Free recode, version 3.5

The character set converter
Edition 3.5, 28 August 2000

François Pinard

1 Quick Tutorial

So, really, you just are in a hurry to use `recode`, and do not feel like studying this manual? Even reading this paragraph slows you down? We might have a problem, as you will have to do some guess work, and might not become very proficient unless you have a very solid intuition....

Let me use here, as a quick tutorial, an actual reply of mine to a `recode` user, who writes:

> My situation is this—I occasionally get email with special characters in it. Sometimes this mail is from a user using IBM software and sometimes it is a user using Mac software. I myself am on a SPARC Solaris machine.

Your situation is similar to mine, except that I *often* receive email needing recoding, that is, much more than *occasionally*! The usual recodings I do are Mac to Latin-1, IBM page codes to Latin-1, Easy-French to Latin-1, remove Quoted-Printable, remove Base64. These are so frequent that I made myself a few two-keystroke Emacs commands to filter the Emacs region. This is very convenient for me. I also resort to many other email conversions, yet more rarely than the frequent cases above.

> It *seems* like this should be doable using `recode`. However, when I try something like '`grecode mac macfile.txt`' I get nothing out—no error, no output, nothing.

Presuming you are using some recent version of `recode`, the command:

```
recode mac macfile.txt
```

is a request for recoding '`macfile.txt`' over itself, overwriting the original, from Macintosh usual character code and Macintosh end of lines, to Latin-1 and Unix end of lines. This is overwrite mode. If you want to use `recode` as a filter, which is probably what you need, rather do:

```
recode mac
```

and give your Macintosh file as standard input, you'll get the Latin-1 file on standard output. The above command is an abbreviation for any of:

```
recode mac..
recode mac..l1
recode mac..Latin-1
recode mac/CR..Latin-1/
recode Macintosh..ISO_8859-1
recode Macintosh/CR..ISO_8859-1/
```

That is, a `CR` surface, encoding newlines with ASCII ⟨CR⟩, is first to be removed (this is a default surface for '`mac`'), then the Macintosh charset is converted to Latin-1 and no surface is added to the result (there is no default surface for '`l1`'). If you want '`mac`' code converted, but you know that newlines are already coded the Unix way, just do:

```
recode mac/
```

the slash then overriding the default surface with empty, that is, none. Here are other easy recipes:

```
recode pc          to filter IBM-PC code and CR-LF (default) to Latin-1
recode pc/         to filter IBM-PC code to Latin-1
```

```
recode 850         to filter code page 850 and CR-LF (default) to Latin-1
recode 850/        to filter code page 850 to Latin-1
recode /qp         to remove quoted printable
```

The last one is indeed equivalent to any of:

```
recode /qp..
recode l1/qp..l1/
recode ISO_8859-1/Quoted-Printable..ISO_8859-1/
```

Here are some reverse recipes:

```
recode ..mac       to filter Latin-1 to Macintosh code and CR (default)
recode ..mac/      to filter Latin-1 to Macintosh code
recode ..pc        to filter Latin-1 to IBM-PC code and CR-LF (default)
recode ..pc/       to filter Latin-1 to IBM-PC code
recode ..850       to filter Latin-1 to code page 850 and CR-LF (default)
recode ..850/      to filter Latin-1 to code page 850
recode ../qp       to force quoted printable
```

In all the above calls, replace '`recode`' by '`recode -f`' if you want to proceed despite recoding errors. If you do not use '`-f`' and there is an error, the recoding output will be interrupted after first error in filter mode, or the file will not be replaced by a recoded copy in overwrite mode.

You may use '`recode -l`' to get a list of available charsets and surfaces, and '`recode --help`' to get a quick summary of options. The above output is meant for those having already read this manual, so let me dare a suggestion: why could not you find a few more minutes in your schedule to peek further down, right into the following chapters!

2 Terminology and purpose

A few terms are used over and over in this manual, our wise reader will learn their meaning right away. Both ISO (International Organization for Standardisation) and IETF (Internet Engineering Task Force) have their own terminology, this document does not try to stick to either one in a strict way, while it does not want to throw more confusion in the field. On the other hand, it would not be efficient using paraphrases all the time, so `recode` coins a few short words, which are explained below.

A *charset*, in the context of `recode`, is a particular association between computer codes on one side, and a repertoire of intended characters on the other side. Codes are usually taken from a set of consecutive small integers, starting at 0. Some characters have a graphical appearance (glyph) or displayable effect, others have special uses like, for example, to control devices or to interact with neighbouring codes to specify them more precisely. So, a *charset* is roughly one of those tables, giving a meaning to each of the codes from the set of allowable values. MIME also uses the term charset with approximately the same meaning. It does *not* exactly corresponds to what ISO calls a *coded character set*, that is, a set of characters with an encoding for them. An coded character set does not necessarily use all available code positions, while a MIME charset usually tries to specify them all. A MIME charset might be the union of a few disjoint coded character sets.

A *surface* is a term used in `recode` only, and is a short for surface transformation of a charset stream. This is any kind of mapping, usually reversible, which associates physical bits in some medium for a stream of characters taken from one or more charsets (usually one). A surface is a kind of varnish added over a charset so it fits in actual bits and bytes. How end of lines are exactly encoded is not really pertinent to the charset, and so, there is surface for end of lines. `Base64` is also a surface, as we may encode any charset in it. Other examples would `DES` enciphering, or `gzip` compression (even if `recode` does not offer them currently): these are ways to give a real life to theoretical charsets. The *trivial* surface consists into putting characters into fixed width little chunks of bits, usually eight such bits per character. But things are not always that simple.

This `recode` library, and the program by that name, have the purpose of converting files between various charsets and surfaces. When this cannot be done in exact ways, as it is often the case, the program may get rid of the offending characters or fall back on approximations. This library recognises or produces around 175 such charsets under 500 names, and handle a dozen surfaces. Since it can convert each charset to almost any other one, many thousands of different conversions are possible.

The `recode` program and library do not usually know how to split and sort out textual and non-textual information which may be mixed in a single input file. For example, there is no surface which currently addresses the problem of how lines are blocked into physical records, when the blocking information is added as binary markers or counters within files. So, `recode` should be given textual streams which are rather *pure*.

This tool pays special attention to superimposition of diacritics for some French representations. This orientation is mostly historical, it does not impair the usefulness, generality or extensibility of the program.

The program `recode` has been written by François Pinard. With time, it got to reuse works from other contributors, and notably, those of Keld Simonsen.

2.1 Overview of charsets

Recoding is currently possible between many charsets, the bulk of which is described by RFC 1345 tables. See Chapter 6 [Tabular], page 36. The `recode` library also handles some charsets in some specialised ways. These are:

- 6-bit charsets based on CDC display code: 6/12 code from NOS; bang-bang code from Université de Montréal;
- 7-bit ASCII: without any diacritics, or else: using backspace for overstriking; Unisys' Icon convention; TeX/LaTeX coding; easy French conventions for electronic mail;
- 8-bit extensions to ASCII: ISO Latin-1, Atari ST code, IBM's code for the PC, Apple's code for the Macintosh;
- 8-bit non-ASCII codes: three flavours of EBCDIC;
- 16-bit or 31-bit universal characters, and their transfer encodings.

The introduction of RFC 1345 in `recode` has brought with it a few charsets having the functionality of older ones, but yet being different in subtle ways. The effects have not been fully investigated yet, so for now, clashes are avoided, the old and new charsets are kept well separate.

2.2 Overview of surfaces

For various practical considerations, it sometimes happens that the codes making up a text, written in a particular charset, cannot simply be put out in a file one after another without creating problems or breaking other things. Sometimes, 8-bit codes cannot be written on a 7-bit medium, variable length codes need kind of envelopes, newlines require special treatment, etc. We sometimes have to apply *surfaces* to a stream of codes, which surfaces are kind of tricks used to fit the charset into those practical constraints. Moreover, similar surfaces or tricks may be useful for many unrelated charsets, and many surfaces can be used at once over a single charset.

So, `recode` has machinery to describe a combination of a charset with surfaces used over it in a file. We would use the expression *pure charset* for referring to a charset free of any surface, that is, the conceptual association between integer codes and character intents.

It is not always clear if some transformation will yield a charset or a surface, especially for those transformations which are only meaningful over a single charset. The `recode` library is not overly picky as identifying surfaces as such: when it is practical to consider a specialised surface as if it were a charset, this is preferred, and done.

2.3 Contributions and bug reports

Even being the `recode` author and current maintainer, I am no specialist in charset standards. I only made `recode` along the years to solve my own needs, but felt it was applicable for the needs of others. Some FSF people liked the program structure and suggested to make it more widely available. I often rely on `recode` users suggestions to decide what is best to be done next.

Properly protecting `recode` about possible copyright fights is a pain for me and for contributors, but we cannot avoid addressing the issue in the long run. Besides, the Free

Software Foundation, which mandates the GNU project, is very sensible to this matter. GNU standards suggest that we stay cautious before looking at copyrighted code. The safest and simplest way for me is to gather ideas and reprogram them anew, even if this might slow me down considerably. For contributions going beyond a few lines of code here and there, the FSF definitely requires employer disclaimers and copyright assignments in writing.

When you contribute something to `recode`, *please* explain what it is about. Do not take for granted that I know those charsets which are familiar to you. Once again, I'm no expert, and you have to help me. Your explanations could well find their way into this documentation, too. Also, for contributing new charsets or new surfaces, as much as possible, please provide good, solid, verifiable references for the tables you used[1].

Many users contributed to `recode` already, I am grateful to them for their interest and involvement. Some suggestions can be integrated quickly while some others have to be delayed, I have to draw a line somewhere when time comes to make a new release, about what would go in it and what would go in the next.

Please send suggestions, documentation errors and bug reports to `recode-bugs@iro.umontreal.ca` or, if you prefer, directly to François Pinard (`pinard@iro.umontreal.ca`). Do not be afraid to report details, because this program is the mere aggregation of hundreds of details.

---

[1] I'm not prone at accepting a charset you just invented, and which nobody uses yet: convince your friends and community first!

3 How to use this program

With the synopsis of the `recode` call, we stress the difference between using this program as a file filter, or recoding many files at once. The first parameter of any call states the recoding request, and this deserves a section on its own. Options are then presented, but somewhat grouped according to the related functionalities they control.

3.1 Synopsis of `recode` call

The general format of the program call is one of:

recode [*option*]... [*charset* | *request* [*file*]...]

Some calls are used only to obtain lists produced by `recode` itself, without actually recoding any file. They are recognised through the usage of listing options, and these options decide what meaning should be given to an optional *charset* parameter. See Section 3.3 [Listings], page 9.

In other calls, the first parameter (*request*) always explains which transformations are expected on the files. There are many variations to the aspect of this parameter. We will discuss more complex situations later (see Section 3.2 [Requests], page 7), but for many simple cases, this parameter merely looks like this[1]:

before..*after*

where *before* and *after* each gives the name of a charset. Each *file* will be read assuming it is coded with charset *before*, it will be recoded over itself so to use the charset *after*. If there is no *file* on the `recode` command, the program rather acts as a Unix filter and transforms standard input onto standard output.

The capability of recoding many files at once is very convenient. For example, one could easily prepare a distribution from Latin-1 to MSDOS, this way:

```
mkdir package
cp -p Makefile *.[ch] package
recode Latin-1..MSDOS package/*
zoo ah package.zoo package/*
rm -rf package
```

(In this example, the non-mandatory '`-p`' option to `cp` is for preserving timestamps, and the `zoo` program is an archiver from Rahul Dhesi which once was quite popular.)

The filter operation is especially useful when the input files should not be altered. Let us make an example to illustrate this point. Suppose that someone has a file named '`datum.txt`', which is almost a TeX file, except that diacriticised characters are written using Latin-1. To complete the recoding of the diacriticised characters *only* and produce a file '`datum.tex`', without destroying the original, one could do:

[1] In previous versions or `recode`, a single colon '`:`' was used instead of the two dots '`..`' for separating charsets, but this was creating problems because colons are allowed in official charset names. The old request syntax is still recognised for compatibility purposes, but is deprecated.

```
cp -p datum.txt datum.tex
recode -d l1..tex datum.tex
```

However, using `recode` as a filter will achieve the same goal more neatly:

```
recode -d l1..tex <datum.txt >datum.tex
```

This example also shows that `l1` could be used instead of `Latin-1`; charset names often have such aliases.

3.2 The *request* parameter

In the case where the *request* is merely written as *before*..*after*, then *before* and *after* specify the start charset and the goal charset for the recoding.

For `recode`, charset names may contain any character, besides a comma, a forward slash, or two periods in a row. But in practice, charset names are currently limited to alphabetic letters (upper or lower case), digits, hyphens, underlines, periods, colons or round parentheses.

The complete syntax for a valid *request* allows for unusual things, which might surprise at first. (Do not pay too much attention to these facilities on first reading.) For example, *request* may also contain intermediate charsets, like in the following example:

before`..`*interim1*`..`*interim2*`..`*after*

meaning that `recode` should internally produce the *interim1* charset from the start charset, then work out of this *interim1* charset to internally produce *interim2*, and from there towards the goal charset. In fact, `recode` internally combines recipes and automatically uses interim charsets, when there is no direct recipe for transforming *before* into *after*. But there might be many ways to do it. When many routes are possible, the above *chaining* syntax may be used to more precisely force the program towards a particular route, which it might not have naturally selected otherwise. On the other hand, because `recode` tries to choose good routes, chaining is only needed to achieve some rare, unusual effects.

Moreover, many such requests (sub-requests, more precisely) may be separated with commas (but no spaces at all), indicating a sequence of recodings, where the output of one has to serve as the input of the following one. For example, the two following requests are equivalent:

before`..`*interim1*`..`*interim2*`..`*after*
before`..`*interim1*`,`*interim1*`..`*interim2*`,`*interim2*`..`*after*

In this example, the charset input for any recoding sub-request is identical to the charset output by the preceding sub-request. But it does not have to be so in the general case. One might wonder what would be the meaning of declaring the charset input for a recoding sub-request of being of different nature than the charset output by a preceding sub-request, when recodings are chained in this way. Such a strange usage might have a meaning and be useful for the `recode` expert, but they are quite uncommon in practice.

More useful is the distinction between the concept of charset, and the concept of surfaces. An encoded charset is represented by:

pure-charset`/`*surface1*`/`*surface2*...

using slashes to introduce surfaces, if any. The order of application of surfaces is usually important, they cannot be freely commuted. In the given example, *surface1* is first applied over the *pure-charset*, then *surface2* is applied over the result. Given this request:

> *before*/*surface1*/*surface2*..*after*/*surface3*

the `recode` program will understand that the input files should have *surface2* removed first (because it was applied last), then *surface1* should be removed. The next step will be to translate the codes from charset *before* to charset *after*, prior to applying *surface3* over the result.

Some charsets have one or more *implied* surfaces. In this case, the implied surfaces are automatically handled merely by naming the charset, without any explicit surface to qualify it. Let's take an example to illustrate this feature. The request '`pc..l1`' will indeed decode MS-DOS end of lines prior to converting IBM-PC codes to Latin-1, because '`pc`' is the name of a charset[2] which has `CR-LF` for its usual surface. The request '`pc/..l1`' will *not* decode end of lines, since the slash introduces surfaces, and even if the surface list is empty, it effectively defeats the automatic removal of surfaces for this charset. So, empty surfaces are useful, indeed!

Both charsets and surfaces may have predefined alternate names, or aliases. However, and this is rather important to understand, implied surfaces are attached to individual aliases rather than on genuine charsets. Consequently, the official charset name and all of its aliases do not necessarily share the same implied surfaces. The charset and all its aliases may each have its own different set of implied surfaces.

Charset names, surface names, or their aliases may always be abbreviated to any unambiguous prefix. Internally in `recode`, disambiguating tables are kept separate for charset names and surface names.

While recognising a charset name or a surface name (or aliases thereof), `recode` ignores all characters besides letters and digits, so for example, the hyphens and underlines being part of an official charset name may safely be omitted (no need to un-confuse them!). There is also no distinction between upper and lower case for charset or surface names.

One of the *before* or *after* keywords may be omitted. If the double dot separator is omitted too, then the charset is interpreted as the *before* charset.[3]

When a charset name is omitted or left empty, the value of the `DEFAULT_CHARSET` variable in the environment is used instead. If this variable is not defined, the `recode` library uses the *usual charset* which is installation dependent. When this variable is defined but has an empty value, charset names may not be omitted. Unless overriden when `recode` was installed, the usual charset is `Latin-1` for Unix systems or `IBM-PC` for MS-DOS/MS-Windows machines.

If the charset name is omitted but followed by surfaces, the surfaces then qualify the usual or default charset. For example, the request '`../x`' is sufficient for applying an hexadecimal surface to the input text[4].

---

[2] More precisely, `pc` is an alias for the charset `IBM-PC`.

[3] Both *before* and *after* may be omitted, in which case the double dot separator is mandatory. This is not very useful, as the recoding reduces to a mere copy in that case.

[4] MS-DOS is one of those systems for which the default charset has implied surfaces, `CR-LF` here. Such surfaces are automatically removed or applied whenever the default charset

The allowable values for *before* or *after* charsets, and various surfaces, are described in the remainder of this document.

3.3 Asking for various lists

Many options control listing output generated by `recode` itself, they are not meant to accompany actual file recodings. These options are:

'`--version`'
: The program merely prints its version numbers on standard output, and exits without doing anything else.

'`--help`'
: The program merely prints a page of help on standard output, and exits without doing any recoding.

'`-C`'
'`--copyright`'
: Given this option, all other parameters and options are ignored. The program prints briefly the copyright and copying conditions. See the file '`COPYING`' in the distribution for full statement of the Copyright and copying conditions.

'`-h[`*language*`/][`*name*`]`'
'`--header[=[`*language*`/][`*name*`]]`'
: Instead of recoding files, `recode` writes a *language* source file on standard output and exits. This source is meant to be included in a regular program written in the same programming *language*: its purpose is to declare and initialise an array, named *name*, which represents the requested recoding. The only acceptable values for *language* are '`c`' or '`perl`', and may may be abbreviated. If *language* is not specified, '`c`' is assumed. If *name* is not specified, then it defaults to '*before_after*'. Strings *before* and *after* are cleaned before being used according to the syntax of *language*.

  Even if `recode` tries its best, this option does not always succeed in producing the requested source table. It will however, provided the recoding can be internally represented by only one step after the optimisation phase, and if this merged step conveys a one-to-one or a one-to-many explicit table. Also, when attempting to produce sources tables, `recode` relaxes its checking a tiny bit: it ignores the algorithmic part of some tabular recodings, it also avoids the processing of implied surfaces. But this is all fairly technical. Better try and see!

  Beware that other options might affect the produced source tables, these are: '`-d`', '`-g`' and, particularly, '`-s`'.

---

is read or written, exactly as it would go for any other charset. In the example above, on such systems, the hexadecimal surface would then *replace* the implied surfaces. For *adding* an hexadecimal surface without removing any, one should write the request as '`/../x`'.

'-k *pairs*'
'--known=*pairs*'

This particular option is meant to help identifying an unknown charset, using as hints some already identified characters of the charset. Some examples will help introducing the idea.

Let's presume here that `recode` is run on Unix, where Latin-1 is the usual charset, and that `DEFAULT_CHARSET` is unset in the environment. Suppose you have guessed that code 130 (decimal) of the unknown charset represents a lower case '`e`' with an acute accent. That is to say that this code should map to code 233 (decimal) in the usual charset. By executing:

```
recode -k 130:233
```

you should obtain a listing similar to:

```
AtariST atarist
CWI cphu cwi cwi2
IBM437 437 cp437 ibm437
IBM850 850 cp850 ibm850
IBM851 851 cp851 ibm851
IBM852 852 cp852 ibm852
IBM857 857 cp857 ibm857
IBM860 860 cp860 ibm860
IBM861 861 cp861 cpis ibm861
IBM863 863 cp863 ibm863
IBM865 865 cp865 ibm865
```

You can give more than one clue at once, to restrict the list further. Suppose you have *also* guessed that code 211 of the unknown charset represents an upper case '`E`' with diaeresis, that is, code 203 in the usual charset. By requesting:

```
recode -k 130:233,211:203
```

you should obtain:

```
IBM850 850 cp850 ibm850
IBM852 852 cp852 ibm852
IBM857 857 cp857 ibm857
```

The usual charset may be overridden by specifying one non-option argument. For example, to request the list of charsets for which code 130 maps to code 142 for the Macintosh, you may ask:

```
recode -k 130:142 mac
```

and get:

```
AtariST atarist
CWI cphu cwi cwi2
IBM437 437 cp437 ibm437
IBM850 850 cp850 ibm850
IBM851 851 cp851 ibm851
IBM852 852 cp852 ibm852
IBM857 857 cp857 ibm857
IBM860 860 cp860 ibm860
IBM861 861 cp861 cpis ibm861
```

```
IBM863 863 cp863 ibm863
IBM865 865 cp865 ibm865
```

which, of course, is identical to the result of the first example, since the code 142 for the Macintosh is a small '`e`' with acute.

More formally, option '`-k`' lists all possible *before* charsets for the *after* charset given as the sole non-option argument to `recode`, but subject to restrictions given in *pairs*. If there is no non-option argument, the *after* charset is taken to be the default charset for this `recode`.

The restrictions are given as a comma separated list of pairs, each pair consisting of two numbers separated by a colon. The numbers are taken as decimal when the initial digit is between '`1`' and '`9`'; '`0x`' starts an hexadecimal number, or else '`0`' starts an octal number. The first number is a code in any *before* charset, while the second number is a code in the specified *after* charset. If the first number would not be transformed into the second number by recoding from some *before* charset to the *after* charset, then this *before* charset is rejected. A *before* charset is listed only if it is not rejected by any pair. The program will only test those *before* charsets having a tabular style internal description (see Chapter 6 [Tabular], page 36), so should be the selected *after* charset.

The produced list is in fact a subset of the list produced by the option '`-l`'. As for option '`-l`', the non-option argument is interpreted as a charset name, possibly abbreviated to any non ambiguous prefix.

'`-l[`*format*`]`'
'`--list[=`*format*`]`'

This option asks for information about all charsets, or about one particular charset. No file will be recoded.

If there is no non-option arguments, `recode` ignores the *format* value of the option, it writes a sorted list of charset names on standard output, one per line. When a charset name have aliases or synonyms, they follow the true charset name on its line, sorted from left to right. Each charset or alias is followed by its implied surfaces, if any. This list is over two hundred lines. It is best used with '`grep -i`', as in:

```
recode -l | grep -i greek
```

There might be one non-option argument, in which case it is interpreted as a charset name, possibly abbreviated to any non ambiguous prefix. This particular usage of the '`-l`' option is obeyed *only* for charsets having a tabular style internal description (see Chapter 6 [Tabular], page 36). Even if most charsets have this property, some do not, and the option '`-l`' cannot be used to detail these particular charsets. For knowing if a particular charset can be listed this way, you should merely try and see if this works. The *format* value of the option is a keyword from the following list. Keywords may be abbreviated by dropping suffix letters, and even reduced to the first letter only:

'`decimal`' This format asks for the production on standard output of a concise tabular display of the charset, in which character code values are expressed in decimal.

'octal' This format uses octal instead of decimal in the concise tabular display of the charset.

'hexadecimal'
This format uses hexadecimal instead of decimal in the concise tabular display of the charset.

'full' This format requests an extensive display of the charset on standard output, using one line per character showing its decimal, hexadecimal, octal and `UCS-2` code values, and also a descriptive comment which should be the 10646 name for the character.

The descriptive comment is given in English and ASCII, yet if the English description is not available but a French one is, then the French description is given instead, using Latin-1. However, if the `LANGUAGE` or `LANG` environment variable begins with the letters '`fr`', then listing preference goes to French when both descriptions are available.

When option '`-l`' is used together with a *charset* argument, the *format* defaults to `decimal`.

'-T'
'--find-subsets'
This option is a maintainer tool for evaluating the redundancy of those charsets, in `recode`, which are internally represented by an `UCS-2` data table. After the listing has been produced, the program exits without doing any recoding. The output is meant to be sorted, like this: '`recode -T | sort`'. The option triggers `recode` into comparing all pairs of charsets, seeking those which are subsets of others. The concept and results are better explained through a few examples. Consider these three sample lines from '`-T`' output:

```
[  0] IBM891 == IBM903
[  1] IBM1004 < CP1252
[ 12] INVARIANT < CSA_Z243.4-1985-1
```

The first line means that `IBM891` and `IBM903` are completely identical as far as `recode` is concerned, so one is fully redundant to the other. The second line says that `IBM1004` is wholly contained within `CP1252`, yet there is a single character which is in `CP1252` without being in `IBM1004`. The third line says that `INVARIANT` is wholly contained within `CSA_Z243.4-1985-1`, but twelve characters are in `CSA_Z243.4-1985-1` without being in `INVARIANT`. The whole output might most probably be reduced and made more significant through a transitivity study.

3.4 Controlling how files are recoded

The following options have the purpose of giving the user some fine grain control over the recoding operation themselves.

'-c'
'--colons'
: With Texte Easy French conventions, use the column *:* instead of the double-quote " for marking diaeresis. See Section 11.6 [Texte], page 58.

'-g'
'--graphics'
: This option is only meaningful while getting *out* of the IBM-PC charset. In this charset, characters 176 to 223 are used for constructing rulers and boxes, using simple or double horizontal or vertical lines. This option forces the automatic selection of ASCII characters for approximating these rulers and boxes, at cost of making the transformation irreversible. Option '-g' implies '-f'.

'-t'
'--touch'
: The *touch* option is meaningful only when files are recoded over themselves. Without it, the time-stamps associated with files are preserved, to reflect the fact that changing the code of a file does not really alter its informational contents. When the user wants the recoded files to be time-stamped at the recoding time, this option inhibits the automatic protection of the time-stamps.

'-v'
'--verbose'
: Before doing any recoding, the program will first print on the stderr stream the list of all intermediate charsets planned for recoding, starting with the *before* charset and ending with the *after* charset. It also prints an indication of the recoding quality, as one of the word 'reversible', 'one to one', 'one to many', 'many to one' or 'many to many'.

This information will appear once or twice. It is shown a second time only when the optimisation and step merging phase succeeds in replacing many single steps by a new one.

This option also has a second effect. The program will print on stderr one message per recoded *file*, so as to keep the user informed of the progress of its command.

An easy way to know beforehand the sequence or quality of a recoding is by using the command such as:

```
recode -v before..after < /dev/null
```

using the fact that, in recode, an empty input file produces an empty output file.

'-x *charset*'
'--ignore=*charset*'
: This option tells the program to ignore any recoding path through the specified *charset*, so disabling any single step using this charset as a start or end point. This may be used when the user wants to force recode into using an alternate recoding path (yet using chained requests offers a finer control, see Section 3.2 [Requests], page 7).

charset may be abbreviated to any unambiguous prefix.

3.5 Reversibility issues

The following options are somewhat related to reversibility issues:

'-f'
'--force'
: With this option, irreversible or otherwise erroneous recodings are run to completion, and `recode` does not exit with a non-zero status if it would be only because irreversibility matters. See Section 3.5 [Reversibility], page 14.

  Without this option, `recode` tries to protect you against recoding a file irreversibly over itself[5]. Whenever an irreversible recoding is met, or any other recoding error, `recode` produces a warning on standard error. The current input file does not get replaced by its recoded version, and `recode` then proceeds with the recoding of the next file.

  When the program is merely used as a filter, standard output will have received a partially recoded copy of standard input, up to the first error point. After all recodings have been done or attempted, and if some recoding has been aborted, `recode` exits with a non-zero status.

  In releases of `recode` prior to version 3.5, this option was always selected, so it was rather meaningless. Nevertheless, users were invited to start using '-f' right away in scripts calling `recode` whenever convenient, in preparation for the current behaviour.

'-q'
'--quiet'
'--silent'
: This option has the sole purpose of inhibiting warning messages about irreversible recodings, and other such diagnostics. It has no other effect, in particular, it does *not* prevent recodings to be aborted or `recode` to return a non-zero exit status when irreversible recodings are met.

  This option is set automatically for the children processes, when recode splits itself in many collaborating copies. Doing so, the diagnostic is issued only once by the parent. See option '-p'.

'-s'
'--strict'
: By using this option, the user requests that `recode` be very strict while recoding a file, merely losing in the transformation any character which is not explicitly mapped from a charset to another. Such a loss is not reversible and so, will bring `recode` to fail, unless the option '-f' is also given as a kind of counter-measure.

  Using '-s' without '-f' might render the `recode` program very susceptible to the slighest file abnormalities. Despite the fact that it might be irritating to some users, such paranoia is sometimes wanted and useful.

Even if `recode` tries hard to keep the recodings reversible, you should not develop an unconditional confidence in its ability to do so. You *ought* to keep only reasonable expectations about reverse recodings. In particular, consider:

---

[5] There are still some cases of ambiguous output which are rather difficult to detect, and for which the protection is not active.

- Most transformations are fully reversible for all inputs, but lose this property whenever '`-s`' is specified.
- A few transformations are not meant to be reversible, by design.
- Reversibility sometimes depends on actual file contents and cannot be ascertained beforehand, without reading the file.
- Reversibility is never absolute across successive versions of this program. Even correcting a small bug in a mapping could induce slight discrepancies later.
- Reversibility is easily lost by merging. This is best explained through an example. If you reversibly recode a file from charset *A* to charset *B*, then you reversibly recode the result from charset *B* to charset *C*, you cannot expect to recover the original file by merely recoding from charset *C* directly to charset *A*. You will instead have to recode from charset *C* back to charset *B*, and only then from charset *B* to charset *A*.
- Faulty files create a particular problem. Consider an example, recoding from `IBM-PC` to `Latin-1`. End of lines are represented as '`\r\n`' in `IBM-PC` and as '`\n`' in `Latin-1`. There is no way by which a faulty `IBM-PC` file containing a '`\n`' not preceded by '`\r`' be translated into a `Latin-1` file, and then back.
- There is another difficulty arising from code equivalences. For example, in a `LaTeX` charset file, the string '`\^\i{}`' could be recoded back and forth through another charset and become '`\^{\i}`'. Even if the resulting file is equivalent to the original one, it is not identical.

Unless option '`-s`' is used, `recode` automatically tries to fill mappings with invented correspondences, often making them fully reversible. This filling is not made at random. The algorithm tries to stick to the identity mapping and, when this is not possible, it prefers generating many small permutation cycles, each involving only a few codes.

For example, here is how `IBM-PC` code 186 gets translated to *`control-U`* in `Latin-1`. *`Control-U`* is 21. Code 21 is the `IBM-PC` section sign, which is 167 in `Latin-1`. `recode` cannot reciprocate 167 to 21, because 167 is the masculine ordinal indicator within `IBM-PC`, which is 186 in `Latin-1`. Code 186 within `IBM-PC` has no `Latin-1` equivalent; by assigning it back to 21, `recode` closes this short permutation loop.

As a consequence of this map filling, `recode` may sometimes produce *funny* characters. They may look annoying, they are nevertheless helpful when one changes his (her) mind and wants to revert to the prior recoding. If you cannot stand these, use option '`-s`', which asks for a very strict recoding.

This map filling sometimes has a few surprising consequences, which some users wrongly interpreted as bugs. Here are two examples.

1. In some cases, `recode` seems to copy a file without recoding it. But in fact, it does. Consider a request:

   ```
   recode l1..us < File-Latin1 > File-ASCII
   cmp File-Latin1 File-ASCII
   ```

   then `cmp` will not report any difference. This is quite normal. `Latin-1` gets correctly recoded to ASCII for charsets commonalities (which are the first 128 characters, in this case). The remaining last 128 `Latin-1` characters have no ASCII correspondent. Instead of losing them, `recode` elects to map them to unspecified characters of ASCII, so making the recoding reversible. The simplest way of achieving this is merely to keep those last 128 characters unchanged. The overall effect is copying the file verbatim.

If you feel this behaviour is too generous and if you do not wish to care about reversibility, simply use option '`-s`'. By doing so, `recode` will strictly map only those `Latin-1` characters which have an ASCII equivalent, and will merely drop those which do not. Then, there is more chance that you will observe a difference between the input and the output file.

2. Recoding the wrong way could sometimes give the false impression that recoding has *almost* been done properly. Consider the requests:

```
recode 437..l1 < File-Latin1 > Temp1
recode 437..l1 < Temp1 > Temp2
```

so declaring wrongly '`File-Latin1`' to be an IBM-PC file, and recoding to `Latin-1`. This is surely ill defined and not meaningful. Yet, if you repeat this step a second time, you might notice that many (not all) characters in '`Temp2`' are identical to those in '`File-Latin1`'. Sometimes, people try to discover how `recode` works by experimenting a little at random, rather than reading and understanding the documentation; results such as this are surely confusing, as they provide those people with a false feeling that they understood something.

Reversible codings have this property that, if applied several times in the same direction, they will eventually bring any character back to its original value. Since `recode` seeks small permutation cycles when creating reversible codings, besides characters unchanged by the recoding, most permutation cycles will be of length 2, and fewer of length 3, etc. So, it is just expectable that applying the recoding twice in the same direction will recover most characters, but will fail to recover those participating in permutation cycles of length 3. On the other end, recoding six times in the same direction would recover all characters in cycles of length 1, 2, 3 or 6.

3.6 Selecting sequencing methods

This program uses a few techniques when it is discovered that many passes are needed to comply with the *request*. For example, suppose that four elementary steps were selected at recoding path optimisation time. Then `recode` will split itself into four different interconnected tasks, logically equivalent to:

step1 `<`*input* `|` *step2* `|` *step3* `|` *step4* `>`*output*

The splitting into subtasks is often done using Unix pipes. But the splitting may also be completely avoided, and rather simulated by using memory buffer, or intermediate files. The various '`--sequence=`*strategy*' options gives you control over the flow methods, by replacing *strategy* with '`memory`', '`pipe`' or '`files`'. So, these options may be used to override the default behaviour, which is also explained below.

'`--sequence=memory`'
: When the recoding requires a combination of two or more elementary recoding steps, this option forces many passes over the data, using in-memory buffers to hold all intermediary results.

'-i'
'--sequence=files'

When the recoding requires a combination of two or more elementary recoding steps, this option forces many passes over the data, using intermediate files between passes. This is the default behaviour when files are recoded over themselves. If this option is selected in filter mode, that is, when the program reads standard input and writes standard output, it might take longer for programs further down the pipe chain to start receiving some recoded data.

'-p'
'--sequence=pipe'

When the recoding requires a combination of two or more elementary recoding steps, this option forces the program to fork itself into a few copies interconnected with pipes, using the `pipe(2)` system call. All copies of the program operate in parallel. This is the default behaviour in filter mode. If this option is used when files are recoded over themselves, this should also save disk space because some temporary files might not be needed, at the cost of more system overhead.

If, at installation time, the `pipe(2)` call is said to be unavailable, selecting option '-p' is equivalent to selecting option '-i'. (This happens, for example, on MS-DOS systems.)

3.7 Using mixed charset input

In real life and practice, textual files are often made up of many charsets at once. Some parts of the file encode one charset, while other parts encode another charset, and so forth. Usually, a file does not toggle between more than two or three charsets. The means to distinguish which charsets are encoded at various places is not always available. The `recode` program is able to handle only a few simple cases of mixed input.

The default `recode` behaviour is to expect pure charset files, to be recoded as other pure charset files. However, the following options allow for a few precise kinds of mixed charset files.

'-d'
'--diacritics'

While converting to or from one of `HTML` or `LaTeX` charset, limit conversion to some subset of all characters. For `HTML`, limit conversion to the subset of all non-ASCII characters. For `LaTeX`, limit conversion to the subset of all non-English letters. This is particularly useful, for example, when people create what would be valid `HTML`, TeX or LaTeX files, if only they were using provided sequences for applying diacritics instead of using the diacriticised characters directly from the underlying character set.

While converting to `HTML` or `LaTeX` charset, this option assumes that characters not in the said subset are properly coded or protected already, `recode` then transmit them literally. While converting the other way, this option prevents translating back coded or protected versions of characters not in the said subset. See Section 11.1 [HTML], page 56. See Section 11.2 [LaTeX], page 57.

'-S[*language*]'
'--source[=*language*]'

The bulk of the input file is expected to be written in ASCII, except for parts, like comments and string constants, which are written using another charset than ASCII. When *language* is 'c', the recoding will proceed only with the contents of comments or strings, while everything else will be copied without recoding. When *language* is 'po', the recoding will proceed only within translator comments (those having whitespace immediately following the initial '#') and with the contents of msgstr strings.

For the above things to work, the non-ASCII encoding of the comment or string should be such that an ASCII scan will successfully find where the comment or string ends.

Even if ASCII is the usual charset for writing programs, some compilers are able to directly read other charsets, like UTF-8, say. There is currently no provision in recode for reading mixed charset sources which are not based on ASCII. It is probable that the need for mixed recoding is not as pressing in such cases.

For example, after one does:

```
recode -Spo pc/..u8 < input.po > output.po
```

file '*output*.po' holds a copy of '*input*.po' in which *only* translator comments and the contents of msgstr strings have been recoded from the IBM-PC charset to pure UTF-8, without attempting conversion of end-of-lines. Machine generated comments and original msgid strings are not to be touched by this recoding.

If *language* is not specified, 'c' is assumed.

3.8 Using recode within Emacs

The fact recode is a filter makes it quite easy to use from within GNU Emacs. For example, recoding the whole buffer from the IBM-PC charset to current charset (Latin-1 on Unix) is easily done with:

```
C-x h C-u M-| recode ibmpc RET
```

'C-x h' selects the whole buffer, and 'C-u M-|' filters and replaces the current region through the given shell command. Here is another example, binding the keys 'C-c T' to the recoding of the current region from Easy French to Latin-1 (on Unix) and the key 'C-u C-c T' from Latin-1 (on Unix) to Easy French:

```
(global-set-key "\C-cT" 'recode-texte)

(defun recode-texte (flag)
  (interactive "P")
  (shell-command-on-region
   (region-beginning) (region-end)
   (concat "recode " (if flag "..txte" "txte")) t)
  (exchange-point-and-mark))
```

4 A recoding library

The program named `recode` is just an application of its recoding library. The recoding library is available separately for other C programs. A good way to acquire some familiarity with the recoding library is to get acquainted with the `recode` program itself.

To use the recoding library once it is installed, a C program needs to have a line:

```
#include <recode.h>
```

near its beginning, and the user should have '`-lrecode`' on the linking call, so modules from the recoding library are found.

The library is still under development. As it stands, it contains four identifiable sets of routines: the outer level functions, the request level functions, the task level functions and the charset level functions. There are discussed in separate sections.

For effectively using the recoding library in most applications, it should be rarely needed to study anything beyond the main initialisation function at outer level, and then, various functions at request level.

4.1 Outer level functions

The outer level functions mainly prepare the whole recoding library for use, or do actions which are unrelated to specific recodings. Here is an example of a program which does not really make anything useful.

```
#include <stdbool.h>
#include <recode.h>

int
main ()
{
  RECODE_OUTER outer = recode_new_outer (true);

  recode_delete_outer (outer);
  exit (0);
}
```

The header file `<recode.h>` declares an opaque `RECODE_OUTER` structure, which the programmer should use for allocating a variable in his program (let's assume the programmer is a male, here, no prejudice intended). This '`outer`' variable is given as a first argument to all outer level functions.

The `<recode.h>` header file uses the Boolean type setup by the system header file `<stdbool.h>`. But this header file is still fairly new in C standards, and likely does not exist everywhere. If you system does not offer this system header file yet, the proper compilation of the `<recode.h>` file could be guaranteed through the replacement of the inclusion line by:

```
typedef enum {false = 0, true = 1} bool;
```

People wanting wider portability, or Autoconf lovers, might arrange their '`configure.in`' for being able to write something more general, like:

```
#if STDC_HEADERS
# include <stdlib.h>
#endif

/* Some systems do not define EXIT_*, even with STDC_HEADERS.  */
#ifndef EXIT_SUCCESS
# define EXIT_SUCCESS 0
#endif
#ifndef EXIT_FAILURE
# define EXIT_FAILURE 1
#endif
/* The following test is to work around the gross typo in systems like Sony
   NEWS-OS Release 4.0C, whereby EXIT_FAILURE is defined to 0, not 1.  */
#if !EXIT_FAILURE
# undef EXIT_FAILURE
# define EXIT_FAILURE 1
#endif

#if HAVE_STDBOOL_H
# include <stdbool.h>
#else
typedef enum {false = 0, true = 1} bool;
#endif

#include <recode.h>

int
main ()
{
  RECODE_OUTER outer = recode_new_outer (true);

  recode_term_outer (outer);
  exit (EXIT_SUCCESS);
}
```

but we will not insist on such details in the examples to come.

- Initialisation functions

  ```
  RECODE_OUTER recode_new_outer (auto_abort);
  bool recode_delete_outer (outer);
  ```

  The recoding library absolutely needs to be initialised before being used, and `recode_new_outer` has to be called once, first. Besides the *outer* it is meant to initialise, the function accepts a Boolean argument whether or not the library should automatically issue diagnostics on standard and abort the whole program on errors. When *auto_abort* is `true`, the library later conveniently issues diagnostics itself, and aborts the calling program on errors. This is merely a convenience, because if this parameter was `false`, the calling program should always take care of checking the return value of all other calls to the recoding library functions, and when any error is detected, issue a diagnostic and abort processing itself.

Regardless of the setting of *auto_abort*, all recoding library functions return a success status. Most functions are geared for returning `false` for an error, and `true` if everything went fine. Functions returning structures or strings return `NULL` instead of the result, when the result cannot be produced. If *auto_abort* is selected, functions either return `true`, or do not return at all.

As in the example above, `recode_new_outer` is called only once in most cases. Calling `recode_new_outer` implies some overhead, so calling it more than once should preferably be avoided.

The termination function `recode_delete_outer` reclaims the memory allocated by `recode_new_outer` for a given *outer* variable. Calling `recode_delete_outer` prior to program termination is more aesthetic then useful, as all memory resources are automatically reclaimed when the program ends. You may spare this terminating call if you prefer.

4.2 Request level functions

The request level functions are meant to cover most recoding needs programmers may have; they should provide all usual functionality. Their API is almost stable by now. To get started with request level functions, here is a full example of a program which sole job is to filter `ibmpc` code on its standard input into `latin1` code on its standard output.

```
#include <stdio.h>
#include <stdbool.h>
#include <recode.h>

int
main ()
{
  RECODE_OUTER outer = recode_new_outer (true);
  RECODE_REQUEST request = recode_new_request (outer);
  bool success;

  recode_scan_request (request, "ibmpc..latin1");

  success = recode_file_to_file (request, stdin, stdout);

  recode_delete_request (request);
  recode_delete_outer (outer);

  exit (success ? 0 : 1);
}
```

The header file `<recode.h>` declares a `RECODE_REQUEST` structure, which the programmer should use for allocating a variable in his program. This *request* variable is given as a first argument to all request level functions, and in most cases, may be considered as opaque.

- Initialisation functions

```
RECODE_REQUEST recode_new_request (outer);
bool recode_delete_request (request);
```

No *request* variable may not be used in other request level functions of the recoding library before having been initialised by `recode_new_request`. There may be many such *request* variables, in which case, they are independent of one another and they all need to be initialised separately. To avoid memory leaks, a *request* variable should not be initialised a second time without calling `recode_delete_request` to "un-initialise" it.

Like for `recode_delete_outer`, calling `recode_delete_request` prior to program termination, in the example above, may be left out.

- Fields of `struct recode_request`

  Here are the fields of a `struct recode_request` which may be meaningfully changed, once a *request* has been initialised by `recode_new_request`, but before it gets used. It is not very frequent, in practice, that these fields need to be changed. To access the fields, you need to include '`recodext.h`' *instead* of '`recode.h`', in which case there also is a greater chance that you need to recompile your programs if a new version of the recoding library gets installed.

  `verbose_flag`
  : This field is initially `false`. When set to `true`, the library will echo to stderr the sequence of elementary recoding steps needed to achieve the requested recoding.

  `diaeresis_char`
  : This field is initially the ASCII value of a double quote `"`, but it may also be the ASCII value of a colon `:`. In `texte` charset, some countries use double quotes to mark diaeresis, while other countries prefer colons. This field contains the diaeresis character for the `texte` charset.

  `make_header_flag`
  : This field is initially `false`. When set to `true`, it indicates that the program is merely trying to produce a recoding table in source form rather than completing any actual recoding. In such a case, the optimisation of step sequence can be attempted much more aggressively. If the step sequence cannot be reduced to a single step, table production will fail.

  `diacritics_only`
  : This field is initially `false`. For `HTML` and `LaTeX` charset, it is often convenient to recode the diacriticized characters only, while just not recoding other HTML code using ampersands or angular brackets, or LaTeX code using backslashes. Set the field to `true` for getting this behaviour. In the other charset, one can edit text as well as HTML or LaTeX directives.

  `ascii_graphics`
  : This field is initially `false`, and relate to characters 176 to 223 in the `ibmpc` charset, which are use to draw boxes. When set to `true`, while getting out of `ibmpc`, ASCII characters are selected so to graphically approximate these boxes.

- Study of request strings

  ```
  bool recode_scan_request (request, "string");
  ```

The main role of a *request* variable is to describe a set of recoding transformations. Function `recode_scan_request` studies the given *string*, and stores an internal representation of it into *request*. Note that *string* may be a full-fledged `recode` request, possibly including surfaces specifications, intermediary charsets, sequences, aliases or abbreviations (see Section 3.2 [Requests], page 7).

The internal representation automatically receives some pre-conditioning and optimisation, so the *request* may then later be used many times to achieve many actual recodings. It would not be efficient calling `recode_scan_request` many times with the same *string*, it is better having many *request* variables instead.

- Actual recoding jobs

  Once the *request* variable holds the description of a recoding transformation, a few functions use it for achieving an actual recoding. Either input or output of a recoding may be string, an in-memory buffer, or a file.

  Functions with names like `recode_`*input-type*`_to_`*output-type* request an actual recoding, and are described below. It is easy to remember which arguments each function accepts, once grasped some simple principles for each possible *type*. However, one of the recoding function escapes these principles and is discussed separately, first.

  ```
  recode_string (request, string);
  ```

  The function `recode_string` recodes *string* according to *request*, and directly returns the resulting recoded string freshly allocated, or `NULL` if the recoding could not succeed for some reason. When this function is used, it is the responsibility of the programmer to ensure that the memory used by the returned string is later reclaimed.

  ```
  char *recode_string_to_buffer (request,
    input_string,
    &output_buffer, &output_length, &output_allocated);
  bool recode_string_to_file (request,
    input_file,
    output_file);
  bool recode_buffer_to_buffer (request,
    input_buffer, input_length,
    &output_buffer, &output_length, &output_allocated);
  bool recode_buffer_to_file (request,
    input_buffer, input_length,
    output_file);
  bool recode_file_to_buffer (request,
    input_file,
    &output_buffer, &output_length, &output_allocated);
  bool recode_file_to_file (request,
    input_file,
    output_file);
  ```

  All these functions return a `bool` result, `false` meaning that the recoding was not successful, often because of reversibility issues. The name of the function well indicates on which types it reads and which type it produces. Let's discuss these three types in turn.

  string

A string is merely an in-memory buffer which is terminated by a `NUL` character (using as many bytes as needed), instead of being described by a byte length. For input, a pointer to the buffer is given through one argument.

It is notable that there is no `to_string` functions. Only one function recodes into a string, and it is `recode_string`, which has already been discussed separately, above.

buffer

A buffer is a sequence of bytes held in computer memory. For input, two arguments provide a pointer to the start of the buffer and its byte size. Note that for charsets using many bytes per character, the size is given in bytes, not in characters.

For output, three arguments provide the address of three variables, which will receive the buffer pointer, the used buffer size in bytes, and the allocated buffer size in bytes. If at the time of the call, the buffer pointer is `NULL`, then the allocated buffer size should also be zero, and the buffer will be allocated afresh by the recoding functions. However, if the buffer pointer is not `NULL`, it should be already allocated, the allocated buffer size then gives its size. If the allocated size gets exceeded while the recoding goes, the buffer will be automatically reallocated bigger, probably elsewhere, and the allocated buffer size will be adjusted accordingly.

The second variable, giving the in-memory buffer size, will receive the exact byte size which was needed for the recoding. A `NUL` character is guaranteed at the end of the produced buffer, but is not counted in the byte size of the recoding. Beyond that `NUL`, there might be some extra space after the recoded data, extending to the allocated buffer size.

file

A file is a sequence of bytes held outside computer memory, but buffered through it. For input, one argument provides a pointer to a file already opened for read. The file is then read and recoded from its current position until the end of the file, effectively swallowing it in memory if the destination of the recoding is a buffer. For reading a file filtered through the recoding library, but only a little bit at a time, one should rather use `recode_filter_open` and `recode_filter_close` (these two functions are not yet available).

For output, one argument provides a pointer to a file already opened for write. The result of the recoding is written to that file starting at its current position.

The following special function is still subject to change:

```
void recode_format_table (request, language, "name");
```

and is not documented anymore for now.

4.3 Task level functions

The task level functions are used internally by the request level functions, they allow more explicit control over files and memory buffers holding input and output to recoding processes. The interface specification of task level functions is still subject to change a bit.

To get started with task level functions, here is a full example of a program which sole job is to filter `ibmpc` code on its standard input into `latin1` code on its standard output. That is, this program has the same goal as the one from the previous section, but does its things a bit differently.

```
#include <stdio.h>
#include <stdbool.h>
#include <recodext.h>

int
main ()
{
  RECODE_OUTER outer = recode_new_outer (false);
  RECODE_REQUEST request = recode_new_request (outer);
  RECODE_TASK task;
  bool success;

  recode_scan_request (request, "ibmpc..latin1");

  task = recode_new_task (request);
  task->input.file = "";
  task->output.file = "";
  success = recode_perform_task (task);

  recode_delete_task (task);
  recode_delete_request (request);
  recode_delete_outer (outer);

  exit (success ? 0 : 1);
}
```

The header file `<recode.h>` declares a `RECODE_TASK` structure, which the programmer should use for allocating a variable in his program. This `task` variable is given as a first argument to all task level functions. The programmer ought to change and possibly consult a few fields in this structure, using special functions.

- Initialisation functions

  ```
  RECODE_TASK recode_new_task (request);
  bool recode_delete_task (task);
  ```

  No *task* variable may be used in other task level functions of the recoding library without having first been initialised with `recode_new_task`. There may be many such *task* variables, in which case, they are independent of one another and they all need to be initialised separately. To avoid memory leaks, a *task* variable should not be initialised a second time without calling `recode_delete_task` to "un-initialise" it. This function also accepts a *request* argument and associates the request to the task.

In fact, a task is essentially a set of recoding transformations with the specification for its current input and its current output.

The *request* variable may be scanned before or after the call to `recode_new_task`, it does not matter so far. Immediately after initialisation, before further changes, the *task* variable associates *request* empty in-memory buffers for both input and output. The output buffer will later get allocated automatically on the fly, as needed, by various task processors.

Even if a call to `recode_delete_task` is not strictly mandatory before ending the program, it is cleaner to always include it. Moreover, in some future version of the recoding library, it might become required.

- Fields of `struct task_request`

  Here are the fields of a `struct task_request` which may be meaningfully changed, once a *task* has been initialised by `recode_new_task`. In fact, fields are expected to change. Once again, to access the fields, you need to include '`recodext.h`' *instead* of '`recode.h`', in which case there also is a greater chance that you need to recompile your programs if a new version of the recoding library gets installed.

  `request`

  : The field `request` points to the current recoding request, but may be changed as needed between recoding calls, for example when there is a need to achieve the construction of a resulting text made up of many pieces, each being recoded differently.

  `input.name`
  `input.file`

  : If `input.name` is not `NULL` at start of a recoding, this is a request that a file by that name be first opened for reading and later automatically closed once the whole file has been read. If the file name is not `NULL` but an empty string, it means that standard input is to be used. The opened file pointer is then held into `input.file`.

    If `input.name` is `NULL` and `input.file` is not, than `input.file` should point to a file already opened for read, which is meant to be recoded.

  `input.buffer`
  `input.cursor`
  `input.limit`

  : When both `input.name` and `input.file` are `NULL`, three pointers describe an in-memory buffer containing the text to be recoded. The buffer extends from `input.buffer` to `input.limit`, yet the text to be recoded only extends from `input.cursor` to `input.limit`. In most situations, `input.cursor` starts with the value that `input.buffer` has. (Its value will internally advance as the recoding goes, until it reaches the value of `input.limit`.)

  `output.name`
  `output.file`

  : If `output.name` is not `NULL` at start of a recoding, this is a request that a file by that name be opened for write and later automatically closed after the recoding is done. If the file name is not `NULL` but an empty string, it means that standard output is to be used. The opened file pointer is then held into `output.file`. If several passes with intermediate files are needed to produce the recoding, the `output.name` file is opened only for the final pass.

If `output.name` is `NULL` and `output.file` is not, then `output.file` should point to a file already opened for write, which will receive the result of the recoding.

`output.buffer`
`output.cursor`
`output.limit`
: When both `output.name` and `output.file` are `NULL`, three pointers describe an in-memory buffer meant to receive the text, once it is recoded. The buffer is already allocated from `output.buffer` to `output.limit`. In most situations, `output.cursor` starts with the value that `output.buffer` has. Once the recoding is done, `output.cursor` will point at the next free byte in the buffer, just after the recoded text, so another recoding could be called without changing any of these three pointers, for appending new information to it. The number of recoded bytes in the buffer is the difference between `output.cursor` and `output.buffer`.

  Each time `output.cursor` reaches `output.limit`, the buffer is reallocated bigger, possibly at a different location in memory, always held up-to-date in `output.buffer`. It is still possible to call a task level function with no output buffer at all to start with, in which case all three fields should have `NULL` as a value. This is the situation immediately after a call to `recode_new_task`.

`strategy`
: This field, which is of type `enum recode_sequence_strategy`, tells how various recoding steps (passes) will be interconnected. Its initial value is `RECODE_STRATEGY_UNDECIDED`, which is a constant defined in the header file '`<recodext.h>`'. Other possible values are:

  `RECODE_SEQUENCE_IN_MEMORY`
  : Keep intermediate recodings in memory.

  `RECODE_SEQUENCE_WITH_FILES`
  : Do not fork, use intermediate files.

  `RECODE_SEQUENCE_WITH_PIPE`
  : Fork processes connected with `pipe(2)`.

  The best for now is to leave this field alone, and let the recoding library decide its strategy, as many combinations have not been tested yet.

`byte_order_mark`
: This field, which is preset to `true`, indicates that a byte order mark is to be expected at the beginning of any canonical `UCS-2` or `UTF-16` text, and that such a byte order mark should be also produced for these charsets.

`fail_level`
: This field, which is of type `enum recode_error` (see Section 4.5 [Errors], page 29), sets the error level at which task level functions should report a failure. If an error being detected is equal or greater than `fail_level`, the function will eventually return `false` instead of `true`. The preset value for this field is `RECODE_NOT_CANONICAL`, that means that if not reset to another value, the library will report failure on *any* error.

`abort_level`
: This field, which is of type `enum recode_error` (see Section 4.5 [Errors], page 29), sets the error level at which task level functions should immediately interrupt their processing. If an error being detected is equal or greater than `abort_level`, the function returns immediately, but the returned value (`true` or `false`) is still is decided from the setting of `fail_level`, not `abort_level`. The preset value for this field is `RECODE_MAXIMUM_ERROR`, that means that is not reset to another value, the library will never interrupt a recoding task.

`error_so_far`
: This field, which is of type `enum recode_error` (see Section 4.5 [Errors], page 29), maintains the maximum error level met so far while the recoding task was proceeding. The preset value is `RECODE_NO_ERROR`.

- Task execution

  ```
  recode_perform_task (task);
  recode_filter_open (task, file);
  recode_filter_close (task);
  ```

  The function `recode_perform_task` reads as much input as possible, and recode all of it on prescribed output, given a properly initialised *task*.

  Functions `recode_filter_open` and `recode_filter_close` are only planned for now. They are meant to read input in piecemeal ways. Even if functionality already exists informally in the library, it has not been made available yet through such interface functions.

4.4 Charset level functions

Many functions are internal to the recoding library. Some of them have been made external and available, for the `recode` program had to retain all its previous functionality while being transformed into a mere application of the recoding library. These functions are not really documented here for the time being, as we hope that many of them will vanish over time. When this set of routines will stabilise, it would be convenient to document them as an API for handling charset names and contents.

```
RECODE_CHARSET find_charset (name, cleaning-type);
bool list_all_charsets (charset);
bool list_concise_charset (charset, list-format);
bool list_full_charset (charset);
```

4.5 Handling errors

The `recode` program, while using the `recode` library, needs to control whether recoding problems are reported or not, and then reflect these in the exit status. The program should also instruct the library whether the recoding should be abruptly interrupted when an error is met (so sparing processing when it is known in advance that a wrong result would be discarded anyway), or if it should proceed nevertheless. Here is how the library groups errors into levels, listed here in order of increasing severity.

`RECODE_NO_ERROR`
: No error was met on previous library calls.

`RECODE_NOT_CANONICAL`
: The input text was using one of the many alternative codings for some phenomenon, but not the one `recode` would have canonically generated. So, if the reverse recoding is later attempted, it would produce a text having the same *meaning* as the original text, yet not being byte identical.

  For example, a `Base64` block in which end-of-lines appear elsewhere that at every 76 characters is not canonical. An e-circumflex in TEX which is coded as '`\^{e}`' instead of '`\^e`' is not canonical.

`RECODE_AMBIGUOUS_OUTPUT`
: It has been discovered that if the reverse recoding was attempted on the text output by this recoding, we would not obtain the original text, only because an ambiguity was generated by accident in the output text. This ambiguity would then cause the wrong interpretation to be taken.

  Here are a few examples. If the `Latin-1` sequence '`e^`' is converted to Easy French and back, the result will be interpreted as e-circumflex and so, will not reflect the intent of the original two characters. Recoding an `IBM-PC` text to `Latin-1` and back, where the input text contained an isolated *LF*, will have a spurious *CR* inserted before the *LF*.

  Currently, there are many cases in the library where the production of ambiguous output is not properly detected, as it is sometimes a difficult problem to accomplish this detection, or to do it speedily.

`RECODE_UNTRANSLATABLE`
: One or more input character could not be recoded, because there is just no representation for this character in the output charset.

  Here are a few examples. Non-strict mode often allows `recode` to compute on-the-fly mappings for unrepresentable characters, but strict mode prohibits such attribution of reversible translations: so strict mode might often trigger such an error. Most `UCS-2` codes used to represent Asian characters cannot be expressed in various Latin charsets.

`RECODE_INVALID_INPUT`
: The input text does not comply with the coding it is declared to hold. So, there is no way by which a reverse recoding would reproduce this text, because `recode` should never produce invalid output.

Here are a few examples. In strict mode, `ASCII` text is not allowed to contain characters with the eight bit set. `UTF-8` encodings ought to be minimal[1].

`RECODE_SYSTEM_ERROR`
: The underlying system reported an error while the recoding was going on, likely an input/output error. (This error symbol is currently unused in the library.)

`RECODE_USER_ERROR`
: The programmer or user requested something the recoding library is unable to provide, or used the API wrongly. (This error symbol is currently unused in the library.)

`RECODE_INTERNAL_ERROR`
: Something really wrong, which should normally never happen, was detected within the recoding library. This might be due to genuine bugs in the library, or maybe due to un-initialised or overwritten arguments to the API. (This error symbol is currently unused in the library.)

`RECODE_MAXIMUM_ERROR`
: This error code should never be returned, it is only internally used as a sentinel for the list of all possible error codes.

One should be able to set the error level threshold for returning failure at end of recoding, and also the threshold for immediate interruption. If many errors occur while the recoding proceed, which are not severe enough to interrupt the recoding, then the most severe error is retained, while others are forgotten[2]. So, in case of an error, the possible actions currently are:

- do nothing and let go, returning success at end of recoding,
- just let go for now, but return failure at end of recoding,
- interrupt recoding right away and return failure now.

See Section 4.3 [Task level], page 25, and particularly the description of the fields `fail_level`, `abort_level` and `error_so_far`, for more information about how errors are handled.

---

[1] The minimality of an `UTF-8` encoding is guaranteed on output, but currently, it is not checked on input.

[2] Another approach would have been to define the level symbols as masks instead, and to give masks to threshold setting routines, and to retain all errors—yet I never met myself such a need in practice, and so I fear it would be overkill. On the other hand, it might be interesting to maintain counters about how many times each kind of error occurred.

5 The universal charset

Standard ISO 10646 defines a universal character set, intended to encompass in the long run all languages written on this planet. It is based on wide characters, and offer possibilities for two billion characters (2^31).

This charset was to become available in `recode` under the name `UCS`, with many external surfaces for it. But in the current version, only surfaces of `UCS` are offered, each presented as a genuine charset rather than a surface. Such surfaces are only meaningful for the `UCS` charset, so it is not that useful to draw a line between the surfaces and the only charset to which they may apply.

`UCS` stands for Universal Character Set. `UCS-2` and `UCS-4` are fixed length encodings, using two or four bytes per character respectively. `UTF` stands for `UCS` Transformation Format, and are variable length encodings dedicated to `UCS`. `UTF-1` was based on ISO 2022, it did not succeed[1]. `UTF-2` replaced it, it has been called `UTF-FSS` (File System Safe) in Unicode or Plan9 context, but is better known today as `UTF-8`. To complete the picture, there is `UTF-16` based on 16 bits bytes, and `UTF-7` which is meant for transmissions limited to 7-bit bytes. Most often, one might see `UTF-8` used for external storage, and `UCS-2` used for internal storage.

When `recode` is producing any representation of `UCS`, it uses the replacement character `U+FFFD` for any *valid* character which is not representable in the goal charset[2]. This happens, for example, when `UCS-2` is not capable to echo a wide `UCS-4` character, or for a similar reason, an `UTF-8` sequence using more than three bytes. The replacement character is meant to represent an existing character. So, it is never produced to represent an invalid sequence or ill-formed character in the input text. In such cases, `recode` just gets rid of the noise, while taking note of the error in its usual ways.

Even if `UTF-8` is an encoding, really, it is the encoding of a single character set, and nothing else. It is useful to distinguish between an encoding (a *surface* within `recode`) and a charset, but only when the surface may be applied to several charsets. Specifying a charset is a bit simpler than specifying a surface in a `recode` request. There would not be a practical advantage at imposing a more complex syntax to `recode` users, when it is simple to assimilate `UTF-8` to a charset. Similar considerations apply for `UCS-2`, `UCS-4`, `UTF-16` and `UTF-7`. These are all considered to be charsets.

5.1 Universal Character Set, 2 bytes

One surface of `UCS` is usable for the subset defined by its first sixty thousand characters (in fact, $31 * 2\hat{\ }11$ codes), and uses exactly two bytes per character. It is a mere dump of the internal memory representation which is *natural* for this subset and as such, conveys with it endianness problems.

A non-empty `UCS-2` file normally begins with a so called *byte order mark*, having value `0xFEFF`. The value `0xFFFE` is not an `UCS` character, so if this value is seen at the beginning of

[1] It is not probable that `recode` will ever support `UTF-1`.

[2] This is when the goal charset allows for 16-bits. For shorter charsets, the '`--strict`' ('`-s`') option decides what happens: either the character is dropped, or a reversible mapping is produced on the fly.

a file, `recode` reacts by swapping all pairs of bytes. The library also properly reacts to other occurrences of `0xFEFF` or `0xFFFE` elsewhere than at the beginning, because concatenation of `UCS-2` files should stay a simple matter, but it might trigger a diagnostic about non canonical input.

By default, when producing an `UCS-2` file, `recode` always outputs the high order byte before the low order byte. But this could be easily overridden through the `21-Permutation` surface (see Section 12.1 [Permutations], page 61). For example, the command:

```
recode u8..u2/21 < input > output
```

asks for an `UTF-8` to `UCS-2` conversion, with swapped byte output.

Use `UCS-2` as a genuine charset. This charset is available in `recode` under the name `ISO-10646-UCS-2`. Accepted aliases are `UCS-2`, `BMP`, `rune` and `u2`.

The `recode` library is able to combine `UCS-2` some sequences of codes into single code characters, to represent a few diacriticized characters, ligatures or diphtongs which have been included to ease mapping with other existing charsets. It is also able to explode such single code characters into the corresponding sequence of codes. The request syntax for triggering such operations is rudimentary and temporary. The `combined-UCS-2` pseudo character set is a special form of `UCS-2` in which known combinings have been replaced by the simpler code. Using `combined-UCS-2` instead of `UCS-2` in an *after* position of a request forces a combining step, while using `combined-UCS-2` instead of `UCS-2` in a *before* position of a request forces an exploding step. For the time being, one has to resort to advanced request syntax to achieve other effects. For example:

```
recode u8..co,u2..u8 < input > output
```

copies an `UTF-8` *input* over *output*, still to be in `UTF-8`, yet merging combining characters into single codes whenever possible.

5.2 Universal Character Set, 4 bytes

Another surface of `UCS` uses exactly four bytes per character, and is a mere dump of the internal memory representation which is *natural* for the whole charset and as such, conveys with it endianness problems.

Use it as a genuine charset. This charset is available in `recode` under the name `ISO-10646-UCS-4`. Accepted aliases are `UCS`, `UCS-4`, `ISO_10646`, `10646` and `u4`.

5.3 Universal Transformation Format, 7 bits

`UTF-7` comes from IETF rather than ISO, and is described by RFC 1642, in the MIME series. The `UTF-7` encoding is meant to fit `UCS-2` over channels limited to seven bits per byte. It proceeds from a mix between the spirit of `Quoted-Printable` and methods of `Base64`, adapted to Unicode contexts.

This charset is available in `recode` under the name `UNICODE-1-1-UTF-7`. Accepted aliases are `UTF-7`, `TF-7` and `u7`.

5.4 Universal Transformation Format, 8 bits

Even if `UTF-8` does not originally come from IETF, there is now RFC 2279 to describe it. In letters sent on 1995-01-21 and 1995-04-20, Markus Kuhn writes:

> `UTF-8` is an `ASCII` compatible multi-byte encoding of the ISO 10646 universal character set (`UCS`). `UCS` is a 31-bit superset of all other character set standards. The first 256 characters of `UCS` are identical to those of ISO 8859-1 (Latin-1). The `UCS-2` encoding of UCS is a sequence of bigendian 16-bit words, the `UCS-4` encoding is a sequence of bigendian 32-bit words. The `UCS-2` subset of ISO 10646 is also known as "Unicode". As both `UCS-2` and `UCS-4` require heavy modifications to traditional `ASCII` oriented system designs (e.g. Unix), the `UTF-8` encoding has been designed for these applications.
>
> In `UTF-8`, only `ASCII` characters are encoded using bytes below 128. All other non-ASCII characters are encoded as multi-byte sequences consisting only of bytes in the range 128-253. This avoids critical bytes like *`NUL`* and `/` in `UTF-8` strings, which makes the `UTF-8` encoding suitable for being handled by the standard C string library and being used in Unix file names. Other properties include the preserved lexical sorting order and that `UTF-8` allows easy self-synchronisation of software receiving `UTF-8` strings.

`UTF-8` is the most common external surface of `UCS`, each character uses from one to six bytes, and is able to encode all 2^31 characters of the `UCS`. It is implemented as a charset, with the following properties:

- Strict 7-bit `ASCII` is completely invariant under `UTF-8`, and those are the only one-byte characters. `UCS` values and `ASCII` values coincide. No multi-byte characters ever contain bytes less than 128. `NUL` *is* `NUL`. A multi-byte character always starts with a byte of 192 or more, and is always followed by a number of bytes between 128 to 191. That means that you may read at random on disk or memory, and easily discover the start of the current, next or previous character. You can count, skip or extract characters with this only knowledge.
- If you read the first byte of a multi-byte character in binary, it contains many '`1`' bits in successions starting with the most significant one (from the left), at least two. The length of this '`1`' sequence equals the byte size of the character. All succeeding bytes start by '`10`'. This is a lot of redundancy, making it fairly easy to guess that a file is valid `UTF-8`, or to safely state that it is not.
- In a multi-byte character, if you remove all leading '`1`' bits of the first byte of a multi-byte character, and the initial '`10`' bits of all remaining bytes (so keeping 6 bits per byte for those), the remaining bits concatenated are the UCS value.

These properties also have a few nice consequences:

- Conversion to/from values is algorithmically simple, and reasonably speedy.
- A sequence of N bytes can hold characters needing up to 2 + $5N$ bits in their `UCS` representation. Here, N is a number between 1 and 6. So, `UTF-8` is most economical when mapping ASCII (1 byte), followed by `UCS-2` (1 to 3 bytes) and `UCS-4` (1 to 6 bytes).
- The lexicographic sorting order of `UCS` strings is preserved.

- Bytes with value 254 or 255 never appear, and because of that, these are sometimes used when escape mechanisms are needed.

In some case, when little processing is done on a lot of strings, one may choose for efficiency reasons to handle `UTF-8` strings directly even if variable length, as it is easy to get start of characters. Character insertion or replacement might require moving the remainder of the string in either direction. In most cases, it is faster and easier to convert from `UTF-8` to `UCS-2` or `UCS-4` prior to processing.

This charset is available in `recode` under the name `UTF-8`. Accepted aliases are `UTF-2`, `UTF-FSS`, `FSS_UTF`, `TF-8` and `u8`.

5.5 Universal Transformation Format, 16 bits

Another external surface of `UCS` is also variable length, each character using either two or four bytes. It is usable for the subset defined by the first million characters ($17 * 2\hat{}16$) of `UCS`.

Martin J. Dürst writes (to `comp.std.internat`, on 1995-03-28):

> `UTF-16` is another method that reserves two times 1024 codepoints in Unicode and uses them to index around one million additional characters. `UTF-16` is a little bit like former multibyte codes, but quite not so, as both the first and the second 16-bit code clearly show what they are. The idea is that one million codepoints should be enough for all the rare Chinese ideograms and historical scripts that do not fit into the Base Multilingual Plane of ISO 10646 (with just about 63,000 positions available, now that 2,000 are gone).

This charset is available in `recode` under the name `UTF-16`. Accepted aliases are `Unicode`, `TF-16` and `u6`.

5.6 Frequency count of characters

A device may be used to obtain a list of characters in a file, and how many times each character appears. Each count is followed by the `UCS-2` value of the character and, when known, the RFC 1345 mnemonic for that character.

This charset is available in `recode` under the name `count-characters`.

This `count` feature has been implemented as a charset. This may change in some later version, as it would sometimes be convenient to count original bytes, instead of their `UCS-2` equivalent.

5.7 Fully interpreted UCS dump

Another device may be used to get fully interpreted dumps of an `UCS-2` stream of characters, with one `UCS-2` character displayed on a full output line. Each line receives the RFC 1345 mnemonic for the character if it exists, the `UCS-2` value of the character, and a descriptive comment for that character. As each input character produces its own output line, beware that the output file from this conversion may be much, much bigger than the input file.

This charset is available in `recode` under the name `dump-with-names`.

This `dump-with-names` feature has been implemented as a charset rather than a surface. This is surely debatable. The current implementation allows for dumping charsets other than `UCS-2`. For example, the command '`recode l2..full <` *input*' implies a necessary conversion from `Latin-2` to `UCS-2`, as `dump-with-names` is only connected out from `UCS-2`. In such cases, `recode` does not display the original `Latin-2` codes in the dump, only the corresponding `UCS-2` values. To give a simpler example, the command

```
echo 'Hello, world!' | recode us..dump
```

produces the following output:

```
UCS2   Mne   Description

0048   H     latin capital letter h
0065   e     latin small letter e
006C   l     latin small letter l
006C   l     latin small letter l
006F   o     latin small letter o
002C   ,     comma
0020   SP    space
0077   w     latin small letter w
006F   o     latin small letter o
0072   r     latin small letter r
006C   l     latin small letter l
0064   d     latin small letter d
0021   !     exclamation mark
000A   LF    line feed (lf)
```

The descriptive comment is given in English and `ASCII`, yet if the English description is not available but a French one is, then the French description is given instead, using `Latin-1`. However, if the `LANGUAGE` or `LANG` environment variable begins with the letters '`fr`', then listing preference goes to French when both descriptions are available.

Here is another example. To get the long description of the code 237 in Latin-5 table, one may use the following command.

```
echo -n 237 | recode l5/d..dump
```

If your `echo` does not grok '`-n`', use '`echo 237\c`' instead. Here is how to see what Unicode `U+03C6` means, while getting rid of the title lines.

```
echo -n 0x03C6 | recode u2/x2..dump | tail +3
```

6 Tabular sources (RFC 1345)

An important part of the tabular charset knowledge in `recode` comes from RFC 1345 or, alternatively, from the `chset` tools, both maintained by Keld Simonsen. The RFC 1345 document:

> "Character Mnemonics & Character Sets", K. Simonsen, Request for Comments no. 1345, Network Working Group, June 1992.

defines many character mnemonics and character sets. The `recode` library implements most of RFC 1345, however:

- It does not recognise those charsets which overload character positions: `dk-us` and `us-dk`. However, see Section 3.7 [Mixed], page 17.
- It does not recognise those charsets which combine two characters for representing a third: `ANSI_X3.110-1983`, `ISO_6937-2-add`, `T.101-G2`, `T.61-8bit`, `iso-ir-90` and `videotex-suppl`.
- It does not recognise 16-bits charsets: `GB_2312-80`, `JIS_C6226-1978`, `JIS_C6226-1983`, `JIS_X0212-1990` and `KS_C_5601-1987`.
- It interprets the charset `isoir91` as `NATS-DANO` (alias `iso-ir-9-1`), *not* as `JIS_C6229-1984-a` (alias `iso-ir-91`). It also interprets the charset `isoir92` as `NATS-DANO-ADD` (alias `iso-ir-9-2`), *not* as `JIS_C6229-1984-b` (alias `iso-ir-92`). It might be better just avoiding these two alias names.

Keld Simonsen `keld@dkuug.dk` did most of RFC 1345 himself, with some funding from Danish Standards and Nordic standards (INSTA) project. He also did the character set design work, with substantial input from Olle Jaernefors. Keld typed in almost all of the tables, some have been contributed. A number of people have checked the tables in various ways. The RFC lists a number of people who helped.

Keld and the `recode` maintainer have an arrangement by which any new discovered information submitted by `recode` users, about tabular charsets, is forwarded to Keld, eventually merged into Keld's work, and only then, reimported into `recode`. Neither the `recode` program nor its library try to compete, nor even establish themselves as an alternate or diverging reference: RFC 1345 and its new drafts stay the genuine source for most tabular information conveyed by `recode`. Keld has been more than collaborative so far, so there is no reason that we act otherwise. In a word, `recode` should be perceived as the application of external references, but not as a reference in itself.

Internally, RFC 1345 associates which each character an unambiguous mnemonic of a few characters, taken from ISO 646, which is a minimal ASCII subset of 83 characters. The charset made up by these mnemonics is available in `recode` under the name `RFC1345`. It has `mnemonic` and `1345` for aliases. As implemened, this charset exactly corresponds to `mnemonic+ascii+38`, using RFC 1345 nomenclature. Roughly said, ISO 646 characters represent themselves, except for the ampersand (`&`) which appears doubled. A prefix of a single ampersand introduces a mnemonic. For mnemonics using two characters, the prefix is immediately by the mnemonic. For longer mnemonics, the prefix is followed by an underline (`_`), the mmemonic, and another underline. Conversions to this charset are usually reversible.

Currently, `recode` does not offer any of the many other possible variations of this family of representations. They will likely be implemented in some future version, however.

`ANSI_X3.4-1968`
: `367`, `ANSI_X3.4-1986`, `ASCII`, `CP367`, `IBM367`, `ISO646-US`, `ISO_646.irv:1991`, `US-ASCII`, `iso-ir-6` and `us` are aliases for this charset. Source: ISO 2375 registry.

`ASMO_449`
: `ISO_9036`, `arabic7` and `iso-ir-89` are aliases for this charset. Source: ISO 2375 registry.

`BS_4730`
: `ISO646-GB`, `gb`, `iso-ir-4` and `uk` are aliases for this charset. Source: ISO 2375 registry.

`BS_viewdata`
: `iso-ir-47` is an alias for this charset. Source: ISO 2375 registry.

`CP1250`
: `1250`, `ms-ee` and `windows-1250` are aliases for this charset. Source: UNICODE 1.0.

`CP1251`
: `1251`, `ms-cyrl` and `windows-1251` are aliases for this charset. Source: UNICODE 1.0.

`CP1252`
: `1252`, `ms-ansi` and `windows-1252` are aliases for this charset. Source: UNICODE 1.0.

`CP1253`
: `1253`, `ms-greek` and `windows-1253` are aliases for this charset. Source: UNICODE 1.0.

`CP1254`
: `1254`, `ms-turk` and `windows-1254` are aliases for this charset. Source: UNICODE 1.0.

`CP1255`
: `1255`, `ms-hebr` and `windows-1255` are aliases for this charset. Source: UNICODE 1.0.

`CP1256`
: `1256`, `ms-arab` and `windows-1256` are aliases for this charset. Source: UNICODE 1.0.

`CP1257`
: `1257`, `WinBaltRim` and `windows-1257` are aliases for this charset. Source: CEN/TC304 N283.

`CSA_Z243.4-1985-1`
: `ISO646-CA`, `ca`, `csa7-1` and `iso-ir-121` are aliases for this charset. Source: ISO 2375 registry.

`CSA_Z243.4-1985-2`
: `ISO646-CA2`, `csa7-2` and `iso-ir-122` are aliases for this charset. Source: ISO 2375 registry.

`CSA_Z243.4-1985-gr`
: `iso-ir-123` is an alias for this charset. Source: ISO 2375 registry.

`CSN_369103`
: `KOI-8_L2`, `iso-ir-139` and `koi8l2` are aliases for this charset. Source: ISO 2375 registry.

CWI
: `CWI-2` and `cp-hu` are aliases for this charset. Source: Computerworld Sza'mita'stechnika vol 3 issue 13 1988-06-29.

DEC-MCS
: `dec` is an alias for this charset. VAX/VMS User's Manual, Order Number: AI-Y517A-TE, April 1986.

DIN_66003
: `ISO646-DE`, `de` and `iso-ir-21` are aliases for this charset. Source: ISO 2375 registry.

DS_2089
: `DS2089`, `ISO646-DK` and `dk` are aliases for this charset. Source: Danish Standard, DS 2089, February 1974.

EBCDIC-AT-DE
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-AT-DE-A
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-CA-FR
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-DK-NO
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-DK-NO-A
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-ES
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-ES-A
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-ES-S
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-FI-SE
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-FI-SE-A
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-FR
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-IS-FRISS
: `friss` is an alias for this charset. Source: Skyrsuvelar Rikisins og Reykjavikurborgar, feb 1982.

EBCDIC-IT
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-PT
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-UK
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

EBCDIC-US
: Source: IBM 3270 Char Set Ref Ch 10, GA27-2837-9, April 1987.

ECMA-cyrillic
: ECMA-113, ECMA-113:1986 and iso-ir-111 are aliases for this charset. Source: ISO 2375 registry.

ES
: ISO646-ES and iso-ir-17 are aliases for this charset. Source: ISO 2375 registry.

ES2
: ISO646-ES2 and iso-ir-85 are aliases for this charset. Source: ISO 2375 registry.

GB_1988-80
: ISO646-CN, cn and iso-ir-57 are aliases for this charset. Source: ISO 2375 registry.

GOST_19768-87
: ST_SEV_358-88 and iso-ir-153 are aliases for this charset. Source: ISO 2375 registry.

IBM037
: 037, CP037, ebcdic-cp-ca, ebcdic-cp-nl, ebcdic-cp-us and ebcdic-cp-wt are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM038
: 038, CP038 and EBCDIC-INT are aliases for this charset. Source: IBM 3174 Character Set Ref, GA27-3831-02, March 1990.

IBM1004
: 1004, CP1004 and os2latin1 are aliases for this charset. Source: CEN/TC304 N283, 1994-02-04.

IBM1026
: 1026 and CP1026 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM1047
: 1047 and CP1047 are aliases for this charset. Source: IBM Character Data Representation Architecture. Registry SC09-1391-00 p 150.

IBM256
: 256, CP256 and EBCDIC-INT1 are aliases for this charset. Source: IBM Registry C-H 3-3220-050.

IBM273
: 273 and CP273 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM274
: 274, CP274 and EBCDIC-BE are aliases for this charset. Source: IBM 3174 Character Set Ref, GA27-3831-02, March 1990.

IBM275
: 275, CP275 and EBCDIC-BR are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM277
: EBCDIC-CP-DK and EBCDIC-CP-NO are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM278
: 278, CP278, ebcdic-cp-fi and ebcdic-cp-se are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM280
: 280, CP280 and ebcdic-cp-it are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM281
: 281, CP281 and EBCDIC-JP-E are aliases for this charset. Source: IBM 3174 Character Set Ref, GA27-3831-02, March 1990.

IBM284
: 284, CP284 and ebcdic-cp-es are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM285
: 285, CP285 and ebcdic-cp-gb are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM290
: 290, CP290 and EBCDIC-JP-kana are aliases for this charset. Source: IBM 3174 Character Set Ref, GA27-3831-02, March 1990.

IBM297
: 297, CP297 and ebcdic-cp-fr are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM420
: 420, CP420 and ebcdic-cp-ar1 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990. IBM NLS RM p 11-11.

IBM423
: 423, CP423 and ebcdic-cp-gr are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM424
: 424, CP424 and ebcdic-cp-he are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM437
: 437 and CP437 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM500
: 500, 500V1, CP500, ebcdic-cp-be and ebcdic-cp-ch are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM850
: 850 and CP850 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990. Source: UNICODE 1.0.

IBM851
: 851 and CP851 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM852
: 852, CP852, pcl2 and pclatin2 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM855
: 855 and CP855 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM857
: 857 and CP857 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM860
: 860 and CP860 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM861
: 861, CP861 and cp-is are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM862 — 862 and CP862 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM863 — 863 and CP863 are aliases for this charset. Source: IBM Keyboard layouts and code pages, PN 07G4586 June 1991.

IBM864 — 864 and CP864 are aliases for this charset. Source: IBM Keyboard layouts and code pages, PN 07G4586 June 1991.

IBM865 — 865 and CP865 are aliases for this charset. Source: IBM DOS 3.3 Ref (Abridged), 94X9575 (Feb 1987).

IBM868 — 868, CP868 and cp-ar are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM869 — 869, CP869 and cp-gr are aliases for this charset. Source: IBM Keyboard layouts and code pages, PN 07G4586 June 1991.

IBM870 — 870, CP870, ebcdic-cp-roece and ebcdic-cp-yu are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM871 — 871, CP871 and ebcdic-cp-is are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM875 — 875, CP875 and EBCDIC-Greek are aliases for this charset. Source: UNICODE 1.0.

IBM880 — 880, CP880 and EBCDIC-Cyrillic are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM891 — 891 and CP891 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM903 — 903 and CP903 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM904 — 904 and CP904 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IBM905 — 905, CP905 and ebcdic-cp-tr are aliases for this charset. Source: IBM 3174 Character Set Ref, GA27-3831-02, March 1990.

IBM918 — 918, CP918 and ebcdic-cp-ar2 are aliases for this charset. Source: IBM NLS RM Vol2 SE09-8002-01, March 1990.

IEC_P27-1 — iso-ir-143 is an alias for this charset. Source: ISO 2375 registry.

INIS — iso-ir-49 is an alias for this charset. Source: ISO 2375 registry.

INIS-8 — iso-ir-50 is an alias for this charset. Source: ISO 2375 registry.

INIS-cyrillic — iso-ir-51 is an alias for this charset. Source: ISO 2375 registry.

INVARIANT — iso-ir-170 is an alias for this charset.

ISO-8859-1
: 819, CP819, IBM819, ISO_8859-1, ISO_8859-1:1987, iso-ir-100, l1 and latin1 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-10
: ISO_8859-10, ISO_8859-10:1993, iso-ir-157, l6 and latin6 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-13
: ISO_8859-13, ISO_8859-13:1998, iso-baltic, iso-ir-179a, l7 and latin7 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-14
: ISO_8859-14, ISO_8859-14:1998, iso-celtic, iso-ir-199, l8 and latin8 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-15
: ISO_8859-15, ISO_8859-15:1998, iso-ir-203, l9 and latin9 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-2
: 912, CP912, IBM912, ISO_8859-2, ISO_8859-2:1987, iso-ir-101, l2 and latin2 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-3
: ISO_8859-3, ISO_8859-3:1988, iso-ir-109, l3 and latin3 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-4
: ISO_8859-4, ISO_8859-4:1988, iso-ir-110, l4 and latin4 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-5
: ISO_8859-5, ISO_8859-5:1988, cyrillic and iso-ir-144 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-6
: ASMO-708, ECMA-114, ISO_8859-6, ISO_8859-6:1987, arabic and iso-ir-127 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-7
: ECMA-118, ELOT_928, ISO_8859-7, ISO_8859-7:1987, greek, greek8 and iso-ir-126 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-8
: ISO_8859-8, ISO_8859-8:1988, hebrew and iso-ir-138 are aliases for this charset. Source: ISO 2375 registry.

ISO-8859-9
: ISO_8859-9, ISO_8859-9:1989, iso-ir-148, l5 and latin5 are aliases for this charset. Source: ISO 2375 registry.

ISO_10367-box
: iso-ir-155 is an alias for this charset. Source: ISO 2375 registry.

ISO_2033-1983
: e13b and iso-ir-98 are aliases for this charset. Source: ISO 2375 registry.

ISO_5427
: iso-ir-37 is an alias for this charset. Source: ISO 2375 registry.

ISO_5427-ext
: ISO_5427:1981 and iso-ir-54 are aliases for this charset. Source: ISO 2375 registry.

ISO_5428
: ISO_5428:1980 and iso-ir-55 are aliases for this charset. Source: ISO 2375 registry.

ISO_646.basic
: ISO_646.basic:1983 and ref are aliases for this charset. Source: ISO 2375 registry.

ISO_646.irv
: ISO_646.irv:1983, irv and iso-ir-2 are aliases for this charset. Source: ISO 2375 registry.

ISO_6937-2-25
: iso-ir-152 is an alias for this charset. Source: ISO 2375 registry.

ISO_8859-supp
: iso-ir-154 and latin1-2-5 are aliases for this charset. Source: ISO 2375 registry.

IT
: ISO646-IT and iso-ir-15 are aliases for this charset. Source: ISO 2375 registry.

JIS_C6220-1969-jp
: JIS_C6220-1969, iso-ir-13, katakana and x0201-7 are aliases for this charset. Source: ISO 2375 registry.

JIS_C6220-1969-ro
: ISO646-JP, iso-ir-14 and jp are aliases for this charset. Source: ISO 2375 registry.

JIS_C6229-1984-a
: jp-ocr-a is an alias for this charset. Source: ISO 2375 registry.

JIS_C6229-1984-b
: ISO646-JP-OCR-B and jp-ocr-b are aliases for this charset. Source: ISO 2375 registry.

JIS_C6229-1984-b-add
: iso-ir-93 and jp-ocr-b-add are aliases for this charset. Source: ISO 2375 registry.

JIS_C6229-1984-hand
: iso-ir-94 and jp-ocr-hand are aliases for this charset. Source: ISO 2375 registry.

JIS_C6229-1984-hand-add
: iso-ir-95 and jp-ocr-hand-add are aliases for this charset. Source: ISO 2375 registry.

`JIS_C6229-1984-kana`
: `iso-ir-96` is an alias for this charset. Source: ISO 2375 registry.

`JIS_X0201`
: `X0201` is an alias for this charset.

`JUS_I.B1.002`
: `ISO646-YU`, `iso-ir-141`, `js` and `yu` are aliases for this charset. Source: ISO 2375 registry.

`JUS_I.B1.003-mac`
: `iso-ir-147` and `macedonian` are aliases for this charset. Source: ISO 2375 registry.

`JUS_I.B1.003-serb`
: `iso-ir-146` and `serbian` are aliases for this charset. Source: ISO 2375 registry.

`KOI-7`
: Source: Andrey A. Chernov <ache@nagual.pp.ru>.

`KOI-8`
: `GOST_19768-74` is an alias for this charset. Source: Andrey A. Chernov <ache@nagual.pp.ru>.

`KOI8-R`
: Source: RFC1489 via Gabor Kiss <kissg@sztaki.hu>. And Andrey A. Chernov <ache@nagual.pp.ru>.

`KOI8-RU`
: Source: http://cad.ntu-kpi.kiev.ua/multiling/koi8-ru/.

`KOI8-U`
: Source: RFC 2319. Mibenum: 2088. Source: http://www.net.ua/KOI8-U/.

`KSC5636`
: `ISO646-KR` is an alias for this charset.

`Latin-greek-1`
: `iso-ir-27` is an alias for this charset. Source: ISO 2375 registry.

`MSZ_7795.3`
: `ISO646-HU`, `hu` and `iso-ir-86` are aliases for this charset. Source: ISO 2375 registry.

`NATS-DANO`
: `iso-ir-9-1` is an alias for this charset. Source: ISO 2375 registry.

`NATS-DANO-ADD`
: `iso-ir-9-2` is an alias for this charset. Source: ISO 2375 registry.

`NATS-SEFI`
: `iso-ir-8-1` is an alias for this charset. Source: ISO 2375 registry.

`NATS-SEFI-ADD`
: `iso-ir-8-2` is an alias for this charset. Source: ISO 2375 registry.

`NC_NC00-10`
: `ISO646-CU`, `NC_NC00-10:81`, `cuba` and `iso-ir-151` are aliases for this charset. Source: ISO 2375 registry.

`NF_Z_62-010`
: `ISO646-FR`, `fr` and `iso-ir-69` are aliases for this charset. Source: ISO 2375 registry.

`NF_Z_62-010_(1973)`
: `ISO646-FR1` and `iso-ir-25` are aliases for this charset. Source: ISO 2375 registry.

`NS_4551-1`
: `ISO646-NO`, `iso-ir-60` and `no` are aliases for this charset. Source: ISO 2375 registry.

`NS_4551-2`
: `ISO646-NO2`, `iso-ir-61` and `no2` are aliases for this charset. Source: ISO 2375 registry.

`NeXTSTEP`
: `next` is an alias for this charset. Source: Peter Svanberg - psv@nada.kth.se.

`PT`
: `ISO646-PT` and `iso-ir-16` are aliases for this charset. Source: ISO 2375 registry.

`PT2`
: `ISO646-PT2` and `iso-ir-84` are aliases for this charset. Source: ISO 2375 registry.

`SEN_850200_B`
: `FI`, `ISO646-FI`, `ISO646-SE`, `SS636127`, `iso-ir-10` and `se` are aliases for this charset. Source: ISO 2375 registry.

`SEN_850200_C`
: `ISO646-SE2`, `iso-ir-11` and `se2` are aliases for this charset. Source: ISO 2375 registry.

`T.61-7bit`
: `iso-ir-102` is an alias for this charset. Source: ISO 2375 registry.

`baltic`
: `iso-ir-179` is an alias for this charset. Source: ISO 2375 registry. &g1esc x2d56 &g2esc x2e56 &g3esc x2f56.

`greek-ccitt`
: `iso-ir-150` is an alias for this charset. Source: ISO 2375 registry.

`greek7`
: `iso-ir-88` is an alias for this charset. Source: ISO 2375 registry.

`greek7-old`
: `iso-ir-18` is an alias for this charset. Source: ISO 2375 registry.

`hp-roman8`
: `r8` and `roman8` are aliases for this charset. Source: LaserJet IIP Printer User's Manual,. HP part no 33471-90901, Hewlet-Packard, June 1989.

`latin-greek`
: `iso-ir-19` is an alias for this charset. Source: ISO 2375 registry.

`mac-is`

`macintosh`
: `mac` is an alias for this charset. Source: The Unicode Standard ver 1.0, ISBN 0-201-56788-1, Oct 1991.

`macintosh_ce`
: `macce` is an alias for this charset. Source: Macintosh CE fonts.

`sami`
: `iso-ir-158`, `lap` and `latin-lap` are aliases for this charset. Source: ISO 2375 registry.

7 ASCII and some derivatives

7.1 Usual ASCII

This charset is available in `recode` under the name `ASCII`. In fact, it's true name is `ANSI_X3.4-1968` as per RFC 1345, accepted aliases being `ANSI_X3.4-1986`, `ASCII`, `IBM367`, `ISO646-US`, `ISO_646.irv:1991`, `US-ASCII`, `cp367`, `iso-ir-6` and `us`. The shortest way of specifying it in `recode` is `us`.

This documentation used to include ASCII tables. They have been removed since the `recode` program can now recreate these easily:

```
recode -lf us                  for commented ASCII
recode -ld us                  for concise decimal table
recode -lo us                  for concise octal table
recode -lh us                  for concise hexadecimal table
```

7.2 ASCII extended by Latin Alphabets

There are many Latin charsets. The following has been written by Tim Lasko `lasko@video.dec.com`, a long while ago:

> ISO Latin-1, or more completely ISO Latin Alphabet No 1, is now an international standard as of February 1987 (IS 8859, Part 1). For those American USEnet'rs that care, the 8-bit ASCII standard, which is essentially the same code, is going through the final administrative processes prior to publication. ISO Latin-1 (IS 8859/1) is actually one of an entire family of eight-bit one-byte character sets, all having ASCII on the left hand side, and with varying repertoires on the right hand side:
>
> - Latin Alphabet No 1 (caters to Western Europe - now approved).
> - Latin Alphabet No 2 (caters to Eastern Europe - now approved).
> - Latin Alphabet No 3 (caters to SE Europe + others - in draft ballot).
> - Latin Alphabet No 4 (caters to Northern Europe - in draft ballot).
> - Latin-Cyrillic alphabet (right half all Cyrillic - processing currently suspended pending USSR input).
> - Latin-Arabic alphabet (right half all Arabic - now approved).
> - Latin-Greek alphabet (right half Greek + symbols - in draft ballot).
> - Latin-Hebrew alphabet (right half Hebrew + symbols - proposed).

The ISO Latin Alphabet 1 is available as a charset in `recode` under the name `Latin-1`. In fact, it's true name is `ISO_8859-1:1987` as per RFC 1345, accepted aliases being `CP819`, `IBM819`, `ISO-8859-1`, `ISO_8859-1`, `iso-ir-100`, `l1` and `Latin-1`. The shortest way of specifying it in `recode` is `l1`.

It is an eight-bit code which coincides with ASCII for the lower half. This documentation used to include Latin-1 tables. They have been removed since the `recode` program can now recreate these easily:

```
recode -lf l1                      for commented ISO Latin-1
recode -ld l1                      for concise decimal table
recode -lo l1                      for concise octal table
recode -lh l1                      for concise hexadecimal table
```

7.3 ASCII 7-bits, *BS* to overstrike

This charset is available in `recode` under the name `ASCII-BS`, with `BS` as an acceptable alias.

The file is straight ASCII, seven bits only. According to the definition of ASCII, diacritics are applied by a sequence of three characters: the letter, one *BS*, the diacritic mark. We deviate slightly from this by exchanging the diacritic mark and the letter so, on a screen device, the diacritic will disappear and let the letter alone. At recognition time, both methods are acceptable.

The French quotes are coded by the sequences: `<` *BS* `"` or `"` *BS* `<` for the opening quote and `>` *BS* `"` or `"` *BS* `>` for the closing quote. This artificial convention was inherited in straight `ASCII-BS` from habits around `Bang-Bang` entry, and is not well known. But we decided to stick to it so that `ASCII-BS` charset will not lose French quotes.

The `ASCII-BS` charset is independent of `ASCII`, and different. The following examples demonstrate this, knowing at advance that '`!2`' is the `Bang-Bang` way of representing an `e` with an acute accent. Compare:

```
% echo \!2 | recode -v bang..l1/d
Request: Bang-Bang..ISO-8859-1/Decimal-1
233,  10
```

with:

```
% echo \!2 | recode -v bang..bs/d
Request: Bang-Bang..ISO-8859-1..ASCII-BS/Decimal-1
 39,   8, 101,  10
```

In the first case, the `e` with an acute accent is merely transmitted by the `Latin-1..ASCII` mapping, not having a special recoding rule for it. In the `Latin-1..ASCII-BS` case, the acute accent is applied over the `e` with a backspace: diacriticised characters have special rules. For the `ASCII-BS` charset, reversibility is still possible, but there might be difficult cases.

7.4 ASCII without diacritics nor underline

This charset is available in `recode` under the name `flat`.

This code is ASCII expunged of all diacritics and underlines, as long as they are applied using three character sequences, with *BS* in the middle. Also, despite slightly unrelated, each control character is represented by a sequence of two or three graphic characters. The newline character, however, keeps its functionality and is not represented.

Note that charset `flat` is a terminal charset. We can convert *to* `flat`, but not *from* it.

8 Some IBM or Microsoft charsets

The `recode` program provides various IBM or Microsoft code pages (see Chapter 6 [Tabular], page 36). An easy way to find them all at once out of the `recode` program itself is through the command:

```
recode -l | egrep -i '(CP|IBM)[0-9]'
```

But also, see few special charsets presented in the incoming sections.

8.1 EBCDIC code

This charset is the IBM's External Binary Coded Decimal for Interchange Coding. This is an eight bits code. The following three variants were implemented in `recode` independently of RFC 1345:

`EBCDIC`
: In `recode`, the `us..ebcdic` conversion is identical to '`dd conv=ebcdic`' conversion, and `recode ebcdic..us` conversion is identical to '`dd conv=ascii`' conversion. This charset also represents the way Control Data Corporation relates EBCDIC to 8-bits ASCII.

`EBCDIC-CCC`
: In `recode`, the `us..ebcdic-ccc` or `ebcdic-ccc..us` conversions represent the way Concurrent Computer Corporation (formerly Perkin Elmer) relates EBCDIC to 8-bits ASCII.

`EBCDIC-IBM`
: In `recode`, the `us..ebcdic-ibm` conversion is *almost* identical to the GNU '`dd conv=ibm`' conversion. Given the exact '`dd conv=ibm`' conversion table, `recode` once said:

```
Codes  91 and 213 both recode to 173
Codes  93 and 229 both recode to 189
No character recodes to  74
No character recodes to 106
```

So I arbitrarily chose to recode 213 by 74 and 229 by 106. This makes the `EBCDIC-IBM` recoding reversible, but this is not necessarily the best correction. In any case, I think that GNU `dd` should be amended. `dd` and `recode` should ideally agree on the same correction. So, this table might change once again.

RFC 1345 brings into `recode` 15 other EBCDIC charsets, and 21 other charsets having EBCDIC in at least one of their alias names. You can get a list of all these by executing:

```
recode -l | grep -i ebcdic
```

Note that `recode` may convert a pure stream of EBCDIC characters, but it does not know how to handle binary data between records which is sometimes used to delimit them and build physical blocks. If end of lines are not marked, fixed record size may produce something readable, but `VB` or `VBS` blocking is likely to yield some garbage in the converted results.

8.2 IBM's PC code

This charset is available in `recode` under the name `IBM-PC`, with `dos`, `MSDOS` and `pc` as acceptable aliases. The shortest way of specifying it in `recode` is `pc`.

The charset is aimed towards a PC microcomputer from IBM or any compatible. This is an eight-bit code. This charset is fairly old in `recode`, its tables were produced a long while ago by mere inspection of a printed chart of the IBM-PC codes and glyph.

It has `CR-LF` as its implied surface. This means that, if the original end of lines have to be preserved while going out of `IBM-PC`, they should currently be added back through the usage of a surface on the other charset, or better, just never removed. Here are examples for both cases:

```
recode pc..l2/cl < input > output
recode pc/..l2 < input > output
```

RFC 1345 brings into `recode` 44 '`IBM`' charsets or code pages, and also 8 other code pages. You can get a list of these all these by executing:[1]

```
recode -l | egrep -i '(CP|IBM)[0-9]'
```

All charset or aliases beginning with letters '`CP`' or '`IBM`' also have `CR-LF` as their implied surface. The same is true for a purely numeric alias in the same family. For example, all of `819`, `CP819` and `IBM819` imply `CR-LF` as a surface. Note that `ISO-8859-1` does *not* imply a surface, despite it shares the same tabular data as `819`.

There are a few discrepancies between this `IBM-PC` charset and the very similar RFC 1345 charset `ibm437`, which have not been analysed yet, so the charsets are being kept separate for now. This might change in the future, and the `IBM-PC` charset might disappear. Wizards would be interested in comparing the output of these two commands:

```
recode -vh IBM-PC..Latin-1
recode -vh IBM437..Latin-1
```

The first command uses the charset prior to RFC 1345 introduction. Both methods give different recodings. These differences are annoying, the fuzziness will have to be explained and settle down one day.

8.3 Unisys' Icon code

This charset is available in `recode` under the name `Icon-QNX`, with `QNX` as an acceptable alias.

The file is using Unisys' Icon way to represent diacritics with code 25 escape sequences, under the system QNX. This is a seven-bit code, even if eight-bit codes can flow through as part of IBM-PC charset.

---

[1] On DOS/Windows, stock shells do not know that apostrophes quote special characters like `|`, so one need to use double quotes instead of apostrophes.

9 Charsets for CDC machines

What is now `recode` evolved out, through many transformations really, from a set of programs which were originally written in *COMPASS*, Control Data Corporation's assembler, with bits in FORTRAN, and later rewritten in CDC 6000 Pascal. The CDC heritage shows by the fact some old CDC charsets are still supported.

The `recode` author used to be familiar with CDC Scope-NOS/BE and Kronos-NOS, and many CDC formats. Reading CDC tapes directly on other machines is often a challenge, and `recode` does not always solve it. It helps having tapes created in coded mode instead of binary mode, and using `S` (Stranger) tapes instead of `I` (Internal) tapes. ANSI labels and multi-file tapes might be the source of trouble. There are ways to handle a few Cyber Record Manager formats, but some of them might be quite difficult to decode properly after the transfer is done.

The `recode` program is usable only for a small subset of NOS text formats, and surely not with binary textual formats, like `UPDATE` or `MODIFY` sources, for example. `recode` is not especially suited for reading 8/12 or 56/60 packing, yet this could easily arranged if there was a demand for it. It does not have the ability to translate Display Code directly, as the ASCII conversion implied by tape drivers or FTP does the initial approximation. `recode` can decode 6/12 caret notation over Display Code already mapped to ASCII.

9.1 Control Data's Display Code

This code is not available in `recode`, but repeated here for reference. This is a 6-bit code used on CDC mainframes.

```
Octal display code to graphic       Octal display code to octal ASCII

00  :    20  P    40  5    60  #    00 072  20 120  40 065  60 043
01  A    21  Q    41  6    61  [    01 101  21 121  41 066  61 133
02  B    22  R    42  7    62  ]    02 102  22 122  42 067  62 135
03  C    23  S    43  8    63  %    03 103  23 123  43 070  63 045
04  D    24  T    44  9    64  "    04 104  24 124  44 071  64 042
05  E    25  U    45  +    65  _    05 105  25 125  45 053  65 137
06  F    26  V    46  -    66  !    06 106  26 126  46 055  66 041
07  G    27  W    47  *    67  &    07 107  27 127  47 052  67 046
10  H    30  X    50  /    70  '    10 110  30 130  50 057  70 047
11  I    31  Y    51  (    71  ?    11 111  31 131  51 050  71 077
12  J    32  Z    52  )    72  <    12 112  32 132  52 051  72 074
13  K    33  0    53  $    73  >    13 113  33 060  53 044  73 076
14  L    34  1    54  =    74  @    14 114  34 061  54 075  74 100
15  M    35  2    55       75  \    15 115  35 062  55 040  75 134
16  N    36  3    56  ,    76  ^    16 116  36 063  56 054  76 136
17  O    37  4    57  .    77  ;    17 117  37 064  57 056  77 073
```

In older times, *:* used octal 63, and octal 0 was not a character. The table above shows the ASCII glyph interpretation of codes 60 to 77, yet these 16 codes were once defined differently.

There is no explicit end of line in Display Code, and the Cyber Record Manager introduced many new ways to represent them, the traditional end of lines being reachable by setting `RT` to '`Z`'. If 6-bit bytes in a file are sequentially counted from 1, a traditional end of line does exist if bytes $10*n+9$ and $10n+10$ are both zero for a given n, in which case these two bytes are not to be interpreted as *::*. Also, up to 9 immediately preceeding zero bytes, going backward, are to be considered as part of the end of line and not interpreted as *:*[1].

9.2 ASCII 6/12 from NOS

This charset is available in `recode` under the name `CDC-NOS`, with `NOS` as an acceptable alias.

This is one of the charsets in use on CDC Cyber NOS systems to represent ASCII, sometimes named *NOS 6/12* code for coding ASCII. This code is also known as *caret ASCII*. It is based on a six bits character set in which small letters and control characters are coded using a `^` escape and, sometimes, a `@` escape.

The routines given here presume that the six bits code is already expressed in ASCII by the communication channel, with embedded ASCII `^` and `@` escapes.

Here is a table showing which characters are being used to encode each ASCII character.

000	^5	020	^#	040		060	0	100	@A	120	P	140	@G	160	^P
001	^6	021	^[	041	!	061	1	101	A	121	Q	141	^A	161	^Q
002	^7	022	^]	042	"	062	2	102	B	122	R	142	^B	162	^R
003	^8	023	^%	043	#	063	3	103	C	123	S	143	^C	163	^S
004	^9	024	^"	044	$	064	4	104	D	124	T	144	^D	164	^T
005	^+	025	^_	045	%	065	5	105	E	125	U	145	^E	165	^U
006	^-	026	^!	046	&	066	6	106	F	126	V	146	^F	166	^V
007	^*	027	^&	047	'	067	7	107	G	127	W	147	^G	167	^W
010	^/	030	^'	050	(	070	8	110	H	130	X	150	^H	170	^X
011	^(	031	^?	051	)	071	9	111	I	131	Y	151	^I	171	^Y
012	^)	032	^<	052	*	072	@D	112	J	132	Z	152	^J	172	^Z
013	^$	033	^>	053	+	073	;	113	K	133	[	153	^K	173	^0
014	^=	034	^@	054	,	074	<	114	L	134	\	154	^L	174	^1
015	^	035	^\	055	-	075	=	115	M	135	]	155	^M	175	^2
016	^,	036	^^	056	.	076	>	116	N	136	@B	156	^N	176	^3
017	^.	037	^;	057	/	077	?	117	O	137	_	157	^O	177	^4

9.3 ASCII "bang bang"

This charset is available in `recode` under the name `Bang-Bang`.

This code, in use on Cybers at Université de Montréal mainly, served to code a lot of French texts. The original name of this charset is *ASCII codé Display*. This code is also known as *Bang-bang*. It is based on a six bits character set in which capitals, French

[1] This convention replaced an older one saying that up to 4 immediately preceeding *pairs* of zero bytes, going backward, are to be considered as part of the end of line and not interpreted as *::*.

diacritics and a few others are coded using an *!* escape followed by a single character, and control characters using a double *!* escape followed by a single character.

The routines given here presume that the six bits code is already expressed in ASCII by the communication channel, with embedded ASCII *!* escapes.

Here is a table showing which characters are being used to encode each ASCII character.

000	!!@	020	!!P	040		060	0	100	@	120	!P	140	!@	160	P
001	!!A	021	!!Q	041	!"	061	1	101	!A	121	!Q	141	A	161	Q
002	!!B	022	!!R	042	"	062	2	102	!B	122	!R	142	B	162	R
003	!!C	023	!!S	043	#	063	3	103	!C	123	!S	143	C	163	S
004	!!D	024	!!T	044	$	064	4	104	!D	124	!T	144	D	164	T
005	!!E	025	!!U	045	%	065	5	105	!E	125	!U	145	E	165	U
006	!!F	026	!!V	046	&	066	6	106	!F	126	!V	146	F	166	V
007	!!G	027	!!W	047	'	067	7	107	!G	127	!W	147	G	167	W
010	!!H	030	!!X	050	(	070	8	110	!H	130	!X	150	H	170	X
011	!!I	031	!!Y	051	)	071	9	111	!I	131	!Y	151	I	171	Y
012	!!J	032	!!Z	052	*	072	:	112	!J	132	!Z	152	J	172	Z
013	!!K	033	!![	053	+	073	;	113	!K	133	[	153	K	173	![
014	!!L	034	!!\	054	,	074	<	114	!L	134	\	154	L	174	!\
015	!!M	035	!!]	055	-	075	=	115	!M	135	]	155	M	175	!]
016	!!N	036	!!^	056	.	076	>	116	!N	136	^	156	N	176	!^
017	!!O	037	!!_	057	/	077	?	117	!O	137	_	157	O	177	!_

10 Other micro-computer charsets

The `NeXT` charset, which used to be especially provided in releases of `recode` before 3.5, has been integrated since as one RFC 1345 table.

10.1 Apple's Macintosh code

This charset is available in `recode` under the name `Apple-Mac`. The shortest way of specifying it in `recode` is `ap`.

The charset is aimed towards a Macintosh micro-computer from Apple. This is an eight bit code. The file is the data fork only. This charset is fairly old in `recode`, its tables were produced a long while ago by mere inspection of a printed chart of the Macintosh codes and glyph.

It has `CR` as its implied surface. This means that, if the original end of lines have to be preserved while going out of `Apple-Mac`, they should currently be added back through the usage of a surface on the other charset, or better, just never removed. Here are examples for both cases:

```
recode ap..l2/cr < input > output
recode ap/..l2 < input > output
```

RFC 1345 brings into `recode` 2 other Macintosh charsets. You can discover them by using `grep` over the output of '`recode -l`':

```
recode -l | grep -i mac
```

Charsets `macintosh` and `macintosh_ce`, as well as their aliases `mac` and `macce` also have `CR` as their implied surface.

There are a few discrepancies between the `Apple-Mac` charset and the very similar RFC 1345 charset `macintosh`, which have not been analysed yet, so the charsets are being kept separate for now. This might change in the future, and the `Apple-Mac` charset might disappear. Wizards would be interested in comparing the output of these two commands:

```
recode -vh Apple-Mac..Latin-1
recode -vh macintosh..Latin-1
```

The first command use the charset prior to RFC 1345 introduction. Both methods give different recodings. These differences are annoying, the fuzziness will have to be explained and settle down one day.

As a side note, some people ask if there is a Macintosh port of the `recode` program. I'm not aware of any. I presume that if the tool fills a need for Macintosh users, someone will port it one of these days?

10.2 Atari ST code

This charset is available in `recode` under the name `AtariST`.

This is the character set used on the Atari ST/TT/Falcon. This is similar to `IBM-PC`, but differs in some details: it includes some more accented characters, the graphic characters are mostly replaced by Hebrew characters, and there is a true German *`sharp s`* different from Greek *`beta`*.

About the end-of-line conversions: the canonical end-of-line on the Atari is '`\r\n`', but unlike `IBM-PC`, the OS makes no difference between text and binary input/output; it is up to the application how to interpret the data. In fact, most of the libraries that come with compilers can grok both '`\r\n`' and '`\n`' as end of lines. Many of the users who also have access to Unix systems prefer '`\n`' to ease porting Unix utilities. So, for easing reversibility, `recode` tries to let '`\r`' undisturbed through recodings.

11 Various other charsets

Even if these charsets were originally added to recode for handling texts written in French, they find other uses. We did use them a lot for writing French diacriticised texts in the past, so recode knows how to handle these particularly well for French texts.

11.1 World Wide Web representations

This charset is available in recode under the name HTML, with h4 as an acceptable alias.

HTML texts used by World Wide Web often use special sequences, beginning with an ampersand & and ending with a semicolon ;, for representing characters. The sequence may itself start with a number sigh # and be followed by digits, so forming a *numeric character reference*, or else be an alphabetic identifier, so forming a *character entity reference*.

Printable characters from Latin-1 may be used directly in an HTML text. However, partly because people have deficient keyboards, partly because people want to transmit HTML texts over non 8-bit clean channels while not using MIME, it is common (yet debatable) to use character entity references even for Latin-1 characters, when they fall outside ASCII (that is, when they have the 8th bit set).

When you recode from another charset to HTML, beware that all occurrences of double quotes, ampersands, and left or right angle brackets are translated into special sequences. However, in practice, people often use ampersands and angle brackets in the other charset for introducing HTML commands, compromising it: it is not pure HTML, not it is pure other charset. These particular translations can be rather inconvenient, they may be specifically inhibited through the command option '-d' (see Section 3.7 [Mixed], page 17).

Codes not having a mnemonic entity are output by recode using the '&#*nnn*;' notation, where *nnn* is a decimal representation of the UCS code value. When there is an entity name for a character, it is always preferred over a numeric character reference. ASCII printable characters are always generated directly. So is the newline. While reading HTML, recode supports numeric character reference as alternate writings, even when written as hexadecimal numbers, as in '&#xfffd'. This is documented in:

http://www.w3.org/TR/REC-html40/intro/sgmltut.html#h-3.2.3

About levels of HTML, Franois Yergeau yergeau@alis.com writes:

- HTML 1.0 was never really documented.
- HTML 2.0 entities are listed in RFC 1866. Basically, there is an entity for each *alphabetical* character in the right part of ISO 8859-1. In addition, there are four entities for syntax-significant ASCII characters: '&', '>', '<' and '"'.
- RFC 2070 (HTML-i18n) added entities to cover the whole right part of ISO 8859-1. The list is conveniently accessible at http://www.alis.com:8085/ietf/html/html-latin1.sgml. In addition, four i18n-related entities were added: '‌' ('‌'), '‍' ('‍'), '‎' ('&#8206') and '‏' ('‏').
- HTML 3.2 (http://www.w3.org/TR/REC-html32.html) took up the full Latin-1 list but not the i18n-related entities from RFC 2070.
- HTML 4.0 (http://www.w3.org/TR/REC-html40/) has the whole Latin-1 list, a set of entities for symbols, mathematical symbols, and Greek letters, and another set for

markup-significant and internationalization characters comprising the 4 ASCII entities, the 4 i18n-related from RFC 2070 plus some more.

When `recode` translates to HTML, the translation occurs according to `http://www.w3.org/TR/REC-html40/sgml/entities.html`. It is also assumed that RFC 1866 has an equivalent contents. When translating *from* HTML, `recode` accepts some alternative special sequences, to be forgiving when files use older HTML tables.

The `recode` program can be used to *normalise* an HTML file using oldish conventions. For example, it accepts '`&AE;`', as this once was a valid writing, somewhere. However, it should always produce '`Æ`' instead of '`&AE;`'. Yet, this is not completely true. If one does:

```
recode h3..h3 < input
```

the operation will be optimised into a mere copy, and you can get '`&AE;`' this way, if you had some in your input file. But if you explicitly defeat the optimisation, like this maybe:

```
recode h3..u2,u2..h3 < input
```

then '`&AE;`' should be normalised into '`Æ`' by the operation.

11.2 LaTeX macro calls

This charset is available in `recode` under the name `LaTeX` and has `ltex` as an alias. It is used for ASCII files coded to be read by LaTeX or, in certain cases, by TeX.

Whenever you recode from another charset to `LaTeX`, beware that all occurrences of backslashes `\` are translated into the string '`\backslash{}`'. However, in practice, people often use backslashes in the other charset for introducing TeX commands, compromising it: it is not pure TeX, nor it is pure other charset. This translation of backslashes into '`\backslash{}`' can be rather inconvenient, it may be inhibited through the command option '`-d`' (see Section 3.7 [Mixed], page 17).

11.3 GNU project documentation files

This charset is available in `recode` under the name `Texinfo` and has `texi` and `ti` for aliases. It is used by the GNU project for its documentation. Texinfo files may be converted into Info files by the `makeinfo` program and into nice printed manuals by the TeX system.

Even if `recode` may transform other charsets to Texinfo, it may not read Texinfo files yet. In these times, usages are also changing between versions of Texinfo, and `recode` only partially succeeds in correctly following these changes. So, for now, Texinfo support in `recode` should be considered as work still in progress (!).

11.4 African charsets

Some African character sets are available for a few languages, when these are heavily used in countries where French is also currently spoken.

One African charset is usable for Bambara, Ewondo and Fulfude, as well as for French. This charset is available in `recode` under the name `AFRFUL-102-BPI_OCIL`. Accepted aliases are `bambara`, `bra`, `ewondo` and `fulfude`. Transliterated forms of the same are available under the name AFRFUL-103-BPI_OCIL. Accepted aliases are t-bambara, t-bra, t-ewondo and t-fulfude.

Another African charset is usable for Lingala, Sango and Wolof, as well as for French. This charset is available in recode under the name AFRLIN-104-BPI_OCIL. Accepted aliases are lingala, lin, sango and wolof. Transliterated forms of the same are available under the name AFRLIN-105-BPI_OCIL. Accepted aliases are t-lingala, t-lin, t-sango and t-wolof.

To ease exchange with ISO-8859-1, there is a charset conveying transliterated forms for Latin-1 in a way which is compatible with the other African charsets in this series. This charset is available in recode under the name AFRL1-101-BPI_OCIL. Accepted aliases are t-fra and t-francais.

11.5 Cyrillic charsets

The following Cyrillic charsets are already available in recode through RFC 1345 tables: CP1251 with aliases 1251, ms-cyrl and windows-1251; CSN_369103 with aliases ISO-IR-139 and KOI8_L2; ECMA-cyrillic with aliases ECMA-113, ECMA-113:1986 and iso-ir-111, IBM880 with aliases 880, CP880 and EBCDIC-Cyrillic; INIS-cyrillic with alias iso-ir-51; ISO-8859-5 with aliases cyrillic, ISO-8859-5:1988 and iso-ir-144; KOI-7; KOI-8 with alias GOST_19768-74; KOI8-R; KOI8-RU and finally KOI8-U.

There seems to remain some confusion in Cyrillic charsets, and because a few users requested it repeatedly, recode now offers special services in that area. Consider these charsets as experimental and debatable, as the extraneous tables describing them are still a bit fuzzy or non-standard. Hopefully, in the long run, Cyrillic will be covered in Keld Simonsen's works to the satisfaction of everybody, and this section will merely disappear.

KEYBCS2
: This charset is available under the name KEYBCS2, with Kamenicky as an accepted alias.

CORK
: This charset is available under the name CORK, with T1 as an accepted alias.

KOI-8_CS2
: This charset is available under the name KOI-8_CS2.

11.6 Easy French conventions

This charset is available in recode under the name Texte and has txte for an alias. It is a seven bits code, identical to ASCII-BS, save for French diacritics which are noted using a slightly different convention.

At text entry time, these conventions provide a little speed up. At read time, they slightly improve the readability over a few alternate ways of coding diacritics. Of course, it would better to have a specialised keyboard to make direct eight bits entries and fonts for immediately displaying eight bit ISO Latin-1 characters. But not everybody is so fortunate. In a few mailing environments, and sadly enough, it still happens that the eight bit is often willing-fully destroyed.

Easy French has been in use in France for a while. I only slightly adapted it (the diaeresis option) to make it more comfortable to several usages in Québec originating from Université de Montréal. In fact, the main problem for me was not to necessarily to invent Easy French, but to recognise the "best" convention to use, (best is not being defined, here) and to try to solve the main pitfalls associated with the selected convention. Shortly said, we have:

`e'` for `e` (and some other vowels) with an acute accent,

`` e` `` for `e` (and some other vowels) with a grave accent,

`e^` for `e` (and some other vowels) with a circumflex accent,

`e"` for `e` (and some other vowels) with a diaeresis,

`c,` for `c` with a cedilla.

There is no attempt at expressing the `ae` and `oe` diphthongs. French also uses tildes over `n` and `a`, but seldomly, and this is not represented either. In some countries, `:` is used instead of `"` to mark diaeresis. `recode` supports only one convention per call, depending on the '`-c`' option of the `recode` command. French quotes (sometimes called "angle quotes") are noted the same way English quotes are noted in TeX, *id est* by ``` `` ``` and `''`. No effort has been put to preserve Latin ligatures (æ, œ) which are representable in several other charsets. So, these ligatures may be lost through Easy French conventions.

The convention is prone to losing information, because the diacritic meaning overloads some characters that already have other uses. To alleviate this, some knowledge of the French language is boosted into the recognition routines. So, the following subtleties are systematically obeyed by the various recognisers.

1. A comma which follows a `c` is interpreted as a cedilla only if it is followed by one of the vowels `a`, `o` or `u`.
2. A single quote which follows a `e` does not necessarily means an acute accent if it is followed by a single other one. For example:

   `e'` will give an `e` with an acute accent.

   `e''` will give a simple `e`, with a closing quotation mark.

   `e'''` will give an `e` with an acute accent, followed by a closing quotation mark.

   There is a problem induced by this convention if there are English quotations with a French text. In sentences like:

   ```
   There's a meeting at Archie's restaurant.
   ```

   the single quotes will be mistaken twice for acute accents. So English contractions and suffix possessives could be mangled.
3. A double quote or colon, depending on '`-c`' option, which follows a vowel is interpreted as diaeresis only if it is followed by another letter. But there are in French several words that *end* with a diaeresis, and the `recode` library is aware of them. There are words ending in "igue", either feminine words without a relative masculine (besaiguë and ciguë), or feminine words with a relative masculine[1] (aiguë, ambiguë, contiguë, exiguë, subaiguë and suraiguë). There are also words not ending in "igue", but instead, either

[1] There are supposed to be seven words in this case. So, one is missing.

ending by "i"[2] (aï, congaï, goï, haïkaï, inouï, saï, samuraï, thaï and tokaï), ending by "e" (canoë) or ending by "u"[3] (Esaü).

Just to complete this topic, note that it would be wrong to make a rule for all words ending in "igue" as needing a diaerisis, as there are counter-examples (becfigue, bèsigue, bigue, bordigue, bourdigue, brigue, contre-digue, digue, d'intrigue, fatigue, figue, garrigue, gigue, igue, intrigue, ligue, prodigue, sarigue and zigue).

11.7 Mule as a multiplexed charset

This version of `recode` barely starts supporting multiplexed or super-charsets, that is, those encoding methods by which a single text stream may contain a combination of more than one constituent charset. The only multiplexed charset in `recode` is `Mule`, and even then, it is only very partially implemented: the only correspondence available is with `Latin-1`. The author fastly implemented this only because he needed this for himself. However, it is intended that Mule support to become more real in subsequent releases of `recode`.

Multiplexed charsets are not to be confused with mixed charset texts (see Section 3.7 [Mixed], page 17). For mixed charset input, the rules allowing to distinguish which charset is current, at any given place, are kind of informal, and driven from the semantics of what the file contains. On the other side, multiplexed charsets are *designed* to be interpreted fairly precisely, and quite independently of any informational context.

The spelling `Mule` originally stands for *multilingual enhancement to GNU Emacs*, it is the result of a collective effort orchestrated by Handa Ken'ichi since 1993. When `Mule` got rewritten in the main development stream of GNU Emacs 20, the FSF renamed it `MULE`, meaning *multilingual environment in GNU Emacs*. Even if the charset `Mule` is meant to stay internal to GNU Emacs, it sometimes breaks loose in external files, and as a consequence, a recoding tool is sometimes needed. Within Emacs, `Mule` comes with `Leim`, which stands for *libraries of emacs input methods*. One of these libraries is named `quail`[4].

---

[2] Look at one of the following sentences (the second has to be interpreted with the '`-c`' option):

```
"Ai"e!  Voici le proble`me que j'ai"
Ai:e!  Voici le proble`me que j'ai:
```

There is an ambiguity between an aï, the small animal, and the indicative future of *avoir* (first person singular), when followed by what could be a diaeresis mark. Hopefully, the case is solved by the fact that an apostrophe always precedes the verb and almost never the animal.

[3] I did not pay attention to proper nouns, but this one showed up as being fairly evident.

[4] Usually, quail means quail egg in Japanese, while egg alone is usually chicken egg. Both quail egg and chicken egg are popular food in Japan. The `quail` input system has been named because it is smaller that the previous `EGG` system. As for `EGG`, it is the translation of `TAMAGO`. This word comes from the Japanese sentence *takusan matasete gomennasai*, meaning *sorry to have let you wait so long*. Of course, the publication of `EGG` has been delayed many times... (Story by Takahashi Naoto)

12 All about surfaces

The *trivial surface* consists of using a fixed number of bits (often eight) for each character, the bits together hold the integer value of the index for the character in its charset table. There are many kinds of surfaces, beyond the trivial one, all having the purpose of increasing selected qualities for the storage or transmission. For example, surfaces might increase the resistance to channel limits (`Base64`), the transmission speed (`gzip`), the information privacy (`DES`), the conformance to operating system conventions (`CR-LF`), the blocking into records (`VB`), and surely other things as well[1]. Many surfaces may be applied to a stream of characters from a charset, the order of application of surfaces is important, and surfaces should be removed in the reverse order of their application.

Even if surfaces may generally be applied to various charsets, some surfaces were specifically designed for a particular charset, and would not make much sense if applied to other charsets. In such cases, these conceptual surfaces have been implemented as `recode` charsets, instead of as surfaces. This choice yields to cleaner syntax and usage. See Chapter 5 [Universal], page 31.

Surfaces are implemented within `recode` as special charsets which may only transform to or from the `data` special charset. Clever users may use this knowledge for writing surface names in requests exactly as if they were pure charsets, when the only need is to change surfaces without any kind of recoding between real charsets. In such contexts, `data` may also be used as if it were some kind of generic, anonymous charset: the request '`data..`*surface*' merely adds the given *surface*, while the request '*surface*`..data`' removes it.

We are only beginning to experiment with surfaces in `recode`, but the concept opens the doors to many avenues; it is not clear yet which ones are worth pursuing, and which should be abandoned. This chapter presents all surfaces currently available.

12.1 Permuting groups of bytes

A permutation is a surface transformation which reorders groups of eight-bit bytes. A *21* permutation exchanges pairs of successive bytes. If the text contains an odd number of bytes, the last byte is merely copied. An *4321* permutation inverts the order of quadruples of bytes. If the text does not contains a multiple of four bytes, the remaining bytes are nevertheless permuted as *321* if there are three bytes, *21* if there are two bytes, or merely copied otherwise.

`21`
: This surface is available in `recode` under the name `21-Permutation` and has `swabytes` for an alias.

`4321`
: This surface is available in `recode` under the name `4321-Permutation`.

12.2 Representation for end of lines

The same charset might slightly differ, from one system to another, for the single fact that end of lines are not represented identically on all systems. The representation for an

[1] These are mere examples to explain the concept, `recode` only has `Base64` and `CR-LF`, actually.

end of line within `recode` is the `ASCII` or `UCS` code with value 10, or *LF*. Other conventions for representing end of lines are available through surfaces.

`CR`
: This convention is popular on Apple's Macintosh machines. When this surface is applied, each line is terminated by *CR*, which has `ASCII` value 13. Unless the library is operating in strict mode, adding or removing the surface will in fact *exchange* *CR* and *LF*, for better reversibility. However, in strict mode, the exchange does not happen, any *CR* will be copied verbatim while applying the surface, and any *LF* will be copied verbatim while removing it.

  This surface is available in `recode` under the name `CR`, it does not have any aliases. This is the implied surface for the Apple Macintosh related charsets.

`CR-LF`
: This convention is popular on Microsoft systems running on IBM PCs and compatible. When this surface is applied, each line is terminated by a sequence of two characters: one *CR* followed by one *LF*, in that order.

  For compatibility with oldish MS-DOS systems, removing a `CR-LF` surface will discard the first encountered *C-z*, which has `ASCII` value 26, and everything following it in the text. Adding this surface will not, however, append a *C-z* to the result.

  This surface is available in `recode` under the name `CR-LF` and has `cl` for an alias. This is the implied surface for the IBM or Microsoft related charsets or code pages.

Some other charsets might have their own representation for an end of line, which is different from *LF*. For example, this is the case of various `EBCDIC` charsets, or `Icon-QNX`. The recoding of end of lines is intimately tied into such charsets, it is not available separately as surfaces.

12.3 MIME contents encodings

RFC 1521 defines two 7-bit surfaces, meant to prepare 8-bit messages for transmission. Base64 is especially usable for binary entities, while Quoted-Printable is especially usable for text entities, in those case the lower 128 characters of the underlying charset coincide with ASCII.

`Base64`
: This surface is available in `recode` under the name `Base64`, with `b64` and `64` as acceptable aliases.

`Quoted-Printable`
: This surface is available in `recode` under the name `Quoted-Printable`, with `quote-printable` and `QP` as acceptable aliases.

Note that `UTF-7`, which may be also considered as a MIME surface, is provided as a genuine charset instead, as it necessary relates to `UCS-2` and nothing else. See Section 5.3 [UTF-7], page 32.

A little historical note, also showing the three levels of acceptance of Internet standards. MIME changed from a “Proposed Standard” to a “Draft Standard” in 1993, and only became a “Full Standard” during 1996-12.

12.4 Interpreted character dumps

Dumps are surfaces meant to express, in ways which are a bit more readable, the bit patterns used to represent characters. They allow the inspection or debugging of character streams, but also, they may assist a bit the production of C source code which, once compiled, would hold in memory a copy of the original coding. However, `recode` does not attempt, in any way, to produce complete C source files in dumps. User hand editing or '`Makefile`' trickery is still needed for adding missing lines. Dumps may be given in decimal, hexadecimal and octal, and be based over chunks of either one, two or four eight-bit bytes. Formatting has been chosen to respect the C language syntax for number constants, with commas and newlines inserted appropriately.

However, when dumping two or four byte chunks, the last chunk may be incomplete. This is observable through the usage of narrower expression for that last chunk only. Such a shorter chunk would not be compiled properly within a C initialiser, as all members of an array share a single type, and so, have identical sizes.

`Octal-1`
: This surface corresponds to an octal expression of each input byte.

  It is available in `recode` under the name `Octal-1`, with `o1` and `o` as acceptable aliases.

`Octal-2`
: This surface corresponds to an octal expression of each pair of input bytes, except for the last pair, which may be short.

  It is available in `recode` under the name `Octal-2` and has `o2` for an alias.

`Octal-4`
: This surface corresponds to an octal expression of each quadruple of input bytes, except for the last quadruple, which may be short.

  It is available in `recode` under the name `Octal-4` and has `o4` for an alias.

`Decimal-1`
: This surface corresponds to an decimal expression of each input byte.

  It is available in `recode` under the name `Decimal-1`, with `d1` and `d` as acceptable aliases.

`Decimal-2`
: This surface corresponds to an decimal expression of each pair of input bytes, except for the last pair, which may be short.

  It is available in `recode` under the name `Decimal-2` and has `d2` for an alias.

`Decimal-4`
: This surface corresponds to an decimal expression of each quadruple of input bytes, except for the last quadruple, which may be short.

  It is available in `recode` under the name `Decimal-4` and has `d4` for an alias.

`Hexadecimal-1`
: This surface corresponds to an hexadecimal expression of each input byte.

  It is available in `recode` under the name `Hexadecimal-1`, with `x1` and `x` as acceptable aliases.

`Hexadecimal-2`
: This surface corresponds to an hexadecimal expression of each pair of input bytes, except for the last pair, which may be short.

  It is available in `recode` under the name `Hexadecimal-2`, with `x2` for an alias.

`Hexadecimal-4`
: This surface corresponds to an hexadecimal expression of each quadruple of input bytes, except for the last quadruple, which may be short.

  It is available in `recode` under the name `Hexadecimal-4`, with `x4` for an alias.

When removing a dump surface, that is, when reading a dump results back into a sequence of bytes, the narrower expression for a short last chunk is recognised, so dumping is a fully reversible operation. However, in case you want to produce dumps by other means than through `recode`, beware that for decimal dumps, the library has to rely on the number of spaces to establish the original byte size of the chunk.

Although the library might report reversibility errors, removing a dump surface is a rather forgiving process: one may mix bases, group a variable number of data per source line, or use shorter chunks in places other than at the far end. Also, source lines not beginning with a number are skipped. So, `recode` should often be able to read a whole C header file, wrapping the results of a previous dump, and regenerate the original byte string.

12.5 Artificial data

A few pseudo-surfaces exist to generate debugging data out of thin air. These surfaces are only meant for the expert `recode` user, and are only useful in a few contexts, like for generating binary permutations from the recoding or acting on them.

Debugging surfaces, *when removed*, insert their generated data at the beginning of the output stream, and copy all the input stream after the generated data, unchanged. This strange removal constraint comes from the fact that debugging surfaces are usually specified in the *before* position instead of the *after* position within a request. With debugging surfaces, one often recodes file '`/dev/null`' in filter mode. Specifying many debugging surfaces at once has an accumulation effect on the output, and since surfaces are removed from right to left, each generating its data at the beginning of previous output, the net effect is an *impression* that debugging surfaces are generated from left to right, each appending to the result of the previous. In any case, any real input data gets appended after what was generated.

`test7`
: When removed, this surface produces 128 single bytes, the first having value 0, the second having value 1, and so forth until all 128 values have been generated.

`test8`
: When removed, this surface produces 256 single bytes, the first having value 0, the second having value 1, and so forth until all 256 values have been generated.

`test15`
: When removed, this surface produces 64509 double bytes, the first having value 0, the second having value 1, and so forth until all values have been generated, but excluding risky `UCS-2` values, like all codes from the surrogate `UCS-2` area (for `UTF-16`), the byte order mark, and values known as invalid `UCS-2`.

`test16` When removed, this surface produces 65536 double bytes, the first having value 0, the second having value 1, and so forth until all 65536 values have been generated.

As an example, the command '`recode l5/test8..dump < /dev/null`' is a convoluted way to produce an output similar to '`recode -lf l5`'. It says to generate all possible 256 bytes and interpret them as `ISO-8859-9` codes, while converting them to `UCS-2`. Resulting `UCS-2` characters are dumped one per line, accompanied with their explicative name.

13 Internal aspects

The incoming explanations of the internals of `recode` should help people who want to dive into `recode` sources for adding new charsets. Adding new charsets does not require much knowledge about the overall organisation of `recode`. You can rather concentrate of your new charset, letting the remainder of the `recode` mechanics take care of interconnecting it with all others charsets.

If you intend to play seriously at modifying `recode`, beware that you may need some other GNU tools which were not required when you first installing `recode`. If you modify or create any '`.l`' file, then you need Flex, and some better `awk` like `mawk`, GNU `awk`, or `nawk`. If you modify the documentation (and you should!), you need `makeinfo`. If you are really audacious, you may also want Perl for modifying tabular processing, then `m4`, Autoconf, Automake and `libtool` for adjusting configuration matters.

13.1 Overall organisation

The `recode` mechanics slowly evolved for many years, and it would be tedious to explain all problems I met and mistakes I did all along, yielding the current behaviour. Surely, one of the key choices was to stop trying to do all conversions in memory, one line or one buffer at a time. It has been fruitful to use the character stream paradigm, and the elementary recoding steps now convert a whole stream to another. Most of the control complexity in `recode` exists so that each elementary recoding step stays simple, making easier to add new ones. The whole point of `recode`, as I see it, is providing a comfortable nest for growing new charset conversions.

The main `recode` driver constructs, while initialising all conversion modules, a table giving all the conversion routines available (*single steps*) and for each, the starting charset and the ending charset. If we consider these charsets as being the nodes of a directed graph, each single step may be considered as oriented arc from one node to the other. A cost is attributed to each arc: for example, a high penalty is given to single steps which are prone to losing characters, a lower penalty is given to those which need studying more than one input character for producing an output character, etc.

Given a starting code and a goal code, `recode` computes the most economical route through the elementary recodings, that is, the best sequence of conversions that will transform the input charset into the final charset. To speed up execution, `recode` looks for subsequences of conversions which are simple enough to be merged, and then dynamically creates new single steps to represent these mergings.

A *double step* in `recode` is a special concept representing a sequence of two single steps, the output of the first single step being the special charset `UCS-2`, the input of the second single step being also `UCS-2`. Special `recode` machinery dynamically produces efficient, reversible, merge-able single steps out of these double steps.

I made some statistics about how many internal recoding steps are required between any two charsets chosen at random. The initial recoding layout, before optimisation, always uses between 1 and 6 steps. Optimisation could sometimes produce mere copies, which are counted as no steps at all. In other cases, optimisation is unable to save any step. The number of steps after optimisation is currently between 0 and 6 steps. Of course, the *expected* number of steps is affected by optimisation: it drops from 2.2 to 0.9.[1] This means that `recode` uses a theoretical average of a bit less than one step per recoding job. This looks good. This was computed using reversible recodings. In strict mode, optimisation might be defeated somewhat. Number of steps run between 1 and 6, both before and after optimisation, and the expected number of steps decreases by a lesser amount, going from 2.2 to 1.3. This is still manageable.

13.2 Adding new charsets

The main part of `recode` is written in C, as are most single steps. A few single steps need to recognise sequences of multiple characters, they are often better written in Flex. It is easy for a programmer to add a new charset to `recode`. All it requires is making a few functions kept in a single '`.c`' file, adjusting '`Makefile.am`' and remaking `recode`.

One of the function should convert from any previous charset to the new one. Any previous charset will do, but try to select it so you will not lose too much information while converting. The other function should convert from the new charset to any older one. You do not have to select the same old charset than what you selected for the previous routine. Once again, select any charset for which you will not lose too much information while converting.

If, for any of these two functions, you have to read multiple bytes of the old charset before recognising the character to produce, you might prefer programming it in Flex in a separate '`.l`' file. Prototype your C or Flex files after one of those which exist already, so to keep the sources uniform. Besides, at `make` time, all '`.l`' files are automatically merged into a single big one by the script '`mergelex.awk`'.

There are a few hidden rules about how to write new `recode` modules, for allowing the automatic creation of '`decsteps.h`' and '`initsteps.h`' at `make` time, or the proper merging of all Flex files. Mimetism is a simple approach which relieves me of explaining all these rules! Start with a module closely resembling what you intend to do. Here is some advice for picking up a model. First decide if your new charset module is to be be driven by algorithms rather than by tables. For algorithmic recodings, see '`iconqnx.c`' for C code, or '`txtelat1.l`' for Flex code. For table driven recodings, see '`ebcdic.c`' for one-to-one style recodings, '`lat1html.c`' for one-to-many style recodings, or '`atarist.c`' for double-step style recodings. Just select an example from the style that better fits your application.

Each of your source files should have its own initialisation function, named `module_`*charset*, which is meant to be executed *quickly* once, prior to any recoding. It should declare the name of your charsets and the single steps (or elementary recodings) you provide, by calling `declare_step` one or more times. Besides the charset names, `declare_step` expects a description of the recoding quality (see '`recodext.h`') and two functions you also provide.

The first such function has the purpose of allocating structures, pre-conditioning conversion tables, etc. It is also the way of further modifying the `STEP` structure. This function is executed if and only if the single step is retained in an actual recoding sequence. If you do not need such delayed initialisation, merely use `NULL` for the function argument.

[1] The above numbers were for `recode` version 3.4. After UCS additions, it drops from 2.3 to 1.1.

The second function executes the elementary recoding on a whole file. There are a few cases when you can spare writing this function:

- Some single steps do nothing else than a pure copy of the input onto the output, in this case, you can use the predefined function `file_one_to_one`, while having a delayed initialisation for presetting the `STEP` field `one_to_one` to the predefined value `one_to_same`.
- Some single steps are driven by a table which recodes one character into another; if the recoding does nothing else, you can use the predefined function `file_one_to_one`, while having a delayed initialisation for presetting the `STEP` field `one_to_one` with your table.
- Some single steps are driven by a table which recodes one character into a string; if the recoding does nothing else, you can use the predefined function `file_one_to_many`, while having a delayed initialisation for presetting the `STEP` field `one_to_many` with your table.

If you have a recoding table handy in a suitable format but do not use one of the predefined recoding functions, it is still a good idea to use a delayed initialisation to save it anyway, because `recode` option '`-h`' will take advantage of this information when available.

Finally, edit '`Makefile.am`' to add the source file name of your routines to the `C_STEPS` or `L_STEPS` macro definition, depending on the fact your routines is written in C or in Flex.

13.3 Adding new surfaces

Adding a new surface is technically quite similar to adding a new charset. See Section 13.2 [New charsets], page 67. A surface is provided as a set of two transformations: one from the predefined special charset `data` to the new surface, meant to apply the surface, the other from the new surface to the predefined special charset `data`, meant to remove the surface.

Internally in `recode`, function `declare_step` especially recognises when a charset is so related to `data`, and then takes appropriate actions so that charset gets indeed installed as a surface.

13.4 Comments on the library design

- Why a shared library?

  There are many different approaches to reduce system requirements to handle all tables needed in the `recode` library. One of them is to have the tables in an external format and only read them in on demand. After having pondered this for a while, I finally decided against it, mainly because it involves its own kind of installation complexity, and it is not clear to me that it would be as interesting as I first imagined.

  It looks more efficient to see all tables and algorithms already mapped into virtual memory from the start of the execution, yet not loaded in actual memory, than to go through many disk accesses for opening various data files once the program is already started, as this would be needed with other solutions. Using a shared library also has the indirect effect of making various algorithms handily available, right in the same modules providing the tables. This alleviates much the burden of the maintenance.

Of course, I would like to later make an exception for only a few tables, built locally by users for their own particular needs once `recode` is installed. `recode` should just go and fetch them. But I do not perceive this as very urgent, yet useful enough to be worth implementing.

Currently, all tables needed for recoding are precompiled into binaries, and all these binaries are then made into a shared library. As an initial step, I turned `recode` into a main program and a non-shared library, this allowed me to tidy up the API, get rid of all global variables, etc. It required a surprising amount of program source massaging. But once this cleaned enough, it was easy to use Gordon Matzigkeit's `libtool` package, and take advantage of the Automake interface to neatly turn the non-shared library into a shared one.

Sites linking with the `recode` library, whose system does not support any form of shared libraries, might end up with bulky executables. Surely, the `recode` library will have to be used statically, and might not very nicely usable on such systems. It seems that progress has a price for those being slow at it.

There is a locality problem I did not address yet. Currently, the `recode` library takes many cycles to initialise itself, calling each module in turn for it to set up associated knowledge about charsets, aliases, elementary steps, recoding weights, etc. *Then*, the recoding sequence is decided out of the command given. I would not be surprised if initialisation was taking a perceivable fraction of a second on slower machines. One thing to do, most probably not right in version 3.5, but the version after, would have `recode` to pre-load all tables and dump them at installation time. The result would then be compiled and added to the library. This would spare many initialisation cycles, but more importantly, would avoid calling all library modules, scattered through the virtual memory, and so, possibly causing many spurious page exceptions each time the initialisation is requested, at least once per program execution.

- Why not a central charset?

  It would be simpler, and I would like, if something like ISO 10646 was used as a turning template for all charsets in `recode`. Even if I think it could help to a certain extent, I'm still not fully sure it would be sufficient in all cases. Moreover, some people disagree about using ISO 10646 as the central charset, to the point I cannot totally ignore them, and surely, `recode` is not a mean for me to force my own opinions on people. I would like that `recode` be practical more than dogmatic, and reflect usage more than religions.

  Currently, if you ask `recode` to go from *charset1* to *charset2* chosen at random, it is highly probable that the best path will be quickly found as:

  charset1`..UCS-2..`*charset2*

  That is, it will almost always use the `UCS` as a trampoline between charsets. However, `UCS-2` will be immediately be optimised out, and *charset1..charset2* will often be performed in a single step through a permutation table generated on the fly for the circumstance[2].

[2] If strict mapping is requested, another efficient device will be used instead of a permutation.

In those few cases where `UCS-2` is not selected as a conceptual intermediate, I plan to study if it could be made so. But I guess some cases will remain where `UCS-2` is not a proper choice. Even if `UCS` is often the good choice, I do not intend to forcefully restrain `recode` around `UCS-2` (nor `UCS-4`) for now. We might come to that one day, but it will come out of the natural evolution of `recode`. It will then reflect a fact, rather than a preset dogma.

- Why not `iconv`?

  The `iconv` routine and library allows for converting characters from an input buffer to an input buffer, synchronously advancing both buffer cursors. If the output buffer is not big enough to receive all of the conversion, the routine returns with the input cursor set at the position where the conversion could later be resumed, and the output cursor set to indicate until where the output buffer has been filled. Despite this scheme is simple and nice, the `recode` library does not offer it currently. Why not?

  When long sequences of decodings, stepwise recodings, and re-encodings are involved, as it happens in true life, synchronising the input buffer back to where it should have stopped, when the output buffer becomes full, is a difficult problem. Oh, we could make it simpler at the expense of loosing space or speed: by inserting markers between each input character and counting them at the output end; by processing only one character in a time through the whole sequence; by repeatedly attempting to recode various subsets of the input buffer, binary searching on their length until the output just fits. The overhead of such solutions looks fully prohibitive to me, and the gain very minimal. I do not see a real advantage, nowadays, imposing a fixed length to an output buffer. It makes things so much simpler and efficient to just let the output buffer size float a bit.

  Of course, if the above problem was solved, the `iconv` library should be easily emulated, given that `recode` has similar knowledge about charsets, of course. This either solved or not, the `iconv` program remains trivial (given similar knowledge about charsets). I also presume that the `genxlt` program would be easy too, but I do not have enough detailed specifications of it to be sure.

  A lot of years ago, `recode` was using a similar scheme, and I found it rather hard to manage for some cases. I rethought the overall structure of `recode` for getting away from that scheme, and never regretted it. I perceive `iconv` as an artificial solution which surely has some elegances and virtues, but I do not find it really useful as it stands: one always has to wrap `iconv` into something more refined, extending it for real cases. From past experience, I think it is unduly hard to fully implement this scheme. It would be awkward that we do contortions for the sole purpose of implementing exactly its specification, without real, fairly sounded reasons (other then the fact some people once thought it was worth standardising). It is much better to immediately aim for the refinement we need, without uselessly forcing us into the dubious detour `iconv` represents.

  Some may argue that if `recode` was using a comprehensive charset as a turning template, as discussed in a previous point, this would make `iconv` easier to implement. Some may be tempted to say that the cases which are hard to handle are not really needed, nor interesting, anyway. I feel and fear a bit some pressure wanting that `recode` be split into the part that well fits the `iconv` model, and the part that does not fit, considering this second part less important, with the idea of dropping it one of these days, maybe. My guess is that users of the `recode` library, whatever its form, would not like to have such arbitrary limitations. In the long run, we should not have to explain to our users that some recodings may not be made available just because they do not fit the simple model we had in mind when we did it. Instead, we should try to stay opened to the difficulties of real life. There is still a lot of complex needs for Asian people, say, that `recode` does not currently address, while it should. Not only the doors should stay open, but we should force them wider!

Concept Index

Option Index

This is an alphabetical list of all command-line options accepted by `recode`.

Library Index

This is an alphabetical index of important functions, data structures, and variables in the recode library.

Charset and Surface Index

This is an alphabetical list of all the charsets and surfaces supported by `recode`, and their aliases.

H

I

J

K

L

Table of Contents