Liblouis User's and Programmer's Manual

for version 2.5.2, 17 December 2012

by John J. Boyer

This manual is for liblouis (version 2.5.2, 17 December 2012), a Braille Translation and Back-Translation Library derived from the Linux screen reader BRLTTY.

Copyright © 1999-2006 by the BRLTTY Team.

Copyright © 2004-2007 ViewPlus Technologies, Inc. www.viewplus.com.

Copyright © 2007,2009 Abilitiessoft, Inc. www.abilitiessoft.com.

This file is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser (or library) General Public License (LGPL) as published by the Free Software Foundation; either version 3, or (at your option) any later version.

This file is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser (or Library) General Public License LGPL for more details.

You should have received a copy of the GNU Lesser (or Library) General Public License (LGPL) along with this program; see the file COPYING. If not, write to the Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Table of Contents

1 Introduction 1

2 Test Programs 2

2.1 lou_debug 2
2.2 lou_trace 3
2.3 lou_checktable 4
2.4 lou_allround 4
2.5 lou_translate 5
2.6 lou_checkhyphens 5

3 How to Write Translation Tables 6

3.1 Hyphenation Tables 8
3.2 Character-Definition Opcodes 8
3.3 Braille Indicator Opcodes 10
3.4 Emphasis Opcodes 11
3.5 Special Symbol Opcodes 14
3.6 Special Processing Opcodes 14
3.7 Translation Opcodes 14
3.8 Character-Class Opcodes 19
3.9 Swap Opcodes 20
3.10 The Context and Multipass Opcodes 20
3.11 The correct Opcode 23
3.12 Miscellaneous Opcodes 23
3.13 Deprecated Opcodes 24

4 How to test Translation Tables 26

4.1 Translation Table Test Harness 26
4.2 Translation Table Doctests 27

5 Notes on Back-Translation 28

6 Programming with liblouis 29

6.1 License 29
6.2 Overview 29
6.3 Data structure of liblouis tables 30
6.4 lou_version 31
6.5 lou_translateString 31
6.6 lou_translate 32
6.7 lou_backTranslateString 33
6.8 lou_backTranslate 33
6.9 lou_hyphenate 34

6.10 lou_compileString ... 34
6.11 lou_dotsToChar ... 34
6.12 lou_charToDots ... 35
6.13 lou_logFile ... 35
6.14 lou_logPrint ... 35
6.15 lou_logEnd ... 35
6.16 lou_setDataPath ... 35
6.17 lou_getDataPath ... 36
6.18 lou_getTable ... 36
6.19 lou_readCharFromFile ... 36
6.20 lou_free ... 36
6.21 Python bindings ... 36

Opcode Index ... 37

Function Index ... 39

Program Index ... 40

1 Introduction

Liblouis is an open-source braille translator and back-translator derived from the translation routines in the BRLTTY screen reader for Linux. It has, however, gone far beyond these routines. It is named in honor of Louis Braille. In Linux and Mac OSX it is a shared library, and in Windows it is a DLL. For installation instructions see the README file. Please report bugs and oddities to the maintainer, john.boyer@abilitiessoft.com

This documentation is derived from Chapter 7 of the BRLTTY manual, but it has been extensively rewritten to cover new features.

Please read the following copyright and warranty information. Note that this information also applies to all source code, tables and other files in this distribution of liblouis. It applies similarly to the sister library liblouisxml.

This file is maintained by John J. Boyer john.boyer@abilitiessoft.com.

Persons who wish to program with liblouis but will not be writing translation tables may want to skip ahead to Chapter 6 [Programming with liblouis], page 29.

2 Test Programs

A number of test programs are provided as part of the liblouis package. They are intended for testing liblouis and for debugging tables. None of them is suitable for braille transcription. An application that can be used for transcription is `xml2brl`, which is part of the liblouisxml package (see Section "Introduction" in *Liblouisxml User's and Programmer's Manual*). The source code of the test programs can be studied to learn how to use the liblouis library and they can be used to perform the following functions.

All of these programs recognize the '`--help`' and '`--version`' options.

'`--help`'
'`-h`' Print a usage message listing all available options, then exit successfully.

'`--version`'
'`-v`' Print the version number, then exit successfully.

2.1 lou_debug

The `lou_debug` tool is intended for debugging liblouis translation tables. The command line for `lou_debug` is:

```
lou_debug [OPTIONS] TABLE[,TABLE,...]
```

The command line options that are accepted by `lou_debug` are described in [common options], page 2.

The table (or comma-separated list of tables) is compiled. If no errors are found a brief command summary is printed, then the prompt '`Command:`'. You can then input one of the command letters and get output, as described below.

Most of the commands print information in the various arrays of `TranslationTableHeader`. Since these arrays are pointers to chains of hashed items, the commands first print the hash number, then the first item, then the next item chained to it, and so on. After each item there is a prompt indicated by '`=>`'. You can then press enter (`RET`) to see the next item in the chain or the first item in the next chain. Or you can press *`h`* (for next-(h)ash) to skip to the next hash chain. You can also press *`e`* to exit the command and go back to the '`command:`' prompt.

`h` Brings up a screen of somewhat more extensive help.

`f` Display the first forward-translation rule in the first non-empty hash bucket. The number of the bucket is displayed at the beginning of the chain. Each rule is identified by the word '`Rule:`'. The fields are displayed by phrases consisting of the name of the field, an equal sign, and its value. The before and after fields are displayed only if they are nonzero. Special opcodes such as the `correct` opcode (see [correct], page 23) and the multipass opcodes are shown with the code that instructs the virtual machine that interprets them. If you want to see only the rules for a particular character string you can type *`p`* at the '`command:`' prompt. This will take you to the '`particular:`' prompt, where you can press *`f`* and then type in the string. The whole hash chain containing the string will be displayed.

`b` Display back-translation rules. This display is very similar to that of forward translation rules except that the dot pattern is displayed before the character string.

`c` Display character definitions, again within their hash chains.

`d` Displays single-cell dot definitions. If a character-definition opcode gives a multi-cell dot pattern, it is displayed among the back-translation rules.

`C` Display the character-to-dots map. This is set up by the character-definition opcodes and can also be influenced by the `display` opcode (see [display], page 24).

`D` Display the dot to character map, which shows which single-cell dot patterns map to which characters.

`z` Show the multi-cell dot patterns which have been assigned to the characters from 0 to 255 to comply with computer braille codes such as a 6-dot code. Note that the character-definition opcodes should use 8-dot computer braille.

`p` Bring up a secondary ('`particular:`') prompt from which you can examine particular character strings, dot patterns, etc. The commands (given in its own command summary) are very similar to those of the main '`command:`' prompt, but you can type a character string or dot pattern. They include `h`, `f`, `b`, `c`, `d`, `C`, `D`, `z` and `x` (to exit this prompt), but not `p`, `i` and `m`.

`i` Show braille indicators. This shows the dot patterns for various opcodes such as the `capsign` opcode (see [capsign], page 10) and the `numsign` opcode (see [numsign], page 11). It also shows emphasis dot patterns, such as those for the `italword`, the `firstletterbold` opcode (see [firstletterbold], page 13), etc. If a given opcode has not been used nothing is printed for it.

`m` Display various miscellaneous information about the table, such as the number of passes, whether certain opcodes have been used, and whether there is a hyphenation table.

`q` Exit the program.

2.2 lou_trace

When working on translation tables it is sometimes useful to determine what rules were applied when translating a string. `lou_trace` helps with exactly that. It list all the the applied rules for a given translation table and an input string.

```
lou_trace [OPTIONS] TABLE[,TABLE,...]
```

`lou_trace` accepts all the standard options (see [common options], page 2). Once started you can type an input string followed by `RET`. `lou_trace` will print the braille translation followed by list of rules that were applied to produce the translation. A possible invocation is listed in the following example:

```
$ lou_trace tables/en-us-g2.ctb
the u.s. postal service
! u4s4 po/al s}vice
1.      largesign       the     2346
```

```
2.      repeated                0
3.      lowercase       u       136
4.      punctuation     .       46
5.      context _$1["."]$1      @256
6.      lowercase       s       234
7.      postpunc        .       256
8.      repeated                0
9.      begword post    1234-135-34
10.     largesign       a       1
11.     lowercase       l       123
12.     repeated                0
13.     lowercase       s       234
14.     always  er      12456
15.     lowercase       v       1236
16.     lowercase       i       24
17.     lowercase       c       14
18.     lowercase       e       15
19.     pass2   $s1-10  @0
20.     pass2   $s1-10  @0
21.     pass2   $s1-10  @0
```

2.3 lou_checktable

To use this program type the following:

```
lou_checktable [OPTIONS] TABLE
```

Aside from the standard options (see [common options], page 2) `lou_checktable` also accepts the following options:

‘`--quiet`’
‘`-q`’ Do not write to standard error if there are no errors.

If the table contains errors, appropriate messages will be displayed. If there are no errors the message ‘`no errors found.`’ will be shown.

2.4 lou_allround

This program tests every capability of the liblouis library. It is completely interactive. Invoke it as follows:

```
lou_allround [OPTIONS]
```

The command line options that are accepted by `lou_allround` are described in [common options], page 2.

You will see a few lines telling you how to use the program. Pressing one of the letters in parentheses and then enter will take you to a message asking for more information or for the answer to a yes/no question. Typing the letter ‘`r`’ and then `RET` will take you to a screen where you can enter a line to be processed by the library and then view the results.

2.5 lou_translate

This program translates whatever is on the standard input unit and prints it on the standard output unit. It is intended for large-scale testing of the accuracy of translation and back-translation. The command line for `lou_translate` is:

```
lou_translate [OPTION] TABLE[,TABLE,...]
```

Aside from the standard options (see [common options], page 2) this program also accepts the following options:

'`--forward`'
'`-f`' Do a forward translation.

'`--backward`'
'`-b`' Do a backward translation.

To use it to translate or back-translate a file use a line like

```
lou_translate --forward en-us-g2.ctb <liblouis.txt >testtrans
```

2.6 lou_checkhyphens

This program checks the accuracy of hyphenation in Braille translation for both translated and untranslated words. It is completely interactive. Invoke it as follows:

```
lou_checkhyphens [OPTIONS]
```

The command line options that are accepted by `lou_checkhyphens` are described in [common options], page 2.

You will see a few lines telling you how to use the program.

3 How to Write Translation Tables

Many translation (contraction) tables have already been made up. They are included in this distribution in the tables directory and should be studied as part of the documentation. The most helpful (and normative) are listed in the following table:

‘`chardefs.cti`’
: Character definitions for U.S. tables

‘`compress.ctb`’
: Remove excessive whitespace

‘`en-us-g1.ctb`’
: Uncontracted American English

‘`en-us-g2.ctb`’
: Contracted or Grade 2 American English

‘`en-us-brf.dis`’
: Make liblouis output conform to BRF standard

‘`en-us-comp8.ctb`’
: 8-dot computer braille for use in coding examples

‘`en-us-comp6.ctb`’
: 6-dot computer braille

‘`nemeth.ctb`’
: Nemeth Code translation for use with liblouisxml

‘`nemeth_edit.ctb`’
: Fixes errors at the boundaries of math and text

The names used for files containing translation tables are completely arbitrary. They are not interpreted in any way by the translator. Contraction tables may be 8-bit ASCII files, UTF-8, 16-bit big-endian Unicode files or 16-bit little-endian Unicode files. Blank lines are ignored. Any leading and trailing whitespace (any number of blanks and/or tabs) is ignored. Lines which begin with a number sign or hatch mark (‘`#`’) are ignored, i.e. they are comments. If the number sign is not the first non-blank character in the line, it is treated as an ordinary character. If the first non-blank character is less-than (‘`<`’) the line is also treated as a comment. This makes it possible to mark up tables as xhtml documents. Lines which are not blank or comments define table entries. The general format of a table entry is:

```
opcode operands comments
```

Table entries may not be split between lines. The opcode is a mnemonic that specifies what the entry does. The operands may be character sequences, braille dot patterns or occasionally something else. They are described for each opcode, please see [Opcode Index], page 37. With some exceptions, opcodes expect a certain number of operands. Any text on the line after the last operand is ignored, and may be a comment. A few opcodes accept a variable number of operands. In this case a number sign begins a comment unless it is preceded by a backslash (‘\’).

Here are some examples of table entries.

```
# This is a comment.
always world 456-2456 A word and the dot pattern of its contraction
```

Most opcodes have both a "characters" operand and a "dots" operand, though some have only one and a few have other types.

The characters operand consists of any combination of characters and escape sequences proceeded and followed by whitespace. Escape sequences are used to represent difficult characters. They begin with a backslash (‘\‘). They are:

`\`	backslash
`\f`	form feed
`\n`	new line
`\r`	carriage return
`\s`	blank (space)
`\t`	horizontal tab
`\v`	vertical tab
`\e`	"escape" character (hex 1b, dec 27)
`\xhhhh`	4-digit hexadecimal value of a character

If liblouis has been compiled for 32-bit Unicode the following are also recognized.

`\yhhhhh`	5-digit (20 bit) character
`\zhhhhhhhh`	Full 32-bit value.

The dots operand is a braille dot pattern. The real braille dots, 1 through 8, must be specified with their standard numbers. liblouis recognizes "virtual dots," which are used for special purposes, such as distinguishing accent marks. There are seven virtual dots. They are specified by the number 9 and the letters ‘`a`’ through ‘`f`’. For a multi-cell dot pattern, the cell specifications must be separated from one another by a dash (‘`-`’). For example, the contraction for the English word ‘`lord`’ (the letter ‘`l`’ preceded by dot 5) would be specified as 5-123. A space may be specified with the special dot number 0.

An opcode which is helpful in writing translation tables is `include`. Its format is:

```
include filename
```

It reads the file indicated by `filename` and incorporates or includes its entries into the table. Included files can include other files, which can include other files, etc. For an example, see what files are included by the entry `include en-us-g1.ctb` in the table ‘`en-us-g2.ctb`’. If the included file is not in the same directory as the main table, use a full path name for filename. Tables can also be specified in a table list, in which the table names are separated by commas and given as a single table name in calls to the translation functions.

The order of the various types of opcodes or table entries is important. Character-definition opcodes should come first. However, if the optional `display` opcode (see [display], page 24) is used it should precede character-definition opcodes. Braille-indicator opcodes should come next. Translation opcodes should follow. The `context` opcode (see

[context], page 20) is a translation opcode, even though it is considered along with the multipass opcodes. These latter should follow the translation opcodes. The `correct` opcode (see [correct], page 23) can be used anywhere after the character-definition opcodes, but it is probably a good idea to group all `correct` opcodes together. The `include` opcode (see [include], page 23) can be used anywhere, but the order of entries in the combined table must conform to the order given above. Within each type of opcode, the order of entries is generally unimportant. Thus the translation entries can be grouped alphabetically or in any other order that is convenient. Hyphenation tables may be specified either with an `include` opcode or as part of a table list. They should come after everything else. Character-definition opcodes are necessary for hyphenation tables to work.

3.1 Hyphenation Tables

Hyphenation tables are necessary to make opcodes such as the `nocross` opcode (see [nocross], page 16) function properly. There are no opcodes for hyphenation table entries because these tables have a special format. Therefore, they cannot be specified as part of an ordinary table. Rather, they must be included using the `include` opcode (see [include], page 23) or as part of a table list. The liblouis hyphenation algorithm was adopted from the one used by OpenOffice. Note that Hyphenation tables must follow character definitions and should preferably be the last. For an example of a hyphenation table, see '`hyph_en_US.dic`'.

3.2 Character-Definition Opcodes

These opcodes are needed to define attributes such as digit, punctuation, letter, etc. for all characters and their dot patterns. liblouis has no built-in character definitions, but such definitions are essential to the operation of the `context` opcode (see [context], page 20), the `correct` opcode (see [correct], page 23), the multipass opcodes and the back-translator. If the dot pattern is a single cell, it is used to define the mapping between dot patterns and characters, unless a `display` opcode (see [display], page 24) for that character-dot-pattern pair has been used previously. If only a single-cell dot pattern has been given for a character, that dot pattern is defined with the character's own attributes. If more than one cell is given and some of them have not previously been defined as single cells, the undefined cells are entered into the dots table with the space attribute. This is done for backward compatibility with old tables, but it may cause problems with the above opcodes or back-translation. For this reason, every single-cell dot pattern should be defined before it is used in a multi-cell character representation. The best way to do this is to use the 8-dot computer braille representation for the particular braille code. If a character or dot pattern used in any rule, except those with the `display` opcode, the `repeated` opcode (see [repeated], page 16) or the `replace` opcode (see [replace], page 15), is not defined by one of the character-definition opcodes, liblouis will give an error message and refuse to continue until the problem is fixed. If the translator or back-translator encounters an undefined character in its input it produces a succinct error indication in its output, and the character is treated as a space.

`space character dots`

Defines a character as a space and also defines the dot pattern as such. for example:

```
space \s 0 \s is the escape sequence for blank; 0 means no dots.
```

`punctuation character dots`
: Associates a punctuation mark in the particular language with a braille representation and defines the character and dot pattern as punctuation. For example:

```
punctuation . 46 dot pattern for period in NAB computer braille
```

`digit character dots`
: Associates a digit with a dot pattern and defines the character as a digit. For example:

```
digit 0 356 NAB computer braille
```

`uplow characters dots [,dots]`
: The characters operand must be a pair of letters, of which the first is uppercase and the second lowercase. The first dots suboperand indicates the dot pattern for the upper-case letter. It may have more than one cell. The second dots suboperand must be separated from the first by a comma and is optional, as indicated by the square brackets. If present, it indicates the dot pattern for the lower-case letter. It may also have more than one cell. If the second dots suboperand is not present the first is used for the lower-case letter as well as the upper-case letter. This opcode is needed because not all languages follow a consistent pattern in assigning Unicode codes to upper and lower case letters. It should be used even for languages that do. The distinction is important in the forward translator. for example:

```
uplow Aa 17,1
```

`grouping name characters dots ,dots`
: This opcode is used to indicate pairs of grouping symbols used in processing mathematical expressions. These symbols are usually generated by the MathML interpreter in liblouisxml. They are used in multipass opcodes. The name operand must contain only letters, but they may be upper- or lower-case. The characters operand must contain exactly two Unicode characters. The dots operand must contain exactly two braille cells, separated by a comma. Note that grouping dot patterns also need to be declared with the `exactdots` opcode (see [exactdots], page 18). The characters may need to be declared with the `math` opcode (see [math], page 10).

```
grouping mrow \x0001\x0002 1e,2e
grouping mfrac \x0003\x0004 3e,4e
```

`letter character dots`
: Associates a letter in the language with a braille representation and defines the character as a letter. This is intended for letters which are neither uppercase nor lowercase.

`lowercase character dots`
: Associates a character with a dot pattern and defines the character as a lowercase letter. Both the character and the dot pattern have the attributes lowercase and letter.

`uppercase character dots`
: Associates a character with a dot pattern and defines the character as an uppercase letter. Both the character and the dot pattern have the attributes uppercase and letter. `lowercase` and `uppercase` should be used when a letter has only one case. Otherwise use the `uplow` opcode (see [uplow], page 9).

`litdigit digit dots`
: Associates a digit with the dot pattern which should be used to represent it in literary texts. For example:

```
litdigit 0 245
litdigit 1 1
```

`sign character dots`
: Associates a character with a dot pattern and defines both as a sign. This opcode should be used for things like at sign (‘`@`’), percent (‘`%`’), dollar sign (‘`$`’), etc. Do not use it to define ordinary punctuation such as period and comma. For example:

```
sign % 4-25-1234 literary percent sign
```

`math character dots`
: Associates a character and a dot pattern and defines them as a mathematical symbol. It should be used for less than (‘`<`’), greater than(‘`>`’), equals(‘`=`’), plus(‘`+`’), etc. For example:

```
math + 346 plus
```

3.3 Braille Indicator Opcodes

Braille indicators are dot patterns which are inserted into the braille text to indicate such things as capitalization, italic type, computer braille, etc. The opcodes which define them are followed only by a dot pattern, which may be one or more cells.

`capsign dots`
: The dot pattern which indicates capitalization of a single letter. In English, this is dot 6. For example:

```
capsign 6
```

`begcaps dots`
: The dot pattern which begins a block of capital letters. For example:

```
begcaps 6-6
```

`endcaps dots`
: The dot pattern which ends a block of capital letters within a word. For example:

```
endcaps 6-3
```

`letsign dots`
: This indicator is needed in Grade 2 to show that a single letter is not a contraction. It is also used when an abbreviation happens to be a sequence of letters that is the same as a contraction. For example:

```
letsign 56
```

`noletsign letters`
: The letters in the operand will not be proceeded by a letter sign. More than one `noletsign` opcode can be used. This is equivalent to a single entry containing all the letters. In addition, if a single letter, such as '`a`' in English, is defined as a `word` (see [word], page 16) or `largesign` (see [largesign], page 16), it will be treated as though it had also been specified in a `noletsign` entry.

`noletsignbefore characters`
: If any of the characters proceeds a single letter without a space a letter sign is not used. By default the characters apostrophe ('`'`') and period ('`.`') have this property. Use of a `noletsignbefore` entry cancels the defaults. If more than one `noletsignbefore` entry is used, the characters in all entries are combined.

`noletsignafter characters`
: If any of the characters follows a single letter without a space a letter sign is not used. By default the characters apostrophe ('`'`') and period ('`.`') have this property. Use of a `noletsignafter` entry cancels the defaults. If more than one `noletsignafter` entry is used the characters in all entries are combined.

`numsign dots`
: The translator inserts this indicator before numbers made up of digits defined with the `litdigit` opcode (see [litdigit], page 10) to show that they are a number and not letters or some other symbols. For example:

```
numsign 3456
```

3.4 Emphasis Opcodes

These also define braille indicators, but they require more explanation. There are four sets, for italic, bold, underline and computer braille. In each of the first three sets there are seven opcodes, for use before the first word of a phrase, for use before the last word, for use after the last word, for use before the first letter (or character) if emphasis starts in the middle of a word, for use after the last letter (or character) if emphasis ends in the middle of a word, before a single letter (or character), and to specify the length of a phrase to which the first-word and last-word-before indicators apply. This rather elaborate set of emphasis opcodes was devised to try to meet all contingencies. It is unlikely that a translation table will contain all of them. The translator checks for their presence. If they are present, it first looks to see if the single-letter indicator should be used. Then it looks at the word (or phrase) indicators and finally at the multi-letter indicators.

The translator will apply up to two emphasis indicators to each phrase or string of characters, depending on what the `typeform` parameter in its calling sequence indicates (see Chapter 6 [Programming with liblouis], page 29).

For computer braille there are only two braille indicators, for the beginning and end of a sequence of characters to be rendered in computer braille. Such a sequence may also have other emphasis. The computer braille indicators are applied not only when computer braille is indicated in the `typeform` parameter, but also when a sequence of characters is determined to be computer braille because it contains a subsequence defined by the `compbrl` opcode (see [compbrl], page 15) or the `literal` opcode (see [literal], page 25).

Here are the various emphasis opcodes.

`firstwordital dots`
: This is the braille indicator to be placed before the first word of an italicized phrase that is longer than the value given in the `lenitalphrase` opcode (see [lenitalphrase], page 12). For example:

```
firstwordital 46-46 English indicator
```

`lastworditalbefore dots`
: This is the braille indicator to be placed before the last word of an italicized phrase. In addition, if `firstwordital` is not used, this braille indicator is doubled and placed before the first word. Do not use `lastworditalbefore` and `lastworditalafter` in the same table. For example:

```
lastworditalbefore 4-6
```

`lastworditalafter dots`
: This is the braille indicator to be placed after the last word of an italicized phrase. Do not use `lastworditalbefore` and `lastworditalafter` in the same table. See also the `lenitalphrase` opcode (see [lenitalphrase], page 12) for more information.

`firstletterital dots`
: This is the braille indicator to be placed before the first letter (or character) if italicization begins in the middle of a word.

`lastletterital dots`
: This is the braille indicator to be placed after the last letter (or character) when italicization ends in the middle of a word.

`singleletterital dots`
: This braille indicator is used if only a single letter (or character) is italicized.

`lenitalphrase number`
: If `lastworditalbefore` is used, an italicized phrase is checked to see how many words it contains. If this number is less than or equal to the number given in the `lenitalphrase` opcode, the `lastworditalbefore` sign is placed in front of each word. If it is greater, the `firstwordital` indicator is placed before the first word and the `lastworditalbefore` indicator is placed after the last word. Note that if the `firstwordital` opcode is not used its indicator is made up by doubling the dot pattern given in the `lastworditalbefore` entry. For example:

```
lenitalphrase 4
```

`firstwordbold dots`
: This is the braille indicator to be placed before the first word of a bold phrase. For example:

```
firstwordbold 456-456
```

`lastwordboldbefore dots`
: This is the braille indicator to be placed before the last word of a bold phrase. In addition, if `firstwordbold` is not used, this braille indicator is doubled and placed before the first word. Do not use `lastwordboldbefore` and `lastwordboldafter` in the same table. For example:

```
lastwordboldbefore 456
```

`lastwordboldafter dots`
: This is the braille indicator to be placed after the last word of a bold phrase. Do not use `lastwordboldbefore` and `lastwordboldafter` in the same table.

`firstletterbold dots`
: This is the braille indicator to be placed before the first letter (or character) if bold emphasis begins in the middle of a word.

`lastletterbold dots`
: This is the braille indicator to be placed after the last letter (or character) when bold emphasis ends in the middle of a word.

`singleletterbold dots`
: This braille indicator is used if only a single letter (or character) is in boldface.

`lenboldphrase number`
: If `lastwordboldbefore` is used, a bold phrase is checked to see how many words it contains. If this number is less than or equal to the number given in the `lenboldphrase` opcode, the `lastwordboldbefore` sign is placed in front of each word. If it is greater, the `firstwordbold` indicator is placed before the first word and the `lastwordboldbefore` indicator is placed after the last word. Note that if the `firstwordbold` opcode is not used its indicator is made up by doubling the dot pattern given in the `lastwordboldbefore` entry.

`firstwordunder dots`
: This is the braille indicator to be placed before the first word of an underlined phrase.

`lastwordunderbefore dots`
: This is the braille indicator to be placed before the last word of an underlined phrase. In addition, if `firstwordunder` is not used, this braille indicator is doubled and placed before the first word.

`lastwordunderafter dots`
: This is the braille indicator to be placed after the last word of an underlined phrase.

`firstletterunder dots`
: This is the braille indicator to be placed before the first letter (or character) if underline emphasis begins in the middle of a word.

`lastletterunder dots`
: This is the braille indicator to be placed after the last letter (or character) when underline emphasis ends in the middle of a word.

`singleletterunder dots`
: This braille indicator is used if only a single letter (or character) is underlined.

`lenunderphrase number`
: If `lastwordunderbefore` is used, an underlined phrase is checked to see how many words it contains. If this number is less than or equal to the number given in the `lenunderphrase` opcode, the `lastwordunderbefore` sign is placed

in front of each word. If it is greater, the `firstwordunder` indicator is placed before the first word and the `lastwordunderbefore` indicator is placed after the last word. Note that if the `firstwordunder` opcode is not used its indicator is made up by doubling the dot pattern given in the `lastwordunderbefore` entry.

`begcomp dots`
: This braille indicator is placed before a sequence of characters translated in computer braille, whether this sequence is indicated in the `typeform` parameter (see Chapter 6 [Programming with liblouis], page 29) or inferred because it contains a subsequence specified by the `compbrl` opcode (see [compbrl], page 15).

`endcomp dots`
: This braille indicator is placed after a sequence of characters translated in computer braille, whether this sequence is indicated in the `typeform` parameter (see Chapter 6 [Programming with liblouis], page 29) or inferred because it contains a subsequence specified by the `compbrl` opcode (see [compbrl], page 15).

3.5 Special Symbol Opcodes

These opcodes define certain symbols, such as the decimal point, which require special treatment.

`decpoint character dots`
: This opcode defines the decimal point. The character operand must have only one character. For example, in '`en-us-g1.ctb`' we have:

```
decpoint . 46
```

`hyphen character dots`
: This opcode defines the hyphen, that is, the character used in compound words such as have-nots. The back-translator uses it to determine the end of individual words.

3.6 Special Processing Opcodes

These opcodes cause special processing to be carried out.

`capsnocont`
: This opcode has no operands. If it is specified, words or parts of words in all caps are not contracted. This is needed for languages such as Norwegian.

3.7 Translation Opcodes

These opcodes define the braille representations for character sequences. Each of them defines an entry within the contraction table. These entries may be defined in any order except, as noted below, when they define alternate representations for the same character sequence.

Each of these opcodes specifies a condition under which the translation is legal, and each also has a characters operand and a dots operand. The text being translated is processed strictly from left to right, character by character, with the most eligible entry for each position being used. If there is more than one eligible entry for a given position in the text,

then the one with the longest character string is used. If there is more than one eligible entry for the same character string, then the one defined first is is tested for legality first. (This is the only case in which the order of the entries makes a difference.)

The characters operand is a sequence or string of characters preceded and followed by whitespace. Each character can be entered in the normal way, or it can be defined as a four-digit hexadecimal number preceded by '`\x`'.

The dots operand defines the braille representation for the characters operand. It may also be specified as an equals sign ('`=`'). This means that the the default representation for each character (see Section 3.2 [Character-Definition Opcodes], page 8) within the sequence is to be used.

In what follows the word '`characters`' means a sequence of one or more consecutive letters between spaces and/or punctuation marks.

`noback opcode ...`
: This is an opcode prefix, that is to say, it modifies the operation of the opcode that follows it on the same line. noback specifies that no back-translation is to be done using this line.

```
noback always ;\s; 0
```

`nofor opcode ...`
: This is an opcode prefix which modifies the operation of the opcode following it on the same line. nofor specifies that forward translation is not to use the information on this line.

`compbrl characters`
: If the characters are found within a block of text surrounded by whitespace the entire block is translated according to the default braille representations defined by the Section 3.2 [Character-Definition Opcodes], page 8, if 8-dot computer braille is enabled or according to the dot patterns given in the `comp6` opcode (see [comp6], page 15), if 6-dot computer braille is enabled. For example:

```
compbrl www translate URLs in computer braille
```

`comp6 character dots`
: This opcode specifies the translation of characters in 6-dot computer braille. It is necessary because the translation of a single character may require more than one cell. The first operand must be a character with a decimal representation from 0 to 255 inclusive. The second operand may specify as many cells as necessary. The opcode is somewhat of a misnomer, since any dots, not just dots 1 through 6, can be specified. This even includes virtual dots.

`nocont characters`
: Like `compbrl`, except that the string is uncontracted. `prepunc` opcode (see [prepunc], page 18) and `postpunc` opcode (see [postpunc], page 18) rules are applied, however. This is useful for specifying that foreign words should not be contracted in an entire document.

`replace characters {characters}`
: Replace the first set of characters, no matter where they appear, with the second. Note that the second operand is *NOT* a dot pattern. It is also optional.

If it is omitted the character(s) in the first operand will be discarded. This is useful for ignoring characters. It is possible that the "ignored" characters may still affect the translation indirectly. Therefore, it is preferable to use `correct` opcode (see [correct], page 23).

`always characters dots`
: Replace the characters with the dot pattern no matter where they appear. Do *NOT* use an entry such as `always a 1`. Use the `uplow`, `letter`, etc. character definition opcodes instead. For example:

```
always world 456-2456 unconditional translation
```

`repeated characters dots`
: Replace the characters with the dot pattern no matter where they appear. Ignore any consecutive repetitions of the same character sequence. This is useful for shortening long strings of spaces or hyphens or periods. For example:

```
repeated --- 36-36-36 shorten separator lines made with hyphens
```

`repword characters dots`
: When characters are encountered check to see if the word before this string matches the word after it. If so, replace characters with dots and eliminate the second word and any word following another occurrence of characters that is the same. This opcode is used in Malaysian braille. In this case the rule is:

```
repword - 123456
```

`largesign characters dots`
: Replace the characters with the dot pattern no matter where they appear. In addition, if two words defined as large signs follow each other, remove the space between them. For example, in '`en-us-g2.ctb`' the words '`and`' and '`the`' are both defined as large signs. Thus, in the phrase '`the cat and the dog`' the space would be deleted between '`and`' and '`the`', with the result '`the cat andthe dog`'. Of course, '`and`' and '`the`' would be properly contracted. The term `largesign` is a bit of braille jargon that pleases braille experts.

`word characters dots`
: Replace the characters with the dot pattern if they are a word, that is, are surrounded by whitespace and/or punctuation.

`syllable characters dots`
: As its name indicates, this opcode defines a "syllable" which must be represented by exactly the dot patterns given. Contractions may not cross the boundaries of this "syllable" either from left or right. The character string defined by this opcode need not be a lexical syllable, though it usually will be. The equal sign in the following example means that the the default representation for each character within the sequence is to be used (see Section 3.7 [Translation Opcodes], page 14):

```
syllable horse = sawhorse, horseradish
```

`nocross characters dots`
: Replace the characters with the dot pattern if the characters are all in one syllable (do not cross a syllable boundary). For this opcode to work, a hyphenation

table must be included. If this is not done, `nocross` behaves like the `always` opcode (see [`always`], page 16). For example, if the English Grade 2 table is being used and the appropriate hyphenation table has been included `nocross sh 146` will cause the '`sh`' in '`monkshood`' not to be contracted.

`joinword characters dots`
: Replace the characters with the dot pattern if they are a word which is followed by whitespace and a letter. In addition remove the whitespace. For example, '`en-us-g2.ctb`' has `joinword to 235`. This means that if the word '`to`' is followed by another word the contraction is to be used and the space is to be omitted. If these conditions are not met, the word is translated according to any other opcodes that may apply to it.

`lowword characters dots`
: Replace the characters with the dot pattern if they are a word preceded and followed by whitespace. No punctuation either before or after the word is allowed. The term `lowword` derives from the fact that in English these contractions are written in the lower part of the cell. For example:

```
lowword were 2356
```

`contraction characters`
: If you look at '`en-us-g2.ctb`' you will see that some words are actually contracted into some of their own letters. A famous example among braille transcribers is '`also`', which is contracted as '`al`'. But this is also the name of a person. To take another example, '`altogether`' is contracted as '`alt`', but this is the abbreviation for the alternate key on a computer keyboard. Similarly '`could`' is contracted into '`cd`', but this is the abbreviation for compact disk. To prevent confusion in such cases, the letter sign (see `letsign` opcode (see [`letsign`], page 10)) is placed before such letter combinations when they actually are abbreviations, not contractions. The `contraction` opcode tells the translator to do this.

`sufword characters dots`
: Replace the characters with the dot pattern if they are either a word or at the beginning of a word.

`prfword characters dots`
: Replace the characters with the dot pattern if they are either a word or at the end of a word.

`begword characters dots`
: Replace the characters with the dot pattern if they are at the beginning of a word.

`begmidword characters dots`
: Replace the characters with the dot pattern if they are either at the beginning or in the middle of a word.

`midword characters dots`
: Replace the characters with the dot pattern if they are in the middle of a word.

`midendword characters dots`
: Replace the characters with the dot pattern if they are either in the middle or at the end of a word.

`endword characters dots`
: Replace the characters with the dot pattern if they are at the end of a word.

`partword characters dots`
: Replace the characters with the dot pattern if the characters are anywhere in a word, that is, if they are proceeded or followed by a letter.

`exactdots @dots`
: Note that the operand must begin with an at sign ('`@`'). The dot pattern following it is evaluated for validity. If it is valid, whenever an at sign followed by this dot pattern appears in the source document it is replaced by the characters corresponding to the dot pattern in the output. This opcode is intended for use in liblouisxml semantic-action files to specify exact dot patterns, as in mathematical codes. For example:

```
exactdots @4-46-12356
```

  will produce the characters with these dot patterns in the output.

`prepunc characters dots`
: Replace the characters with the dot pattern if they are part of punctuation at the beginning of a word.

`postpunc characters dots`
: Replace the characters with the dot pattern if they are part of punctuation at the end of a word.

`begnum characters dots`
: Replace the characters with the dot pattern if they are at the beginning of a number, that is, before all its digits. For example, in '`en-us-g1.ctb`' we have `begnum # 4`.

`midnum characters dots`
: Replace the characters with the dot pattern if they are in the middle of a number. For example, '`en-us-g1.ctb`' has `midnum . 46`. This is because the decimal point has a different dot pattern than the period.

`endnum characters dots`
: Replace the characters with the dot pattern if they are at the end of a number. For example '`en-us-g1.ctb`' has `endnum th 1456`. This handles things like '`4th`'. A letter sign is *NOT* inserted.

`joinnum characters dots`
: Replace the characters with the dot pattern. In addition, if whitespace and a number follows omit the whitespace. This opcode can be used to join currency symbols to numbers for example:

```
joinnum \x20AC 15 (EURO SIGN)
joinnum \x0024 145 (DOLLAR SIGN)
joinnum \x00A3 1234 (POUND SIGN)
joinnum \x00A5 13456 (YEN SIGN)
```

3.8 Character-Class Opcodes

These opcodes define and use character classes. A character class associates a set of characters with a name. The name then refers to any character within the class. A character may belong to more than one class.

The basic character classes correspond to the character definition opcodes, with the exception of the `uplow` opcode (see [uplow], page 9), which defines characters belonging to the two classes `uppercase` and `lowercase`. These classes are:

`space` Whitespace characters such as blank and tab

`digit` Numeric characters

`letter` Both uppercase and lowercase alphabetic characters

`lowercase`
: Lowercase alphabetic characters

`uppercase`
: Uppercase alphabetic characters

`punctuation`
: Punctuation marks

`sign` Signs such as percent (‘`%`’)

`math` Mathematical symbols

`litdigit` Literary digit

`undefined`
: Not properly defined

The opcodes which define and use character classes are shown below. For examples see ‘`fr-abrege.ctb`’.

`class name characters`
: Define a new character class. The characters operand must be specified as a string. A character class may not be used until it has been defined.

`after class opcode ...`
: The specified opcode is further constrained in that the matched character sequence must be immediately preceded by a character belonging to the specified class. If this opcode is used more than once on the same line then the union of the characters in all the classes is used.

`before class opcode ...`
: The specified opcode is further constrained in that the matched character sequence must be immediately followed by a character belonging to the specified class. If this opcode is used more than once on the same line then the union of the characters in all the classes is used.

3.9 Swap Opcodes

The swap opcodes are needed to tell the `context` opcode (see [context], page 20), the `correct` opcode (see [correct], page 23) and multipass opcodes which dot patterns to swap for which characters. There are three, `swapcd`, `swapdd` and `swapcc`. The first swaps dot patterns for characters. The second swaps dot patterns for dot patterns and the third swaps characters for characters. The first is used in the `context` opcode and the second is used in the multipass opcodes. Dot patterns are separated by commas and may contain more than one cell.

`swapcd name characters dots, dots, dots, ...`
: See above paragraph for explanation. For example:

```
swapcd dropped 0123456789 356,2,23,...
```

`swapdd name dots, dots, dots ... dotpattern1, dotpattern2, dotpattern3, ...`
: The `swapdd` opcode defines substitutions for the multipass opcodes. In the second operand the dot patterns must be single cells, but in the third operand multi-cell dot patterns are allowed. This is because multi-cell patterns in the second operand would lead to ambiguities.

`swapcc name characters characters`
: The `swapcc` opcode swaps characters in its second operand for characters in the corresponding places in its third operand. It is intended for use with `correct` opcodes and can solve problems such as formatting phone numbers.

3.10 The Context and Multipass Opcodes

The `context` and multipass opcodes (`pass2`, `pass3` and `pass4`) provide translation capabilities beyond those of the basic translation opcodes (see Section 3.7 [Translation Opcodes], page 14) discussed previously. The multipass opcodes cause additional passes to be made over the string to be translated. The number after the word `pass` indicates in which pass the entry is to be applied. If no multipass opcodes are given, only the first translation pass is made. The `context` opcode is basically a multipass opcode for the first pass. It differs slightly from the multipass opcodes per se. The format of all these opcodes is `opcode test action`. The specific opcodes are invoked as follows:

```
context test action
pass2 test action
pass3 test action
pass4 test action
```

The `test` and `action` operands have suboperands. Each suboperand begins with a non-alphanumeric character and ends when another non-alphanumeric character is encountered. The suboperands and their initial characters are as follows.

`" (double quote)`
: a string of characters. This string must be terminated by another double quote. It may contain any characters. If a double quote is needed within the string, it must be preceded by a backslash (‘\’). If a space is needed, it must be represented by the escape sequence \s. This suboperand is valid only in the test part of the `context` opcode.

`@ (at sign)`
: a sequence of dot patterns. Cells are separated by hyphens as usual. This suboperand is not valid in the test part of the context and correct opcodes.

`‘ (accent mark)`
: If this is the beginning of the string being translated this suboperand is true. It is valid only in the test part and must be the first thing in this operand.

`~ (tilde)`
: If this is the end of the string being translated this suboperand is true. It is valid only in the test part and must be the last thing in this operand.

`$ (dollar sign)`
: a string of attributes, such as ‘`d`’ for digit, ‘`l`’ for letter, etc. More than one attribute can be given. If you wish to check characters with any attribute, use the letter ‘`a`’. Input characters are checked to see if they have at least one of the attributes. The attribute string can be followed by numbers specifying how many characters are to be checked. If no numbers are given, 1 is assumed. If two numbers separated by a hyphen are given, the input is checked to make sure that at least the first number of characters with the attributes are present, but no more than the second number. If only one number is present, then exactly that many characters must have the attributes. A period instead of the numbers indicates an indefinite number of characters (for technical reasons the number of characters that are actually matched is limited to 65535).

  This suboperand is valid in all test parts but not in action parts. For the characters which can be used in attribute strings, see the following table.

`! (exclamation point)`
: reverses the logical meaning of the suboperand which follows. For example, !$d is true only if the character is *NOT* a digit. This suboperand is valid in test parts only.

`% (percent sign)`
: the name of a class defined by the `class` opcode (see [`class`], page 19) or the name of a swap set defined by the swap opcodes (see Section 3.9 [Swap Opcodes], page 20). Names may contain only letters. The letters may be upper or lower-case. The case matters. Class names may be used in test parts only. Swap names are valid everywhere.

`{ (left brace)`
: Name: the name of a grouping pair. The left brace indicates that the first (or left) member of the pair is to be used in matching. If this is between replacement brackets it must be the only item. This is also valid in the action part.

`} (right brace)`
: Name: the name of a grouping pair. The right brace indicates that the second (or right) member is to be used in matching. See the remarks on the left brace immediately above.

`/ (slash)`
: Search the input for the expression following the slash and return true if found. This can be used to set a variable.

_ (underscore)
: Move backward. If a number follows, move backward that number of characters. The program never moves backward beyond the beginning of the input string. This suboperand is valid only in test parts.

[(left bracket)
: start replacement here. This suboperand must always be paired with a right bracket and is valid only in test parts. Multiple pairs of square brackets in a single expression are not allowed.

] (right bracket)
: end replacement here. This suboperand must always be paired with a left bracket and is valid only in test parts.

 # (number sign or crosshatch)
: test or set a variable. Variables are referred to by numbers 1 to 50, for example, `#1`, `#2`, `#25`. Variables may be set by one `context` or multipass opcode and tested by another. Thus, an operation that occurs at one place in a translation can tell an operation that occurs later about itself. This feature will be used in math translation, and it may also help to alleviate the need for new opcodes. This suboperand is valid everywhere.

  Variables are set in the action part. To set a variable use an expression like `#1=1`, `#2=5`, etc. Variables are also incremented and decremented in the action part with expressions like `#1+`, `#3-`, etc. These operators increment or decrement the variable by 1.

  Variables are tested in the test part with expressions like `#1=2`, `#3<4`, `#5>6`, etc.

** (asterisk)*
: Copy the characters or dot patterns in the input within the replacement brackets into the output and discard anything else that may match. This feature is used, for example, for handling numeric subscripts in Nemeth. This suboperand is valid only in action parts.

? (question mark)
: Valid only in the action part. The characters to be replaced are simply ignored. That is, they are replaced with nothing. If either member of a grouping pair is in the replace brackets the other member at the same level is also removed.

The characters which can be used in attribute strings are as follows:

`a` any attribute

`d` digit

`D` literary digit

`l` letter

`m` math

`p` punctuation

`S` sign

`s`	space
`U`	uppercase
`u`	lowercase
`w`	first user-defined class
`x`	second user-defined class
`y`	third user-defined class
`z`	fourth user-defined class

The following illustrates the algorithm how text is evaluated with multipass expressions:
Loop over context, pass2, pass3 and pass4 and do the following for each pass:

a. Match the text following the cursor against all expressions in the current pass
b. If there is no match: shift the cursor one position to the right and continue the loop
c. If there is a match: choose the longest match
d. Do the replacement (everything between square brackets)
e. Place the cursor after the replaced text
f. continue loop

3.11 The correct Opcode

`correct test action`

Because some input (such as that from an OCR program) may contain systematic errors, it is sometimes advantageous to use a pre-translation pass to remove them. The errors and their corrections are specified by the `correct` opcode. If there are no `correct` opcodes in a table, the pre-translation pass is not used. The format of the `correct` opcode is very similar to that of the `context` opcode (see [context], page 20). The only difference is that in the action part strings may be used and dot patterns may not be used. Some examples of `correct` opcode entries are:

```
correct "\\" ? Eliminate backslashes
correct "cornf" "comf" fix a common "scano"
correct "cornm" "comm"
correct "cornp" "comp"
correct "*" ? Get rid of stray asterisks
correct "|" ? ditto for vertical bars
correct "\s?" "?" drop space before question mark
```

3.12 Miscellaneous Opcodes

`include filename`

Read the file indicated by `filename` and incorporate or include its entries into the table. Included files can include other files, which can include other files, etc. For an example, see what files are included by the entry include '`en-us-g1.ctb`' in the table '`en-us-g2.ctb`'. If the included file is not in the same directory as the main table, use a full path name for filename.

`locale characters`
: Not implemented, but recognized and ignored for backward compatibility.

`undefined dots`
: If this opcode is used in a table any characters which have not been defined in the table but are encountered in the text will be replaced by the dot pattern. If this opcode is not used, any undefined characters are replaced by `'\xhhhh'`, where the h's are hexadecimal digits.

`display character dots`
: Associates dot patterns with the characters which will be sent to a braille embosser, display or screen font. The character must be in the range 0-255 and the dots must specify a single cell. Here are some examples:

```
# When the character a is sent to the embosser or display,
# it will produce a dot 1.
display a 1

# When the character L is sent to the display or embosser
# it will produce dots 1-2-3.
display L 123
```

  The `display` opcode is optional. It is used when the embosser or display has a different mapping of characters to dot patterns than that given in Section 3.2 [Character-Definition Opcodes], page 8. If used, display entries must proceed character-definition entries.

  A possible use case would be to define display opcodes so that the result is Unicode braille for use on a display and a second set of display opcodes (in a different file) to produce plain ASCII braille for use with an embosser.

`multind dots opcode opcode ...`
: The `multind` opcode tells the back-translator that a sequence of braille cells represents more than one braille indicator. For example, in '`en-us-g1.ctb`' we have `multind 56-6 letsign capsign`. The back-translator can generally handle single braille indicators, but it cannot apply them when they immediately follow each other. It recognizes the letter sign if it is followed by a letter and takes appropriate action. It also recognizes the capital sign if it is followed by a letter. But when there is a letter sign followed by a capital sign it fails to recognize the letter sign unless the sequence has been defined with `multind`. A `multind` entry may not contain a comment because liblouis would attempt to interpret it as an opcode.

3.13 Deprecated Opcodes

The following opcodes are an early attempt to handle emphasis. They have been deprecated by more specific opcodes, but are kept for backward compatibility.

`italsign dots`
: This opcode is deprecated. Use the `lastworditalbefore` opcode (see [lastworditalbefore], page 12) instead.

`begital dots`
: This opcode is deprecated. Use the `firstletterital` opcode (see [`firstletterital`], page 12) instead.

`endital dots`
: This opcode is deprecated. Use the `lastletterital` opcode (see [`lastletterital`], page 12) instead.

`boldsign dots`
: This opcode is deprecated. Use the `lastwordboldbefore` opcode (see [`lastwordboldbefore`], page 12) instead.

`begbold dots`
: This opcode is deprecated. Use the `firstletterbold` opcode (see [`firstletterbold`], page 13) instead.

`endbold dots`
: This opcode is deprecated. Use the `lastletterbold` opcode (see [`lastletterbold`], page 13) instead.

`undersign dots`
: This opcode is deprecated. Use the `lastwordunderbefore` opcode (see [`lastwordunderbefore`], page 13) instead.

`begunder dots`
: This opcode is deprecated. Use the `firstletterunder` opcode (see [`firstletterunder`], page 13) instead.

`endunder dots`
: This opcode is deprecated. Use the `lastletterunder` opcode (see [`lastletterunder`], page 13) instead.

`literal characters`
: This opcode is deprecated. Use the `compbrl` opcode (see [`compbrl`], page 15) instead.

4 How to test Translation Tables

There are a number of automated tests for liblouis and they are proving to be of tremendous value. When changing the code the developers can run the tests to see if anything broke.

For testing the translation tables there are basically two approaches: there are the harness tests and the doctests. They were created at roughly the same time using different technologies, have influenced each other and have gone through improvements and technology changes. For now they are both based on Python so you need to have that installed. The philosophies of the two are slightly different:

Harness tests
: The harness tests are data driven, i.e. you give the test data, i.e. a string to translate and the expected output. The data is in a standard format, i.e. json. They work with both Python2 and Python3, however since the format is json it is perceivable that somebody would write some C code which takes the data in the harness file and runs it through liblouis so they could also run without Python and without ucs4.

Doctests
: The doctests on the other hand are based on a technology used in Python where you define your tests as if you were sitting at a terminal session with a Python interpreter. So the tests look like you typed a command and got some output, e.g.

```
>>> translate(['table.ctb'], "Hello", mode=compbrlLeftCursor)
("HELLO", [0,1,2,3], [0,1,2,3], 0)
```

There is a convenience wrapper which hides away much of the complexity of above example so you can write stuff like

```
>>> t.braille('the cat sat on the mat')
u'! cat sat on ! mat'
```

But essentially you are writing code, so the doctests allow you to do more flexible tests that are much closer to the raw iron. For technical reasons the doctests will probably only ever work in either Python2 or Python3 but not both and they will never run from C.

To sum it up, the recommendation is that for normal table testing you should use the test harness. It has a lot of momentum and the format is a standard. If you want to be closer to the raw Python API of liblouis, if you want to test some more intricate scenarios (involving inpos, modes, etc) then the doctests are for you.

4.1 Translation Table Test Harness

Each harness file is a simple utf8 encoded json file, which has two entries.

`tables`
: A list containing table names, which the tests should be run against. This is usually just one table, but for some situations more than one table is required.

`tests`
: A list of sections of tests, which should be processed independantly. Each test section is a dictionary of two items.

`flags`
: The flags that apply for all the test cases in this section. For example, they could all be forward translation tests, or they should all be run as computer braille tests.

`data` A list of test cases, each one containing the specific test data needed to perform a test.

These are the valid fields for the flags section:

`comment` A field describing the reason for the tests, the transformation rule or any useful info that might be needed in case the test breaks (optional).

`cursorPos`
The position of the cursor within the given text (optional). Useful when simulating screenreader interaction, to debug contraction and cursor behaviour.

`mode` The liblouis translation mode that should be used for this test (optional). If not defined defaults to 0.

`outputUniBrl`
For a forward translation test, the output should be in unicode braille. For a backward translation test, the input is in unicode braille.

`testmode` The optional testmode field can have three values: `"translate"` (default if undeclaired), `"backtranslate"` or `"hyphenate"`. Declares what tests should be performed on the test data.

Each test case has the following entries:

`input` The unicode text to be tested (required).

`output` The expected braille output (required). The dots should be encoded in the liblouis ascii-braille like encoding.

`brlCursorPos`
The expected position of the braille cursor in the braille output (optional). Useful when simulating screenreader interaction, to debug contraction and cursor behaviour.

Variables defined in the flags section can be overwridden by individual test cases, but if several tests need the same options, they should idealy be split into their own section, complete with their own flags and data.

For examples please see '`*_harness.txt`' in the harness directory in the source distribution.

4.2 Translation Table Doctests

For examples on how to create doctests please see '`*_test.txt`' in the doctest directory in the source distribution.

5 Notes on Back-Translation

Back-translation is carried out by the function `lou_backTranslateString`. Its calling sequence is described in Chapter 6 [Programming with liblouis], page 29. Tables containing no `context` opcode (see [context], page 20), `correct` opcode (see [correct], page 23) or multipass opcodes can be used for both forward and backward translation. If these opcodes are needed different tables will be required. `lou_backTranslateString` first performs `pass4`, if present, then `pass3`, then `pass2`, then the backtranslation, then corrections. Note that this is exactly the inverse of forward translation.

6 Programming with liblouis

6.1 License

Liblouis may contain code borrowed from the Linux screen reader BRLTTY, Copyright © 1999-2006 by the BRLTTY Team.

Copyright © 2004-2007 ViewPlus Technologies, Inc. www.viewplus.com.

Copyright © 2007,2009 Abilitiessoft, Inc. www.abilitiessoft.com.

Liblouis is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Liblouis is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Liblouis. If not, see http://www.gnu.org/licenses/.

6.2 Overview

You use the liblouis library by calling the following functions, `lou_translateString`, `lou_backTranslateString`, `lou_logFile`, `lou_logPrint`, `lou_endLog`, `lou_getTable`, `lou_translate`, `lou_backTranslate`, `lou_hyphenate`, `lou_charToDots`, `lou_dotsToChar`, `lou_compileString`, `lou_readCharFromFile`, `lou_version` and `lou_free`. These are described below. The header file, '`liblouis.h`', also contains brief descriptions. Liblouis is written in straight C. It has just three code modules, '`compileTranslationTable.c`', '`lou_translateString.c`' and '`lou_backTranslateString.c`'. In addition, there are two header files, '`liblouis.h`', which defines the API, and '`louis.h`', used only internally and by liblouisxml. The latter includes '`liblouis.h`'.

Persons who wish to use liblouis from Python may want to skip ahead to Section 6.21 [Python bindings], page 36.

'`compileTranslationTable.c`' keeps track of all translation tables which an application has used. It is called by the translation, hyphenation and checking functions when they start. If a table has not yet been compiled '`compileTranslationTable.c`' checks it for correctness and compiles it into an efficient internal representation. The main entry point is `lou_getTable`. Since it is the module that keeps track of memory usage, it also contains the `lou_free` function. In addition, it contains the `lou_logFile`, `lou_logPrint` and `lou_endLog` functions, plus some utility functions which are used by the other modules.

By default, liblouis handles all characters internally as 16-bit unsigned integers. It can be compiled for 32-bit characters as explained below. The meanings of these integers are not hard-coded. Rather they are defined by the character-definition opcodes. However, the standard printable characters, from decimal 32 to 126 are recognized for the purpose of processing the opcodes. Hence, the following definition is included in '`liblouis.h`'. It is correct for computers with at least 32-bit processors.

```
#define widechar unsigned short int
```

To make liblouis handle 32-bit Unicode simply remove the word `short` in the above `define`. This will cause the translate and back-translate functions to expect input in 32-bit form and to deliver their output in this form. The input to the compiler (tables) is unaffected except that two new escape sequences for 20-bit and 32-bit characters are recognized.

Here are the definitions of the eleven liblouis functions and their parameters. They are given in terms of 16-bit Unicode. If liblouis has been compiled for 32-bit Unicode simply read 32 instead of 16.

6.3 Data structure of liblouis tables

The data structure `TranslationTableHeader` is defined by a `typedef` statement in '`louis.h`'. To find the beginning, search for the word '`header`'. As its name implies, this is actually the table header. Data are placed in the `ruleArea` array, which is the last item defined in this structure. This array is declared with a length of 1 and is expanded as needed. The table header consists mostly of arrays of pointers of size `HASHNUM`. These pointers are actually offsets into `ruleArea` and point to chains of items which have been placed in the same hash bucket by a simple hashing algorithm. `HASHNUM` should be a prime and is currently 1123. The structure of the table was chosen to optimize speed rather than memory usage.

The first part of the table contains miscellaneous information, such as the number of passes and whether various opcodes have been used. It also contains the amount of memory allocated to the table and the amount actually used.

The next section contains pointers to various braille indicators and begins with `capitalSign`. The rules pointed to contain the dot pattern for the indicator and an opcode which is used by the back-translator but does not appear in the list of opcodes. The braille indicators also include various kinds of emphasis, such as italic and bold and information about the length of emphasized phrases. The latter is contained directly in the table item instead of in a rule.

After the braille indicators comes information about when a letter sign should be used.

Next is an array of size `HASHNUM` which points to character definitions. These are created by the character-definition opcodes.

Following this is a similar array pointing to definitions of single-cell dot patterns. This is also created from the character-definition opcodes. If a character definition contains a multi-cell dot pattern this is compiled into ordinary forward and backward rules. If such a multi-cell dot pattern contains a single cell which has not previously been defined that cell is placed in this array, but is given the attribute `space`.

Next come arrays that map characters to single-cell dot patterns and dots to characters. These are created from both character-definition opcodes and display opcodes.

Next is an array of size 256 which maps characters in this range to dot patterns which may consist of multiple cells. It is used, for example, to map '`{`' to dots 456-246. These mappings are created by the `compdots` or the `comp6` opcode (see [comp6], page 15).

Next are two small arrays that held pointers to chains of rules produced by the `swapcd` opcode (see [swapcd], page 20) and the `swapdd` opcode (see [swapdd], page 20) and by some multipass, `context` and `correct` opcodes.

Now we get to an array of size `HASHNUM` which points to chains of rules for forward translation.

Following this is a similar array for back-translation.

Finally is the `ruleArea`, an array of variable size to which various structures are mapped and to which almost everything else points.

6.4 lou_version

```
char *lou_version ()
```

This function returns a pointer to a character string containing the version of liblouis, plus other information, such as the release date and perhaps notable changes.

6.5 lou_translateString

```
int lou_translateString (
    const char * tableList,
    const widechar * inbuf,
    int *inlen,
    widechar *outbuf,
    int *outlen,
    char *typeform,
    char *spacing,
    int mode);
```

This function takes a string of 16-bit Unicode characters in `inbuf` and translates it into a string of 16-bit characters in `outbuf`. Each 16-bit character produces a particular dot pattern in one braille cell when sent to an embosser or braille display or to a screen type font. Which 16-bit character represents which dot pattern is indicated by the character-definition and display opcodes in the translation table.

The `tableList` parameter points to a list of translation tables separated by commas. If only one table is given, no comma should be used after it. It is these tables which control just how the translation is made, whether in Grade 2, Grade 1, or something else.

liblouis knows where to find all the tables that have been distributed with it. So you can just give a table name such as `en-us-g2.ctb` and liblouis will load it. You can also give a table name which includes a path. If this is the first table in a list, all the tables in the list must be on the same path. You can specify a path on which liblouis will look for table names by setting the environment variable `LOUIS_TABLEPATH`. This environment variable can contain one or more paths separated by commas. On receiving a table name liblouis first checks to see if it can be found on any of these paths. If not, it then checks to see if it can be found in the current directory, or, if the first (or only) name in a table list, if it contains a path name, can be found on that path. If not, it checks to see if it can be found on the path where the distributed tables have been installed. If a table has already been loaded and compiled this path-checking is skipped.

The tables in a list are all compiled into the same internal table. The list is then regarded as the name of this table. As explained in Chapter 3 [How to Write Translation Tables], page 6, each table is a file which may be plain text, big-endian Unicode or little-endian Unicode. A table (or list of tables) is compiled into an internal representation the first time

it is used. Liblouis keeps track of which tables have been compiled. For this reason, it is essential to call the `lou_free` function at the end of your application to avoid memory leaks. Do *NOT* call `lou_free` after each translation. This will force liblouis to compile the translation tables each time they are used, leading to great inefficiency.

Note that both the `*inlen` and `*outlen` parameters are pointers to integers. When the function is called, these integers contain the maximum input and output lengths, respectively. When it returns, they are set to the actual lengths used.

The `typeform` parameter is used to indicate italic type, boldface type, computer braille, etc. It is a string of characters with the same length as the input buffer pointed to by `*inbuf`. However, it is used to pass back character-by-character results, so enough space must be provided to match the `*outlen` parameter. Each character indicates the typeform of the corresponding character in the input buffer. The values are as follows: 0 plain-text; 1 italic; 2 bold; 4 underline; 8 computer braille. These values can be added for multiple emphasis. If this parameter is `NULL`, no checking for type forms is done. In addition, if this parameter is not `NULL`, it is set on return to have an 8 at every position corresponding to a character in `outbuf` which was defined to have a dot representation containing dot 7, dot 8 or both, and to 0 otherwise.

The `spacing` parameter is used to indicate differences in spacing between the input string and the translated output string. It is also of the same length as the string pointed to by `*inbuf`. If this parameter is `NULL`, no spacing information is computed.

The `mode` parameter specifies how the translation should be done. The valid values of mode are listed in '`liblouis.h`'. They are all powers of 2, so that a combined mode can be specified by adding up different values.

The function returns 1 if no errors were encountered and 0 if a complete translation could not be done.

6.6 lou_translate

```
int lou_translate (
    const char * tableList,
    const widechar * const inbuf,
    int *inlen,
    widechar * outbuf,
    int *outlen,
    char *typeform,
    char *spacing,
    int *outputPos,
    int *inputPos,
    int *cursorPos,
    int mode);
```

This function adds the parameters `outputPos`, `inputPos` and `cursorPos`, to facilitate use in screen reader programs. The `outputPos` parameter must point to an array of integers with at least `inlen` elements. On return, this array will contain the position in `outbuf` corresponding to each input position. Similarly, `inputPos` must point to an array of integers of at least `outlen` elements. On return, this array will contain the position in `inbuf` corresponding to each position in `outbuf`. `cursorPos` must point to an integer containing

the position of the cursor in the input. On return, it will contain the cursor position in the output. Any parameter after `outlen` may be `NULL`. In this case, the actions corresponding to it will not be carried out. The `mode` parameter, however, must be present and must be an integer, not a pointer to an integer. If the `compbrlAtCursor` bit is set in the `mode` parameter the space-bounded characters containing the cursor will be translated in computer braille. If the `compbrlLeftCursor` bit is set only the characters to the left of the cursor will be in computer braille. This bit overrides `compbrlAtCursor`. When the `dotsIO` bit is set, during translation, produce output as dot patterns. During back-translation accept input as dot patterns. Note that the produced dot patterns are affected if you have any `display` opcode (see [display], page 24) defined in any of your tables. The `ucBrl` (Unicode Braille) bit is used by `lou_charToDots` and `lou_translate`. It causes the dot patterns to be Unicode Braille rather than the liblouis representation. Note that you will not notice any change when setting `ucBrl` unless `dotsIO` is also set. `lou_dotsToChar` and `lou_backTranslate` recognize Unicode braille automatically.

The `otherTrans` mode needs special description. If it is set liblouis will attempt to call a wrapper for another translator. These other translators are usually for Asian languages. The calling sequence is the same as for liblouis itself except that the `trantab` parameter gives the name of the other translator, possibly abbreviated, followed by a colon, followed by whatever other information the other translator needs. This is specific for each translator. If no such information is needed the colon should be omitted. The result of calling either the translate or back-translate functions with this mode bit set will be the same as calling without it set. That is, the wrapper for the other translator simulates a call to liblouis. Note that the wrappers are not implemented at this time. Setting this mode bit will result in failure (return value of 0).

6.7 lou_backTranslateString

```
int lou_backTranslateString (
    const char * tableList,
    const widechar * inbuf,
    int *inlen,
    widechar *outbuf,
    int *outlen,
    char *typeform,
    char *spacing,
    int mode);
```

This is exactly the opposite of `lou_translateString`. `inbuf` is a string of 16-bit Unicode characters representing braille. `outbuf` will contain a string of 16–bit Unicode characters. `typeform` will indicate any emphasis found in the input string, while `spacing` will indicate any differences in spacing between the input and output strings. The `typeform` and `spacing` parameters may be `NULL` if this information is not needed. `mode` again specifies how the back-translation should be done.

6.8 lou_backTranslate

```
int lou_backTranslate (
    const char * tableList,
```

```
    const widechar * inbufx,
    int *inlen,
    widechar * outbuf,
    int *outlen,
    char *typeform,
    char *spacing,
    int *outputPos,
    int *inputPos,
    int *cursorPos,
    int mode);
```

This function is exactly the inverse of `lou_translate`.

6.9 lou_hyphenate

```
int lou_hyphenate (
    const char *tableList,
    const widechar *inbuf,
    int inlen,
    char *hyphens,
    int mode);
```

This function looks at the characters in `inbuf` and if it finds a sequence of letters attempts to hyphenate it as a word. Note that lou_hyphenate operates on single words only, and spaces or punctuation marks between letters are not allowed. Leading and trailing punctuation marks are ignored. The table named by the `tableList` parameter must contain a hyphenation table. If it does not, the function does nothing. `inlen` is the length of the character string in `inbuf`. `hyphens` is an array of characters and must be of size `inlen` + 1 (to account for the NULL terminator). If hyphenation is successful it will have a 1 at the beginning of each syllable and a 0 elsewhere. If the `mode` parameter is 0 `inbuf` is assumed to contain untranslated characters. Any nonzero value means that `inbuf` contains a translation. In this case, it is back-translated, hyphenation is performed, and it is re-translated so that the hyphens can be placed correctly. The `lou_translate` and `lou_backTranslate` functions are used in this process. `lou_hyphenate` returns 1 if hyphenation was successful and 0 otherwise. In the latter case, the contents of the `hyphens` parameter are undefined. This function was provided for use in liblouisxml.

6.10 lou_compileString

```
int lou_compileString (const char *tableList, const char *inString)
```

This function enables you to compile a table entry on the fly at run-time. The new entry is added to `tableList` and remains in force until `lou_free` is called. If `tableList` has not previously been loaded it is loaded and compiled. `inString` contains the table entry to be added. It may be anything valid. Error messages will be produced if it is invalid. The function returns 1 on success and 0 on failure.

6.11 lou_dotsToChar

```
int lou_dotsToChar (const char *tableList, const widechar *inbuf, widechar
```

```
*outbuf, int length, int)
```

This function takes a widechar string in `inbuf` consisting of dot patterns and converts it to a widechar string in `outbuf` consisting of characters according to the specifications in `tableList`. `length` is the length of both `inbuf` and `outbuf`. The dot patterns in `inbuf` can be in either liblouis format or Unicode braille. The function returns 1 on success and 0 on failure.

6.12 lou_charToDots

```
int lou_charToDots (const char *tableList, const widechar *inbuf, widechar
*outbuf, int length, int mode)
```

This function is the inverse of `lou_dotsToChar`. It takes a widechar string in `inbuf` consisting of characters and converts it to a widechar string in `outbuf` consisting of dot patterns according to the specifications in `tableList`. `length` is the length of both `inbuf` and `outbuf`. The dot patterns in `outbufbuf` are in liblouis format if the mode bit `ucBrl` is not set and in Unicode format if it is set. The function returns 1 on success and 0 on failure.

6.13 lou_logFile

```
void lou_logFile (char *fileName);
```

This function is used when it is not convenient either to let messages be printed on stderr or to use redirection, as when liblouis is used in a GUI application or in liblouisxml. Any error messages generated will be printed to the file given in this call. The entire path name of the file must be given.

6.14 lou_logPrint

```
void lou_logPrint (char *format, ...);
```

This function is called like `fprint`. It can be used by other libraries to print messages to the file specified by the call to `lou_logFile`. In particular, it is used by the companion library liblouisxml.

6.15 lou_logEnd

```
lou_logEnd ();
```

This function is used at the end of processing a document to close the log file, so that it can be read by the rest of the program.

6.16 lou_setDataPath

```
char * lou_setDataPath (char *path);
```

This function is used to tell liblouis and liblouisutdml where tables and files are located. It thus makes them completely relocatable, even on Linux. The `path` is the directory where the subdirectories `liblouis/tables` and `liblouisutdml/lbu_files` are rooted or located. The function returns a pointer to the `path`.

6.17 lou_getDataPath

```
char * lou_getDataPath ();
```

This function returns a pointer to the path set by `lou_setDataPath`. If no path has been set it returns `NULL`.

6.18 lou_getTable

```
void *lou_getTable (char *tablelist);
```

`tablelist` is a list of names of table files separated by commas, as explained previously (see [`tableList` parameter in `lou_translateString`], page 31). If no errors are found this function returns a pointer to the compiled table. If errors are found messages are printed to the log file, which is stderr unless a different filename has been given using the `lou_logFile` function. Errors result in a `NULL` pointer being returned.

6.19 lou_readCharFromFile

```
int lou_readCharFromFile (const char *fileName, int *mode);
```

This function is provided for situations where it is necessary to read a file which may contain little-endian or big-endian 16-bit Unicode characters or ASCII8 characters. The return value is a little-endian character, encoded as an integer. The `fileName` parameter is the name of the file to be read. The `mode` parameter is a pointer to an integer which must be set to 1 on the first call. After that, the function takes care of it. On end-of-file the function returns `EOF`.

6.20 lou_free

```
void lou_free ();
```

This function should be called at the end of the application to free all memory allocated by liblouis. Failure to do so will result in memory leaks. Do *NOT* call `lou_free` after each translation. This will force liblouis to compile the translation tables every time they are used, resulting in great inefficiency.

6.21 Python bindings

There are Python bindings for `lou_translateString`, `lou_translate` and `lou_version`. For installation instructions see the the '`README`' file in the '`python`' directory. Usage information is included in the Python module itself.

Opcode Index

A

after 19
always 16

B

before 19
begbold 25
begcaps 10
begcomp 14
begital 24
begmidword 17
begnum 18
begunder 25
begword 17
boldsign 25

C

capsign 10
capsnocont 14
class 19
comp6 15
compbrl 15
context 20
contraction 17
correct 23

D

decpoint 14
digit 9
display 24

E

endbold 25
endcaps 10
endcomp 14
endital 25
endnum 18
endunder 25
endword 18
exactdots 18

F

firstletterbold 13
firstletterital 12
firstletterunder 13
firstwordbold 12
firstwordital 12
firstwordunder 13

G

grouping 9

H

hyphen 14

I

include 23
italsign 24

J

joinnum 18
joinword 17

L

largesign 16
lastletterbold 13
lastletterital 12
lastletterunder 13
lastwordboldafter 13
lastwordboldbefore 12
lastworditalafter 12
lastworditalbefore 12
lastwordunderafter 13
lastwordunderbefore 13
lenboldphrase 13
lenitalphrase 12
lenunderphrase 13
letsign 10
letter 9
litdigit 10
literal 25
locale 23
lowercase 9
lowword 17

M

math 10
midendword 17
midnum 18
midword 17
multind 24

N

noback 15
nocont 15
nocross 16

nofor . . . 15
noletsign . . . 10
noletsignafter . . . 11
noletsignbefore . . . 11
numsign . . . 11

P

partword . . . 18
pass2 . . . 20
pass3 . . . 20
pass4 . . . 20
postpunc . . . 18
prepunc . . . 18
prfword . . . 17
punctuation . . . 9

R

repeated . . . 16
replace . . . 15
repword . . . 16

S

sign . . . 10
singleletterbold . . . 13
singleletterital . . . 12
singleletterunder . . . 13
space . . . 8
sufword . . . 17
swapcc . . . 20
swapcd . . . 20
swapdd . . . 20
syllable . . . 16

U

undefined . . . 24
undersign . . . 25
uplow . . . 9
uppercase . . . 9

W

word . . . 16

Function Index

lou_backTranslate 33
lou_backTranslateString 33
lou_charToDots 35
lou_compileString 34
lou_dotsToChar 34
lou_free 36
lou_getDataPath 36
lou_getTable 36
lou_hyphenate 34
lou_logEnd 35
lou_logFile 35
lou_logPrint 35
lou_readCharFromFile 36
lou_setDataPath 35
lou_translate 32
lou_translateString 31
lou_version 31

Program Index

lou_allround 4
lou_checkhyphens 5
lou_checktable 4
lou_debug 2
lou_trace 3
lou_translate 5