GNU gettext tools, version 0.10

Native Language Support Library and Tools
Edition 0.10, 26 November

Ulrich Drepper
Jim Meyering
Francois Pinard

1 Introduction

> This manual is still in *DRAFT* state. Some sections are still empty, or almost. We keep merging material from other sources (essentially email folders) while the proper integration of this material is delayed.

In this manual, we use *he* when speaking of the programmer or maintainer, *she* when speaking of the translator, and *they* when speaking of the installers or end users of the translated program. This is only a convenience for clarifying the documentation. It is absolutely not meant to imply that some roles are more appropriate to males or females. Besides, as you might guess, GNU `gettext` is meant to be useful for people using computers, whatever their sex, race, religion or nationality!

This chapter explains what are the goals seeked by the mere existence of GNU `gettext`. Then, it explains a few wide concepts around Native Language Support, and situates message translation in regard to other aspects of national and cultural variance, as applicable to programs. It also surveys what are those files used to convey translations. It explains how the various tools interrelate in the initial generation for these files, and later, how the maintenance cycle usually operate.

1.1 The Purpose of GNU `gettext`

Usually, programs are written and documented in English, and use English at execution time for interacting with users. This is true not only from within GNU, but also in a great deal of commercial and free software. Using a common language is quite handy for communication between developers, maintainers and users from all countries. On the other hand, most people are less comfortable with English than with their own native language, and would rather prefer using their mother tongue for day to day's work, as far as possible. Many would simply *love* seeing their computer screen showing a lot less of English, and far more of their own spoken language.

However, to some people, this dream might appear so far fetched that they may believe it is not even worth spending time thinking about it, and they have no confidence at all that the dream might ever become true. Many did not loose hope yet, and organized themselves. The GNU Translation Project is a formalization of this hope into a workable structure, which has a good chance to get all of us nearer the achievement of a truly multi-lingual set of programs.

GNU `gettext` is an important step for the GNU Translation Project, as it is an asset on which we may build many other steps. This package offers to programmers, translators and even users, a well integrated set of tools and documentation. Specifically, the GNU `gettext` utilities are a set of tools that provides a framework to help other GNU packages produce multi-lingual messages. These tools include a set of conventions about how programs should be written to support message catalogs, a directory and file naming organization for the message catalogs themselves, a runtime library supporting the retrieval of translated messages, and a few stand-alone programs to massage in various ways the sets of translatable strings, or already translated strings. A special GNU Emacs mode also helps interested parties into preparing these sets, or bringing them up to date.

GNU `gettext` is designed so it minimizes the impact of internationalization on program sources, keeping this impact as small and hardly noticeable as possible. Internationalization has better chances of succeeding if it is very light weighted, or at least, appear to be so, when looking at program sources.

The GNU Translation Project also uses the GNU `gettext` distribution as a vehicle for documenting its structure and methods, even if this goes beyond the technicalities of the GNU `gettext` proper. By doing so, translators will find in a single place, as far as possible, all they need to know for properly doing their translating work. Also, this supplementary documentation might also help programmers, and even curious users, at understanding how GNU `gettext` is related to the remainder of the GNU Translation Project, and consequently, have a glimpse at the *big picture*.

1.2 I18n, L10n, and Such

Two long words appear all the time when we discuss support of native language in programs, and these words have a precise meaning, worth being explained here, once and for all in this document. The words are *internationalization* and *localization*. Many people, tired of writing these long words over and over again, took the habit of writing *i18n* and *l10n* instead, quoting the first and last letter of each word, and replacing the run of intermediate letters by a number merely telling how many such letters there are. But in this manual, in the sake of clarity, we will patiently write the names in full, each time...

By *internationalization*, one refers to the operation by which a program, or a set of programs turned into a package, is made aware and able to support multiple languages. This is a generalization process, by which the programs are untied from using only English strings or other English specific habits, and connected to generic ways of doing the same, instead. Program developers may use various techniques to internationalize their programs, some of them have been standardized. GNU `gettext` offers one of these standards. See Chapter 8 [Programmers], page 38.

By *localization*, one means the operation by which, in a set of programs already internationalized, one gives the program all needed information so that it can bend itself to handle its input and output in a fashion which is correct for some native language and cultural habits. This is a particularisation process, by which generic methods already implemented in an internationalized program are used in specific ways. The programming environment puts several functions to the programmers disposal which allow this runtime configuration. The formal description of specific set of cultural habits for some country, together with all associated translations targeted to the same native language, is called the *locale* for this language or country. Users achieve localization of programs by setting proper values to special environment variables, prior to executing those programs, identifying which locale should be used.

In fact, locale message support is only one component of the cultural data that makes up a particular locale. There are a whole host of routines and functions provided to aid programmers in developing internationalized software and which allows them to access the data stored in a particular locale. When someone presently refers to a particular locale, they are obviously referring to the data stored within that particular locale. Similarly, if a programmer is referring to “accessing the locale routines”, they are referring to the complete suite of routines that access all of the locale’s information.

One uses the expression *Native Language Support*, or merely NLS, for speaking of the overall activity or feature encompassing both internationalization and localization, allowing for multi-lingual interactions in a program. In a nutshell, one could say that internationalization is the operation by which further localizations are made possible.

Also, very roughly said, when it comes to multi-lingual messages, internationalization is usually taken care of by programmers, and localization is usually taken care of by translators.

1.3 Aspects in Native Language Support

For a totally multi-lingual distribution, there are many things to translate beyond output messages.

- As of today, GNU `gettext` offers a complete toolset for translating messages output by C programs. Perl scripts and shell scripts also need to be translated. Even if there are some hooks so this can be done, these hooks are not integrated as well as they should be.
- Some programs, like `autoconf` or `bison`, are able to produce other programs (or scripts). Even if the generating programs themselves are internationalized, the generated programs they produce may need internationalization on their own, and this indirect internationalization could be automated right from the generating program. In fact, quite usually, generating and generated programs could be internationalized independently, as the effort needed is fairly orthogonal.
- A few programs include textual tables which might need translation themselves, independently of the strings contained in the program itself. For example, RFC 1345 gives an English description for each character which GNU `recode` is able to reconstruct at execution. Since these descriptions are extracted from the RFC by mechanical means, translating them properly would require a prior translation of the RFC itself.
- Almost all programs accept options, which are often worded out so to be descriptive for the English readers; one might want to consider offering translated versions for program options as well.
- Many programs read, interpret, compile, or are somewhat driven by input files which are texts containing keywords, identifiers, or replies which are inherently translatable. For example, one may want `gcc` to allow diacriticized characters in identifiers or use translated keywords; '`rm -i`' might accept something else than '`y`' or '`n`' for replies, etc. Even if the program will eventually make most of its output in the foreign languages, one has to decide whether the input syntax, option values, etc., are to be localized or not.
- The manual accompanying a package, as well as all documentation files in the distribution, could surely be translated, too. Translating a manual, with the intent of later keeping up with updates, is a major undertaking in itself, generally.

As we already stressed, translation is only one aspect of locales. Other internationalization aspects are not currently handled by GNU `gettext`, but perhaps may be handled in future versions. There are many attributes that are needed to define a country's cultural conventions. These attributes include beside the country's native language, the formatting of the date and time, the representation of numbers, the symbols for currency, etc. These local *rules* are termed the country's locale. The locale represents the knowledge needed to support the country's native attributes.

There are a few major areas which may vary between countries and hence, define what a locale must describe. The following list helps putting multi-lingual messages into the proper context of other tasks related to locales, and also presents some other areas which GNU `gettext` might eventually tackle, maybe, one of these days.

Characters and Codesets
: The codeset most commonly used through out the USA and most English speaking parts of the world is the ASCII codeset. However, there are many characters needed by various locales that are not found within this codeset. The 8-bit ISO 8859-1 code set has most of the special characters needed to handle the major European languages. However, in many cases, the ISO 8859-1 font is not adequate. Hence each locale will need to specify which codeset they need to use and will need to have the appropriate character handling routines to cope with the codeset.

Currency
: The symbols used vary from country to country as does the position used by the symbol. Software needs to be able to transparently display currency figures in the native mode for each locale.

Dates
: The format of date varies between locales. For example, Christmas day in 1994 is written as 12/25/94 in the USA and as 25/12/94 in Australia. Other countries might use ISO 8061 dates, etc.

Time of the day may be noted as *hh*:*mm*, *hh*.*mm*, or otherwise. Some locales require time to be specified in 24-hour mode rather than as AM or PM. Further, the nature and yearly extent of the Daylight Saving correction vary widely between countries.

Numbers
: Numbers can be represented differently in different locales. For example, the following numbers are all written correctly for their respective locales:

```
12,345.67       English
12.345,67       French
1,2345.67       Asia
```

Some programs could go further and use different unit systems, like English units or Metric units, or even take into account variants about how numbers are spelled in full.

Messages
: The most obvious area is the language support within a locale. This is where GNU `gettext` provide an ease for developers and users to easily change the language that the software uses to communicate to the user.

In the near future we see no chance that beside message handling more components of locale will be made available for use in other GNU packages. The reason for this is that most modern system provide a more or less reasonable support for at least some of the missing components. Another point is that the GNU libc and Linux will get a new and complete implementation of the whole locale functionality which could be adopted by system lacking a reasonable locale support.

1.4 Files Conveying Translations

The letters PO in '.po' files means Portable Object, to distinguish it from '.mo' files, where MO stands for Machine Object. This paradigm, as well as the PO file format, is inspired by the NLS standard developed by Uniforum, and implemented by Sun in their Solaris system.

PO files are meant to be read and edited by humans, and associate each original, translatable string of a given package with its translation in a particular target language. A single PO file is dedicated to a single target language. If a package supports many languages, there is one such PO file per language supported, and each package has its own set of PO files. These PO files are best created by the `xgettext` program, and later updated or refreshed through the `tupdate` program. Program `xgettext` extracts all marked messages from a set of C files and initializes a PO file with empty translations. Program `tupdate` takes care of adjusting PO files between releases of the corresponding sources, commenting obsolete entries, initializing new ones, and updating all source line references. Files ending with '.pot' are kind of base translation files found in distributions, in PO file format, and '.pox' files are often temporary PO files.

MO files are meant to be read by programs, and are binary in nature. A few systems already offer tools for creating and handling MO files as part of the Native Language Support coming with the system, but the format of these MO files is often different from system to system, and non-portable. They do not necessary use '.mo' for file extensions, but since system libraries are also used for accessing these files, it works as long as the system is self-consistent about it. If GNU `gettext` is able to interface with the tools already provided with systems, it will consequently let these provided tools take care of generating the MO files. Or else, if such tools are not found or do not seem usable, GNU `gettext` will use its own ways and its own format for MO files. Files ending with '.gmo' are really MO files, when it is known that these files use the GNU format.

1.5 Overview of GNU `gettext`

The following diagram summarizes the relation between the files handled by GNU `gettext` and the tools acting on these files. It is followed by a somewhat detailed explanations, which you should read while keeping an eye on the diagram. Having a clear understanding of these interrelations would surely help programmers, translators and maintainers.

```
Original C Sources ---> PO mode ---> Marked C Sources ---.
                                                         |
              .---------<--- GNU gettext Library         |
.--- make <---+                                          |
|             `---------<--------------------+-----------'
|                                            |
|   .-----<--- PACKAGE.pot <--- xgettext <---'   .---<--- PO Compendium
|   |                                            |              ^
|   |                                            `---.          |
|   `---.                                            +---> PO mode ---.
|       +----> tupdate -------> LANG.pox --->--------'                |
|   .---'                                                             |
|   |                                                                 |
|   `-------------<---------------.                                   |
|                                 +--- LANG.po <--- New LANG.pox <----'
|   .--- LANG.gmo <--- msgfmt <---'
|   |
|   `---> install ---> /.../LANG/PACKAGE.mo ---.
|                                              +---> "Hello world!"
`-------> install ---> /.../bin/PROGRAM -------'
```

The indication '`PO mode`' appears in two places in this picture, and you may safely read it as merely meaning "hand editing", using any editor of your choice, really. However, for those of you being the lucky users of GNU Emacs, PO mode has been specifically created for providing a cosy environment for editing or modifying PO files. While editing a PO file, PO mode allows for the easy browsing of auxiliary and compendium PO files, as well as following references into the set of C program sources from which PO files has been derived. It has a few special features, among which the interactive marking of program strings as translatable, and the validatation of PO files with easy repositioning to PO file lines showing errors.

As a programmer, the first step into bringing GNU `gettext` into your package is identifying, right in the C sources, which strings are meant to be translatable, and which are untranslatable. This tedious job can be done a little more comfortably using PO mode, but you can use any means being usual to you for modifying your C sources. Some other simple, standard changes are also needed to properly initialize the translation library. See Chapter 3 [Sources], page 16, for more information about all this.

Once the C sources have been modified, the `xgettext` program is used to find and extract all translatable strings, and create an initial PO file out of all these. This '*package*`.pot`' file contains all original program strings, it has sets of pointers to exactly where in C sources each string is used, and all translations are set to empty. The letter `t` in '`.pot`' marks that this is a Template PO file, not yet oriented towards any particular language. See Section 4.1 [xgettext Invocation], page 22, for more details about how one calls the `xgettext` program. If you are *really* lazy, you might be interested at working a lot more right away, and preparing the whole distribution setup (see Chapter 10 [Maintainers], page 54). By doing so, you spare typing the `xgettext` command yourself, as `make` should now generate the proper things automatically for you!

The first time through, there is no '*lang*.po' yet, so the tupdate step may be skipped and replaced by a mere copy of '*package*.pot' to '*lang*.pox', where *lang* represents the target language.

Then comes the initial translation of messages. Translation in itself is a whole matter, still exclusively meant for humans, and whose complexity far overwhelms the level of this manual. Nevertheless, a few hints are given in some other chapter of this manual (see Chapter 9 [Translators], page 48). You will also find there indications about how to contact translating teams, or becoming part of them, for sharing your translating concerns with others who target the same native language.

While adding the translated messages into the '*lang*.pox' PO file, if you do not have GNU Emacs handy, you are on your own for ensuring that your fully respect the PO file format, and quoting conventions (see Section 2.2 [PO Files], page 9). This is surely not an impossible task, as this is the way many people handled PO files already for Uniforum or Solaris. On the other hand, using PO mode in GNU Emacs, most details of PO file format are taken care for you, but you have to acquire some familiarity with PO mode itself. Besides main PO mode commands (see Section 2.3 [Main PO Commands], page 11), you should know how to move between entries (see Section 2.4 [Entry Positioning], page 12), and how to handle untranslated entries (see Section 5.2 [Untranslated Entries], page 26).

If some common translations have already been saved into a compendium PO file, translators may use PO mode for initializing untranslated entries from the compendium, and also save selected translations into the compendium, updating it (see Section 4.3 [Compendium], page 25). Compendium files are meant to be exchanged between members of a given translation team.

Programs, or packages of programs, are dynamic in nature: users write bug reports and suggestion for improvements, maintainers react by modifying programs in various ways. The fact that a package has already been internationalized should not make maintainers shy of adding new strings, or modifying strings already translated. They just do their job the best they can. For the GNU Translation Project to work smoothly, it is important that maintainers do not carry translation concerns on their already loaded shoulders, and that translators be kept as free as possible of programmatic concerns.

The only concern maintainers should have is carefully marking new strings are translatable, when they should be, and do not otherwise worry about them being translated, as this will come in proper time. Consequently, when programs and their strings are adjusted in various ways by maintainers, and for matters usually unrelated to translation, xgettext would construct '*package*.pot' files which are evolving over time, so the translations carried by '*lang*.po' are slowly fading out of date.

It is important for translators (and even maintainers) to understand that package translation is a continuous process in the lifetime of a package, and not something which is done once and for all at the start. After an initial burst of translation activity for a given package, interventions are needed once in a while, because here and there, translated entries become obsolete, and new untranslated entries appear, needing translation.

The tupdate program has the purpose of refreshing an already existing '*lang*.po' file, by comparing it with a newer '*package*.pot' template file, extracted by xgettext out of recent C sources. The refreshing operation adjusts all references to C source locations for strings, since these strings move as programs are modified. Also, tupdate comments out as obsolete, in '*lang*.pox', those already translated entries which are no longer used in the program sources (see Section 5.3 [Obsolete Entries], page 27. It finally discovers new strings and insert them in the resulting PO file as untranslated entries (see Section 5.2 [Untranslated Entries], page 26. See Section 5.1 [tupdate Invocation], page 26, for more information about what `tupdate` really does.

Whatever route or means taken, the goal is obtaining an updated '*lang*.pox' file offering translations for all strings. When this is properly achieved, this file '*lang*.pox' may take the place of the previous official '*lang*.po' file.

The time mobility, or fluidity of PO files, is an integral part of the translation game, and should be well understood, and accepted. People resisting it will have a hard time participating in the GNU Translation Project, or will give a hard time to other participants! In particular, maintainers should relax and include all available PO files in their distributions, even if these have not recently been updated, without banging or otherwise trying to exert pressure on the translator teams to get the job done. The pressure should rather come from the community of users speaking a particular language, and maintainers should consider themselves fairly relieved of any concern about the adequacy of translation files. On the other hand, translators should reasonably try updating the PO files they are responsible for, while the package is undergoing pretest, prior to an official distribution.

Once the PO file is complete and dependable, the `msgfmt` program is used for turning the PO file into a machine-oriented format, which may yield efficient retrieval of translations by the programs of the package, whenever needed at runtime (see Section 6.2 [MO Files], page 33). See Section 6.1 [msgfmt Invocation], page 32, for more information about all modalities of execution for the `msgfmt` program.

Finally, the modified and marked C sources are compiled and linked with the GNU `gettext` library, usually through the operation of `make`, given a suitable '`Makefile`' exists for the project, and the resulting executable is installed somewhere users will find it. The MO files themselves should also be properly installed. Given the appropriate environment variables are set (see Section 7.3 [End Users], page 37), the program should localize itself automatically, whenever it executes.

The remaining of this manual has the purpose of deepening the various steps outlined in this section.

2 PO Files and PO Mode Basics

The GNU `gettext` toolset helps programmers and translators at producing, updating and using translation files, mainly those PO files which are textual, editable files. This chapter insists on the format of PO files, and contains a PO mode starter. PO mode description is spread over this manual instead of being concentrated in one place, this chapter presents only the basics of PO mode.

2.1 Completing GNU gettext Installation

Once you have received, unpacked, configured and compiled the GNU `gettext` distribution, the '`make install`' command puts in place the programs `xgettext`, `msgfmt`, `gettext`, and `tupdate`, as well as their available message catalogs. For completing a comfortable installation, you might also want to make the PO mode available to your GNU Emacs users.

To finish the installation of the PO mode, you might want modify your file '`.emacs`', once and for all, so it contains a few lines looking like:

```
(setq auto-mode-alist
      (cons '("\\.pox?\\'" . po-mode) auto-mode-alist))
(autoload 'po-mode "po-mode")
```

Later, whenever you edit some '`.po`' or '`.pox`' file, Emacs loads '`po-mode.elc`' (or '`po-mode.el`') as needed, and automatically activate PO mode commands for the associated buffer. The string *PO* appears in the mode line for any buffer for which PO mode is active. Many PO files may be active at once in a single Emacs session.

2.2 The Format of PO Files

A PO file is made up of many entries, each entry holding the relation between an original untranslated string and its corresponding translation. All entries in a given PO file usually pertain to a single project, and all translations are expressed in a single target language. One PO file *entry* has the following schematic structure:

white-space
`#` *translator-comments*
`#.` *automatic-comments*
`#:` *reference...*
`msgid` *untranslated-string*
`msgstr` *translated-string*

The general structure of a PO file should be well understood by the translator. When using PO mode, very little has to be known about the format details, as PO mode takes care of them for her.

Entries begin with some optional white space. Usually, when generated through GNU `gettext` tools, there is exactly one blank line between entries. Then comments follow, on lines all starting with the character `#`. There are two kinds of comments: those which have some white space immediately following the `#`, which comments are created and maintained exclusively by the translator, and those which have some non-white character just after the

`#`, which comments are created and maintained automatically by GNU `gettext` tools. All comments, of any kind, are optional.

After white space and comments, entries show two strings, giving first the untranslated string as it appears in the original program sources, and then, the translation of this string. The original string is introduced by the keyword `msgid`, and the translation, by `msgstr`. The two strings, untranslated and translated, are quoted in various ways in the PO file, using `"` delimiters and `\` escapes, but the translator does not really have to pay attention to the precise quoting format, as PO mode fully intend to take care of quoting for her.

The `msgid` strings, as well as automatic comments, are produced and managed by other GNU `gettext` tools, and PO mode does not provide means for the translator to alter these. The most she can do is merely deleting them, and only by deleting the whole entry. On the other hand, the `msgstr` string, as well as translator comments, are really meant for the translator, and PO mode gives her the full control she needs.

It happens that some lines, usually whitespace or comments, follow the very last entry of a PO file. Such lines are not part of any entry, and PO mode is unable to take action on those lines. By using the PO mode function *`M-x po-normalize`*, the translator may get rid of those spurious lines. See Section 2.5 [Normalizing], page 14.

The remainder of this section may be safely skipped for those using PO mode, yet it may be interesting for everybody to have a better idea of the precise format of a PO file. On the other hand, those not having GNU Emacs handy should carefully continue reading on.

Each of *untranslated-string* and *translated-string* respects the C syntax for a character string, including the surrounding quotes and imbedded backslashed escape sequences. When the time comes to write multi-line strings, one should not use escaped newlines. Instead, a closing quote should follow the last character on the line to be continued, and an opening quote should resume the string at the beginning of the following PO file line. For example:

```
msgid ""
"Here is an example of how one might continue a very long string\n"
"for the common case the string represents multi-line output.\n"
```

In this example, the empty string is used on the first line, for allowing the better alignment of the *`H`* from the word '`Here`' over the *`f`* from the word '`for`'. In this example, the `msgid` keyword is followed by three strings, which are meant to be concatenated. Concatenating the empty string does not change the resulting overall string, but it is a way for us to comply with the necessity of `msgid` to be followed by a string on the same line, while keeping the multi-line presentation left-justified, as we find this to be cleaner disposition. The empty string could have been omitted, but only if the string starting with '`Here`' was promoted on the first line, right after `msgid`.[1] It was not really necessary either to switch between the two last quoted strings immediately after the newline '`\n`', the switch could have occurred after *any* other character, we just did it this way because it is neater.

One should carefully distinguish between end of lines marked as '`\n`' *inside* quotes, which are part of the represented string, and end of lines in the PO file itself, outside string quotes, which have no incidence on the represented string.

[1] This limitation is not imposed by GNU `gettext`, but comes from the `msgfmt` implementation on Solaris.

Outside strings, white lines and comments may be used freely. Comments start at the beginning of a line with '`#`' and extend until the end of the PO file line. Comments written by translators should have the initial '`#`' immediately followed by some white space. If the '`#`' is not immediately followed by white space, this comment is most likely generated and managed by specialized GNU tools, and might disappear or be replaced unexpectandly when the PO file is given to `tupdate`.

2.3 Main Commands

When Emacs finds a PO file in a window, PO mode is activated for that window. This puts the window read-only and establishes a po-mode-map, which is a genuine Emacs mode, in that way that it is not derived from text mode in any way.

The main PO commands are those who do not fit in the other categories in subsequent sections, they allow for quitting PO mode or managing windows in special ways.

`u` Undo last modification to the PO file.

`q` Quit processing and save the PO file.

`o` Temporary leave the PO file window.

`h` Show help about PO mode.

`=` Give some PO file statistics.

`v` Batch validate the format of the whole PO file.

The command *`u`* (`po-undo`) interfaces to the GNU Emacs *undo* facility. See section "Undoing Changes" in *The Emacs Editor*. Each time *`u`* is typed, modifications the translator did to the PO file are undone a little more. For the purpose of undoing, each PO mode command is atomic. This is especially true for the ⟨RET⟩ command: the whole edition made by using a single use of this command is undone at once, even if the edition itself implied several actions. However, while in the editing window, one can undo the edition work quite parsimoniously.

The command *`q`* (`po-quit`) is used when the translator is done with the PO file. If the file has been modified, it is saved on disk first. However, prior to all this, the command checks if some untranslated message remains in the PO file and, if yes, the translator is asked if she really wants to leave working with this PO file. This is the preferred way of getting rid of an Emacs PO file buffer. Merely killing it through the usual command *`C-x k`* (`kill-buffer`), say, has the unnice effect of leaving a PO internal work buffer behind.

The command *`o`* (`po-other-window`) is another, softer way, to leave PO mode, temporarily. It just moves the cursor in some other Emacs window, and pops one if necessary. For example, if the translator just got PO mode to show some source context in some other, she might discover some apparent bug in the program source that needs correction. This command allows the translator to change sex, become a programmer, and have the cursor right into the window containing the program she (or rather *he*) wants to modify. By later getting the cursor back in the PO file window, or by asking Emacs to edit this file once again, PO mode is then recovered.

The command *h* (`po-help`) displays a summary of all available PO mode commands. The translator should then type any character to resume normal PO mode operations. The command *?* has the same effect as *h*.

The command `=` (`po-statistics`) computes the total number of entries in the PO file, the ordinal of the current entry (counted from 1), the number of untranslated entries, the number of obsolete entries, and displays all these numbers.

The command *v* (`po-validate`) launches `msgfmt` in verbose mode over the current PO file. This command first offers to save the current PO file on disk. The `msgfmt` tool, from GNU `gettext`, has the purpose of creating an MO file out of a PO file, and PO mode uses the features of this program for checking the overall format of a PO file, as well as all individual entries.

The program `msgfmt` runs asynchronously with Emacs, so the translator regains control immediately while her PO file is being studied. Error output is collected in the GNU Emacs '`*compilation*`' buffer, displayed in another window. The regular GNU Emacs command *C-x`* (`next-error`), as well as other usual compile commands, allow the translator to reposition quickly to the offending parts of the PO file. Once the cursor on the line in error, the translator may decide for any PO mode action which would help correcting the error.

2.4 Entry Positioning

The cursor in a PO file window is almost always part of an entry. The only exceptions are the special case when the cursor is after the last entry in the file, or when the PO file is empty. The entry where the cursor is found to be is said to be the current entry. Many PO mode commands operate on the current entry, so moving the cursor does more than allowing the translator to browse the PO file, this also selects on which entry commands operate.

Some PO mode commands alter the position of the cursor in a specialized way. A few of those special purpose positioning are described here, the others are described in following sections.

. Redisplay the current entry.

n
SPC Select the entry after the current one.

p
DEL Select the entry before the current one.

< Select the first entry in the PO file.

> Select the last entry in the PO file.

m Record the location of the current entry for later use.

l Return to a previously saved entry location.

x Exchange the current entry location with the previously saved one.

Any GNU Emacs command able to reposition the cursor may be used to select the current entry in PO mode, including commands which move by characters, lines, paragraphs, screens or pages, and search commands. However, there is a kind of standard way to display the current entry in PO mode, which usual GNU Emacs commands moving the cursor do not especially try to enforce. The command `.` (`po-current-entry`) has the sole purpose of redisplaying the current entry properly, after the current entry has been changed by means external to PO mode, or the Emacs screen otherwise altered.

It is yet to decide if PO mode would help the translator, or otherwise irritate her, by forcing a more fixed window disposition while she is doing her work. We originally had quite precise ideas about how windows should behave, but on the other hand, anyone used to GNU Emacs is often happy to keep full control. Maybe a fixed window disposition might be offered as a PO mode option that the translator might activate or deactivate at will, so it could be offered on an experimental basis. If nobody feels a real need for using it, or a compulsion for writing it, we might as well drop this whole idea. The incentive for doing it should come from translators rather than programmers, as opinions from an experienced translator are surely more worth to me than opinions from programmers *thinking* about how *others* should do translation.

The commands *n* (`po-next-entry`) and *p* (`po-previous-entry`) move the cursor the entry following, or preceding, the current one. If *n* is given while the cursor is on the last entry of the PO file, or if *p* is given while the cursor is on the first entry, no move is done. ⟨SPC⟩ and ⟨DEL⟩ are alternate keys for *n* and *p*, respectively.

The commands `<` (`po-first-entry`) and `>` (`po-last-entry`) move the cursor to the first entry, or last entry, of the PO file. When the cursor is located past the last entry in a PO file, most PO mode commands will return an error saying '`After last entry`'. However, the commands `<` and `>` have the special property of being able to work even when the cursor is not into some PO file entry, and you may use them for nicely correcting this situation. But even these commands will fail on a truly empty PO file. There are development plans for PO mode for it to interactively fill an empty PO file from sources. See Section 3.3 [Marking], page 18.

The translator may decide, before working at the translation of a particular entry, that she needs browsing the remainder of the PO file, maybe for finding the terminology or phraseology used in related entries. She can of course use the standard Emacs idioms for saving the current cursor location in some register, and use that register for getting back, or else, to use the location ring.

PO mode offers another approach, by which cursor locations may be saved onto a special stack. The command *m* (`po-push-location`) merely adds the location of current entry to the stack, pushing the already saved locations under the new one. The command *l* (`po-pop-location`) consumes the top stack element and reposition the cursor to the entry associated with that top element. This position is then lost, for the next *l* will move the cursor to the previously saved location, and so on until locations remain on the stack.

If the translator wants the position to be kept on the location stack, maybe for taking a mere look at the entry associated with the top element, then go elsewhere with the intent of getting back later, she ought to use *m* immediately after *l*.

The command *x* (`po-exchange-location`) simultaneously reposition the cursor to the entry associated with the top element of the stack of saved locations, and replace that top element with the location of the current entry before the move. Consequently, repeating the `x` command toggles alternatively between two entries. For achieving this, the translator will position the cursor on the first entry, use `m`, then position to the second entry, and merely use `x` for making the switch.

2.5 Normalizing Strings in Entries

There are many different ways for encoding a particular string into a PO file entry, because there are so many different ways to split and quote multi-line strings, and even, to represent special characters by backslahsed escaped sequences. Some features of PO mode rely on the ability for PO mode to scan an already existing PO file for a particular string encoded into the `msgid` field of some entry. Even if PO mode has internally all the built-in machinery for implementing this recognition easily, doing it fast is technically difficult. For facilitating a solution to this efficiency problem, we decided for a canonical representation for strings.

A conventional representation of strings in a PO file is currently under discussion, and PO mode experiments a canonical representation. Having both `xgettext` and PO mode converging towards a uniform way of representing equivalent strings would be useful, as the internal normalization needed by PO mode could be automatically satisfied when using `xgettext` from GNU `gettext`. An explicit PO mode normalization should then be only necessary for PO files imported from elsewhere, or for when the convention itself evolves.

So, for achieving normalization of at least the strings of a given PO file needing a canonical representation, the following PO mode command is available:

`M-x po-normalize`
: Tidy the whole PO file by making entries more uniform.

The special command *`M-x po-normalize`*, which has no associate keys, revises all entries, ensuring that strings of both original and translated entries use uniform internal quoting in the PO file. It also removes any crumb after the last entry. This command may be useful for PO files freshly imported from elsewhere, or if we ever improve on the canonical quoting format we use. This canonical format is not only meant for getting cleaner PO files, but also for greatly speeding up `msgid` string lookup for some other PO mode commands.

`M-x po-normalize` presently makes three passes over the entries. The first implements heuristics for converting PO files for GNU `gettext` 0.6 and earlier, in which `msgid` and `msgstr` fields were using K&R style C string syntax for multi-line strings. These heuristics may fail for comments not related to obsolete entries and ending with a backslash; they also depend on subsequent passes for finalizing the proper commenting of continued lines for obsolete entries. This first pass might disappear once all oldish PO files would have been adjusted. The second and third pass normalize all `msgid` and `msgstr` strings respectively. They also clean out those trailing backslashes used by XView's `msgfmt` for continued lines.

Having such an explicit normalizing command allows for importing PO files from other sources, but also eases the evolution of the current convention, evolution driven mostly by aesthetic concerns, as of now. It is all easy to make suggested adjustments at a later time, as the normalizing command and eventually, other GNU `gettext` tools should greatly automate conformance. A description of the canonical string format is given below, for the particular benefit of those not having GNU Emacs handy, and who would nevertheless want to handcraft their PO files in nice ways.

Right now, in PO mode, strings are single line or multi-line. A string goes multi-line if and only if it has *embedded* newlines, that is, if it matches '`[^\n]\n+[^\n]`'. So, we would have:

```
msgstr "\n\nHello, world!\n\n\n"
```

but, replacing the space by a newline, this becomes:

```
msgstr ""
"\n"
"\n"
"Hello,\n"
"world!\n"
"\n"
"\n"
```

We are deliberately using a caricatural example, here, to make the point clearer. Usually, multi-lines are not that bad looking. It is probable that we will implement the following suggestion. We might lump together all initial newlines into the empty string, and also all newlines introducing empty lines (that is, for $n > 1$, the n-1'th last newlines would go together on a separate string), so making the previous example appear:

```
msgstr "\n\n"
"Hello,\n"
"world!\n"
"\n\n"
```

There are a few yet undecided little points about string normalization, to be documented in this manual, once these questions settle.

3 Preparing Program Sources

For the programmer, changes to the C source code fall into three categories. First, you have to make the localization functions known to all modules needing message translation. Second, you should properly trigger the operation of GNU `gettext` when the program initializes, usually from the `main` function. Last, you should identify and especially mark all constant strings in your program needing translation.

Presuming that your set of programs, or package, has been adjusted so all needed GNU `gettext` files are available, and your '`Makefile`' files are adjusted (see Chapter 10 [Maintainers], page 54), each C module having translated C strings should contain the line:

```
#include <libintl.h>
```

The remaining changes to your C sources are discussed in the further sections of this chapter.

3.1 Triggering gettext Operations

The initialization of locale data should be done with more or less the same code in every program, as demonstrated below:

```
int
main (argc, argv)
     int argc;
     char argv;
{
  ...
  setlocale (LC_ALL, "");
  bindtextdomain (PACKAGE, LOCALEDIR);
  textdomain (PACKAGE);
  ...
}
```

PACKAGE and *LOCALEDIR* should be provided either by '`config.h`' or by the Makefile. For now consult the `gettext` sources for more information.

The use of `LC_ALL` might not be appropriate for you. `LC_ALL` includes all locale categories and especially `LC_CTYPE`. This later category is responsible for determining character classes with the `isalnum` etc. functions from '`ctype.h`' which could especially for programs, which process some kind of input language, be wrong. For example this would mean that a source code using the c (cedille character) is runnable in France but not in the U.S.

So it is sometimes necessary to replace the `LC_ALL` line in the code above by a sequence of `setlocale` lines

```
{
  ...
  setlocale (LC_TIME, "");
  setlocale (LC_MESSAGES, "");
  ...
}
```

or to switch for and back to the character class in question.

3.2 How Marks Appears in Sources

The C sources should mark all strings requiring translation. Marking is done in such a way that each translatable string appears to be the sole argument of some function or preprocessor macro. There are only a few such possible functions or macros meant for translation, and their names are said to be marking keywords. The marking is attached to strings themselves, rather than to what we do with them. This approach has more uses. A blatant example is an error message produced by formatting. The format string needs translation, as well as some strings inserted through some '`%s`' specification in the format, while the result from `sprintf` may have so many different instances that it is unpractical to list them all in some '`error_string_out()`' routine, say.

This marking operation has two goals. The first goal of marking is for triggering the retrieval of the translation, at run time. The keyword are possibly resolved into a routine able to dynamically return the proper translation, as far as possible or wanted, for the argument string. Most localizable strings are found into executable positions, that is, affected to variables or given as parameter to functions. But this is not universal usage, and some translatable strings appear in structured initializations. See Section 3.4 [Special cases], page 20.

The second goal of the marking operation is to help `xgettext` at properly extracting all translatable strings when it scans a set of program sources and produces PO file templates.

The canonical keyword for marking translatable strings is '`gettext`', it gave its name to the whole GNU `gettext` package. For packages making only light use of the '`gettext`' keyword, macro or function, it is easily used *as is*. However, for packages using the `gettext` interface more heavily, it is usually more convenient giving the main keyword a shorter, less obtrusive name. Indeed, the keyword might appear on a lot of strings all over the package, and programmers usually do not want nor need that their program sources remind them loud, all the time, that they are internationalized. Further, a long keyword has the disadvantage of using more horizontal space, forcing more indentation work on sources for those trying to keep them within 79 or 80 columns.

Many GNU packages use '`_`' (a simple underline) as a keyword, and write '`_("Translatable string")`' instead of '`gettext ("Translatable string")`'. Further, the usual GNU coding rule wanting that there is a space between the keyword and the opening parenthesis is relaxed, in practice, for this particular usage. So, the textual overhead per translatable string is reduced to only three characters: the underline and the two parentheses. However, even if GNU `gettext` uses this convention internally, it does not offer it officially. The real, genuine keyword is truly '`gettext`' indeed. It is fairly easy for those wanting to use '`_`' instead of '`gettext`' to declare:

```
#include <libintl.h>
#define _(String) gettext (String)
```

instead of merely using '`#include <libintl.h>`'.

Later on, the maintenance is relatively easy. If, as a programmer, you add or modify a string, you will have to ask yourself if the new or altered string requires translation, and include it within '`_()`' if you think it should be translated. '`"%s: %d"`' is an example of string *not* requiring translation!

3.3 Marking Translatable Strings

In PO mode, one set of features is meant more for the programmer than for the translator, and allows him to interactively mark which strings, in a set of program sources, are translatable, and which are not. Even if it is a fairly easy job for a programmer to find and mark such strings by other means, using any editor of his choice, PO mode makes this work more comfortable. Further, this gives translators who feel a little like programmers, or programmers who feel a little like translators, a tool letting them work at marking translatable strings in the program sources, while simultaneously producing a set of translation in some language, for the package being internationalized.

The set of program sources, aimed by the PO mode commands describe here, should have an Emacs tags table constructed for your project, prior to using these PO file commands. This is easy to do. In any shell window, change the directory to the root of your project, then execute a command resembling:

```
etags src/*.[hc] lib/*.[hc]
```

presuming here you want to process all '`.h`' and '`.c`' files from the '`src/`' and '`lib/`' directories. This command will explore all said files and create a '`TAGS`' file in your root directory, somewhat summarizing the contents using a special file format Emacs can understand.

For official GNU packages which follow the GNU coding standard there is a make goal `tags` or `TAGS` which construct the tag files in all directories and for all files containing source code.

Once your '`TAGS`' file is ready, the following commands assist the programmer at marking translatable strings in his set of sources. But these commands are necessarily driven from within a PO file window, and it is likely that you do not even have such a PO file yet. This is not a problem at all, as you may safely open a new, empty PO file, mainly for using these commands. This empty PO file will slowly fill in while you mark strings as translatable in your program sources.

`,` Search through program sources for a string which looks like a candidate for translation.

`M-,` Mark the last string found with '`_()`'.

`M-.` Mark the last string found with a keyword taken from a set of possible keywords. This command with a prefix allows some management of these keywords.

The `,` (`po-tags-search`) command search for the next occurrence of a string which looks like a possible candidate for translation, and displays the program source in another Emacs window, positioned in such a way that the string is near the top of this other window. If the string is to big to fit whole in this window, it is rather positioned so only its end is shown. In any case, the cursor is left in the PO file window. If the shown string would be better presented differently in different native languages, you may mark it using *`M-,`* or *`M-.`*. Otherwise, you might rather ignore it and skip to the next string by merely repeating the `,` command.

A string is a good candidate for translation if it contains a sequence of three or more letters. A string containing at most two letters in a row will be considered as a candidate if it has more letters than non-letters. The command disregards strings containing no letters, or isolated letters only. It also disregards strings within comments, or strings already marked with some keyword PO mode knows (see below).

If you have never told Emacs about some 'TAGS' file to use, the command will request that you specify one from the minibuffer, the first time you use the command. You may later change your 'TAGS' file by using the regular Emacs command `M-x visit-tags-table`, which will ask you to name the precise 'TAGS' file you want to use. See section "Tag Tables" in *The Emacs Editor*.

Each time you use the `,` command, the search resumes where it was left over by the previous search, and goes through all program sources, obeying the 'TAGS' file, until all sources have been processed. However, by giving a prefix argument to the command (`C-u ,`), you may request that the search be restarted all over again from the first program source; but in this case, strings that you recently marked as translatable will be automatically skipped.

Using this `,` command does not prevent using of other regular Emacs tags commands. For example, regular `tags-search` or `tags-query-replace` commands may be used without disrupting the independent `,` search sequence. However, as implemented, the *initial* `,` command (or the `,` command is used with a prefix) might also reinitialize the regular Emacs tags searching to the first tags file, this reinitialization might be considered spurious.

The `M-,` (`po-mark-translatable`) command will mark the recently found string with the '_' keyword. The `M-.` (`po-select-mark-and-mark`) command will request that you type one keyword from the minibuffer and use that keyword for marking the string. Both commands will automatically create a new PO file untranslated entry for the string being marked, and make it the current entry (making it easy for you to immediately proceed to its translation, if you feel like doing it right away). It is possible that the modifications made to the program source by `M-,` or `M-.` render some source line longer than 80 columns, forcing you to break and re-indent this line differently. You may use the `o` command from PO mode, or any other window changing command from GNU Emacs, to break out into the program source window, and do any needed adjustments. You will have to use some regular Emacs command to return the cursor to the PO file window, if you want commanding `,` for the next string, say.

The `M-.` command has a few built-in speedups, so you do not have to explicitly type all keywords all the time. The first such speedup is that you are presented with a *preferred* keyword, which you may accept by merely typing ⟨RET⟩ at the prompt. The second speedup is that you may type any non-ambiguous prefix of the keyword you really mean, and the command will complete it automatically for you. This also means that PO mode has to *know* all your possible keywords, and that it will not accept mistyped keywords.

If you reply `?` to the keyword request, the command gives a list of all known keywords, from which you may choose. When the command is prefixed by an argument (`C-u M-.`), it inhibits updating any program source or PO file buffer, and does some simple keyword management instead. In this case, the command asks for a keyword, written in full, which becomes a new allowed keyword for later `M-.` commands. Moreover, this new keyword automatically becomes the *preferred* keyword for later commands. By typing an already known keyword in response to `C-u M-.`, one merely changes the *preferred* keyword and does nothing more.

All keywords known for *M-.* are recognized by the *,* command when scanning for strings, and strings already marked by any of those known keywords are automatically skipped. If many PO files are opened simultaneously, each one has its own independent set of known keywords. There is no provision in PO mode, currently, for deleting a known keyword, you have to quit the file (maybe using *q*) and reopen it afresh. When a PO file is newly brought up in an Emacs window, only '`gettext`' and '`_`' are known as keywords, and '`gettext`' is preferred for the *M-.* command. In fact, this is not useful to prefer '`_`', as this one is already built in the *M-,* command.

3.4 Special Cases of Translatable Strings

The attentive reader might now point out that it is not always possible to mark translatable string with `gettext` or something like this. Consider the following case:

```
{
  static const char *messages[] = {
    "some very meaningful message",
    "and another one"
  };
  const char *string;
  ...
  string
    = index > 1 ? "a default message" : messages[index];

  fputs (string);
  ...
}
```

While it is no problem to mark the string `"a default message"` it is not possible to mark the string initializers for `messages`. What is to do? We have to fulfill two tasks. First we have to mark the strings so that the `xgettext` program (see Section 4.1 [xgettext Invocation], page 22) can find them, and second we have to translate the string at runtime before printing them.

The first task can be fulfilled by creating a new keyword, which names a no-op. For the second we have to mark all access points to a string from the array. So one solution can look like this:

```
#define gettext_noop(String) (String)

{
  static const char *messages[] = {
    gettext_noop ("some very meaningful message"),
    gettext_noop ("and another one")
  };
  const char *string;
  ...
  string
    = index > 1 ? gettext ("a default message") : gettext (messages[index]);

  fputs (string);
  ...
}
```

Please convince yourself that the string which is written by `fputs` is translated in any case. How to get `xgettext` know the additional keyword `gettext_noop` is explained in Section 4.1 [xgettext Invocation], page 22.

The above is of course not the only solution. You could also come along with the following one:

```
#define gettext_noop(String) (String)

{
  static const char *messages[] = {
    gettext_noop ("some very meaningful message",
    gettext_noop ("and another one")
  };
  const char *string;
  ...
  string
    = index > 1 ? gettext_noop ("a default message") : messages[index];

  fputs (gettext (string));
  ...
}
```

But this has some drawbacks. First the programmer has to take care that he uses `gettext_noop` for the string `"a default message"`. A use of `gettext` could have in rare cases unpredictable results. The second reason is found in the internals of the GNU `gettext` Library which will make this solution less efficient.

One advantage is that you need not make control flow analysis to make sure the output is really translated in any case. But this analysis is generally not very difficult. If it should be in any situation you can use this second method in this situation.

4 Making the Initial PO File

4.1 Invoking the `xgettext` Program

`xgettext` [*option*] *inputfile* ...

'`-a`'
'`--extract-all`'
Extract all strings.

'`-c` [*tag*]'
'`--add-comments`[=*tag*]'
Place comment block with *tag* (or those preceding keyword lines) in output file.

'`-C`'
'`--c++`' Recognize C++ style comments.

'`-d` *name*'
'`--default-domain=`*name*'
Use '*name*`.po`' for output (instead of '`messages.po`').

'`-D` *directory*'
'`--directory=`*directory*'
Change to *directory* before beginning to search and scan source files. The resulting '`.po`' file will be written relative to the original directory, though.

'`-f` *file*'
'`--files-from=`*file*'
Read the names of the input files from *file* instead of getting them from the command line.

'`-h`'
'`--help`' Display this help and exit.

'`-I` *list*'
'`--input-path=`*list*'
List of directories searched for input files.

'`-j`'
'`--join-existing`'
Join messages with existing file.

'`-k` *word*'
'`--keyword`[=*word*]'
Additonal keyword to be looked for (without *word* means not to use default keywords).

The default keywords, which are always looked for if not explicitly disabled, are `gettext`, `dgettext`, `dcgettext` and `gettext_noop`.

'`-m` [*string*]'
'`--msgstr-prefix`[=*string*]'
Use *string* or "" as prefix for msgstr entries.

'-M [*string*]'
'--msgstr-suffix[=*string*]'
Use *string* or "" as suffix for msgstr entries.

'--no-location'
Do not write '#: *filename*:*line*' lines.

'-n'
'--add-location'
Generate '#: *filename*:*line*' lines (default).

'--omit-header'
Don't write header with 'msgid ""' entry.

This is useful for testing purposes because it eliminates a source of variance for generated .gmo files. We can ship some of these files in the GNU gettext package, and the result of regenerating them through msgfmt should yield the same values.

'-p *dir*'
'--output-dir=*dir*'
Output files will be placed in directory *dir*.

'-s'
'--sort-output'
Generate sorted output and remove duplicates.

'--strict'
Write out strict Uniforum conforming PO file.

'-v'
'--version'
Output version information and exit.

'-x *file*'
'--exclude-file=*file*'
Entries from *file* are not extracted.

Search path for supplementary PO files is: '/usr/local/share/nls/src/'.

If *inputfile* is '-', standard input is read.

This implementation of xgettext is able to process a few awkward cases, like strings in preprocessor macros, ANSI concatenation of adjacent strings, and escaped end of lines for continued strings.

4.2 C Sources Context

PO mode is particularily powerful when used with PO files created through GNU gettext utilities, as those utilities insert special comments in the PO files they generate. Some of these special comments relate the PO file entry to exactly where the untranslated string appears in the program sources.

When the translator gets to an untranslated entry, she is fairly often faced with an original string which is not as informative as it normally should, being succinct, cryptic, or otherwise ambiguous. Before chosing how to translate the string, she needs to understand better what the string really means and how tight the translation has to be. Most of times, when problems arise, the only way left to make her judgment is looking at the true program sources from where this string originated, searching for surrounding comments the programmer might have put in there, and looking around for helping clues of *any* kind.

Surely, when looking at program sources, the translator will receive more help if she is a fluent programmer. However, even if she is not versed in programming and feels a little lost in C code, the translator should not be shy at taking a look, once in a while. It is most probable that she will still be able to find some of the hints she needs. She will learn quickly to not feel uncomfortable in program code, paying more attention to programmer's comments, variable and function names (if he dared chosing them well), and overall organization, than to programmation itself.

The following commands are meant to help the translator at getting program source context for a PO file entry.

c Resume the display of a program source context, or cycle through them.

M-c Display of a program source context selected by menu.

d Add a directory to the search path for source files.

M-d Delete a directory from the search path for source files.

The commands *c* (`po-cycle-reference`) and *M-c* (`po-select-reference`) both open another window displaying some source program file, and already positioned in such a way that it shows an actual use of the current string to translate. By doing so, the command gives source program context for the string. But if the entry has no source context references, or if all references are unresolved along the search path for program sources, then the command diagnoses this as an error.

Even if *c* (or *M-c*) opens a new window, the cursor stays in the PO file window. If the translator really wants to get into the program source window, she ought to do it explicitly, maybe by using command *o*.

When *c* is typed for the first time, or for a PO file entry which is different of the last one used for getting source context, then the command reacts by giving the first context available for this entry, if any. If some context has already been recently displayed for the current PO file entry, and the translator wandered to do other things, typing *c* again will merely resume, in another window, the context last displayed. In particular, if the translator moved the cursor away from the context in the source file, the command will bring the cursor back to the context. By using *c* many times in a row, with no interning other commands, PO mode will cycle to the next available contexts for this particular entry, getting back to the first context once the last has been shown.

The command *M-c* behaves differently. Instead of cycling through references, it lets the translator choose of particular reference among many, and displays that reference. It is best used with completion, if the translator types *TAB* immediately after *M-c*, in response to the question, she will be offered a menu of all possible references, as a reminder of which are the acceptable answers. This command is useful only where there are really many contexts available for a single string to translate.

Program source files are usually found relative to where the PO file stands. As a special provision, when this fails, the file is also looked for, but relative to the directory immediately above it. Those two cases take proper care of most PO files. However, it might happen that a PO file has been moved, or is edited in a different place than its normal location. When this happens, the translator should tell PO mode in which directory normally sits the genuine PO file. Many such directories may be specified, and all together, they constitute what is called the *search path* for program sources. The command `d` (`po-add-path`) is used to interactively enter a new directory at the front of the search path, and the command `M-d` (`po-delete-path`) is used to select, with completion, one of the directories she does not want anymore on the search path.

4.3 Using Translation Compendiums

Compendiums are yet to be implemented.

An incoming PO mode feature will let the translator maintain a compendium of already achieved translations. A *compendium* is a special PO file containing a set of translations recurring in many different packages. The translator will be given commands for adding entries to her compendium, and later initializing untranslated entries, or updating already translated entries, from translations kept in the compendium. For this to work, however, the compendium would have to be normalized. See Section 2.5 [Normalizing], page 14.

5 Updating Existing PO Files

5.1 Invoking the `tupdate` Program

```
tupdate --help
tupdate --version
tupdate new old
```

File *new* is the last created PO file (generally by `xgettext`). It need not contain any translations. File *old* is the PO file including the old translations which will be taken over to the newly created file as long as they still match.

When English messages change in the programs, this is reflected in the PO file as extracted by `xgettext`. In large messages, that can be hard to detect, and will obviously result in an incomplete translation. One of the virtues of `tupdate` is that it detects such changes, saving the previous translation into a PO file comment, so marking the entry as obsolete, and giving the modified string with an empty translation, that is, marking the entry as untranslated.

5.2 Untranslated Entries

When `xgettext` originally creates a PO file, unless told otherwise, it initializes the `msgid` field with the untranslated string, and leaves the `msgstr` string to be empty. Such entries, having an empty translation, are said to be *untranslated* entries. Later, when the programmer slightly modifies some string right in the program, this change is later reflected in the PO file by the appearance of a new untranslated entry for the modified string.

The usual commands moving from entry to entry consider untranslated entries on the same level as active entries. Untranslated entries are easily recognizable by the fact they end with '`msgstr ""`'.

The work of the translator might be (quite naively) seen as the process of seeking after an untranslated entry, editing a translation for it, and repeating these actions until no untranslated entries remain. Some commands are more specifically related to untranslated entry processing.

e Find the next untranslated entry.

M-e Find the previous untranslated entry.

k Turn the current entry into an untranslated one.

The commands *e* (`po-next-empty-entry`) and *M-e* (`po-previous-empty`) move forwards or backwards, chasing for an obsolete entry. If none is found, the search is extended and wraps around in the PO file buffer.

An entry can be turned back into an untranslated entry by merely emptying its translation, using the command *k* (`po-kill-msgstr`). See Section 5.4 [Modifying Translations], page 28.

Also, when time comes to quit working on a PO file buffer with the *q* command, the translator is asked for confirmation, if some untranslated string still exists.

5.3 Obsolete Entries

By *obsolete* PO file entries, we mean those entries which are commented out, usually by `tupdate` when it found that the translation is not needed anymore by the package being localized.

The usual commands moving from entry to entry consider obsolete entries on the same level as active entries. Obsolete entries are easily recognizable by the fact that all their lines start with `#`, even those lines containing `msgid` or `msgstr`.

Commands exist for emptying the translation or reinitializing it to the original untranslated string. Commands interfacing with the kill ring may force some previously saved text into the translation. The user may interactively edit the translation. All these commands may apply to obsolete entries, carefully leaving the entry obsolete after the fact.

Moreover, some commands are more specifically related to obsolete entry processing.

M-n
M-⟨SPC⟩ Find the next obsolete entry.

M-p
M-⟨DEL⟩ Find the previous obsolete entry.

z Make an active entry obsolete, or zap out an obsolete entry.

The commands *M-n* (`po-next-obsolete-entry`) and *M-p* (`po-previous-obsolete-entry`) move forwards or backwards, chasing for an obsolete entry. If none is found, the search is extended and wraps around in the PO file buffer. The commands *M-⟨SPC⟩* and *M-⟨DEL⟩* are synonymous to *M-n* and *M-p*, respectively.

PO mode does not provide ways for un-commenting an obsolete entry and making it active, because this would reintroduce an original untranslated string which does not correspond to any marked string in the program sources. This goes with the philosophy of never introducing useless `msgid` values.

However, it is possible to comment out an active entry, so making it obsolete. GNU `gettext` utilities will later react to the disappearance of a translation by using the untranslated string. The command *z* (`po-fade-out-entry`) pushes the current entry a little further towards annihilation. If the entry is active, then the entry is merely commented out. If the entry is already obsolete, then it is completely deleted from the PO file. It is easy to recycle the translation so deleted into some other PO file entry, usually one which is untranslated. See Section 5.4 [Modifying Translations], page 28.

Here is a quite interesting problem to solve for later development of PO mode, for those nights you are not sleepy. The idea would be that PO mode might become bright enough, one of these days, to make good guesses at retrieving the most probable candidate, among all obsolete entries, for initializing the translation of a newly appeared string. I think it might be a quite hard problem to do this algorithmically, as we have to develop good and efficient measures of string similarity. Right now, PO mode completely lets the decision to the translator, when the time comes to find the adequate obsolete translation, it merely tries to provide handy tools for helping her to do so.

5.4 Modifying Translations

PO mode prevents direct edition of the PO file, by the usual means Emacs give for altering a buffer's contents. By doing so, it pretends helping the translator to avoid little clerical errors about the overall file format, or the proper quoting of strings, as those errors would be easily made. Other kinds of errors are still possible, but some may be catched and diagnosed by the batch validation process, which the translator may always trigger by the *v* command. For all other errors, the translator has to rely on her own judgment, and also on the linguistic reports submitted to her by the users of the translated package, having the same mother tongue.

When the time comes to create a translation, correct a error diagnosed mechanically or reported by a user, the translator have to resort to using the following commands for modifying the translations.

RET Interactively edit the translation.

TAB Reinitialize the translation with the original, untranslated string.

k Save the translation on the kill ring, and delete it.

w Save the translation on the kill ring, without deleting it.

y Replace the translation, taking the new from the kill ring.

The command *RET* (`po-edit-msgstr`) opens a new Emacs window containing a copy of the translation taken from the current PO file entry, all ready for edition, fully modifiable and with the complete extent of GNU Emacs modifying commands. The string is presented to the translator expunged of all quoting marks, and she will modify the *unquoted* string in this window to heart's content. Once done, the regular Emacs command *M-C-c* (`exit-recursive-edit`) may be used to return the edited translation into the PO file, replacing the original translation. The keys *C-c C-c* are bound so they have the same effect as *M-C-c*.

If the translator becomes unsatisfied with her translation to the extent she prefers keeping the translation which was existent prior to the *RET* command, she may use the regular Emacs command *C-]* (`abort-recursive-edit`) to merely get rid of edition, while preserving the original translation. Another way would be for her to exit normally with *C-c C-c*, then type `u` once for undoing the whole effect of last edition.

While editing her translation, the translator should pay attention at not inserting unwanted ⟨RET⟩ (carriage returns) characters at the end of the translated string if those are not meant to be there, or removing such characters when they are required. Since these characters are not visible in the editing buffer, they are easily to introduce by mistake. To help her, ⟨RET⟩ automatically puts the character `<` at the end of the string being edited, but this `<` is not really part of the string. On exiting the editing window with *C-c C-c*, PO mode automatically removes such `<` and all whitespace added after it. If the translator adds characters after the terminating `<`, it looses its delimiting property and integrally becomes part of the string. If she removes the delimiting `<`, then the edited string is taken *as is*, with all trailing newlines, even if invisible. Also, if the translated string ought to end itself with a genuine `<`, then the delimiting `<` may not be removed; so the string should appear, in the editing window, as ending with two `<` in a row.

When a translation (or a comment) is being edited, the translator may move the cursor back into the PO file buffer and freely move to other entries, and browsing at will. The edited entry will be recovered as soon as the edit ceases, because this is this entry only which is being modified. If, with an edition still opened, the translator wanders in the PO file buffer, she cannot modify any other entry. If she tries to, PO mode will react by suggesting that she aborts the current edit, or else, by inviting her to finish the current edit prior to any other modification.

The command *TAB* (`po-msgid-to-msgstr`) initializes, or reinitializes the translation with the original string. This command is normally used when the translator wants to redo a fresh translation of the original string, disregarding any previous work.

In fact, whether it is best to start a translation with an empty string, or rather with a copy of the original string, is a matter of taste or habit. Sometimes, the source mother tongue language and the target language are so different that is simply best to start writing on an empty page. At other times, the source and target languages are so close that it would be a waste to retype a number of words already being written in the original string. A translator may also like having the original string right under her eyes, as she will progressively overwrite the original text with the translation, even if this requires some extra editing work to get rid of the original.

The command *k* (`po-kill-msgstr`) merely empties the translation string, so turning the entry into an untranslated one. But while doing so, its previous contents is put apart in a special place, known as the kill ring. The command *w* (`po-kill-ring-save-msgstr`) has also the effect of taking a copy of the translation onto the kill ring, but it otherwise leaves the entry alone, and does *not* remove the translation from the entry. Both commands use exactly the Emacs kill ring, which is shared between buffers, and which is well known already to GNU Emacs lovers.

The translator may use *k* or *w* many times in the course of her work, as the kill ring may hold several saved translations. From the kill ring, strings may later be reinserted in various Emacs buffers. In particular, the kill ring may be used for moving translation strings between different entries of a single PO file buffer, or if the translator is handling many such buffers at once, even between PO files.

To facilitate exchanges with buffers which are not in PO mode, the translation string put on the kill ring by the *k* command is fully unquoted before being saved: external quotes are removed, multi-lines strings are concatenated, and backslashed escaped sequences are turned into their corresponding characters. In the special case of obsolete entries, the translation is also uncommented prior to saving.

The command *y* (`po-yank-msgstr`) completely replaces the translation of the current entry by a string taken from the kill ring. Following GNU Emacs terminology, we then say that the replacement string is *yanked* into the PO file buffer. See section "Yanking" in *The Emacs Editor*. The first time *y* is used, the translation receives the value of the most recent addition to the kill ring. If *y* is typed once again, immediately, without intervening keystrokes, the translation just inserted is taken away and replaced by the second most recent addition to the kill ring. By repeating *y* many times in a row, the translator may travel along the kill ring for saved strings, until she finds the string she really wanted.

When a string is yanked into a PO file entry, it is fully and automatically requoted for complying with the format PO files should have. Further, if the entry is obsolete, PO mode then appropriately push the inserted string inside comments. Once again, translators should not burden themselves with quoting considerations besides, of course, the necessity of the translated string itself respective to the program using it.

Note that `k` or `w` are not the only commands pushing strings on the kill ring, as almost any PO mode command replacing translation strings (or the translator comments) automatically save the old string on the kill ring. The main exceptions to this general rule are the yanking commands themselves.

To better illustrate the operation of killing and yanking, let's use an actual example, taken from a common situation. When the programmer slightly modifies some string right in the program, his change is later reflected in the PO file by the appearance of a new untranslated entry for the modified string, and the fact that the entry translating the original or unmodified string becomes obsolete. In many cases, the translator might spare herself some work by retrieving the unmodified translation from the obsolete entry, then initializing the untranslated entry `msgstr` field with this retrieved translation. Once this done, the obsolete entry is not wanted anymore, and may be safely deleted.

When the translator finds an untranslated entry and suspects that a slight variant of the translation exists, she immediately uses `m` to mark the current entry location, then starts chasing obsolete entries with `M-SPC`, hoping to find some translation corresponding to the unmodified string. Once found, she uses the `z` command for deleting the obsolete entry, knowing that `z` also *kills* the translation, that is, pushes the translation on the kill ring. Then, `l` returns to the initial untranslated entry, `y` then *yanks* the saved translation right into the `msgstr` field. The translator is then free to use ⟨RET⟩ for fine tuning the translation contents, and maybe to later use `e`, then `m` again, for going on with the next untranslated string.

When some sequence of keys has to be typed over and over again, the translator may find comfortable to become more acquainted with the GNU Emacs capability of learning these sequences and playing them back under request. See section “Keyboard Macros” in *The Emacs Editor*.

5.5 Modifying Comments

Any translation work done seriously will raise many linguistic difficulties, for which decisions have to be made, and the choices further documented. These documents may be saved within the PO file in form of translator comments, which the translator is free to create, delete, or modify at will. These comments may be useful to herself when she returns to this PO file after a while. Memory forgets!

These commands are somewhat similar to those modifying translations, so the general indications given for these apply here. See Section 5.4 [Modifying Translations], page 28.

`M-RET` Interactively edit the translator comments.

`M-k` Save the translator comments on the kill ring, and delete it.

`M-w` Save the translator comments on the kill ring, without deleting it.

`M-y` Replace the translator comments, taking the new from the kill ring.

Those commands parallel PO mode commands for modifying the translation strings, and behave much the same way as them, except that they handle this part of PO file comments meant for translator usage, rather than the translation strings. So, the descriptions given below are slightly succinct, because the full details have already been given. See Section 5.4 [Modifying Translations], page 28.

The command *M-RET* (`po-edit-comment`) opens a new Emacs window containing a copy of the translator comments the current PO file entry. If there is no such comments, PO mode understands that the translator wants to add a comment to the entry, and she is presented an empty screen. Comment marks (`#`) and the space following them are automatically removed before edition, and reinstated after. For translator comments pertaining to obsolete entries, the uncommenting and recommenting operations are done twice. The command `#` also has the same effect as *M-RET*, and might be easier to type. Once in the editing window, the keys *C-c C-c* allow the translator to tell she is finished with editing the comment.

The command *M-k* (`po-kill-comment`) get rid of all translator comments, while saving those comments on the kill ring. The command *M-w* (`po-kill-ring-save-comment`) takes a copy of the translator comments on the kill ring, but leaves them undisturbed in the current entry. The command *M-y* (`po-yank-comment`) completely replaces the translator comments by a string taken at the front of the kill ring. When this command is immediately repeated, the comments just inserted are withdrawn, and replaced by other strings taken along the kill ring.

On the kill ring, all strings have the same nature. There is no distinction between *translation* strings and *translator comments* strings. So, for example, let's presume the translator has just finished editing a translation, and wants to create a new translator comments for documenting why the previous translation was not good, just to remember what was the problem. Foreseeing that she will do that in her documentation, the translator will want to quote the previous translation in her translator comments. For doing so, she may initialize the translator comments with the previous translation, still at the head of the kill ring. Because editing already pushed the previous translation on the kill ring, she just has to type *M-w* prior to `#`, and the previous translation will be right there, all ready for being introduced by some explanatory text.

On the other hand, presume there are some translator comments already and that the translator wants to add to those comments, instead of wholly replacing them. Then, she should edit the comment right away with `#`. Once inside the editing window, she can use the regular GNU Emacs commands *C-y* (`yank`) and *M-y* (`yank-pop`) for getting the previous translation where she likes.

5.6 Consulting Auxiliary PO Files

An incoming feature of PO mode should help the knowledgeable translator to take advantage of translations already achieved in other languages she just happens to know, by providing these other language translation as additional context for her own work. Each PO file existing for the same package the translator is working on, but targeted to a different mother tongue language, is called an *auxiliary* PO file. Commands will exist for declaring and handling auxiliary PO files, and also for showing contexts for the entry under work. For this to work fully, all auxiliary PO files will have to be normalized.

6 Producing Binary MO Files

6.1 Invoking the msgfmt Program

Usage: msgfmt [*option*] *filename*.po ...

'-a *number*'
'--alignment=*number*'
: Align strings to *number* bytes (default: 1).

'-h'
'--help'
: Display this help and exit.

'-I *list*'
'--input-path=*list*'
: List of directories searched for input files.

'--no-hash'
: Binary file will not include the hash table.

'-o *file*'
'--output-file=*file*'
: Specify output file name as *file*.

'-v'
'--verbose'
: Detect and diagnose input file anomalies which might represent translation errors. The msgid and msgstr strings are studied and compared. It is considered abnormal that one string starts or ends with a newline while the other does not. Also, both strings should have the same number of '%' format specifiers, with matching types. For example, the check will diagnose using '%.*s' against '%s', or '%d' against '%s', or '%d' against '%x'. It can even handle positional parameters.

'-V'
'--version'
: Output version information and exit.

If input file is '-', standard input is read. If output file is '-', output is written to standard output.

The search patch for msgfmt is '/usr/local/share/nls/src/', by default. It represents the path to additional directories where other PO files can be found. This feature could be used for some PO files for standard libraries, in case we would like to spare translating their strings over and over again. The '-x' option could then exclude these strings from the generation.

6.2 The Format of GNU MO Files

The format of the generated MO files is best described by a picture, which appears below.

The first two words serve the identification of the file. The magic number will always signal GNU MO files. The number is stored in the byte order of the generating machine, so the magic number really is two numbers: `0x950412de` and `0xde120495`. The second word describes the current revision of the file format. For now the revision is 0. This might change in future versions, and ensures that the readers of MO files can distinguish new formats from old ones, so that both can be handled correctly. The version is kept separate from the magic number, instead of using different magic numbers for different formats, mainly because '`/etc/magic`' is not updated often. It might be better to have magic separated from internal format version identification.

Follow a number of pointers to later tables in the file, allowing for the extension of the prefix part of MO files without having to recompile programs reading them. This might become useful for later inserting a few flag bits, indication about the charset used, new tables, or other things.

Then, at offset O and offset T in the picture, two tables of string descriptors can be found. In both tables, each string descriptor uses two 32 bits integers, one for the string length, another for the offset of the string in the MO file, counting in bytes from the start of the file. The first table contains descriptors for the original strings, and is sorted so the original strings are in increasing lexicographical order. The second table contains descriptors for the translated strings, and is parallel to the first table: to find the corresponding translation one has to access the array slot in the second array with the same index.

Having the original strings sorted enables the use of simple binary search, for when the MO file does not contain an hashing table, or for when it is not practical to use the hashing table provided in the MO file. This also has another advantage, as the empty string in a PO file GNU `gettext` is usually *translated* into some system information attached to that particular MO file, and the empty string necessarily becomes the first in both the original and translated tables, making the system information very easy to find.

The size S of the hash table can be zero. In this case, the hash table itself is not contained in the MO file. Some people might prefer this because a precomputed hashing table takes disk space, and does not win *that* much speed. The hash table contains indices to the sorted array of strings in the MO file. Conflict resolution is done by double hashing. The precise hashing algorithm used is fairly dependent of GNU `gettext` code, and is not documented here.

As for the strings themselves, they follow the hash file, and each is terminated with a ⟨NUL⟩, and this ⟨NUL⟩ is not counted in the length which appears in the string descriptor. The `msgfmt` program has an option selecting the alignment for MO file strings. With this option, each string is separately aligned so it starts at an offset which is a multiple of the alignment value. On some RISC machines, a correct alignment will speed things up.

Nothing prevents an MO file from having embedded ⟨NUL⟩s in strings. However, the program interface currently used already presumes that strings are ⟨NUL⟩ terminated, so embedded ⟨NUL⟩s are somewhat useless. But MO file format is general enough so other interfaces would be later possible, if for example, we ever want to implement wide characters right in MO files, where ⟨NUL⟩ bytes may accidently appear.

This particular issue has been strongly debated in the GNU `gettext` development forum, and it is expectable that MO file format will evolve or change over time. It is even possible that many formats may later be supported concurrently. But surely, we got to start somewhere, and the MO file format described here is a good start. Nothing is cast in concrete, and the format may later evolve fairly easily, so we should feel comfortable with the current approach.

```
        byte
             +------------------------------------------+
          0  | magic number = 0x950412de                |
             |                                          |
          4  | file format revision = 0                 |
             |                                          |
          8  | number of strings                        |  == N
             |                                          |
         12  | offset of table with original strings    |  == O
             |                                          |
         16  | offset of table with translation strings |  == T
             |                                          |
         20  | size of hashing table                    |  == S
             |                                          |
         24  | offset of hashing table                  |  == H
             |                                          |
             .                                          .
             .    (possibly more entries later)         .
             .                                          .
             |                                          |
          O  | length & offset 0th string  ----------------.
      O + 8  | length & offset 1st string  ------------------.
              ...                                    ...   | |
O + ((N-1)*8)| length & offset (N-1)th string           |  | |
             |                                          |  | |
          T  | length & offset 0th translation  ---------------.
      T + 8  | length & offset 1st translation  -----------------.
              ...                                    ...   | | | |
T + ((N-1)*8)| length & offset (N-1)th translation      |  | | | |
             |                                          |  | | | |
          H  | start hash table                         |  | | | |
              ...                                    ...   | | | |
  H + S * 4  | end hash table                           |  | | | |
             |                                          |  | | | |
             | NUL terminated 0th string  <----------------' | | |
             |                                          |    | | |
             | NUL terminated 1st string  <------------------' | |
             |                                          |      | |
              ...                                    ...       | |
             |                                          |      | |
             | NUL terminated 0th translation  <---------------' |
             |                                          |        |
             | NUL terminated 1st translation  <-----------------'
             |                                          |
              ...                                    ...
             |                                          |
             +------------------------------------------+
```

7 The User's View

When GNU `gettext` will truly have reached is goal, average users should feel some kind of astonished pleasure, seeing the effect of that strange kind of magic that just makes their own native language appear everywhere on their screens. As for naive users, they would ideally have no special pleasure about it, merely taking their own language for *granted*, and becoming rather unhappy otherwise.

So, let's try to describe here how we would like the magic to operate, as we want the users' view to be the simplest, among all ways one could look at GNU `gettext`. All other software engineers: programmers, translators, maintainers, should work together in such a way that the magic becomes possible. This is a long and progressive undertaking, and information is available about the progress of the GNU Translation Project.

When a package is distributed, there are two kind of users: *installers* who fetch the distribution, unpack it, configure it, compile it and install it for themselves or others to use; and *end users* that call programs of the package, once these have been installed at their site. GNU `gettext` is offering magic for both installers and end users.

7.1 The Current 'NLS' Matrix for GNU

Languages are not equally supported in all GNU packages. To know if some GNU package uses GNU `gettext`, one may check the distribution for the '`NLS`' information file, for some '*ll*`.po`' files, often kept together into some '`po/`' directory, or for an '`intl/`' directory. Internationalized packages have usually many '*ll*`.po`' files, where *ll* represents the language. Section 7.3 [End Users], page 37 for a complete description of the format for *ll*.

More generally, a matrix is available for showing the current state of GNU internationalization, listing which packages are prepared for multi-lingual messages, and which languages is supported by each. Because this information changes often, this matrix is not kept within this GNU `gettext` manual. This information is often found in file '`NLS`' from various GNU distributions, but is also as old as the distribution itself. A recent copy of this '`NLS`' file, containing up-to-date information, should generally be found on most GNU archive sites.

7.2 Magic for Installers

By default, packages fully using GNU `gettext`, internally, are installed in such a way that they to allow translation of messages. At *configuration* time, those packages should automatically detect whether the underlying host system provides usable `catgets` or `gettext` functions. If neither is present, the GNU `gettext` library should be automatically prepared and used. Installers may use special options at configuration time for changing this behavior. The command '`./configure --with-gnu-gettext`' bypasses system `catgets` or `gettext` to use GNU `gettext` instead, while '`./configure --disable-nls`' produces program totally unable to translate messages.

Internationalized packages have usually many '*ll*`.po`' files. Unless translations are disabled, all those available are installed together with the package. However, the environment variable `LINGUAS` may be set, prior to configuration, to limit the installed set. `LINGUAS` should then contain a space separated list of two-letter codes, stating which languages are allowed.

7.3 Magic for End Users

We consider here those packages using GNU `gettext` internally, and for which the installers did not disable translation at *configure* time. Then, users only have to set the `LANG` environment variable to the appropriate '*ll*' prior to using the programs in the package. See Section 7.1 [Matrix], page 36. For example, let's presume a German site. At the shell prompt, users merely have to execute '`setenv LANG de`' (in `csh`) or '`export LANG; LANG=de`' (in `sh`). They could even do this from their '`.login`' or '`.profile`' file.

8 The Programmer's View

One aim of the current message catalog implementation provided by GNU `gettext` was to use the systems message catalog handling, if the installer wishes to do so. So we perhaps should first take a look at the solutions we know about. The people in the POSIX committee does not manage to agree on one of the semi-official standards which we'll describe below. In fact they couldn't agree on anything, so nothing decide only to include an example of an interface. The major Unix vendors are split in the usage of the two most important specifications: X/Opens catgets vs. Uniforums gettext interface. We'll describe them both and later explain our solution of this dilemma.

8.1 About `catgets`

The `catgets` implementation is defined in the X/Open Portability Guide, Volume 3, XSI Supplementary Definitions, Chapter 5. But the process of creating this standard seemed to be too slow for some of the Unix vendors so they created their implementations on preliminary versions of the standard. Of course this leads again to problems while writing platform independent programs: even the usage of `catgets` does not guarantee a unique interface.

Another, personal comment on this that only a bunch of committee members could have made this interface. They never really tried to program using this interface. It is a fast, memory-saving implementation, an user can happily live with it. But programmers hate it (at least me and some others do...)

But we must not forget one point: after all the trouble with transfering the rights on Unix(tm) they at last came to X/Open, the very same who published this specifications. This leads me to making the prediction that this interface will be in future Unix standards (e.g. Spec1170) and therefore part of all Unix implementation (implementations, which are *allowed* to wear this name).

8.1.1 The Interface

The interface to the `catgets` implementation consists of three functions which correspond to those used in file access: `catopen` to open the catalog for using, `catgets` for accessing the message tables, and `catclose` for closing after work is done. Prototypes for the functions and the needed definitions are in the `<nl_types.h>` header file.

`catopen` is used like in this:

```
nl_catd catd = catopen ("catalog_name", 0);
```

The function takes as the argument the name of the catalog. This usual refers to the name of the program or the package. The second parameter is not further specified in the standard. I don't even know whether it is implemented consistently among various systems. So the common advice is to use `0` as the value. The return value is a handle to the message catalog, equivalent to handles to file returned by `open`.

This handle is of course used in the `catgets` function which can be used like this:

```
char *translation = catgets (catd, set_no, msg_id, "original string");
```

The first parameter is this catalog descriptor. The second parameter specifies the set of messages in this catalog, in which the message described by `msg_id` is obtained. `catgets` therefore uses a three-stage addressing:

catalog name ⇒ set number ⇒ message ID ⇒ translation

The fourth argument is not used to address the translation. It is given as a default value in case when one of the addressing stages fail. One important thing to remember is that although the return type of catgets is `char *` the resulting string *must not* be changed. It should better `const char *`, but the standard is published in 1988, one year before ANSI C.

The last of these function functions is used and behaves as expected:

```
catclose (catd);
```

After this no `catgets` call using the descriptor is legal anymore.

8.1.2 Problems with the `catgets` Interface?!

Now that this descriptions seemed to be really easy where are the problem we speak of. In fact the interface could be used in a reasonable way, but constructing the message catalogs is a pain. The reason for this lies in the third argument of `catgets`: the unique message ID. This has to be a numeric value for all messages in a single set. Perhaps you could imagine the problems keeping such list while changing the source code. Add a new message here, remove one there. Of course there have been developed a lot of tools helping to organize this chaos but one as the other fails in one aspect or the other. We don't want to say that the other approach has no problems but they are far more easily to manage.

8.2 About `gettext`

The definition of the `gettext` interface comes from a Uniforum proposal and it is followed by at least one major Unix vendor (Sun) in its last developments. It is not specified in any official standard, though.

The main points about this solution is that it does not follow the method of normal file handling (open-use-close) and that it does not burden the programmer so many task, especially the unique key handling. Of course here is also a unique key needed, but this key is the message itself (how long or short it is). See Section 8.3 [Comparison], page 43 for a more detailed comparison of the two methods.

The following section contains a rather detailed description of the interface. We make it that detailed because this is the interface we chose for the GNU `gettext` Library. Programmers interested in using this library will be interested in this description.

8.2.1 The Interface

The minimal functionality an interface must have is a) to select a domain the strings are coming from (a single domain for all programs is not reasonable because its construction and maintenance is difficult, perhaps impossible) and b) to access a string in a selected domain.

This is principally the description of the `gettext` interface. It has an global domain which unqualified usages reference. Of course this domain is selectable by the user.

```
char *textdomain (const char *domain_name);
```

This provides the possibility to change or query the current status of the current global domain of the `LC_MESSAGE` category. The argument is a null-terminated string, whose characters must be legal in the use in filenames. If the *domain_name* argument is `NULL`, the function return the current value. If no value has been set before, the name of the default domain is returned: *messages*. Please note that although the return value of `textdomain` is of type `char *` no changing is allowed. It is also important to know that no checks of the availability are made. If the name is not available you will see this by the fact that no translations are provided.

To use a domain set by `textdomain` the function

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

is to be used. This is the simplest reasonable form one can imagine. The translation of the string *msgid* is returned if it is available in the current domain. If not available the argument itself is returned. If the argument is `NULL` the result is undefined.

One things which should come into mind is that no explicit dependency to the used domain is given. The current value of the domain for the `LC_MESSAGES` locale is used. If this changes between two executions of the same `gettext` call in the program, both calls reference a different message catalog.

For the easiest case, which is normally used in internationalized GNU packages, once at the beginning of execution a call to `textdomain` is issued, setting the domain to a unique name, normally the package name. In the following code all strings which have to be translated are filtered through the gettext function. That's all, the package speaks your language.

8.2.2 Solving Ambiguities

While this single name domain work good for most applications there might be the need to get translations from more than one domain. Of course one could switch between different domains with calls to `textdomain`, but this is really not convenient nor is it fast. A possible situation could be one case discussing while this writing: all error messages of functions in the set of common used functions should go into a separate domain `error`. By this mean we would only need to translate them once.

For this reasons there are two more functions to retrieve strings:

```
char *dgettext (const char *domain_name, const char *msgid);
char *dcgettext (const char *domain_name, const char *msgid,
                 int category);
```

Both take an additional argument at the first place, which corresponds to the argument of `textdomain`. The third argument of `dcgettext` allows to use another locale but `LC_MESSAGES`. But I really don't know where this can be useful. If the *domain_name* is `NULL` or *category* has an value beside the known ones, the result is undefined. It should also be noted that this function is not part of the second known implementation of this function family, the one found in Solaris.

A second ambiguity can arise by the fact, that perhaps more than one domain has the same name. This can be solved by specifying where the needed message catalog files can be found.

```
char *bindtextdomain (const char *domain_name,
                      const char *dir_name);
```

Calling this function binds the given domain to a file in the specified directory (how this file is determined follows below). Esp a file in the systems default place is not favored against the specified file anymore (as it would be by solely using `textdomain`). A `NULL` pointer for the *dir_name* parameter returns the binding associated with *domain_name*. If *domain_name* itself is `NULL` nothing happens and a `NULL` pointer is returned. Here again as for all the other functions is true that none of the return value must be changed!

8.2.3 Locating Message Catalog Files

Because many different languages for many different packages have to be stored we need some way to add these information to file message catalog files. The way usually used in Unix environments is have this encoding in the file name. This is also done here. The directory name given in `bindtextdomain`s second argument (or the default directory), followed by the value and name of the locale and the domain name are concatenated:

dir_name`/`*locale*`/LC_`*category*`/`*domain_name*`.mo`

The default value for *dir_name* is system specific. For the GNU library it's:

```
/usr/local/share/locale
```

locale is the value of the locale whose name is this `LC_`*category*. For `gettext` and `dgettext` this locale is always `LC_MESSAGES`. `dcgettext` specifies the locale by the third argument.[1][2]

8.2.4 Optimization of the *gettext functions

At this point of the discussion we should talk about an advantage of the GNU `gettext` implementation. Some readers might have pointed out that an internationalized program might have a poor performance if some string has to be translated in an inner loop. While this is unavoidable when the string varies from one run of the loop to the other it is simply a waste of time when the string is always the same. Take the following example:

```
{
  while (...)
    {
      puts (gettext ("Hello world"));
    }
}
```

When the locale selection does not change between two runs the resulting string is always the same. One way to use this is:

---

[1] Some system, eg Ultrix, don't have `LC_MESSAGES`. Here we use a more or less arbitrary value for it.

[2] When the system does not support `setlocale` its behavior in setting the locale values is simulated by looking at the environment variables.

```
{
  str = gettext ("Hello world");
  while (...)
    {
      puts (str);
    }
}
```

But this solution is not usable in all situation (e.g. when the locale selection changes) nor is it good readable.

The GNU C compiler, version 2.7 and above, provide another solution for this. To describe this we show here some lines of the 'intl/libgettext.h' file. For an explanation of the expression command block see section "Statements and Declarations in Expressions" in *The GNU CC Manual*.

```
#  if defined __GNUC__ && __GNUC__ == 2 && __GNUC_MINOR__ >= 7
#   define dcgettext(domainname, msgid, category)           \
  (__extension__                                                 \
   ({                                                            \
     char *result;                                               \
     if (__builtin_constant_p (msgid))                           \
       {                                                         \
         extern int _nl_msg_cat_cntr;                            \
         static char *__translation__;                           \
         static int __catalog_counter__;                         \
         if (! __translation__                                   \
             || __catalog_counter__ != _nl_msg_cat_cntr)         \
           {                                                     \
             __translation__ =                                   \
               dcgettext__ ((domainname), (msgid), (category));  \
             __catalog_counter__ = _nl_msg_cat_cntr;             \
           }                                                     \
         result = __translation__;                               \
       }                                                         \
     else                                                        \
       result = dcgettext__ ((domainname), (msgid), (category)); \
     result;                                                     \
    }))
#  endif
```

The interesting thing here is the `__builtin_constant_p` predicate. This is evaluated at compile time and so optimization can take place immediately. Here two cases are distinguished: the argument to `gettext` is not a constant value in which case simply the function `dcgettext__` is called, the real implementation of the `dcgettext` function.

If the string argument *is* constant we can reuse the once gained translation when the locale selection has not changed. This is exactly what is done here. The `_nl_msg_cat_cntr` variable is defined in the 'loadmsgcat.c' which is available in 'libintl.a' and is changed whenever a new message catalog is loaded.

8.3 Comparing the Two Interfaces

The following discussion is perhaps a little bit colored. As said above we implemented GNU `gettext` following the Uniforum proposal and this surely has its reasons. But it should show how we came to this decision.

First we take a look at the developing process. When we write an application using NLS provided by `gettext` we proceed as always. Only when we come to a string which might be seen by the users and thus has to be translated we use `gettext("...")` instead of `"..."`. At the beginning of each source file (or in a central header file) we define

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

Even this definition can be avoided when the system supports the `gettext` function in its C library. When we compile this code the result is the same as if no NLS code is used. When you take a look at the GNU `gettext` code you will see that we use `_("...")` instead of `gettext("...")`. This reduces the number of additional characters per translatable string to *3* (in words: three).

When now a production version of the program is needed we simply replace the definition

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

by

```
#include <libintl.h>
#define _(String) gettext (String)
```

and include the header '`libintl.h`'. Additionally we run the program '`xgettext`' on all source code file which contain translatable strings and we are gone. We have a running program which does not depend on translations to be available, but which can use any that becomes available.

The same procedure can be done for the `gettext_noop` invocations (see Section 3.4 [Special cases], page 20). First you can define `gettext_noop` to a no-op macro and later use the definition from '`libintl.h`'. Because this name is not used in Suns implementation of '`libintl.h`', you should consider the following code for your project:

```
#ifdef gettext_noop
# define N_(Str) gettext_noop (Str)
#else
# define N_(Str) (Str)
#endif
```

`N_` is a short form similar to `_`. The '`Makefile`' in the '`po/`' directory of GNU gettext knows by default both of the mentioned short forms so you are invited to follow this proposal for your own ease.

Now to `catgets`. The main problem is the work for the programmer. Every time he comes to a translatable string he has to define a number (or a symbolic constant) which has also be defined in the message catalog file. He also has to take care for duplicate entries, duplicate message IDs etc. If he wants to have the same quality in the message catalog as the GNU `gettext` program provides he also has to put the descriptive comments for the strings and the location in all source code files in the message catalog. This is nearly a Mission: Impossible.

But there are also some points people might call advantages speaking for `catgets`. If you have a single word in a string and this string is used in different contexts it is likely that in one or the other language the word has different translations. Example:

```
printf ("%s: %d", gettext ("number"), number_of_errors)

printf ("you should see %d %s", number_count,
        number_count == 1 ? gettext ("number") : gettext ("numbers"))
```

Here we have to translate two times the string `"number"`. Even if you do not speak a language beside English it might be possible to recognize that the two words have a different meaning. In German the first appearance has to be translated to `"Anzahl"` and the second to `"Zahl"`.

Now you can say that this example is really esoteric. And you are right! This is exactly how we felt about this problem and decide that it does not weight that much. The solution for the above problem could be very easy:

```
printf (gettext ("number: %d"), number_of_errors)

printf (number_count == 1 ? gettext ("you should see %d number")
                          : gettext ("you should see %d numbers"),
        number_count)
```

We believe that we can solve all conflicts with this method. If it is difficult one can also consider changing one of the conflicting string a little bit. But it is not impossible to overcome.

Translator note: It is perhaps appropriate here to tell those English speaking programmers that the plural form of a noun cannot be formed by appending a single 's'. Most other languages use different methods. So you should at least use the method given in the above example.

But I have been told that some languages have even more complex rules. A good approach might be to consider methods like the one used for `LC_TIME` in the POSIX.2 standard.

8.4 Using libintl.a in own programs

Starting with version 0.9.4 the library `libintl.h` should be more or less self-contained. I.e. you can use it in your own programs. The '`Makefile`' will put the header and the library in directories selected using the `$(prefix)`.

One exception of the above is found on HP-UX systems. Here the C library does not contain the `alloca` function (and the HP compiler does not generate it inlined). But it is not intended to rewrite the whole library just because of this dumb system. Instead include the `alloca` function in all package you use the `libintl.a` in.

8.5 Being a gettext grok

To fully exploit the functionality of the GNU `gettext` library it is surely helpful to read the source code. But for those who don't want to spend that much time in reading the (sometimes complicated) code here is a list comments:

- Changing the language at runtime

  For interactive programs it might be useful to offer a selection of the used language at runtime. To understand how to do this one need to know how the used language is determined while executing the `gettext` function. The method which is presented here only works correctly with the GNU implementation of the `gettext` functions. It is not possible with underlying `catgets` functions or `gettext` functions from the systems C library. The exception is of course the GNU C Library which uses the GNU gettext Library for message handling.

  In the function `dcgettext` at every call the current setting of the highest priority environment variable is determined and used. Highest priority means here the following list with decreasing priority:

  1. `LANGUAGE`
  2. `LC_ALL`
  3. `LC_xxx`, according to selected locale
  4. `LANG`

  Afterwards the path is constructed using the found value and the translation file is loaded if available.

  What is now when the value for, say, `LANGUAGE` changes. According to the process explained above the new value of this variable is found as soon as the `dcgettext` function is called. But this also means the (perhaps) different message catalog file is loaded. In other words: the used language is changed.

  But there is one little hook. The code for gcc-2.7.0 and up provides some optimization. This optimization normally prevents the calling of the `dcgettext` function as long as now new catalog is loaded. But if `dcgettext` is not called we program also cannot find the `LANGUAGE` variable be changed (see Section 8.2.4 [Optimized gettext], page 41). But the solution is very easy. Include the following code in the language switching function.

  ```
  /* Change language.  */
  setenv ("LANGUAGE", "fr", 1);

  /* Make change known.  */
  {
    extern int  _nl_msg_cat_cntr;
    ++_nl_msg_cat_cntr;
  }
  ```

  The variable `_nl_msg_cat_cntr` is defined in '`loadmsgcat.c`'.

8.6 Temporary Notes for the Programmers Chapter

8.6.1 Temporary - Two Possible Implementations

There are two competing methods for language independent messages: the X/Open `catgets` method, and the Uniforum `gettext` method. The `catgets` method indexes messages by integers; the `gettext` method indexes them by their English translations. The `catgets` method has been around longer and is supported by more vendors. The `gettext` method is supported by Sun, and it has been heard that the COSE multi-vendor initiative is supporting it. Neither method is a POSIX standard; the POSIX.1 committee had a lot of disagreement in this area.

Neither one is in the POSIX standard. There was much disagreement in the POSIX.1 committee about using the `gettext` routines vs. `catgets` (XPG). In the end the committee couldn't agree on anything, so no messaging system was included as part of the standard. I believe the informative annex of the standard includes the XPG3 messaging interfaces, "...as an example of a messaging system that has been implemented..."

They were very careful not to say anywhere that you should use one set of interfaces over the other. For more on this topic please see the Programming for Internationalization FAQ.

8.6.2 Temporary - About `catgets`

There have been a few discussions of late on the use of `catgets` as a base. I think it important to present both sides of the argument and hence am opting to play devil's advocate for a little bit.

I'll not deny the fact that `catgets` could have been designed a lot better. It currently has quite a number of limitations and these have already been pointed out.

However there is a great deal to be said for consistency and standardization. A common recurring problem when writing Unix software is the myriad portability problems across Unix platforms. It seems as if every Unix vendor had a look at the operating system and found parts they could improve upon. Undoubtedly, these modifications are probably innovative and solve real problems. However, software developers have a hard time keeping up with all these changes across so many platforms.

And this has prompted the Unix vendors to begin to standardize their systems. Hence the impetus for Spec1170. Every major Unix vendor has committed to supporting this standard and every Unix software developer waits with glee the day they can write software to this standard and simply recompile (without having to use autoconf) across different platforms.

As I understand it, Spec1170 is roughly based upon version 4 of the X/Open Portability Guidelines (XPG4). Because `catgets` and friends are defined in XPG4, I'm led to believe that `catgets` is a part of Spec1170 and hence will become a standardized component of all Unix systems.

8.6.3 Temporary - Why a single implementation

Now it seems kind of wasteful to me to have two different systems installed for accessing message catalogs. If we do want to remedy `catgets` deficiencies why don't we try to expand `catgets` (in a compatible manner) rather than implement an entirely new system. Otherwise, we'll end up with two message catalog access systems installed with an operating system - one set of routines for GNU software, and another set of routines (catgets) for all other software. Bloated?

Supposing another catalog access system is implemented. Which do we recommend? At least for Linux, we need to attract as many software developers as possible. Hence we need to make it as easy for them to port their software as possible. Which means supporting `catgets`. We will be implementing the `glocale` code within our `libc`, but does this mean we also have to incorporate another message catalog access scheme within our `libc` as well? And what about people who are going to be using the `glocale +` non-`catgets` routines. When they port their software to other platforms, they're now going to have to include the front-end (`glocale`) code plus the back-end code (the non-`catgets` access routines) with their software instead of just including the `glocale` code with their software.

Message catalog support is however only the tip of the iceberg. What about the data for the other locale categories. They also have a number of deficiencies. Are we going to abandon them as well and develop another duplicate set of routines (should `glocale` expand beyond message catalog support)?

Like many parts of Unix that can be improved upon, we're stuck with balancing compatibility with the past with useful improvements and innovations for the future.

8.6.4 Temporary - Double layer solution

GNU locale implements a `gettext`-style interface on top of a `catgets`-style interface.

This is not needless complexity. It is absolutely vital, because it enables `gettext` to run on top of `catgets`, which enables Linux International to recommend users use it *today*.

Rewriting `gettext` so that it could use *either* `catgets` *or* some simpler mechanism would not break anything, but would not reduce complexity either. It might be worth doing, but it isn't urgent.

In general, simplicity is not enough of a reason to rewrite a program that works. Simplicity is just one desirable thing. It is not overridingly important.

8.6.5 Temporary - Notes

X/Open agreed very late on the standard form so that many implementations differ from the final form. Both of my system (old Linux catgets and Ultrix-4) have a strange variation.

OK. After incorporating the last changes I have to spend some time on making the GNU/Linux libc gettext functions. So in future Solaris is not the only system having gettext.

9 The Translator's View

9.1 Introduction 0

GNU is going international! The GNU Translation Project is a way to get maintainers, translators and users all together, so GNU will gradually become able to speak many native languages.

The GNU `gettext` tool set contains *everything* maintainers need for internationalizing their packages for messages. It also contains quite useful tools for helping translators at localizing messages to their native language, once a package has already been internationalized.

To achieve the GNU Translation Project, we need many interested people who like their own language and write it well, and who are also able to synergize with other translators speaking the same language. If you'd like to volunteer to *work* at translating messages, please send mail to your translating team.

Each team has its own mailing list, courtesy of Linux International. You may reach your translating team at the address '*ll*`@li.org`', replacing *ll* by the two-letter ISO 639 code for your language. Language codes are *not* the same as country codes given in ISO 3166. The following translating teams exist:

> Chinese `zh`, Czech `cs`, Danish `da`, Dutch `nl`, Esperanto `eo`, Finnish `fi`, French `fr`, Irish `ga`, German `de`, Greek `el`, Italian `it`, Japanese `ja`, Indonesian `in`, Norwegian `no`, Polish `pl`, Portuguese `pt`, Russian `ru`, Spanish `es`, Swedish `sv` and Turkish `tr`.

For example, you may reach the Chinese translating team by writing to '`zh@li.org`'. When you become a member of the translating team for your own language, you may subscribe to its list. For example, Swedish people can send a message to '`sv-request@li.org`', having this message body:

```
subscribe
```

Keep in mind that team members should be interested in *working* at translations, or at solving translational difficulties, rather than merely lurking around. If your team does not exist yet and you want to start one, please write to '`gnu-translation@prep.ai.mit.edu`'; you will then reach the GNU coordinator for all translator teams.

A handful of GNU packages have already been adapted and provided with message translations for several languages. Translation teams have begun to organize, using these packages as a starting point. But there are many more packages and many languages for which we have no volunteer translators. If you would like to volunteer to work at translating messages, please send mail to '`gnu-translation@prep.ai.mit.edu`' indicating what language(s) you can work on.

9.2 Introduction 1

This is now official, GNU is going international! Here is the announcement submitted for the January 1995 GNU Bulletin:

> A handful of GNU packages have already been adapted and provided with message translations for several languages. Translation teams have begun to organize, using these packages as a starting point. But there are many more packages and many languages for which we have no volunteer translators. If you'd like to volunteer to work at translating messages, please send mail to '`gnu-translation@prep.ai.mit.edu`' indicating what language(s) you can work on.

This document should answer many questions for those who are curious about the process or would like to contribute. Please at least skim over it, hoping to cut down a little of the high volume of email generated by this collective effort towards GNU internationalization.

GNU programming is done in English, and currently, English is used as the main communicating language between national communities collaborating to the GNU project. This very document is written in English. This will not change in the foreseeable future.

However, there is a strong appetite from national communities for having more software able to write using national language and habits, and there is an on-going effort to modify GNU software in such a way that it becomes able to do so. The experiments driven so far raised an enthusiastic response from pretesters, so we believe that GNU internationalization is dedicated to succeed.

For suggestion clarifications, additions or corrections to this document, please email to '`gnu-translation@prep.ai.mit.edu`'.

9.3 Discussions

Facing this internationalization effort, a few users expressed their concerns. Some of these doubts are presented and discussed, here.

- Smaller groups

  Some languages are not spoken by a very large number of people, so people speaking them sometimes consider that there may not be all that much demand such versions of GNU packages. Moreover, many people being *into computers*, in some countries, generally seem to prefer English versions of their software.

  On the other end, people might enjoy their own language a lot, and be very motivated at providing to themselves the pleasure of having their beloved GNU software speaking their mother tongue. They do themselves a personal favor, and do not pay that much attention to the number of people beneficiating of their work.
- Misinterpretation

  Other users are shy to push forward their own language, seeing in this some kind of misplaced propaganda. Someone thought there must be some users of the language over the networks pestering other people with it.

  But any spoken language is worth localization, because there are people behind the language for whom the language is important and dear to their hearts.
- Odd translations

  The biggest problem is to find the right translations so that everybody can understand the messages. Translations are usually a little odd. Some people get used to English, to the extent they may find translations into their own language "rather pushy, obnoxious and sometimes even hilarious." As a French speaking man, I have the experience of those instruction manuals for goods, so poorly translated in French in Korea or Taiwan...

The fact is that we sometimes have to create a kind of national computer culture, and this is not easy without the collaboration of many people liking their mother tongue. This is why translations are better achieved by people knowing and loving their own language, and ready to work together at improving the results they obtain.

- Dependencies over the GPL

  Some people wonder if using GNU `gettext` necessarily brings their package under the protective wing of the GNU General Public License, when they do not want to make their program free, or want other kinds of freedom. The simplest answer is yes.

  The mere marking of localizable strings in a package, or conditional inclusion of a few lines for initialization, is not really including GPL'ed code. However, the localization routines themselves are under the GPL and would bring the remainder of the package under the GPL if they were distributed with it. So, I presume that, for those for which this is a problem, it could be circumvented by letting to the end installers the burden of assembling a package prepared for localization, but not providing the localization routines themselves.

9.4 Organization

On a larger scale, the true solution would be to organize some kind of fairly precise set up in which volunteers could participate. I gave some thought to this idea lately, and realize there will be some touchy points. I thought of writing to Richard Stallman to launch such a project, but feel it might be good to shake out the ideas between ourselves first. Most probably that Linux International has some experience in the field already, or would like to orchestrate the volunteer work, maybe. Food for thought, in any case!

I guess we have to setup something early, somehow, that will help many possible contributors of the same language to interlock and avoid work duplication, and further be put in contact for solving together problems particular to their tongue (in most languages, there are many difficulties peculiar to translating technical English). My Swedish contributor acknowledged these difficulties, and I'm well aware of them for French.

This is surely not a technical issue, but we should manage so the effort of locale contributors be maximally useful, despite the national team layer interface between contributors and maintainers.

GNU needs some setup for coordinating language coordinators. Localizing evolving GNU programs will surely become a permanent and continuous activity in GNU, once started. The setup should be minimally completed and tested before GNU `gettext` becomes an official reality. The email address '`gnu-translation@prep.ai.mit.edu`' has been setup for receiving offers from volunteers and general email on these topics. This address reaches the GNU Translation Project coordinator.

9.4.1 Central Coordination

I also think GNU will need sooner than it thinks, that someone setup a way to organize and coordinate these groups. Some kind of group of groups. My opinion is that it would be good that GNU delegate this task to a small group of collaborating volunteers, shortly. Perhaps in '`gnu.announce`' a list of this national committee's can be published.

My role as coordinator would simply be to refer to Ulrich any German speaking volunteer interested to localization of GNU programs, and maybe helping national groups to initially organize, while maintaining national registries for until national groups are ready to take over. In fact, the coordinator should ease volunteers to get in contact with one another for creating national teams, which should then select one coordinator per language, or country (regionalized language). If well done, the coordination should be useful without being an overwhelming task, the time to put delegations in place.

9.4.2 National Teams

I suggest we look for volunteer coordinators/editors for individual languages. These people will scan contributions of translation files for various programs, for their own languages, and will ensure high and uniform standards of diction.

From my current experience with other people in these days, those who provide localizations are very enthusiastic about the process, and are more interested in the localization process than in the program they localize, and want to do many programs, not just one. This seems to confirm that having a coordinator/editor for each language is a good idea.

We need to choose someone who is good at writing clear and concise prose in the language in question. That is hard—we can't check it ourselves. So we need to ask a few people to judge each others' writing and select the one who is best.

I announce my prerelease to a few dozen people, and you would not believe all the discussions it generated already. I shudder to think what will happen when this will be launched, for true, officially, world wide. Who am I to arbitrate between two Czekolsovak users contradicting each other, for example?

I assume that your German is not much better than my French so that I would not be able to judge about these formulations. What I would suggest is that for each language there is a group for people who maintain the PO files and judge about changes. I suspect there will be cultural differences between how such groups of people will behave. Some will have relaxed ways, reach consensus easily, and have anyone of the group relate to the maintainers, while others will fight to death, organize heavy administrations up to national standards, and use strict channels.

The German team is putting out a good example. Right now, they are maybe half a dozen people revising translations of each other and discussing the linguistic issues. I do not even have all the names. Ulrich Drepper is taking care of coordinating the German team. He subscribed to all my pretest lists, so I do not even have to warn him specifically of incoming releases.

I'm sure, that is a good idea to get teams for each language working on translations. That will make the translations better and more consistent.

9.4.2.1 Sub-Cultures

Taking French for example, there are a few sub-cultures around computers which developed diverging vocabularies. Picking volunteers here and there without addressing this problem in an organized way, soon in the project, might produce a distasteful mix of GNU programs, and possibly trigger endless quarrels among those who really care.

Keeping some kind of unity in the way French localization of GNU programs is achieved is a difficult (and delicate) job. Knowing the latin character of French people (:-), if we take this the wrong way, we could end up nowhere, or spoil a lot of energies. Maybe we should begin to address this problem seriously *before* GNU `gettext` become officially published. And I suspect that this means soon!

9.4.2.2 Organizational Ideas

I expect the next big changes after the official release. Please note that I use the German translation of the short GPL message. We need to set a few good examples before the localization goes out for true in GNU. Here are a few points to discuss:

- Each group should have one FTP server (at least one master).
- The files on the server should reflect the latest version (of course!) and it should also contain a RCS directory with the corresponding archives (I don't have this now).
- There should also be a ChangeLog file (this is more useful than the RCS archive but can be generated automatically from the later by Emacs).
- A *core group* should judge about questionable changes (for now this group consists solely by me but I ask some others occasionally; this also seems to work).

9.4.3 Mailing Lists

If we get any inquiries about GNU `gettext`, send them on to:

`'gnu-translation@prep.ai.mit.edu'`

The '`*-pretest`' lists are quite useful to me, maybe the idea could be generalized to all GNU packages. But each maintainer his/her way!

Francois, we have a mechanism in place here at '`gnu.ai.mit.edu`' to track teams, support mailing lists for them and log members. We have a slight preference that you use it. If this is OK with you, I can get you clued in.

Things are changing! A few years ago, when Daniel Fekete and I asked for a mailing list for GNU localization, nested at the FSF, we were politely invited to organize it anywhere else, and so did we. For communicating with my pretesters, I later made a handful of mailing lists located at iro.umontreal.ca and administrated by `majordomo`. These lists have been *very* dependable so far...

I suspect that the German team will organize itself a mailing list located in Germany, and so forth for other countries. But before they organize for true, it could surely be useful to offer mailing lists located at the FSF to each national team. So yes, please explain me how I should proceed to create and handle them.

We should create temporary mailing lists, one per country, to help people organize. Temporary, because once regrouped and structured, it would be fair the volunteers from country bring back *their* list in there and manage it as they want. My feeling is that, in the long run, each team should run its own list, from within their country. There also should be some central list to which all teams could subscribe as they see fit, as long as each team is represented in it.

9.5 Information Flow

There will surely be some discussion about this messages after the packages are finally released. If people now send you some proposals for better messages, how do you proceed? Jim, please note that right now, as I put forward nearly a dozen of localizable programs, I receive both the translations and the coordination concerns about them.

If I put one of my things to pretest, Ulrich receives the announcement and passes it on to the German team, who make last minute revisions. Then he submits the translation files to me *as the maintainer*. For GNU packages I do not maintain, I would not even hear about it. This scheme could be made to work GNU-wide, I think. For security reasons, maybe Ulrich (national coordinators, in fact) should update central registry kept by GNU (Jim, me, or Len's recruits) once in a while.

In December/January, I was aggressively ready to internationalize all of GNU, giving myself the duty of one small GNU package per week or so, taking many weeks or months for bigger packages. But it does not work this way. I first did all the things I'm responsible for. I've nothing against some missionary work on other maintainers, but I'm also loosing a lot of energy over it—same debates over again.

And when the first localized packages are released we'll get a lot of responses about ugly translations :-). Surely, and we need to have beforehand a fairly good idea about how to handle the information flow between the national teams and the package maintainers.

Please start saving somewhere a quick history of each PO file. I know for sure that the file format will change, allowing for comments. It would be nice that each file has a kind of log, and references for those who want to submit comments or gripes, or otherwise contribute. I sent a proposal for a fast and flexible format, but it is not receiving acceptance yet by the GNU deciders. I'll tell you when I have more information about this.

10 The Maintainer's View

The maintainer of a package has many responsibilities. One of them is ensuring that the package will install easily on many platforms, and that the magic we described earlier (see Chapter 7 [Users], page 36) will work for installers and end users.

Of course, there are many possible ways by which GNU `gettext` might be integrated in a distribution, and this chapter does not cover them in all generality. Instead, it details one possible approach which is especially adequate for many GNU distributions, because GNU `gettext` is purposely for helping the internationalization of the whole GNU project. So, the maintainer's view presented here presumes that the package already has a '`configure.in`' file and uses Autoconf.

Nevertheless, GNU `gettext` may surely be useful for non-GNU packages, but the maintainers of such packages might have to show imagination and initiative in organizing their distributions so `gettext` work for them in all situations. There are surely many, out there.

Even if `gettext` methods are now stabilizing, slight adjustments might be needed between successive `gettext` versions, so you should ideally revise this chapter in subsequent releases, looking for changes.

10.1 Flat or Non-Flat Directory Structures

Some GNU packages are distributed as `tar` files which unpack in a single directory, these are said to be *flat* distributions. Other GNU packages have a one level hierarchy of subdirectories, using for example a subdirectory named '`doc/`' for the Texinfo manual and man pages, another called '`lib/`' for holding functions meant to replace or complement C libraries, and a subdirectory '`src/`' for holding the proper sources for the package. These other distributions are said to be *non-flat*.

For now, we cannot say much about flat distributions. A flat directory structure has the disadvantage of increasing the difficulty of updating to a new version of GNU `gettext`. Also, if you have many PO files, this could somewhat pollute your single directory. In the GNU `gettext` distribution, the '`misc/`' directory contains a shell script named '`combine-sh`'. That script may be used for combining all the C files of the '`intl/`' directory into a pair of C files (one '`.c`' and one '`.h`'). Those two generated files would fit more easily in a flat directory structure, and you will then have to add these two files to your project.

Maybe because GNU `gettext` itself has a non-flat structure, we have more experience with this approach, and this is what will be described in the remaining of this chapter. Some maintainers might use this as an opportunity to unflatten their package structure. Only later, once gained more experience adapting GNU `gettext` to flat distributions, we might add some notes about how to proceed in flat situations.

10.2 Prerequisite Works

There are some works which are required for using GNU `gettext` in one of your package. These works have some kind of generality that escape the point by point descriptions used in the remainder of this chapter. So, we describe them here.

- Before attempting to use you should install some other packages first. Ensure that recent versions of GNU `m4`, GNU Autoconf and GNU `gettext` are already installed at your site, and if not, proceed to do this first. If you got to install these things, beware that GNU `m4` must be fully installed before GNU Autoconf is even *configured*.

  Those three packages are only needed to you, as a maintainer; the installers of your own package and end users do not really need any of GNU `m4`, GNU Autoconf or GNU `gettext` for successfully installing and running your package, with messages properly translated. But this is not completely true if you provide internationalized shell scripts within your own package: GNU `gettext` shall then be installed at the user site if the end users want to see the translation of shell script messages.
- Your package should use Autoconf and have a '`configure.in`' file. If it does not, you have to learn how. The Autoconf documentation is quite well written, it is a good idea that you print it and get familiar with it.
- Your C sources should have already been modified according to instructions given earlier in this manual. See Chapter 3 [Sources], page 16.
- Your '`po/`' directory should receive all PO files submitted to you by the translator teams, each having '*ll*`.po`' as a name. This is not usually easy to get translation work done before your package gets internationalized and available! Since the cycle has to start somewhere, the easiest for the maintainer is to start with absolutely no PO files, and wait until various translator teams get interested in your package, and submit PO files.

It is worth adding here a few words about how the maintainer should ideally behave with PO files submissions. As a maintainer, your role is to authentify the origin of the submission as being the representative of the appropriate GNU translating team (forward the submission to '`gnu-translation@prep.ai.mit.edu`' in case of doubt), to ensure that the PO file format is not severely broken and does not prevent successful installation, and for the rest, to merely to put these PO files in '`po/`' for distribution.

As a maintainer, you do not have to take on your shoulders the responsibility of checking if the translations are adequate or complete, and should avoid diving into linguistic matters. Translation teams drive themselves and are fully responsible of their linguistic choices for GNU. Keep in mind that translator teams are *not* driven by maintainers. You can help by carefully redirecting all communications and reports from users about linguistic matters to the appropriate translation team, or explain users how to reach or join their team. The simplest might be to send them the '`NLS`' file.

Maintainers should *never ever* apply PO file bug reports themselves, short-cutting translation teams. If some translator has difficulty to get some of her points through her team, it should not be an issue for her to directly negotiate translations with maintainers. Teams ought to settle their problems themselves, if any. If you, as a maintainer, ever think there is a real problem with a team, please never try to *solve* a team's problem on your own.

10.3 Invoking the `gettextize` Program

Some files are consistently and identically needed in every package internationalized through GNU `gettext`. As a matter of convenience, the `gettextize` program puts all these files right in your package. This program has the following synopsis:

```
gettextize [ option... ] [ directory ]
```

and accepts the following options:

'-f'
'--force' Force replacement of files which already exist.

'-h'
'--help' Display this help and exit.

'--version'
Output version information and exit.

If *directory* is given, this is the top level directory of a package to prepare for using GNU `gettext`. If not given, it is assumed that the current directory is the top level directory of such a package.

The program `gettextize` provides the following files. However, no existing file will be replaced unless the option `--force` (`-f`) is specified.

1. The 'NLS' file is copied in the main directory of your package, the one being at the top level. This file gives the main indications about how to install and use the Native Language Support features of your program. You might elect to use a more recent copy of this 'NLS' file than the one provided through `gettextize`, if you have one handy. You may also fetch a more recent copy of file 'NLS' from most GNU archive sites.
2. A 'po/' directory is created for eventually holding all translation files, but initially only containing the file 'po/Makefile.in.in' from the GNU `gettext` distribution. (beware the double '.in' in the file name). If the 'po/' directory already exists, it will be preserved along with the files it contains, and only 'Makefile.in.in' will be overwritten.
3. A 'intl/' directory is created and filled with most of the files originally in the 'intl/' directory of the GNU `gettext` distribution. Also, if option `--force` (`-f`) is given, the 'intl/' directory is emptied first.

If your site support symbolic links, `gettextize` will not actually copy the files into your package, but establish symbolic links instead. This avoids duplicating the disk space needed in all packages. Merely using the '-h' option while creating the `tar` archive of your distribution will resolve each link by an actual copy in the distribution archive. So, to insist, you really should use '-h' option with `tar` within your `dist` goal of your main 'Makefile.in'.

It is interesting to understand that most new files for supporting GNU `gettext` facilities in one package go in 'intl/' and 'po/' subdirectories. One distinction between these two directories is that 'intl/' is meant to be completely identical in all packages using GNU `gettext`, while all newly created files, which have to be different, go into 'po/'. There is a common 'Makefile.in.in' in 'po/', because the 'po/' directory needs its own 'Makefile', and it has been designed so it can be identical in all packages.

10.4 Files You Must Create or Alter

Besides files which are automatically added through `gettextize`, there are many files needing revision for properly interacting with GNU `gettext`. If you are closely following

GNU standards for Makefile engineering and auto-configuration, the adaptations should be easier to achieve. Here is a point by point description of the changes needed in each.

So, here comes a list of files, each one followed by a description of all alterations it needs. Many examples are taken out from the GNU `gettext` 0.10 distribution itself. You may indeed refer to the source code of the GNU `gettext` package, as it is intended to be a good example and master implementation for using its own functionality.

10.4.1 'POTFILES' in 'po/'

The 'po/' directory should receive a file named 'POTFILES.in'. This file tells which files, among all program sources, have marked strings needing translation. Here is an example of such a file:

```
# List of source files containing translatable strings.
# Copyright (C) 1995 Free Software Foundation, Inc.

# Common library files
lib/error.c
lib/getopt.c
lib/xmalloc.c

# Package source files
src/gettextp.c
src/msgfmt.c
src/xgettext.c
```

Dashed comments and white lines are ignored. All other lines list those source files containing strings marked for translation (see Section 3.2 [Mark Keywords], page 17), in a notation relative to the top level of your whole distribution, rather than the location of the 'POTFILES.in' file itself.

10.4.2 'configure.in' at top level

1. Declare the package and version.

   This is done by a set of lines like these:

   ```
   PACKAGE=gettext
   VERSION=0.10
   AC_DEFINE_UNQUOTED(PACKAGE, "$PACKAGE")
   AC_DEFINE_UNQUOTED(VERSION, "$VERSION")
   AC_SUBST(PACKAGE)
   AC_SUBST(VERSION)
   ```

   Of course, you replace '`gettext`' with the name of your package, and '`0.10`' by its version numbers, exactly as they should appear in the packaged `tar` file name of your distribution ('`gettext-0.10.tar.gz`', here).

2. Declare the available translations.

   This is done by defining `ALL_LINGUAS` to the white separated, quoted list of available languages, in a single line, like this:

```
ALL_LINGUAS="de fr"
```

This example means that German and French PO files are available, so that these languages are currently supported by your package. If you want to further restrict, at installation time, the set of installed languages, this should not be done by modifying `ALL_LINGUAS` in '`configure.in`', but rather by using the `LINGUAS` environment variable (see Section 7.2 [Installers], page 36).

3. Check for internationalization support.

   Here is the main `m4` macro for triggering internationalization support. Just add this line to '`configure.in`':

   ```
   ud_GNU_GETTEXT
   ```

   This call is purposely simple, even if it generates a lot of configure time checking and actions.

4. Obtain some '`libintl.h`' header file.

   Once you called `ud_GNU_GETTEXT` in '`configure.in`', use:

   ```
   AC_LINK_FILES($nls_cv_header_libgt, $nls_cv_header_intl)
   ```

   This will create one header file '`libintl.h`'. The reason for this has to do with the fact that some systems, using the Uniforum message handling functions, already have a file of this name.

   The `AC_LINK_FILES` call has not been integrated into the `ud_GNU_GETTEXT` macro because there can be only one such call in a '`configure`' file. If you already use it, you will have to *merge* the needed `AC_LINK_FILES` within yours, by adding the first argument at the end of the list of your first argument, and adding the second argument at the end of the list of your second argument.

5. Have output files created.

   The `AC_OUTPUT` directive, at the end of your '`configure.in`' file, needs to be modified in two ways:

   ```
   AC_OUTPUT([existing configuration files intl/Makefile po/Makefile.in],
   [sed -e "/POTFILES =/r po/POTFILES" po/Makefile.in > po/Makefile
   existing additional actions])
   ```

   The modification to the first argument to `AC_OUTPUT` asks for substitution in the '`intl/`' and '`po/`' directories. Note the '`.in`' suffix used for '`po/`' only. This is because the distributed file is really '`po/Makefile.in.in`'.

   The modification to the second argument ensures that '`po/Makefile`' gets generated out of the '`po/Makefile.in`' just created, including in it the '`po/POTFILES`' produced by `ud_GNU_GETTEXT`. Two steps are needed because '`po/POTFILES`' can get lengthy in some packages, too lengthy in fact for being able to merely use an Autoconf substituted variable, as many `sed`s cannot handle very long lines.

10.4.3 '`aclocal.m4`' at top level

If you do not have an '`aclocal.m4`' file in your distribution, the simplest is taking a copy of '`aclocal.m4`' from GNU `gettext`. But to be precise, you only need macros `ud_LC_MESSAGES`, `ud_WITH_NLS` and `ud_GNU_GETTEXT`, so you may use an editor and remove macros you do not need.

If you already have an 'aclocal.m4' file, then you will have to merge the said macros into your 'aclocal.m4'. Note that if you are upgrading from a previous release of GNU gettext, you should most probably *replace* the said macros, as they usually change a little from one release of GNU gettext to the next. Their contents may vary as we get more experience with strange systems out there.

These macros check for the internationalization support functions and related informations. Hopefully, once stabilized, these macros might be integrated in the standard Autoconf set, because this piece of m4 code will be the same for all projects using GNU gettext.

10.4.4 'acconfig.h' at top level

If you do not have an 'acconfig.h' file in your distribution, the simplest is use take a copy of 'acconfig.h' from GNU gettext. But to be precise, you only need the lines and comments for ENABLE_NLS, HAVE_CATGETS, HAVE_GETTEXT and HAVE_LC_MESSAGES, so you may use an editor and remove everything else. If you already have an 'acconfig.h' file, then you should merge the said definitions into your 'acconfig.h'.

10.4.5 'Makefile.in' at top level

Here are a few modifications you need to make to your main, top-level 'Makefile.in' file.

1. Add the following lines near the beginning of your 'Makefile.in', so the 'dist:' goal will work properly (as explained further down):

   ```
   PACKAGE = @PACKAGE@
   VERSION = @VERSION@
   ```

2. Add file 'NLS' to the DISTFILES definition, so the file gets distributed.
3. Wherever you process subdirectories in your 'Makefile.in', be sure you also process @INTLSUB@ and @POSUB@, which are replaced respectively by 'intl' and 'po', or empty when the configuration processes decides these directories should not be processed.

   Here is an example of a canonical order of processing. In this example, we also define SUBDIRS in Makefile.in for it to be further used in the 'dist:' goal.

   ```
   SUBDIRS = doc lib @INTLSUB@ src @POSUB@
   ```

   that you will have to adapt to your own package.
4. A delicate point is the 'dist:' goal, as both 'intl/Makefile' and 'po/Makefile' will later assume that the proper directory has been set up from the main 'Makefile'. Here is an example at what the 'dist:' goal might look like:

   ```
   distdir = $(PACKAGE)-$(VERSION)
   dist: Makefile
   rm -fr $(distdir)
   mkdir $(distdir)
   chmod 777 $(distdir)
   for file in $(DISTFILES); do \
     ln $$file $(distdir) 2>/dev/null || cp -p $$file $(distdir); \
   done
   for subdir in $(SUBDIRS); do \
   ```

```
  mkdir $(distdir)/$$subdir || exit 1; \
  chmod 777 $(distdir)/$$subdir; \
  (cd $$subdir && $(MAKE) $@) || exit 1; \
done
tar chozf $(distdir).tar.gz $(distdir)
rm -fr $(distdir)
```

10.4.6 'Makefile.in' in 'src/'

Some of the modifications made in the main 'Makefile.in' will also be needed in the 'Makefile.in' from your package sources, which we assume here to be in the 'src/' subdirectory. Here are all the modifications needed in 'src/Makefile.in':

1. In view of the 'dist:' goal, you should have these lines near the beginning of 'src/Makefile.in':

   ```
   PACKAGE = @PACKAGE@
   VERSION = @VERSION@
   ```

2. If not done already, you should guarantee that `top_srcdir` gets defined. This will serve for `cpp` include files. Just add the line:

   ```
   top_srcdir = @top_srcdir@
   ```

3. You might also want to define `subdir` as 'src', later allowing for almost uniform 'dist:' goals in all your 'Makefile.in'. At list, the 'dist:' goal below assume that you used:

   ```
   subdir = src
   ```

4. You should ensure that the final linking will use `@INTLLIBS@` as a library. An easy way to achieve this is to manage that it gets into `LIBS`, like this:

   ```
   LIBS = @INTLLIBS@ @LIBS@
   ```

   In most GNU packages one will find a directory 'lib/' in which a library containing some helper functions will be build. (You need at least the few functions which the GNU `gettext` Library itself needs.) However some of the functions in the 'lib/' also give messages to the user which of course should be translated, too. Taking care of this it is not enough to place the support library (say 'libsupport.a') just between the `@INTLLIBS@` and `@LIBS@` in the above example. Instead one has to write this:

   ```
   LIBS = ../lib/libsupport.a @INTLLIBS@ ../lib/libsupport.a @LIBS@
   ```

5. You should also ensure that directory 'intl/' will be searched for C preprocessor include files in all circumstances. So, you have to manage so both '-I../intl' and '-I$(top_srcdir)/intl' will be given to the C compiler.

6. Your 'dist:' goal has to conform with others. Here is a reasonable definition for it:

   ```
   distdir = ../$(PACKAGE)-$(VERSION)/$(subdir)
   dist: Makefile $(DISTFILES)
   for file in $(DISTFILES); do \
     ln $$file $(distdir) 2>/dev/null || cp -p $$file $(distdir); \
   done
   ```

11 Concluding Remarks

We would like to conclude this GNU `gettext` manual by presenting an history of the GNU Translation Project so far. We finally give a few pointers for those who want to do further research or readings about Native Language Support matters.

11.1 History of GNU `gettext`

Internationalization concerns and algorithms have been informally and casually discussed for years in GNU, sometimes around GNU `libc`, maybe around the incoming `Hurd`, or otherwise (nobody clearly remembers). And even then, when the work started for real, this was somewhat independently of these previous discussions.

This all began in July 1994, when Patrick D'Cruze had the idea and initiative of internationalizing version 3.9.2 of GNU `fileutils`. He then asked Jim Meyering, the maintainer, how to get those changes folded into an official release. That first draft was full of `#ifdef`s and somewhat disconcerting, and Jim wanted to find nicer ways. Patrick and Jim shared some tries and experimentations in this area. Then, feeling that this might eventually have a deeper impact on GNU, Jim wanted to know what standards were, and contacted Richard Stallman, who very quickly and verbally described an overall design for what was meant to become `glocale`, at that time.

Jim implemented `glocale` and got a lot of exhausting feedback from Patrick and Richard, of course, but also from Mitchum DSouza (who wrote a `catgets`-like package), Roland McGrath, maybe David MacKenzie, Francois Pinard, and Paul Eggert, all pushing and pulling in various directions, not always compatible, to the extent that after a couple of test releases, `glocale` was torn apart.

While Jim took some distance and time and became dad for a second time, Roland wanted to get GNU `libc` internationalized, and got Ulrich Drepper involved in that project. Instead of starting from `glocale`, Ulrich rewrote something from scratch, but more conformant to the set of guidelines who emerged out of the `glocale` effort. Then, Ulrich got people from the previous forum to involve themselves into this new project, and the switch from `glocale` to what was first named `msgutils`, renamed `nlsutils`, and later `gettext`, became officially accepted by Richard in May 1995 or so.

Let's summarize by saying that Ulrich Drepper wrote GNU `gettext` in April 1995. The first official release of the package, including PO mode, occurred in July 1995, and was numbered 0.7. Other people contributed to the effort by providing a discussion forum around Ulrich, writing little pieces of code, or testing. These are quoted in the `THANKS` file which comes with the GNU `gettext` distribution.

While this was being done, Francois adapted half a dozen of GNU packages to `glocale` first, then later to `gettext`, putting them in pretest, so providing along the way an effective user environment for fine tuning the evolving tools. He also took the responsibility of organizing and coordinating the GNU Translation Project. After nearly a year of informal exchanges between people from many countries, translator teams started to exist in May 1995, through the creation and support by Patrick D'Cruze of twenty unmoderated mailing lists for that many native languages, and two moderated lists: one for reaching all teams at once, the other for reaching all maintainers of internationalized packages in GNU.

Francois also wrote PO mode in June 1995 with the collaboration of Greg McGary, as a kind of contribution to Ulrich's package. He also gave a hand with the GNU `gettext` Texinfo manual.

11.2 Related Readings

Eugene H. Dorr ('`dorre@well.com`') maintains an interesting bibliography on internationalization matters, called *Internationalization Reference List*, which is available as:

```
ftp://ftp.ora.com/pub/examples/nutshell/ujip/doc/i18n-books.txt
```

Michael Gschwind ('`mike@vlsivie.tuwien.ac.at`') maintains a Frequently Asked Questions (FAQ) list, entitled *Programming for Internationalisation*. This FAQ discusses writing programs which can handle different language conventions, character sets, etc.; and is applicable to all character set encodings, with particular emphasis on ISO 8859-1. It is regularly published in Usenet groups '`comp.unix.questions`', '`comp.std.internat`', '`comp.software.international`', '`comp.lang.c`', '`comp.windows.x`', '`comp.std.c`', '`comp.answers`' and '`news.answers`'. The home location of this document is:

```
ftp://ftp.vlsivie.tuwien.ac.at/pub/8bit/ISO-programming
```

Patrick D'Cruze ('`pdcruze@li.org`') wrote a tutorial about NLS matters, and Jochen Hein ('`Hein@student.tu-clausthal.de`') took over the responsibility of maintaining it. It may be found as:

```
ftp://sunsite.unc.edu/pub/Linux/utils/nls/catalogs/Incoming/...
     ...locale-tutorial-0.8.txt.gz
```

This site is mirrored in:

```
ftp://ftp.ibp.fr/pub/linux/sunsite/
```

A French version of the same tutorial should be findable at:

```
ftp://ftp.ibp.fr/pub/linux/french/docs/
```

together with French translations of many Linux-related documents.

Table of Contents