount of information it provides is almost useless. The category data describes the strings used for writing "yes" and "no" in the language of the selected locale.

This poor solution can only be found because the standardization committee members could not agree on a general method for handling the translation of messages. This would of course be the overall solution.

The GNU project members agreed on one single method (well, at least some of the members agreed :-). The result is the GNU gettext package which is described in the following section.

The only thing to remember about the LC_MESSAGES locale is that the selected locale is sometimes important for correct execution of the message handling functions. So one should always allow the user to set LC_MESSAGES to the value which corresponds to the language s/he wishes to use.

6 Message Handling

One major problem of computer users is the language the program talks to them in. In the old days when computers were found only in laboratories this was no problem because the users consisted mainly of scientists who are familar with the English language. But computers moved out of the laboratories and many more people, who are not educated to understand this Computer English, are now confronted with them. And the number of computer users will increase even more in the next years.

The bright spirits in the computer industry and at universities certainly recognized this and developed methods to overcome this problem. They all have in common that one single instance of the program can be used in all environments. Only the set of data describing the translated messages of the program has to be changed. This certainly is a much better idea than building a new program for each new language.

There are today three widely used methods. The first is found in the X Window System. Using application default fields one can change the text associated with a single widget. A correctly written application could be completely translated using the method. One drawback of this method is obvious: it is only usable with X programs.

The second method is the function `catgets` and friends. The original definition is found in the XPG 3 standard and implementations are today found in almost all commercial and free Unix implementations. The basic idea is to give each translatable message a number. In a simple data base the translator can later associate with each number a translation. When the user at runtime selects to use a translation the program retrieves through the `catgets` function the appropriate string and uses this instead of the original one.

Maintaining the set of unique numbers for the messages is not that easy, though. People tried to ease this task by allowing symbolic names for each message which tools might later change into a unique number. But this also does not really help to remove the risk of duplicated identifiers and the additional work the programmer has to do while programming. For these reasons GNU decided to use a third method with an available specification.

6.1 The `gettext` function

The description of the `gettext` functions and the tools belonging to them are described and implemented by Sun Microsystems. The first implementation was published in the SunView system and the current one is available in Solaris 2.x. The GNU `gettext` package provides a complete implementation of these functionality, but has *much* more flexibility and many more tools for handling the message catalogs.

The prototype for the simplest of the three functions is

```
const char *gettext (const char *string);
```

The prototype already implies that this function works like a simple filter. When the implicitly defined mapping table contains a translation for `string`, the translated string is returned. If no translation or no mapping table exist, the argument string is returned. This makes the use of this function very easy: one can use `gettext` everywhere a translatable string is used. It always returns a usable string.

Before we get into more details we should first examine why this approach is more promising. Although there is no unique number to index a message catalog, there is of course still an unique key: the string itself. For a programmer this means s/he does not have to keep tables with numbers. Instead each occurence of a translatable string simply has to be wrapped into a call of `gettext`. That's all. And as will be shown later, even this process is eased by the tools in the GNU `gettext` package.

Some people have argued that `catgets` is preferable because it is faster. Using the unique numerical key value the function need not do expensive searches using an argument string. But this argument does not hold: The `gettext` internally use a good hashing technique which makes the string equivalent to an unique numerical key. Even more speed is possible if the GNU C compiler is used. A special feature of this compiler allows to avoid all but the first call to gettext for each string unless another mesage catalog is loaded. This means that many (if not most) programs indeed might run *faster*.

6.2 How to find the message catalog

The description in the previous section only said that the message catalog which gets used in the gettext function is implicitly defined. This is of course true but certainly does not satisfy all needs. It must be possible to specify the catalog currently used.

If no attempt to change the selection is made, the catalog used is

msg-path/*locale*/LC_MESSAGES/messages.po

msg-path is a compile-time defined value. It normally points to /usr/local/share or /usr/share.

locale is the currently selected locale for the category LC_MESSAGES. As we will see later it is also possible to use other categories. Of course the path above will have to be changed to use the name of the other category instead of LC_MESSAGES. In the description of the setlocale function we already saw how the current selection for a specific category is selected. But GNU gettext extends this a bit: instead of relying only on the sequence LC_ALL, LC_xxx, and LANG, we have another environment variable which has the highest precedence of all: LANGUAGE.

Introducing this new variable was necessary because we also extended the format the environment variable values can have. This would not be usable with the previous environment variables because it would break all programs using the locale implementation in the C library.

Before expleining the extended environment variable format, we should first motivate it. Unlike the locale data of the six categories, the message translation has to come with each program anew. There is no general translation. But not all programs come with a complete set of translations. This is especially true for GNU programs because there is no authority demanding a translation to be made. For this reason users should be able to request several languages before falling back to the default language. For example, some Swedes speak better German than English. So the selection for a user like this should be

```
LANGUAGE="sv:de"
```

The used syntax is derived from the PATH variable in the Unix shell and in fact we have to specify something similar to a path. The meaning of this selection is of course "Try to use the Swedish catalog if available; if not, try the German; if even this fails fall back onto the default."

6.3 Selecting the catalog: textdomain and bindtextdomain

From looking at the above implicit file name of the currently used catalog one can see that only one file name is not enough. If all programs used the filename messages.po, that file would have to be extended for each new program. Of course this is not practical and so the programmer has to set a file name for each program separatly using the textdomain functions. The last part of the file name for which we have not presented a way to change is the *msg-path*. Using the bindtextdomain function the programmer can select for each message catalog a new base directory. Having said all this we are now capable of changing all four parts of the file name individually. To illustrate the use of these functions we show here the beginning of a typical GNU program:

```
#include <libintl.h>
#include <locale.h>
int
main (int argc, char *argv[])
{
  /* Set locale via LC_ALL.  */
  setlocale (LC_ALL, "");

  /* Set the text message domain.  */
  bindtextdomain (PACKAGE, LOCALEDIR);
  textdomain (PACKAGE);
  ...
}
```

The setlocale call selects the locale for all categories as the user wants it to be. PACKAGE is normally the name of the GNU package the current program is part of. I.e., all programs of a package share one single message catalog file. By default, LOCALEDIR has the value /usr/local/share but can have an arbitrary value. The only important things are that the subdirectories follow the form needed for the complete file name of the message catalog file and that the installer is allowed to copy the catalogs of the package in the appropriate subdirectories.

After this initialization every call to `gettext` uses the message catalog file

localedir/*locale*/LC_MESSAGES/*PACKAGE*.po

while *locale* is selected from the environment variables.

6.4 The `dgettext` and `dcgettext` functions

Sometimes it is useful to have more than one message catalog available. One can of course switch the name of the catalog file using the `textdomain` function each time it is necessary, but this is confusing. Instead the programmer can use the function `dgettext`. This function takes an additional string as the first argument. This string can be seen as the argument given to an implicit call of `textdomain`. The implicitly selected catalog will not be changed by this function call and so the catalog which is used more often can be the default.

The Sun developers added another possibility to change the used catalog. We already said that a different category than `LC_MESSAGES` can be used. This can be done by using the function `dcgettext` which is like `dgettext` which an additional third argument specifying one of the six available categories. It is hard to think of a useful application of this function, and it also causes a lot of trouble because the C libraries of Sun's Solaris OS before version 2.4 do not contain this function. So the best is merely to avoid it.

6.5 Tool support

It has already been said that one of the big advantages of GNU `gettext` is the good support for the programmer by the utilities. One task which has to be done is the extraction of all the marked strings in the source files. Making it once is error-prone, but doing it after every change to the source file would be extremely painful. Fortunately, there exists the tool `xgettext` which takes over this task completely. This enables the programmer to write rules for the `Makefile` so that s/he never again has to think about extracting the strings. The program recognizes the string as translatable by the wrapping function call. It knows about the standard function names like `gettext` and also can be taught new ones.

The next step after extracting the translatable strings is to translate them. This cannot be done mechanically today. But what happens when we have a translation for all messages and now change the source code? Perhaps we removed, added or changed some of the strings. The extracted list can be updated using `xgettext` but the translation files also have to be updated. This is the task of the `msgmerge` program. Sun provides no counterpart to `msgmerge`, but it is impossible to automate the whole process without it.

`msgmerge` takes two input files: an old translation file and an updated extracted list of the messages. The produced output is the translation file. Writing `Makefile` rules for these program calls is trivial and we successfully used this method in a handful of GNU packages already.

6.6 Usage in GNU packages

Having described all the functions a program needs to provide message translation we now will take a look on the process of making the package `gettext`-aware. If the package follows the GNU coding standards this is very easy. All the maintainer has to do is

- execute the `gettextize` tool, which will copy the `intl/` subdirectory from the GNU `gettext` package, create a new subdirectory `po/`, and copy the `Makefile.in.in` from the GNU `gettext` package into it.
- write a file `po/POTFILES.in` which contains the names of the source files containing translatable strings.
- add to `configure.in` tests needed for the `gettext` library. Until these tests are incorporated in GNU `autoconf` the maintainer has to add the rules found in GNU `gettext`'s `aclocal.m4` file to the own version of this file.
- extend the main `Makefile` to run the appropriate rules in the new `intl` and `po` subdirectory (GNU `automake` can produce correct Makefiles automatically).

- use the Emacs PO mode to mark all translatable strings in the source files of the package (of course this can also be done by hand).

The rest is up to the translators. For translating GNU packages teams are available for a lot of languages. Please consult the documentation of GNU `gettext` for more information on this.

6.6.1 Some conventions

Finally here are some conventions we found useful while developing GNU `gettext` and internationalizing a number of GNU packages. None of the following points are required by the GNU coding standards, so every author is free to follow this or not.

While writing new code it is convenient to leave aside the aspects of internationalization. But to reduce the later work one should at least mark all the translatable strings. The way to do this in C is to use a macro. For the first one could define this macro to return the value of the parameter.

Using a macro has other benefits, so we generally propose to use macros. The macro definition we suggest (because it is supported by the development environment provided in GNU `gettext`) is

```
#define _(String) gettext (String)
```

The short name is very nice to write because it adds only three additional characters for each translatable string and we also haven't found it causes trouble on any system.

And one more argument for using a macro: if it becomes necessary to use more than one translation catalog for a package, the programmer only has to change the macro definiton. In the GNU C library the definition of `_` currently is

```
#define _(String) dgettext("SYS_libc",String)
```

Remark: domain names with a leading `SYS_` as in the above example are reserved for the system.

6.6.2 Special string occurences in C source code

In almost every program one will find situations where the scheme for marking strings as we described it will fail. The following code fragments both will fail:

```
void
f (int idx)
{
  const char *tab[] = {_("one"), _("two"),
                       _("three")};
  puts (tab[idx]);
}
```

and

```
void
f (int idx)
{
  const char *tab[] = {"one", "two",
                       "three"};
  puts (_(tab[idx]));
}
```

The former will fail because initializers cannot be computed at runtime and the second will fail because the strings are not marked and so cannot be extracted by `xgettext`.

The answer is a mixture between both examples:

```
void
f (int idx)
{
  const char *tab[] = {N_("one"), N_("two"),
                       N_("three")};
  puts (_(tab[idx]));
}
```

`N_` is another of the semi-standard macros used in GNU packages. It is defined as the no-op macro with one argument:

```
#define N_(String) (String)
```

The meaning of this macro is to provide a wrapping function call which the `xgettext` program can recognize. So we can extract the strings and at runtime translate them using the `_()` (i.e., `gettext`) function.

6.7 Conclusion

Making a package aware of different languages for its user interface is an easy process. The closer the package follows the GNU coding standard the easier it is. When the GNU `gettext` library and the GNU C compiler are used the result is a more flexible, easier, and faster implementation than any other known to us.

7 All together: GNU C Library

In an ideal system all the functionality we described here should be available by default. The GNU system is intended to be ideal in this respect and so all described functions are part of the GNU C library. Systems running this library are well prepared for a future in which the other side of the earth is only a mouse-click away. This is especially true because we have not described all the internationalization support the GNU C library contains. The following sections give an overview about the other functions and mechanism. But this information should be used with a grain of salt because a lot of this can be changed in the future. The reason for this is that current international standard does not contain official definitions for this functionality. Only drafts exist and ideas vanish as fast as they are invented.

7.1 Extensions to `printf` and `scanf` functions

One thing available for a long time and defined in the XPG 4 standard is another extension to the format strings of `printf` and `scanf`. This extension takes into account one aspect of the grammar of human languages. People knowing more than one language probably noted that the order of the single parts of a sentence differ from one language to another. This influences internationalizing programs when we use message translation. E.g., the correct German translation for the English version

```
printf ("The new year starts at %s %d.",
        "January", 1)
```

is

```
printf ("Das neue Jahr beginnt am %d. %s",
        1, "Januar")
```

We see that the two format specifiers are reversed. How should this case be handled when using `gettext`?

The answer for this are the *positional parameters*. The original program can simply contain

```
printf (_("The new year starts at %s %d."),
        _("January"), 1);
```

The whole point is the translation. One possibility is to translate the format string to `"Das neue Jahr beginnt am %2$d. %1$s"`. I.e., the call to `printf` will actually be

```
printf ("Das neue Jahr beginnt am %2$d. %1$s",
        "Januar", 1)
```

and now we can see what this additional information in the specifiers means. `%2$d` means to use the *second* argument as an integer value to be printed in decimal notation. Similarly `%1$s` means to use the *first* argument as a string pointer.

This is a nice feature that will make language purists moan a bit quieter. Unfortunately we cannot use it in GNU packages. We said that most OSes support this extension, but not all. We don't want to distribute two kinds of translation catalogs for each language in each package. So we would have to ship with each package an implementation of the `printf` and `scanf` functions which can understand this extended format but this is certainly not practical.

7.2 Wide and multi-byte character support

The ISO C standard finally managed to define in its 1995 amendment the handling of character sets that exceed the limited 8 bits currently used. This is one major step but until now there is no complete implementation. Some C libraries already contain the new functions used for this and these functions get their information from the locale data file for category `LC_CTYPE`. The problem is that no POSIX standard correctly defines how to handle all the different kind of character sets out there. Sometimes it is possible to use some special extended character sets (e.g., Unicode or ISO-2022-JP) but it is not possible to generate own locale data in the general way described in POSIX.2.

This is all not very promising but nevertheless it should be described here. First, it will be available in the GNU C library soon and second, some interesting systems will have native support for this, too. Because of the importance of the extended character sets it will be necessary to support them in most GNU utilities. This means that de facto two versions of these programs exist: with or without extended character set support. But as we will see in the description of the functions the C library has to provide, this is not that hard to implement.

7.2.1 New data types

The existing `char` data type is not usable anymore. It is only required to be large enough to store a character from the basic execution character set (which only requires 7 bits). For this reason ISO C defines a new type `wchar_t` which must be large enough to contain a character of the extended character set. What size this means is implementation defined.

Analogous to the promotion of an `char` to an `int` in function calls there exists a new type `wint_t`. Having seen this it is possible to guess a simple way to prepare a program to run using extended character sets. One has to conditionally define some macros which are used wherever today `char` and `int` are used in character and string processing functions.

```
#if ENABLE_WIDE_CHARS
# define CHAR wchar_t
# define INT  wint_t
#else
# define CHAR char
# define INT  int
#endif
```

Obviously there exist a lot of pitfalls. E.g., allocating memory for a string in future might look like this

```
str_ptr =
  (CHAR *) malloc (requested_len
                   * sizeof (CHAR));
```

(please note the additional `sizeof`) but this provides no real problems.

Based on the new character type we also have a new character string type. It is simply an array of `wchar_t`. Instead of using one single byte per character, it now uses a bigger, but fixed number. Usual values are two or four. Using such a string type in current western environments is a bit of a waste of space. The biggest part of the characters used today are still ASCII characters. This means all but the lowest byte of the wide character are unused. And there is another, more severe problem: the interaction of a program with the OS today has to take place using streams of byte characters. While it is possible to think of files as streams of wide characters this is not possible for other devices. Unix sockets cannot be changed to streams of a different unit width without breaking lots of code. The complete implementation of the network layers would have to be rewritten.

Being faced with these arguments, there was a second kind of strings invented: multi-byte strings. These strings are nothing special, only wide character strings encoded using a method whose output is a stream of bytes. There exist quite a lot of possible encodings. One important encoding here is **UTF-8**. It is important to note that there exist *no* definition about which encoding has to be used. The only requirement is that the encoding must never use the null byte for anything but representing itself. The reason is easy to guess: this permits to use the existing string functions of the C library to be used on this new kind of string.

Of course this demand requires that one string representation can be transformed into the other and vice versa. This process is implemented in the functions `mbstowcs` and `wcstombs`. The interface of both functions are similar to those of `strncpy` which means the space for the destination string must have been allocated in advance. Two additional functions (`mbtowc` and `wctomb`) also allow converting single characters, but this is not really used often.

And one last point must be discussed: how to represent constant strings for the new string types. Wide character strings are represented using a format that is already understood by most of today's C compilers: the string constant has the prefix '`L`'. This means the string `L"Hello world!"` does not occupy 12 bytes, but instead 12 times the byte size of a `wchar_t`. But beware that we said *most* systems here: to be portable GNU programs must use macros which get defined to the appropriate values (see below).

For multi-byte character strings there is no such easy solution. This is because no single encoding method for multi-byte characters is prescribed. If one knows the encoding for a special system one can build string constants by writing the corresponding single-byte string. But this is not portable and cannot be used in GNU programs.

Single wide characters can also be defined using the same extension the language. `L'H'` is the wide character form of `'H'`. Please note that using prefixed character and string constants lead to syntax errors when a non-ISO C compiler is used.

All this extended string syntax will require the existing programs to be changed. To remain compatible with systems and compilers not understanding this syntax we have to define conditionally a macro again. There is no written or unwritten standard for this, but using something like

```
#if ENABLE_WIDE_CHARS
# define L_(String) L##String
#else
# define L_(String) String
#endif
```

should be possible. The attentive reader now will notice that this macro deals with the string constants in programs as well as the macro `_()` for the `gettext` function does. You are right, there is a lot more work to do...

At this point it is appropriate to give some advice on using the new types. The choice of what kind of extended string type to be used depends on the situation and the requirements. When the amount of used memory is a determining factor it should be considered to use multi-byte strings. Though this can cause the reverse effect if mainly characters with high codes are used. There is no single best solution for the memory problem.

But the other aspect, speed, can be decided uniquely: wide character strings are *much* faster to process. To understand why it is like this a short look at the problem of determining the length in characters of a multi-byte string suffices. Even if an appropriate function exists it has to test byte for byte whether it is part of a legal multi-byte character. Simply testing for the null byte is completely wrong because this would compute the length in bytes. It is also problematic to look for certain patterns which indicate the beginning of a new multi-byte character. To be correct the function has to test every byte sequence for being a valid multi-byte character. If it is invalid, the computation process has to stop. And these implicit `mbtowc` calls are slow! We can optimize this process if we know the strings are correctly formed, though. You see, the final decision is on the programmer.

7.2.2 Supporting the new types

The new types described in the last section need to be supported by functions equivalent to those existing for the regular character and string types.

New string functions Describing this is easy: all standard string functions are available for wide character strings, only the `str` prefix is replaced by `wcs`. The length of a wide character string `str` can be computed using `wcslen (str)` (note, this is probably not the same as the number of bytes used). Prototypes for all these functions are found in the header file <`wchar.h`>.

The ISO C amendment does not define any of the string functions for multi-byte character strings. Many system nevertheless provide functions like `mbslen` but GNU programs should not rely on this. Of course programmers could use the implementation of the `mbs`... functions which are found in the GNU C library.

Extended character classes and mappings The next group of functions are those corresponding to the `is`... and `to`... functions with prototypes in <`ctype.h`>. It is obvious that the old functions cannot be used because those expect its argument to fit into one byte.

The new functions have prototypes in <`wctype.h`> and expect arguments of the type `wint_t`. The names of the functions only differ from the old functions by an additional 'w' after the prefix: `iswdigit` instead of `isdigit` and `towlower` instead of `tolower`.

But this is not the only new thing with this group of functions. In the 1995 amendment to ISO C the standard C library becomes aware that there is much more in the world but the Latin alphabet. In programs processing Japanese texts, for example, it is necessary to distinguish the characters representing Kanji from those representing Katakana. To make this fast and portable, the ISO C standard requires the system to allow defining additional character classes and mappings. Assume we have a new class `kanji` available. In a program this would be used as in the following example:

```
{
  wchar_t *ws = wide_string;
  wctype_t kanji_class = wctype ("kanji");
  while (*ws != L'\0' &&
         iswctype (*ws, kanji_class))
    ++ws;
  ...
}
```

The function `wctype` extracts from the `LC_CTYPE` locale data the necessary information about this character class. In case the class is not available the function returns a negative value.

The found value for the Kanji class is used in the `iswctype` function, which second argument is of this type `wctype_t`. The return value indicates whether the first argument represents a character of this character class, just like the normal `is`... functions do.

We said above that ISO C also allows you to define your own character mappings. To test for a character mapping the function

```
wctrans_t wctrans (const char *property);
```

is used. A negative result indicates a failure; else the result can be used in the function

```
wint_t towctrans (wint_t wc, wctrans_t desc);
```

which transform its first argument to the appropriate value.

The latest ISO C standard defines some more functions for handling multi-byte and wide character strings. It would go beyond the scope of this paper to describe them all. The functions presented here are the ones one can expect to be widely available soon and so GNU programmers should for now restrict themselves to these functions and identify the locations in their programs where they will have to be used when the program is adapted to use extended character sets.

Input/Output of wide and multi-byte characters Before a program can work on the strings using the functions we just described they have to be written and read in again. This requires that the I/O functions must be extended, too. Again the names are chosen to be very similar to the ones in the standard library: the `printf`-like functions are called ...`wprintf` and the `scanf`-like functions are called ...`wscanf`. The format strings for both new function families are now wide character strings. All known format specifiers are still available. The only difference is that committee decided to fix the security problem in the ...`sprintf` functions: the functions ...`swprintf` now have the interface corresponding to the ...`snprintf` functions.

One stream associated with a `FILE` descriptor at one time can only be used for writing single byte strings (including multi-byte strings) and on the other hand wide-oriented streams. After opening a stream the orientation can be explicitly selected using the `fwide` function or implicitly by using a single-byte or wide character operation. The orientation can be reset by using the `freopen` function.

People writing GNU programs using wide characters should remember that a stream can never be used for single-byte and wide characters at the same time.

The external representation of a wide-oriented stream contains some more pitfalls for the programmer. First, the representation of a wide-oriented stream is not a sequence of wide characters. This would cause problems because of the endianess of different systems and so it was decided to use a multi-byte encoding. The encoding to be used is not specified and this can lead to communication problems between systems with different encodings. The only two points said about the encoding are [6, *section*7.9.3]:

- [*it*] may contain embedded null bytes (unlike multibyte encodings valid for use internal to the program)
- a file need not begin nor end in the initial shift state

The second pitfall comes with file positioning using the `fseek` function. Because the position is counted in bytes it is possible to start reading or writing in the middle of a multi-byte character. Beside this reading after positioning is problematic itself. Because the encoding of the multi-byte characters might depend on a state it is not possible to expect reading to be correct after repositioning the stream. To work around this point the programmer has to use the newer functions fgetpos and fsetpos. These functions take as an argument the pointer to an object of type fpos_t which contains the needed state information. But there is no means for the programmer to set the state herself. Instead an fpos_t has to be initialized by a previous call to fgetpos:

```
{
  fpos_t pos;
  FILE *fp;
  ...
  /* Get current file position.  */
  fgetpos (fp, &pos);
  ...
  /* Reset stream position.  */
  fsetpos (fp, &pos);
  ...
}
```

Please note that the pos variable is not initialized. A value has to be provided by a call to fgetpos. Because this method can already be used in any ISO C environment, programmers should change to use it now. On systems missing these functions one could use

```
#define fpos_t long int
#define fgetpos(fp, pos) \
  ((*(pos) = ftell (fp)) == -1L)
#define fsetpos(fp, pos) \
  fseek ((fp), *(pos), SEEK_SET)
```

You can see that fseek is used only with the third parameter set to SEEK_SET. This limits the functionality but it also makes clear that SEEK_CUR and SEEK_END should never be used with text stream.

User input in the presence of extended character sets While reading a file using a wide character stream is easy we have to take care for one more kind of input: input from the user. How should a user be able to write all the thousands of different characters? There will never be a keyboard with that many keys.

Of course this problem is not new. In Asia where large character sets are used in everyday life for many years exist quite a few methods. Each system brings a new method and no single one can be called standard. The X Window system is kept very general in that it allows the programmer to select (and even provide) the *input method* to use.

Since large character sets are also used in the rest of the world interest in a general solution awoke. One of the most important references is the proposal by Alain LaBonté ([7]). But there is up today no ISO standard available and so user input is another grey zone in internationalization. For GNU programs this is at the first sight not important because the program only receives the cooked characters. But think about documentation for an interactive program in an internationalized environment.

7.2.3 Extended characters in GNU packages

In the last sections we saw that extended character sets can be problematic. At least they require a special handling and we explained what a programmer can do. But we left out three important points which we will enlighten now. All problems are caused by the fact that source packages cannot be automatically transformed during the distribution. The maintainer produces a tar file which is non-textual data and so has to be transported using binary channels.

Non-ASCII characters in documentation files Imagine what happens when a maintainer, call him François, adds a new Change Log entry which includes his name. The cedilla character in his name will have the value 0xE7 as he uses the Latin-1 character set. Let us assume now that the installer uses an Apple computer. Taking a look at the Change Log file s/he will see: FranÁois. This sort of problem is not severe. Nevertheless it should not be ignored.

One possible way around this problem is to state that all packages are shipped encoded using Latin-1. This requires all installers/users to use a transformation tool like GNU recode to view the files. This is not reasonable for several reasons. First, one must not assume such a tool to be available and second, what can authors do when they want to use characters not part of Latin-1?

A better solution would be to assume a unified character set (Unicode or ISO 10646) and use encoded character values in the file. Possible encodings could be MIME quoted printable or **UTF-8**. This solution allows to specify all needed character but still requires a conversion tool or at least a file viewer with decoding capabilities.

Officially no solution to this problem exists but some of the GNU activists intend to use **UTF-8** encoded Unicode characters.

Non-ASCII characters in source files The only place where non-ASCII characters can appear in portable C source files are strings. This can happen for single byte strings but especially for wide character strings. Again here the character codes the installer/compiler will find are the same the author chose.

This problem is different to the previous one because we *must* handle this problem before the program can be used correctly. The GNU coding standard requires that only some very basic utilities can be used for compiling and installing a program and this set does not contain `recode`. Beside this it is not guaranteed that all strings in a source file use the same character set and encoding. So it is not possible to transform the strings in the source file.

This only leaves two ways out: we have to handle this problem at runtime. I.e., all strings containing extended characters must be processed before they can be used in the program. One possibility is to use a translation function. The next release of GNU `recode` will contain such a function, but using it requires to ship its source code along with all packages using it. A second possibility is to use already existing translation facilities: the `gettext` functions. Using this method means shifting the problem from the source file to the translation files.

But still we should require an unique representation for all files in the GNU project. So we propose here to use **UTF-8** encoded Unicode (or ISO 10646) characters wherever needed. To enable the programmer to write the program using the local character set, future GNU `recode` versions will be able to transform C source code into the desired form.

Non-ASCII characters in `gettext` translation files Having non-ASCII characters in translation files is very common. Because the translation problem also must be solved before the program can be successfully used this situation seems to be similar to the one in the last section. But there is one important difference: we always know the character set and encoding used for the translation strings.

Of course it is again possible to use the translation function GNU `recode` will provide. All strings returned by the `gettext` function have to be transformed and so the program will never see an invalid encoded string (please note we said *all* strings will be transformed; this includes the case when no translation is found).

But we have a second option here. It helps to speed up the program by shifting the translation process to the time of installation. When we transform the translation files for `gettext` to the used character set we can return the translations as `gettext` finds them at runtime. We would not need the translation library which reduces the size of the program and saves an (expensive) function call. The drawback of this option is that the program possibly will not work as expected if no translation is used (i.e., the English messages are used) and the messages contain extended characters. But this will certainly be a rare case and could be handled separately.

8 Building locale data files

Now we have seen all the fancy operations which are based on the information available in the locale data of the different categories. It now is time to see how these locale data files are built. But this would cover an whole article on its own and so we restrict ourself to a superficial description of the files, programs and the process. It all is defined in the POSIX standard. P1003.2 defines the basics but the later versions (P1003.2b) correct and extend this.

8.1 The locale definition source files

P1003.2 defines a file format that allows to define all the different informations which are later available in the six categories. These files use a character set independent format. By doing this it is possible to combine it with an (also symbolic) character set description.

Locale definition source files for available countries and also character set descriptions for all major character sets are available via anonymous FTP. Keld Simonsen from the DKUUG takes care of them on behalf of the POSIX working groups.

8.2 The `localedef` program

The `localedef` program takes as input a locale definition source file and a character set description and produces six locale data files, one for each of the categories. The program is tightly coupled with the implementation of the functions in the C library because these have to know about the internal format of the data files. The GNU C library comes with an implementation of the `localedef` program which fulfills all the requirements.

9 Prepare for the future

The previous text tells a lot about the various means to internationalize programs which are widely available. They are likely to be unavailable on only a few systems which are not interesting for the GNU project. We saw ways to adapt programs to the new functions and/or data structures so that they still can be used on old-style systems.

Programs running on GNU systems itself (here we can be more general: systems that use the GNU C library) can take advantage of all known methods because they are (or soon will be) implemented on this system. Because the implementations from the GNU C library can also be used to substitute faulty or missing functions on other systems, programmers could consider to use the fancier functions we presented here if they add useful functionality to the program. It must not be the least developed systems which determine the way of programming. Old systems will most probably never get the functionality we need, e.g., to process wide character strings. Even if the solution is to have two classes of systems, those with and without the needed functionality, GNU has to walk this way. Else the results of the GNU project will not be usable in the future Internet-connected international world.

10 Thank Yous

First of all I want to thank Karl Berry, Roland McGrath, and François Pinard for helping me to write this paper. It would be hardly readable without their work. While developing internationalization support for the GNU C library we had extensive discussions which strongly influenced the result. Roland McGrath, as the main author of the GNU C library, wrote many of the internationalization functions as part of this library. The first try of a message handling package (which later resulted in GNU `gettext`) was written by Jim Meyering and `gettext` itself would not be what it is today without the unselfish help of François Pinard, Peter Miller and the members of the GNITS mailing list. Many thanks to all of them.

11 Biography

Ulrich Drepper, *1968 in Hamm, Germany. Currently student of Computer Science at the University of Karlsruhe, Germany. Author of the GNU `gettext` package, maintainer of GNU `sharutils` and regular contributor to GNU and Linux C libraries. Most recently, involved in internationalization affairs for GNU and international standardization committees. Main interests beside this are optimizing libraries/compilers by neat assembler code. Long-term goal: writing a Go program. Best email address: `drepper@gnu.ai.mit.edu`.

A Bibliography

[1] **POSIX 1003.2** International standard, IEEE, 1992.

[2] **POSIX 1003.2b** Draft, International standard, IEEE, 1995.

[3] **X/Open Portability Guide**, Part 3, Supplementary Definitions, X/Open Company Ltd, Reading UK, ISBN 0-13-685850-3. *Defines* `catgets` *message handling functions.*

[4] **X/Open Portability Guide**, Part 4, Programming Languages, X/Open Company Ltd, Reading UK, ISBN 0-13-685868-6. *Definition of the standard C library.*

[5] **ISO C-1989**, ISO/IEC 9899, International standard. *Definition of the C programming language and its library.*

[6] **ISO C, amendment 1**, 1995, International standard. *Defines extensions to ISO/IEC 9899 to provide better support for extended character sets.*

[7] **Input methods to enter characters from the repertoire of ISO/IEC 10646 with a keyboard or other input devices**, Draft for POSIX working group JTC1/SC18/WG9, Alain LaBonté, 1995. http://www-rocq.inria.fr/˜deschamp/www/divers/ALB-WD.html

B References

GNU programs:

- GNU C Library: C library supporting ISO C-1989, P1003.1-1990, most of P1003.2-1992, upwardly compatible with 4.4BSD, including many System V functions, plus GNU extensions. Available on `prep.ai.mit.edu` in `/pub/gnu/glibc-1.09.tar.gz` via anonymous FTP.
- GNU gettext: Portable library for message translation including tools to aid writing and to allow fully automated building of translation files. Available on `prep.ai.mit.edu` in `/pub/gnu/gettext-0.10.tar.gz` via anonymous FTP.
- GNU recode: Convert files between character sets and usages. When exact transliterations are not possible, it may get rid of the offending characters or fall back on approximations. Available on `prep.ai.mit.edu` in `/pub/gnu/recode-3.4.tar.gz` via anonymous FTP.

Other references:

- Keld Simonsen's archive of POSIX character set and locale definition source files. Take a look at `(ftp://dkuug.dk/i18n)` using anonymous FTP.

C Trademarks

Unix is a registered trademark of X/Open Company Limited.

SunView, Solaris are registered trademarks of Sun Microsystems.