Liblouis User's and Programmer's Manual

for version 3.28.0, 4 December 2023

by John J. Boyer

This manual is for liblouis (version 3.28.0, 4 December 2023), 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.org`.

Copyright © 2014, 2016 Swiss Library for the Blind, Visually Impaired and Print Disabled. `www.sbs.ch`.

> 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

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 mailing list, `liblouis-liblouisxml@freelists.org`

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

1.1 Who is this manual for

This manual has two main audiences: People who want to write or improve a braille translation table and people who want to use the braille translator library in their own programs. This manual is probably not for people who are looking for some turn-key braille translation software.

1.2 How to read this manual

If you are mostly interested in writing braille translation tables then you want to focus on ⟨undefined⟩ [How to Write Translation Tables], page ⟨undefined⟩. You might want to look at ⟨undefined⟩ [Notes on Back-Translation], page ⟨undefined⟩, if you are interested in back-translation. Read ⟨undefined⟩ [Table Metadata], page ⟨undefined⟩, if you want to find out how you can augment your tables with metadata in order to make them discoverable by programs. Finally ⟨undefined⟩ [Testing Translation Tables interactively], page ⟨undefined⟩, and ⟨undefined⟩ [Automated Testing of Translation Tables], page ⟨undefined⟩, will show how your braille translation tables can be tested interactively and also in an automated fashion.

If you want to use the braille translation library in your own program or you are interested in enhancing the braille translation library itself then you will want to look at ⟨undefined⟩ [Programming with liblouis], page ⟨undefined⟩.

2 How to Write Translation Tables

For many languages there is already a translation table, so before creating a new table start by looking at existing tables to modify them as needed.

Typically, a braille translation table consists of several parts. First are header and includes, in which you write what the table is for, license information and include tables you need for your table.

Following this, you'll write various translation rules and lastly you write special rules to handle certain situations.

A translation rule is composed of at least three parts: the opcode (translation command), character(s) and braille dots. An opcode is a command you give to a machine or a program to perform something on your behalf. In liblouis, an opcode tells it which rule to use when translating characters into braille. An operand can be thought of as parameters for the translation rule and is composed of two parts: the character or word to be translated and the braille dots.

For example, suppose you want to read the word '`world`' using braille dots '`456`', followed by the letter '`W`' all the time. Then you'd write:

```
always world 456-2456
```

The word `always` is an opcode which tells liblouis to always honor this translation, that is to say when the word '`world`' (an operand) is encountered, always show braille dots '`456`' followed by the letter '`w`' ('`2456`').

When you write any braille table for any language, we'd recommend working from some sort of official standard, and have a device or a program in which you can test your work.

2.1 Overview

Many translation (contraction) tables have already been made up. They are included in the distribution in the tables directory and can be studied as part of the documentation. Some of the more 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

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 ⟨undefined⟩ [Opcode Index], page ⟨undefined⟩. 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.

> Please take a look at the public directory of the Unicode Character Database (`https://unicode.org/Public/UNIDATA/`) as well as at the Unicode names list with their code points (`https://unicode.org/Public/UNIDATA/NamesList.txt`) to figure out the corresponding Unicode code point for a given Unicode character.

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 ⟨undefined⟩ [`display`], page ⟨undefined⟩) is used it should precede character-definition opcodes. Braille-indicator opcodes should come next. Translation opcodes should follow. The `context` opcode (see ⟨undefined⟩ [`context`], page ⟨undefined⟩) 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 ⟨undefined⟩ [`correct`], page ⟨undefined⟩) 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 ⟨undefined⟩ [`include`], page ⟨undefined⟩) 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.

2.2 Hyphenation Tables

Hyphenation tables are necessary to make opcodes such as the `nocross` opcode (see ⟨undefined⟩ [`nocross`], page ⟨undefined⟩) 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 ⟨undefined⟩ [`include`], page ⟨undefined⟩) 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`.

2.3 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 ⟨undefined⟩ [`context`], page ⟨undefined⟩), the `correct` opcode (see ⟨undefined⟩ [`correct`], page ⟨undefined⟩), 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 ⟨undefined⟩ [`display`], page ⟨undefined⟩) 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.

You may have multiple definitions of a character using the same or different dot patterns. If you use different dot patterns for the same character, only the first dot pattern will be used during forward translation. However, during back-translation, all the relevant dot patterns will back-translate to the character you defined.

You can also define a character multiple times using the same dot pattern for the character, but using different character classes. The following example would define the character '`*`' (star) as both `math` opcode (see ⟨undefined⟩ [`math`], page ⟨undefined⟩) and `sign` opcode (see ⟨undefined⟩ [`sign`], page ⟨undefined⟩).

```
math * 16
sign * 16
```

Likewise, you can define multiple characters as the same dot pattern. The characters you define this way will be forward translated to the same dot pattern. However, when back-translating, the dot pattern will always back-translate to the first character that was defined with this pattern.

This technique may be useful when defining characters that have one representation in the Windows character set (CP1252) and another representation in the Unicode character set, e.g. the Euro sign, '€'. It may also be of use when you have to define several variants of the same letter with different accents, which may be represented in your Braille code by the same dot pattern. This is a very common practice for accented letters that are foreign to the Braille code. In the following example, both e acute ('é') and e grave ('è') are defined as dot 4 followed by dots 1 and 5.

```
lowercase \x00e9 4-15 # E acute
lowercase \x00e8 4-15 # E grave
```

In this example, the dot pattern would always back-translate to e acute, since this is the first definition. You could use the `correct` opcode (see ⟨undefined⟩ [`correct`], page ⟨undefined⟩) to correct at least the most common errors on that account. However, there is no fail-safe way to know what accented letter to use when you back-translate from a dot pattern representing more than one variant.

`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
```

`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.

`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
```

`grouping name characters dots ,dots`
: This opcode is different from the previous ones in that it defines two characters in one rule, and associates them with each other. The opcode is used to indicate pairs of grouping symbols used in processing mathematical expressions.

These symbols are usually generated by the MathML interpreter in liblouisutdml. They are used in multipass opcodes. The name operand must contain only letters (a-z and A-Z). The letters may be upper or lower-case but the case matters. The characters operand must contain exactly two Unicode characters. The dots operand must contain exactly two braille cells, separated by a comma.

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

`base attribute <derived character> <base character>`

This opcode is different in that it does not associate a character with a dot pattern, but it associates a character with another already defined character. The derived character inherits the dot pattern of the base character, and braille indicators (see ⟨undefined⟩ [Braille Indicator Opcodes], page ⟨undefined⟩) are used to distinguish them. The attribute operand refers to the character class (see ⟨undefined⟩ [Character-Class Opcodes], page ⟨undefined⟩) to which the character should be added. By defining braille indicator rules associated with this character class, you can determine the braille indicators to be inserted. The character operands are the derived character and the base character, respectively. A typical use of this opcode is for defining a pair of letters, a lowercase and the corresponding uppercase. For example:

```
lowercase a 1
base uppercase A a
```

2.4 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.

`modeletter attribute dots`
`capsletter dots`

The dot pattern which indicates that a certain mode is entered and ends after a single character. A "mode" is a state in which dot patterns must be interpreted a certain way. For example, in uppercase mode dots '`1`' is to be interpreted as a capital "A" and not a small "a". In numeric mode dots '`1`' is to be interpreted as a "1". The attribute operand identifies the mode and corresponds with the name of the character class that determines when the mode must be entered and exited.

`modeletter` is also used to mark every character when a mode must last for several characters but when there is no `begmode` definition, or when the sequence happens in the middle of a word and `begmodeword` is defined but no `endmodeword` (see ⟨undefined⟩ [`begmode`], page ⟨undefined⟩, ⟨undefined⟩ [`begmodeword`], page ⟨undefined⟩ and ⟨undefined⟩ [`endmodeword`], page ⟨undefined⟩)

`capsletter` is an alias for `modeletter uppercase`. The following two examples are equivalent:

```
capsletter 6
```

```
modeletter uppercase 6
```

`emphletter` (see ⟨undefined⟩ [`emphletter`], page ⟨undefined⟩) is the counterpart of `modeletter` for indication of emphasis.

`begmodeword attribute dots`
`begcapsword dots`

The dot pattern which indicates that a certain mode is entered for the following word or remainder of the current word. The mode is automatically terminated by the first character that is not a letter.

For uppercase mode, you can define a list of characters that can appear within a word in capitals without terminating the block. Do this by using the `capsmodechars` opcode (see ⟨undefined⟩ [`capsmodechars`], page ⟨undefined⟩).

Example:

```
begcapsword 6-6
```

`begemphword` (see ⟨undefined⟩ [`begemphword`], page ⟨undefined⟩) is the counterpart of `begmodeword` for indication of emphasis.

`endmodeword attribute dots`
`endcapsword dots`

The dot pattern which terminates a mode within a word. It is used in cases where the block is not terminated automatically by a word boundary, a number or punctuation. A common case is when an uppercase block is followed directly by a lowercase letter.

For example:

```
endcapsword 6-3
```

`endemphword` (see ⟨undefined⟩ [`endemphword`], page ⟨undefined⟩) is the counterpart of `endmodeword` for indication of emphasis.

`capsmodechars characters`

Normally, any character other than a letter will automatically cancel the `begcapsword` indicator. However, by using the `capsmodechars` opcode, you can specify a list of characters that are legal within a capitalized word. In some Braille codes, this might be the case for the hyphen character, '`-`'.

Example:

```
capsmodechars -
```

`begmode attribute dots`
`begcaps dots`

The dot pattern which indicates that a mode is entered until it is terminated by a `endmode` indicator. It is used in some Braille codes to mark a whole sentence or several words as capital letters. The block can contain capital letters as well as non-alphabetic characters, punctuation, numbers etc.

This is the most general opening mark, i.e. it can be used for opening at any position.

Example:

```
begcaps 6-6-6
```

`begemph` (see ⟨undefined⟩ [`begemph`], page ⟨undefined⟩) is the counterpart of `begmode` for indication of emphasis.

`endmode attribute dots`
`endcaps dots`
The dot pattern which terminates a mode.

```
endcaps 6-3
```

`endemph` (see ⟨undefined⟩ [`endemph`], page ⟨undefined⟩) is the counterpart of `endmode` for indication of emphasis.

`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 ⟨undefined⟩ [`word`], page ⟨undefined⟩) or `largesign` (see ⟨undefined⟩ [`largesign`], page ⟨undefined⟩), 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.

`nocontractsign dots`
The dots in this opcode are used to indicate a letter or a sequence of letters that are not a contraction, e.g. '`CD`' (see ⟨undefined⟩ [`contraction`], page ⟨undefined⟩). The opcode is similar to the `letsign` opcode (see ⟨undefined⟩ [`letsign`], page ⟨undefined⟩).

Note: This opcode was implemented in Liblouis specifically in order to support Unified English Braille (UEB). It may be used in any table, but may have unpredicted side-effects if used outside the intended context. Use with great care, and test thoroughly.

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

is terminated when a space, a letter or any other none-`litdigit` opcode (see ⟨undefined⟩ [`litdigit`], page ⟨undefined⟩) character is encountered.

You can define characters or strings to be part of a number by using the `midnum` opcode (see ⟨undefined⟩ [`midnum`], page ⟨undefined⟩), the `numericmodechars` opcode (see ⟨undefined⟩ [`numericmodechars`], page ⟨undefined⟩) or the `midendnumericmodechars` opcode (see ⟨undefined⟩ [`midendnumericmodechars`], page ⟨undefined⟩).

Example:

```
numsign 3456
```

`nonumsign dots`

Usually the end of a number doesn't need to be indicated as the reader expects the the number to end at a space character. However for mixed number word combinations you might want to have an indicator that lets the reader notice the end of the number.

For a word like "123abc" for example the reader expects an indicator between the number "123" and the characters "abc". This can be achieved using the `nonumsign`.

Example:

```
numsign 56
```

`numericnocontchars characters`

This opcode specifies the characters that require a `nonumsign` opcode (see ⟨undefined⟩ [`nonumsign`], page ⟨undefined⟩) if they appear after a number with no intervening space, e.g. '`1a`' or '`2-B`'.

These characters will typically be the letters a-j, which usually constitute the literary digits (see `litdigit` opcode (see ⟨undefined⟩ [`litdigit`], page ⟨undefined⟩)). However, in some Braille codes, all letters fall in this category.

Note: This opcode is case sensitive. So, if you need a `nonumsign` opcode (see ⟨undefined⟩ [`nonumsign`], page ⟨undefined⟩) to also appear before the capital letters a-j, you should include these letters in the definition. This is especially relevant if you are also using the `begcaps` and `endcaps` opcodes (see ⟨undefined⟩ [`begcaps`], page ⟨undefined⟩ and ⟨undefined⟩ [`endcaps`], page ⟨undefined⟩). In this case, you might otherwise end up having numbers immediately followed by capital letters with no indicator between.

Note: This opcode was implemented in Liblouis specifically in order to support Unified English Braille (UEB). It may be used in any table, but may have unpredicted side-effects if used outside the intended context. Use with great care, and test thoroughly.

Example:

```
numericnocontchars abcdefghij
```

`numericmodechars characters`
`midendnumericmodechars characters`

Any of these characters can appear within a number without terminating the effect of the number sign (see ⟨undefined⟩ [`numsign`], page ⟨undefined⟩). In other words, they don't cancel numeric mode.

The difference between the two opcodes is that `numericmodechars` opcode (see ⟨undefined⟩ [`numericmodechars`], page ⟨undefined⟩) characters can appear anywhere in a number whereas `midendnumericmodechars` opcode (see ⟨undefined⟩ [`midendnumericmodechars`], page ⟨undefined⟩) characters can appear only in the middle or at the end of a number. Like `midendnumericmodechars`, `numericmodechars` characters keep numeric mode active, but in addition they activate numeric mode immediately when at least one digit follows, and the number sign will precede the `numericmodechars` character in this case.

Note: This opcode was implemented in Liblouis specifically in order to support Unified English Braille (UEB). It may be used in any table, but may have unpredicted side-effects if used outside the intended context. Use with great care, and test thoroughly.

Example:

```
numericmodechars .,
midendnumericmodechars -/
```

2.5 Opcodes for *Standing Alone* Sequences

The term "standing alone" comes from the specification of Unified English Braille (UEB). In Liblouis, a letter or letters-sequence is considered to be *standing alone* if it is preceded and followed by a space, and/or other characters that you choose as delimiters, e.g. '`-`'. A *standing alone* sequence can be thought of as a word in a very broad sense. With the opcodes described in this section, you can decide what characters constitute a delimiter, and what characters can attach to the beginning or end of a word or *standing alone* sequence.

Note: The opcodes in this section were implemented in Liblouis specifically in order to support Unified English Braille (UEB). They may be used in any table, but may have unpredicted side-effects if used outside the intended context. Use with great care, and test thoroughly.

`seqdelimiter <characters>`

All the characters listed with this opcode designate a valid beginning and ending to a letter sequence used to determine when a letter sequence is *standing alone*. This again determines whether word contractions (see ⟨undefined⟩ [`word`], page ⟨undefined⟩) or nocontractsign (see ⟨undefined⟩ [`nocontractsign`], page ⟨undefined⟩) should be applied.

Spaces do not need to be listed as they are automatically delimiters.

For example, in UEB (section 2.6.1 page 15), any hyphen or dash count as delimiters.

This opcode can be used several times, but the characters must have already been defined.

Example:

```
seqdelimiter ----
```

`seqbeforechars <characters>`

Characters specified with this opcode may appear between a beginning sequence delimiter and the letter sequence itself.

For example, in UEB (2.6.2, page 15), opening parenthesis and opening quotations and such are allowed.

This opcode can be used several times, but the characters must have already been defined.

Example:

```
seqbeforechars ([{"‘’‘
```

`seqafterchars <chars>`

Characters specified with this opcode may appear between a letter sequence itself and a end sequence delimiter.

For example, in UEB (2.6.3, page 16), closing parenthesis and closing quotations and such are allowed.

This opcode can be used several times, but the characters must have already been defined.

Example:

```
seqafterchars  )]}"’’’.,;:.!?...
```

`seqafterpattern <string>`

Specifies that a specific string of characters can be between the letter sequence itself and an ending sequence delimiter.

For example, in UEB (section 2.6.4, page 18), the ‘`’d`’, ‘`’s`’, ‘`’ll`’, ‘`’ve`’, etc. can be after a letter sequence provided the overall sequence is *standing alone*.

This opcode may be used multiple times, once per pattern.

Example:

```
seqafterpattern ’d
```

2.6 Emphasis Opcodes

In many braille systems emphasis such as bold, italics or underline is indicated using special dot patterns that mark the beginning and if needed also the end. Some braille systems have several indicators for different situations, i.e. an indicator for an emphasized word and another one for an emphasized phrase. To accommodate for all these usage scenarios liblouis provides a number of opcodes.

At the same time some braille systems use different indicators for different kinds of emphasis while others know only one kind of emphasis. For that reason liblouis doesn't hard code any emphasis types but the table author defines which kind of emphasis exist for a specific language using the `emphclass` opcode (see ⟨undefined⟩ [`emphclass`], page ⟨undefined⟩).

2.6.1 Emphasis classes

The `emphclass` opcode defines the classes of emphasis that are relevant for a particular language. For all emphasis that need special indicators an emphasis class has to be declared.

`emphclass <emphasis class>`

Define an emphasis class to be used later in other emphasis related opcodes in the table.

```
emphclass italic
```

```
emphclass underline
emphclass bold
emphclass transnote
```

2.6.2 Emphasis indicators

In order to understand the capabilities of Liblouis for emphasis handling we have to look at the different kinds of indicators that exist and the different scopes. There are various indicators to mark the beginning of emphasis. Where the emphasis ends depends on the type of start indicator (the scope) and on the possible occurrence of an end indicator. Sometimes an indicator appears in the middle of an emphasized part in order to change the current scope.

Which indicators Liblouis uses in which situation depends on the specific combination of indicators that are defined in the table.

2.6.2.1 Permanent indicator

Many languages simply have an indicator for the beginning of emphasis and another one for the end of the emphasis. The emphasis is "permanent" in the sense that it needs an end indicator to cancel it. It is not implicitly canceled by any character or space.

Characters that are defined as *not emphasizable* in braille (see ⟨undefined⟩ [`noemphchars`], page ⟨undefined⟩) are not indicated as such by Liblouis. This means that if an emphasized phrase begins or ends with such characters, they will not be within the part enclosed by the two indicators. Also, if multiple emphasized parts are separated by unemphasizable characters only, it will be indicated as if it was a single emphasized phrase, with one start indicator and one end indicator.

A table can not specify both a permanent indicator and a word indicator (see ⟨undefined⟩ [Word indicator], page ⟨undefined⟩) for a certain emphasis class.

`begemph <emphasis class> <dot pattern>`

Braille dot pattern to indicate the beginning of emphasis.

```
begemph italic 46
```

A `begemph` rule must always be combined with a `endemph` rule.

`endemph <emphasis class> <dot pattern>`

Braille dot pattern to indicate the end of emphasis.

```
endemph italic 46-36
```

`noemphchars <emphasis class> characters`

Normally, emphasis is indicated on all characters except spaces (characters with a `space` attribute, see ⟨undefined⟩ [`space`], page ⟨undefined⟩). You can change this with the `noemphchars` opcode. When this opcode is specified, emphasis is indicated on all characters except the ones in the list. That means that emphasis is also indicated on spaces unless the list contains space characters (escaped, e.g. `\s`).

Example:

```
noemphchars italic \s'()
```

2.6.2.2 Letter indicator

Some languages have special indicators for single letter emphasis. The letter indicator will be chosen over other indicators when the next character is emphasized, but not the characters thereafter.

In some situations the letter indicator is the only way to indicate emphasis. For instance when emphasis ends within the middle of a word, and a word indicator exists (see ⟨undefined⟩ [Word indicator], page ⟨undefined⟩), but no way to cancel it explicitly. In this case Liblouis will use the letter indicator for every emphasized character in the word.

A table is even allowed to define only a letter indicator and no word or permanent indicators (see ⟨undefined⟩ [Permanent indicator], page ⟨undefined⟩), in which case the letter indicator will be used for any emphasis. It is not very likely that there are braille systems that work this way, but this feature can be useful anyway.

`emphletter <emphasis class> <dot pattern>`
: Braille dot pattern to indicate that the next character is emphasized.

```
emphletter italic 46-25
```

2.6.2.3 Word indicator

Many languages have special indicators for emphasized words. The scope of a word indicator is normally until the next unemphasizable character, however this can be changed with the `emphmodechars` opcode (see ⟨undefined⟩ [`noemphchars`], page ⟨undefined⟩ and ⟨undefined⟩ [`emphmodechars`], page ⟨undefined⟩).

Word emphasis can also be canceled with an explicit closing indicator. Usually a word indicator is put at the the beginning of the word, but it may also be used for cases where the emphasis starts in the middle of the word.

`begemphword <emphasis class> <dot pattern>`
: Braille dot pattern to indicate the beginning of an emphasized word or the beginning of emphasized characters within a word.

```
begemphword underline 456-36
```

`endemphword <emphasis class> <dot pattern>`
: Word emphasis ends implicitly when the word ends. When an indication is required to close word emphasis, i.e. when emphasis ends in the middle of a word, then this opcode defines the braille dot pattern that is used.

```
endemphword transnote 6-3
```

When emphasis ends in the middle of a word and there is no `endemphword` definition, a letter indicator must be defined (see ⟨undefined⟩ [Letter indicator], page ⟨undefined⟩).

`emphmodechars <emphasis class> characters`
: Normally, only spaces and unemphasizable characters (see ⟨undefined⟩ [`space`], page ⟨undefined⟩ and ⟨undefined⟩ [`noemphchars`], page ⟨undefined⟩) will cancel the `begemphword` indicator (see ⟨undefined⟩ [`begemphword`], page ⟨undefined⟩). However this can be overruled with the `emphmodechars` opcode. If `emphmodechars` is specified, any character that is not in the specified list and is not a letter (see ⟨undefined⟩ [`uppercase`], page ⟨undefined⟩, ⟨undefined⟩

[lowercase], page ⟨undefined⟩ or ⟨undefined⟩ [letter], page ⟨undefined⟩) will cancel the begemphword indicator. Conversely, letters and characters that are in the list will not cancel the word indicator.

Example:

```
emphmodechars underline -
```

2.6.2.4 Phrase indicator

Some languages have a concept of a phrase where the emphasis is valid for a number of words. The beginning of the phrase is indicated with a braille dot pattern and a closing indicator is put before or after the last word of the phrase. A phrase only contains whole words. The phrase indicator is a special kind of permanent indicator that must be used in combination with a word indicator (see ⟨undefined⟩ [Permanent indicator], page ⟨undefined⟩, and ⟨undefined⟩ [Word indicator], page ⟨undefined⟩).

A word is defined as a character sequence that starts and ends with emphasizable characters and does not contain characters that are both unemphasizable and resetting (see ⟨undefined⟩ [emphmodechars], page ⟨undefined⟩, and ⟨undefined⟩ [noemphchars], page ⟨undefined⟩).

To define how many words are considered a phrase in your language use the lenemphphrase opcode (see ⟨undefined⟩ [lenemphphrase], page ⟨undefined⟩).

`begemphphrase <emphasis class> <dot pattern>`

Braille dot pattern to indicate the beginning of a phrase.

```
begemphphrase bold 456-46-46
```

A begemphphrase rule must always be combined with a endemphphrase rule.

`endemphphrase <emphasis class> before <dot pattern>`

Braille dot pattern to indicate the end of a phrase. The closing indicator will be placed before the last word of the phrase.

```
endemphphrase bold before 456-46
```

If a table specifies endemphphrase before and the dot pattern is the same as that of begemphword, the word scope applies whenever this indicator is used (see ⟨undefined⟩ [Word indicator], page ⟨undefined⟩).

`endemphphrase <emphasis class> after <dot pattern>`

Braille dot pattern to indicate the end of a phrase. The closing indicator will be placed after the last word of the phrase. If both endemphphrase <emphasis class> before and endemphphrase <emphasis class> after are defined an error will be signaled.

```
endemphphrase underline after 6-3
```

`lenemphphrase <emphasis class> <number>`

Define how many words are required before a sequence of words is considered a phrase.

```
lenemphphrase underline 3
```

With the above rule, a sequence of two emphasized words will not be indicated as a phrase, but the words will be indicated individually.

2.6.3 Note about fallback behavior

Contrary to older versions of liblouis there is limited fallback behavior. Generally opcodes have a very specific purpose. This is done for the sake of simplicity. While it would be possible to support more combinations of rules, Liblouis chooses to signal an error when certain combinations of opcodes are used (or not used).

For example, when a `begemphphrase` rule is defined it is required that there is also an `endemphphrase` definition. `begemph` must be combined with `endemph`. A `begemphphrase` rule is only allowed if there is also a `begemphword`. `begemph` and `begemphword` are mutually exclusive. Etc.

When new requirements for indicating emphasis arise that are not supported yet, either more opcode combinations might be enabled, or more specific opcodes might be added.

2.6.4 Computer braille

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 ⟨undefined⟩ [`compbrl`], page ⟨undefined⟩).

`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 ⟨undefined⟩ [`typeform` parameter], page ⟨undefined⟩) or inferred because it contains a subsequence specified by the `compbrl` opcode (see ⟨undefined⟩ [`compbrl`], page ⟨undefined⟩).

`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 ⟨undefined⟩ [`typeform` parameter], page ⟨undefined⟩) or inferred because it contains a subsequence specified by the `compbrl` opcode (see ⟨undefined⟩ [`compbrl`], page ⟨undefined⟩).

2.7 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. It is useful if your Braille code requires the decimal separator to show as a dot pattern different from the normal representation of this character, i.e. period or comma. In addition, it allows the notation '`.001`' to be translated correctly. This notation is common in some languages instead of '`0.001`' (no leading 0). When you use the `decpoint` opcode, the decimal point will be taken to be part of the number and correctly preceded by number sign.

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.

2.8 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.

2.9 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 ⟨undefined⟩ [Character-Definition Opcodes], page ⟨undefined⟩) within the sequence is to be used. It is an error if not all the characters in the rule have been previously defined in a character-definition rule. Note that the '`=`' shortcut for dot patterns has a known bug[1] that might cause problems when back-translating.

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 back-translation is not to use information on this line.

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

[1] See `https://github.com/liblouis/liblouis/issues/500#issuecomment-365753137`.

`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.

`nocross opcode characters ...`

`nocross` is an opcode prefix which modifies the operation of the opcode following it. The following opcode can be `always` opcode (see ⟨undefined⟩ [`always`], page ⟨undefined⟩) or any other translation opcode listed below with `characters` as the first operand. `nocross` specifies that the operation should not be done when the characters are not all in one syllable (when they cross a syllable boundary). For this opcode to work, a hyphenation table must be included. If this is not done, `nocross` is ignored. For example, if the English Grade 2 table is being used and the appropriate hyphenation table has been included `nocross always sh 146` will cause the '`sh`' in '`monkshood`' not to be contracted.

`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 ⟨undefined⟩ [Character-Definition Opcodes], page ⟨undefined⟩, if 8-dot computer braille is enabled or according to the dot patterns given in the `comp6` opcode (see ⟨undefined⟩ [`comp6`], page ⟨undefined⟩), 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. The first operand must be a single character. 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 (see ⟨undefined⟩ [virtual dots], page ⟨undefined⟩).

`nocont characters`

Like `compbrl`, except that the string is uncontracted. `prepunc` opcode (see ⟨undefined⟩ [`prepunc`], page ⟨undefined⟩) and `postpunc` opcode (see ⟨undefined⟩ [`postpunc`], page ⟨undefined⟩) 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 ⟨undefined⟩ [`correct`], page ⟨undefined⟩).

`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 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`

For some braille systems there is the requirement to remove repeated words which are connected by some character. In Malaysian braille for example you want to contract as follows:

Text: '`tasik-tasik`'

Grade 1: '`2345-1-234-24-13-36-2345-1-234-24-13`'

Grade 2: '`2345-1-234-24-13-123456`'

The dash and the second occurrence of '`tasik`' is replaced with dots '`123456`'. To achieve this you can use the repword opcode as follows:

```
repword - 123456
```

`repeatedword characters dots, dots`

Like the `repword` opcode (see ⟨undefined⟩ [`repword`], page ⟨undefined⟩), but for indicating a repetition of a string at the end of a word. When characters are encountered check to see if a part of the word before these characters (but not the whole word) matches the string after them. If so, insert the first dot pattern at the position where the part of the word started, replace characters with the second dot pattern and eliminate the repeated string and any string following another occurrence of characters that is the same. This opcode is used in Malaysian braille. In this case the rule is:

```
repeatedword - 25,123456
```

`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 ⟨undefined⟩ [Translation Opcodes], page ⟨undefined⟩):

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

`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 ⟨undefined⟩ [`letsign`], page ⟨undefined⟩)) 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 liblouisutdml 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)
```

2.10 Character-Class Opcodes

These opcodes define and use character classes, also known as character attributes. 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. 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

The opcodes which define and use character classes are shown below. For examples see `el.ctb`.

`attribute name characters`
Add characters to a character class. The class may be one of the predefined classes listed above, a user-defined class previously created with this opcode, or a new one. The name operand must contain only letters (a-z and A-Z, case matters). For historical reasons and to support the `match` opcode (see ⟨undefined⟩ [`match`], page ⟨undefined⟩) it can also be a number between 0 and 7. The characters operand must be specified as a string. Each character in the string, as well as its dot counterpart if it occupies a single cell, is added to the character class.

A user-defined character class may not be used in other rules until it has been created with a `attribute` rule. Numbered classes can be used in `match` rules (see ⟨undefined⟩ [`match`], page ⟨undefined⟩). Named classes can be used with the `before` and `after` opcodes and in `context` and multipass rules (see ⟨undefined⟩ [The Context and Multipass Opcodes], page ⟨undefined⟩).

`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.

2.11 Swap Opcodes

The swap opcodes are needed to tell the `context` opcode (see ⟨undefined⟩ [`context`], page ⟨undefined⟩), the `correct` opcode (see ⟨undefined⟩ [`correct`], page ⟨undefined⟩) 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.

All the swap opcodes have a name so they can be refered to from the `context`, `correct` and multipass opcodes. The name operand must contain only letters (a-z and A-Z). The letters may be upper or lower-case but the case matters.

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.

2.12 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 ⟨undefined⟩ [Translation Opcodes], page ⟨undefined⟩) 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. When back-translating, the passes are performed in the reverse order, i.e. `pass4`, `pass3`, `pass2`, `context`. Each of these opcodes must be prefixed by either the `noback` opcode (see ⟨undefined⟩ [`noback`], page ⟨undefined⟩) or the `nofor` opcode (see ⟨undefined⟩ [`nofor`], page ⟨undefined⟩). 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 in the test and action parts of the `correct` opcode, in the test part of the `context` opcode when forward translating, and in the action part of the `context` opcode when back translating.

`@ (at sign)`
: a sequence of dot patterns. Cells are separated by hyphens as usual. This suboperand is valid in the test and action parts of the `pass2`, `pass3`, and `pass4` opcodes, in the action part of the `context` opcode when forward translating, and in the test part of the `context` opcode when back translating.

`` ` (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. For a list of all valid attributes see ⟨undefined⟩ [valid attribute characters], page ⟨undefined⟩. 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 character class, predefined or defined using the `attribute` opcode (see ⟨undefined⟩ [`attribute`], page ⟨undefined⟩), or the name of a swap set defined by the swap opcodes (see ⟨undefined⟩ [Swap Opcodes], page ⟨undefined⟩). Names must contain only letters (a-z and A-Z). The letters may be upper or lower-case but the case matters. Character 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.

  The brace actions, `{name` and `}name`, refer to named groupings. A grouping is created with the `grouping` opcode (see ⟨undefined⟩ [`grouping`], page ⟨undefined⟩) and contains exactly two characters which represent the opening character and the matching closing character for a character grouping. The first operand is the grouping name, the second is the two (opening and closing) characters, and the third is the two dot patterns separated by a comma.

  Let's say that you'd like to define the opening and closing parentheses via multipass rules, and that you'd like to use dots '`123478`' for the opening parenthesis and dots '`145678`' for the closing parenthesis. One way to do so is like this:

  ```
  grouping parentheses () 123478,145678
  noback correct {parentheses {parentheses
  noback correct }parentheses }parentheses
  ```

  The references within the test part of the multipass rule match against the characters (the second operand) of the grouping rule, and the references within the action part replace with the dot patterns (the third operand) of the grouping.

} (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 default is to move backward one character. This suboperand is valid only in test parts. The test fails if moving backward beyond the beginning of the input string.

[(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 (0 through 49), e.g. `#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 within the same pass about itself. This feature is used in math translation, and 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`. All of the variables are initialized to 0 at the start of each pass.

Variables can also be incremented and decremented by one in the action part with expressions like `#1+` and `#3-` respectively. An attempt to decrement a variable below 0 is silently ignored.

Variables are tested in the test part with conditional expressions like: `#1=2`, `#3<4`, `#5>6`, `#7<=8`, `#9>=10`.

** (asterisk)*

Copy the input characters or dot patterns within the replacement brackets into the output, and discard anything else that was matched. If there are no replacement brackets then copy all of the matched input. This suboperand is only valid within the action part. It may be specified any number of times. This feature is used, for example, for handling numeric subscripts in Nemeth.

? (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 valid 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 character class
`x`	second user-defined character class
`y`	third user-defined character class
`z`	fourth user-defined character 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. If an expression has square brackets to indicate the part to be replaced, and the opening bracket would correspond with a position before the cursor, it is not a match.
b. If there is no match: shift the cursor one position to the right and continue the loop
c. If there are matches: choose the longest match
d. Do the replacement. If the expression has square brackets, the part of the input that matches the part in between the brackets is replaced with the right-hand side of the rule. If the expression has no square brackets, the whole match is replaced.
e. Place the cursor after the replaced text
f. continue loop

Normally, when a rule is applied, the characters in the input that the rule applies to are "consumed", i.e. the position of the input string is stepped forward, and the characters are no longer available for subsequent rules. However, with the multipass opcodes, the `context` opcode (see ⟨undefined⟩ [`context`], page ⟨undefined⟩) opcode and the `correct` opcode (see ⟨undefined⟩ [`correct`], page ⟨undefined⟩) opcode, it is possible to make rules which don't consume any characters from the input. This could happen, e.g. if you use the `context` opcode (see ⟨undefined⟩ [`context`], page ⟨undefined⟩) opcode to insert a dot pattern before a special group of characters. In these cases, Liblouis will always advance the position by one character to make sure that the program doesn't apply a rule to the same characters again and again.

2.13 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. If any back-translation corrections have been specified then they are applied in a post-translation (i.e. the very last) pass.

Note that like the `context` opcode (see ⟨undefined⟩ [`context`], page ⟨undefined⟩) and multi-pass opcodes, the `correct` opcode must be preceded by `noback` opcode (see ⟨undefined⟩ [`noback`], page ⟨undefined⟩) or `nofor` opcode (see ⟨undefined⟩ [`nofor`], page ⟨undefined⟩).

The format of the `correct` opcode is very similar to that of the `context` opcode (see ⟨undefined⟩ [`context`], page ⟨undefined⟩). 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:

```
noback correct "\\" ? Eliminate backslashes
noback correct "cornf" "comf" fix a common "scano"
noback correct "cornm" "comm"
noback correct "cornp" "comp"
```

```
noback correct "*" ? Get rid of stray asterisks
noback correct "|" ? ditto for vertical bars
noback correct "\s?" "?" drop space before question mark
```

2.14 The match Opcode

For historical reasons despite being fairly similar in functionality both the `context` opcode (see ⟨undefined⟩ [`context`], page ⟨undefined⟩) and the match opcode exist and are in use in modern braille tables. But in the future they might be merged under some common opcode. For that reason consider the match opcode *somewhat experimental*.

`match pre-pattern characters post-pattern dots`

This opcode allows for matching a string of characters via *pre* and *post patterns*. The patterns are specified using an expression syntax somewhat like regular expressions (see ⟨undefined⟩ [pattern expression syntax], page ⟨undefined⟩). A single hyphen (‘`-`’) by itself means no pattern is specified.

The following will replace ‘`xyz`’ with the dots ‘`1346-13456-1356`’ when it appears in the string ‘`abxyzcd`’.

```
match ab xyz cd 1346-13456-1356
```

The following will replace ‘`ONE`’ with ‘`3456-1`’ when it starts the input and is followed by ‘`:`’

```
match ^ ONE : 3456-1
```

The `pre-pattern` and the `post-pattern` can contain any of the following expressions:

‘`[ ]`’ Expression can be any of the characters between the brackets. If only one character present then the brackets are not needed unless it is a special character, in which it should be escaped with the backslash.

‘`.`’ Expression can be any character.

‘`%[ ]`’ Expression is a character with the attributes listed between the brackets. If only one character is present then the brackets are not needed. The set of attributes are specified as follows:

- ‘`_`’ space
- ‘`#`’ digit
- ‘`a`’ letter
- ‘`u`’ uppercase
- ‘`l`’ lowercase
- ‘`.`’ punctuation
- ‘`$`’ sign
- ‘`~`’ seqdelimiter
- ‘`<`’ seqbeforechars
- ‘`>`’ seqafterchars

‘^’ Match at the end of input processing (or beginning depending of the direction pre or post).

‘$’ Same as ‘^’.

For example the following will replace ‘bb’ with the dots ‘23’ when it is between letters.

```
match %a bb %a 23
```

The following will replace ‘con’ with the dots ‘25’ when it is preceded by a space or beginning of input, and followed by an ‘s’ and then any letter.

```
match %[^_] con s%a 25
```

Similar to regular expressions the pattern expressions can contain grouping, quantifiers and even negation:

‘( )’ Expressions between parentheses are grouped together as one expression.

‘!’ The following expression is negated.

‘?’ The previous expression must match zero or one times.

‘*’ The previous expression must match zero or more times.

‘+’ The previous expression must match one or more times.

‘|’ Either the previous or the following expressions must match.

For example the following will replace ‘ing’ with the dots ‘346’ when it is *not* preceded by a space or beginning of input. What follows after the ‘ing’ does not matter, hence the ‘-’.

```
match !%[^_] ing - 346
```

The following will replace ‘con’ with the dots ‘25’ when it is preceded by a space, or beginning of input; then followed by a ‘c’ that is followed by any character but ‘h’.

```
match %[^_] con c!h 25
```

2.15 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.

`undefined dots`

If this opcode is used in a table any characters which have not been handled 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 ⟨undefined⟩ [Character-Definition Opcodes], page ⟨undefined⟩. 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-g2.ctb` we have `multind 56-6 letsign capsletter`. 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 Notes on Back-Translation

3.1 General Notes

Back-translation refers to the process of translating backwards, i.e. from Braille to text. For many years, Liblouis was mainly concerned with forward translation, and so were most of the authors of the translation tables. Today however, Liblouis is being used extensively in conjunction with screen reading programs like NVDA and JAWS for Windows as well as Braille note-takers like BrailleSense from HIMS and BrailleNote from HumanWare. So when writing a translation table for Liblouis, it is indeed relevant to consider how the table will work when used for back-translation, if anything special must be done, or if you want to write separate tables for forward translation and back-translation.

Back-translation is generally harder to do in a computer program than forward translation. Ideally, any text could be translated to Braille and then translated back to text giving exactly the same result as the original. However, many Braille codes omit a lot of information and leaves it to the reader to fill in the missing bits. An example of this is letters with accents. In languages where accents are uncommon, e.g. English, Accented letters are usually just marked with a Braille indicator stating that there is an accent, but not which accent, even though this may be crucial to the meaning of the word or the sentence. Another example of this is when not all capital letters are marked in the Braille code, but only the "important" capital letters. A third example is when a Braille character serves as both a punctuation sign, a math sign, and perhaps even as a contraction, and the Braille code then leaves it up to the reader to use his/her knowledge of the context to decide the meaning of the Braille character.

In some cases, you may need to bend the rules of the Braille code if it is important to create Braille that can be properly back-translated. This may include marking all capital letters instead of just the "important" ones, or perhaps marking a Braille character with an indicator stating that this character should in fact be interpreted as a math sign and not a punctuation or Braille contraction. In some cases, the best solution may be to create two separate sets of tables for forward translation: One set for Braille that must be back-translatable (for use with screen readers and note-takers), and another for good and nice literary Braille (for embossing). But no matter how you bend the Braille code, the back-translation process may not be perfect.

3.2 Back-translation with Liblouis

Back-translation is carried out by the function `lou_backTranslateString`. Its calling sequence is described in ⟨undefined⟩ [Programming with liblouis], page ⟨undefined⟩. `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.

Most opcodes can be preceded by `noback` opcode (see ⟨undefined⟩ [`noback`], page ⟨undefined⟩) or `nofor` opcode (see ⟨undefined⟩ [`nofor`], page ⟨undefined⟩), and the `correct`, `context` and multi-pass opcodes must be preceded with either `noback` or `nofor`. So in most cases, it will be perfectly possible to make one table for translation in both directions,

although a separate table for forward and backward translation might be more readable in some cases.

Most of the opcodes associated with pass 1 have two operands, a character operand to the left and a dots operand to the right. During forward translation, these operands are used to replace the characters with the dot pattern according to the conditions of the opcode. The opcode works from left to right. When back-translating, these opcodes work the opposite way. The dot patterns are replaced by the text. The opcodes work from right to left.

On the other hand, the `correct`, `context` and multi-pass opcodes have a test part to the left and an action part to the right. These opcodes work from left to right in both translation directions. The test is performed, and if true, the action is executed, i.e. replacing, inserting or deleting characters or dots. This is why a translation direction always has to be specified with these opcodes using `noback` or `nofor`.

4 Table Metadata

Translation tables may contain metadata. This makes them discoverable. Programs may for example use the Liblouis function ⟨undefined⟩ [lou_findTable], page ⟨undefined⟩, to find a table based on a special query of which the ⟨undefined⟩ [Query Syntax], page ⟨undefined⟩, is described below.

4.1 Syntax

Metadata must be defined in special comments within the table header. The table header is the area at the top of the file, before the first translation rule, consisting of only comments or empty lines. Any metadata within included tables is ignored.

A metadata field must be defined on its own line, starting with `#+`. It has the following syntax:

```
#+<key>: <value>
```

where '`<key>`' and '`<value>`' are sequences of one or more characters `a` to `z`, `A` to `Z`, `0` to `9`, `.`, `-`, and `_`. The colon that separates the key and value may have zero or more spaces or tabs on either side.

A value is optional. In case of no value the colon must be omitted as well:

```
#+<key>
```

The same key may appear multiple times in a table.

There is no restriction on which keys and values are allowed, as long as the syntax is correct. However in order to be really useful there must be some standard keys and values. A possible grammar is proposed on the wiki page Standard metadata tags (`https://github.com/liblouis/liblouis/wiki/Table-discovery-based-on-table-metadata#standard-metadata-tags`).

4.2 Query Syntax

A query that is passed to the ⟨undefined⟩ [lou_findTable], page ⟨undefined⟩, function must have the following syntax:

```
<feature1> <feature2> <feature3> ...
```

where '`<feature>`' is either:

```
<key>:<value>
```

or:

```
<key>
```

Features are separated by one or more spaces or tabs. No spaces are allowed around colons.

5 Testing Translation Tables interactively

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 `file2brl`, which is part of the liblouisutdml package (see Section "Introduction" in *Liblouisutdml 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.

Most test programs let you specify one or multiple tables to use. These tables are usually found in standard locations in the file system or local to where the command is executed. See ⟨undefined⟩ [How tables are found], page ⟨undefined⟩, for a description on how the tables are located.

5.1 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 ⟨undefined⟩ [common options], page ⟨undefined⟩.

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 ⟨undefined⟩ [`correct`], page ⟨undefined⟩) 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 ⟨undefined⟩ [`display`], page ⟨undefined⟩).

`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 `capsletter` opcode (see ⟨undefined⟩ [`capsletter`], page ⟨undefined⟩) and the `numsign` opcode (see ⟨undefined⟩ [`numsign`], page ⟨undefined⟩). It also shows emphasis dot patterns, such as those for the `begemphword` opcode (see ⟨undefined⟩ [`begemphword`], page ⟨undefined⟩), the `begemphphrase` opcode (see ⟨undefined⟩ [`begemphphrase`], page ⟨undefined⟩), 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.

5.2 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,...]
```

Aside from the standard options (see ⟨undefined⟩ [common options], page ⟨undefined⟩) `lou_trace` also accepts the following options:

`--forward`
`-f` Trace a forward translation.

`--backward`
`-b` Trace a backward translation.

If no options are given forward translation is assumed.

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
```

5.3 lou_checktable

To use this program type the following:

```
lou_checktable [OPTIONS] TABLE
```

Aside from the standard options (see ⟨undefined⟩ [common options], page ⟨undefined⟩) `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.

5.4 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 ⟨undefined⟩ [common options], page ⟨undefined⟩.

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.

5.5 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
```

where '`TABLE`' is either:

`QUERY` a ⟨undefined⟩ [Query Syntax], page ⟨undefined⟩,

`FILE[,FILE,...]`
a comma-separated list of table files

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

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

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

If no options are given forward translation is assumed.

Use the following command to do a forward translation of English text to grade 2 contracted braille according to the U.S. braille standard.

```
lou_translate language:en grade:2 region:en-US < input.txt
```

Use the following command to do a forward translation with translation table `en-us-g2.ctb`.

```
lou_translate --forward en-us-g2.ctb < input.txt
```

When you specify the table as a query, the braille encoding is always Unicode dot patterns. When you specify the table as a file list, the encoding of the resulting braille depends on the character definitions in the given table. It is recommended to use a display table, as in the following example, if you require a specific braille encoding.

The next example illustrates a forward translation with translation table `en-us-g2.ctb` and display table `unicode.dis`. The resulting braille is encoded as Unicode dot patterns (as defined in `unicode.dis`).

```
lou_translate --forward unicode.dis,en-us-g2.ctb < input.txt
```

Use a pipe if you would rather just pass some given text to the translator.

```
echo "The quick brown fox jumps over the lazy dog" | lou_translate -f unicode.dis,en-u
```

The result will be written to standard output:

Backward translation can be done as follows:

```
echo ",! qk br{n fox jumps ov} ! lazy dog" | lou_translate --backward en-us-g2.ctb
```

which results in

```
The quick brown fox jumps over the lazy dog
```

You can also do a backward translation using Unicode dot patterns

```
echo "   " | lou_translate --backward unicode.dis,en-us-g2.ctb
```

resulting in

```
The quick brown fox
```

5.6 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 ⟨undefined⟩ [common options], page ⟨undefined⟩.

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

5.7 lou_checkyaml

This program tests a liblouis table against a corpus of known good Braille translations defined in YAML format. For a description of the format refer to ⟨undefined⟩ [YAML Tests], page ⟨undefined⟩. The program returns 0 if all tests pass or 1 if any of the tests fail. If `libyaml` is not installed the program will simply skip all tests. Invoke it as follows:

```
lou_checkyaml YAML_TEST_FILE
```

The command line options that are accepted by `lou_checkyaml` are described in ⟨undefined⟩ [common options], page ⟨undefined⟩.

Due to some technical limitations the YAML tests work best if the `LOUIS_TABLEPATH` is set up correctly. By running `make` this is all taken care for you. You can also run individual YAML tests as shown in the following example:

```
cd tests
make check TESTS=yaml/en-ueb-g2_backward.yaml
```

6 Automated Testing of 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.

The easiest way to test the translation tables is to write a YAML file where you define the table that is to be tested and any number of words or phrases to translate together with their respective expected translation.

The YAML tests are data driven, i.e. you give the test data, a string to translate and the expected output. The data is in a standard format namely YAML. If you have `libyaml` installed they will automatically be invoked as part of the standard `make check` command.

6.1 YAML Tests

YAML (`http://yaml.org/`) is a human readable data serialization format that allows for an easy and compact way to define tests.

A YAML file first defines which tables are to be used for the tests. Then it optionally defines flags such as the '`testmode`'. Finally all the tests are defined.

You can repeat the cycle as many times as you like (tables, optional flags, tests). You can also define several rounds of tests for any table, with or without the optional flags. Just remember that the flags are reset to their default values each time you start a new round of tests or load a new set of tables.

Let's just look at a simple example how tests could be defined:

(For technical reasons the Unicode braille in the expected translation in the following YAML examples is not displayed correctly. Please refer to the example YAML file `example_test.yaml` *in the* `tests` *directory of the source distribution or read these examples in another version of the documentation such as HTML)*

```
# comments start with '#' anywhere on a line
# first define which tables will be used for your tests
table: [unicode.dis, en-ueb-g1.ctb]

# then optionally define flags such as testmode. If no flags are
# defined forward translation is assumed

# now define the tests
tests:
  - # each test is a list.
    # The first item is the string to translate. Quoting of strings is
    # optional
    - hello
    # The second item is the expected translation
    -
  - # optionally you can define additional parameters in a third
    # item such as typeform or expected failure, etc
    - Hello
    -
```

```
    - {typeform: {italic: '++++ '}, xfail: true}
  - # a simple, no-frills test
    - Good bye
    -
  # same as above using "flow style" notation
  - [Good bye,   ]
```

The four basic components of a test file are as follows:

'`table`' A list containing table files, which the tests should be run against. This is usually just one file, but for some situations more than one file can be required. For example:

```
table: [hu-hu-g1.ctb, hyph_hu_HU.dic]
```

It is also possible to specify a table inline. ⟨undefined⟩ [Inline definition of tables], page ⟨undefined⟩, below explains how to do this.

A third way to specify a table is by its metadata. A table query, which is essentially as list of "features", is matched against the ⟨undefined⟩ [Table Metadata], page ⟨undefined⟩, defined inside the tables contained in `LOUIS_TABLEPATH`. Only the best match is used for the test.

The syntax of the query is a variation of the ⟨undefined⟩ [Query Syntax], page ⟨undefined⟩, used for the ⟨undefined⟩ [lou_findTable], page ⟨undefined⟩, function:

```
table:
  locale: fr
  grade: 1
```

'`display`' A display table, which should be used to encode braille in the test. This item is optional. It is only taken into account for the '`forward`', '`backward`' and '`bothDirections`' test modes. If it is present it should be the first item of the file. If it is not present, the braille encoding of each test is determined by the table that is being tested.

The next example shows how to test the `en-ueb-g1.ctb` table using ASCII notation (as defined in `en-ueb-g1.ctb` itself):

```
table: [en-ueb-g1.ctb]
```

If you wanted to test the `en-ueb-g1.ctb` table using Unicode dot patterns then you would use the following definition:

```
display: unicode.dis
table: [en-ueb-g1.ctb]
```

'`flags`' The flags that apply for all tests in this file. At the moment only the '`testmode`' flag is supported. It can have four possible values:

'`forward`' This indicates that the tests are for forward translation.

'`backward`'
This indicates that the tests are for backward translation.

'`bothDirections`'
Checks that forward translating the input yields the expected output, and backward translating the output yields the input again.

'`display`' Checks that a display table maps characters to the expected dot patterns. The input is a string of characters, the output are the corresponding dot patterns, as a string of Unicode braille symbols. Virtual dots in the output are ignored.

'`hyphenate`'
This indicates that the tests are for hyphenation.

'`hyphenateBraille`'
This indicates that the tests are for hyphenation and the input is braille.

If no flags are defined forward translation is assumed.

'`tests`' A list of tests. Each test consists of a list of two, three or in some cases four items. The first item is the test input. The second item is the expected output. Braille input and output can be either Unicode braille or an ASCII-braille like encoding, depending on the specified test mode, translation mode and display table. Quoting strings is optional. Comments can be inserted almost anywhere using the '`#`' sign. A simple test would look at follows:

```
- # a simple, no-frills test
  - Good bye
  -
```

Using the more compact "flow style" notation it would look like the following:

```
- [Good bye,  ]
```

An optional third item can contain additional options for a test such as the typeform, or whether a test is expected to fail. The following shows a typical example:

```
-
  - Hello
  -
  - {typeform: {italic: '++++ '}, xfail: true}
# same test more compact
- [Hello, , {typeform: {italic: '++++ '}, xfail: true}]
# same test but is only expected to fail for backtranslation
- [Hello, , {typeform: {italic: '++++ '}, xfail: {backward: true}}]
```

The valid additional options for a test are as follows:

'`xfail`' Sometimes it is known that a test is failing. Maybe the table under test doesn't handle that word correctly yet, or maybe backtranslation has not been implemented. '`xfail`' is designed to mark tests as expected failures. There are three ways to mark a test as failing:

1. Simply Mark a test as expected failure by setting '`xfail`' to '`true`'.

   ```
   - [Hello, , {xfail: true}]
   ```

2. Mark a test as expected failure and give a reason for the failure.

   ```
   - [Hello, , {xfail: Test case is not complete}]
   ```

3. Mark a test as expected failure just for backward or forward translation using the notation '`{xfail: {forward: true}}`' or '`{xfail: {backward: true}}`'. If you expect both to fail use '`{xfail: {forward: true, backward: true}}`'.

   To mark just the backward translation of a test as expected failure use the following:

   ```
   - [Hello, , {xfail: {backward: true}}]
   ```

   Again a reason for the expected failure can be given.

   ```
   - [Hello, , {xfail: {backward: Not implemented}}]
   ```

   If you expect both forward and backward translation to fail set both '`forward`' and '`backward`' to '`true`' (or give a reason).

   ```
   - [Hello, , {xfail: {forward: true, backward: true}}]
   # above is equivalent to
   - [Hello, , {xfail: true}]
   ```

'`typeform`'

The typeform used for a translation. It consists of one or more emphasis specifications. For each character in the specifications that is not a space the corresponding emphasis will be set. Valid options for emphasis are '`italic`', '`underline`', '`bold`', '`computer_braille`', '`passage_break`', '`word_reset`', '`script`', '`trans_note`', '`trans_note_1`', '`trans_note_2`', '`trans_note_3`', '`trans_note_4`' or '`trans_note_5`'. The following shows an example where both '`italic`' and '`underline`' are specified:

```
-
  - Hello
  -
  - typeform:
      italic:    '++++ '
      underline: '    +'
```

'`inputPos`'

A list of 0-based input positions, one for each output position. Useful when simulating screen reader interaction, to debug contraction and cursor behavior as in the following example. Note that all positions in this and the following examples start at 0. Also note that in these examples the additional options are not passed using the "flow style" notation.

```
-
  - went
  -
  - inputPos: [0,1,3]
```

'`outputPos`'

A list of 0-based output positions, one for each input position. Useful when simulating screen reader interaction, to debug contraction and cursor behavior as in the following example.

```
-
  - went
  -
  - outputPos: [0,1,1,2]
```

'cursorPos'
: The cursor position for the given translation and optionally an expected cursor position where the cursor is supposed to be after the translation. Useful when simulating screen reader interaction, to debug contraction and cursor behavior:

  The cursor position can take two forms: You can either specify a single number or alternatively you can give a tuple of two numbers.

  single number (e.g. '4')
  : When you simply want to specify the cursor position for the given translation you pass a number as in the following example:

```
-
  - you went to
  -
  - mode: [compbrlAtCursor]
    cursorPos: 4
```

  a tuple (e.g. '[4,2]')
  : When you expect the cursor to be in a particular position after the translation and you want to check this then pass a tuple of cursor positions as in the following example:

```
-
  - you went to
  -
  - mode: [compbrlAtCursor]
    cursorPos: [4,2]
```

'mode'
: A list of translation modes that should be used for this test. If not defined defaults to 0. Valid mode values are 'noContractions', 'compbrlAtCursor', 'dotsIO', 'compbrlLeftCursor', 'ucBrl', 'noUndefined' or 'partialTrans'.

  For a description of the various translation mode flags, please see the function ⟨undefined⟩ [lou_translateString], page ⟨undefined⟩.

'maxOutputLength'
: Define a maximum length of the output. This can be used to test the behavior of liblouis in the face of a limited output buffer, for example the length of the refreshable braille display.

6.1.1 Optional test description

When a test contains three or four items the first item is assumed to be a test description, the second item is the unicode text to be tested and the third item is the expected braille

output. Again an optional fourth item can contain additional options for the test. The following shows an example:

```
-
  - Number-text-transitions with italic
  - 123abc
  -
  - {typeform: '000111'}
```

In case the test fails the description will be printed together with the expected and the actual braille output.

For more examples and inspiration please see the YAML tests (`*.yaml`) in the `tests` directory of the source distribution.

6.1.2 Testing multiple tables within the same YAML test file

Sometimes you are more focused on testing a particular feature across several tables rather than just testing one table. For that reason the following is also allowed:

```
table: ...
tests:
  - [..., ...]
  - [..., ...]
table: ...
tests:
  - [..., ...]
  - [..., ...]
```

If you specify flags for the tests, remember that the flags are reset to their default values when you specify a new table.

6.1.3 Multiple test sections for each table

You can specify several sections of tests for each table, with or without the optional flags. This is useful e.g. if you want to have various tests for both forward and backward translation for the same set of tables, especially if you are defining the table as part of the yaml file (see next section). This feature is also useful if you simply want to divide your tests into multiple sections for better overview. All flags are reset to their default values when you start a new test section.

Thus, a yaml file might look as follows:

```
table: ...
tests:
  - [..., ...]
  - [..., ...]

# Some more tests
  tests:
  - [..., ...]
  - [..., ...]

# Some tests for back-translation - same table
```

```
flags: {testmode: backward}
  - [..., ...]
  - [..., ...]
```

6.1.4 Inline definition of tables

When testing very specific opcode combinations it is sometimes tedious to create specific test tables just for that. Hence the YAML tests allow for specification of table definitions inline. Instead of referring to a table by name you just define the table inline by using what the YAML spec calls a Literal Style Block (`http://www.yaml.org/spec/1.2/spec.html#id2795688`). Start the definition with a '`|`', then list the opcodes with an indentation. The inline table ends when the indentation ends.

```
table: |
  sign a 1
  ...
tests:
  - ...
  - ...
```

6.1.5 Running the same test data on multiple tables

Sometimes you maintain multiple tables which are very similar and basically contain the same test data. Instead of copying the YAML test and changing the table name you can also define multiple tables. This will cause the YAML tests to be checked against both tables.

```
table: nl-NL
table: nl-BE
tests:
  - [..., ...]
  - [..., ...]
```

7 Programming with liblouis

7.1 Overview

You use the liblouis library by calling the following functions, `lou_translateString`, `lou_backTranslateString`, `lou_translate`, `lou_backTranslate`, `lou_registerLogCallback`, `lou_setLogLevel`, `lou_logFile`, `lou_logPrint`, `lou_logEnd`, `lou_getTable`, `lou_findTable`, `lou_indexTables`, `lou_checkTable`, `lou_hyphenate`, `lou_charToDots`, `lou_dotsToChar`, `lou_compileString`, `lou_getTypeformForEmphClass`, `lou_readCharFromFile`, `lou_version`, `lou_free` and `lou_charSize`. These are described below. The header file, `liblouis.h`, also contains brief descriptions. Liblouis is written in straight C. It has four code modules, `compileTranslationTable.c`, `logging.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 liblouisutdml. The latter includes `liblouis.h`.

Persons who wish to use liblouis from Python may want to skip ahead to ⟨undefined⟩ [Python bindings], page ⟨undefined⟩.

`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_checkTable` function, 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.

```
typedef unsigned short int widechar
```

To make liblouis handle 32-bit Unicode simply remove the word `short` in the above `typedef`. 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.

At run time, the width of a character specified during compilation may be obtained using `lou_charSize`.

Here are the definitions of the eleven liblouis functions and their parameters. They are given in terms of Unicode characters, either 16-bit or 32-bit, depening on how liblouis has been compiled.

7.2 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 ⟨undefined⟩ [`comp6`], page ⟨undefined⟩).

Next are two small arrays that held pointers to chains of rules produced by the `swapcd` opcode (see ⟨undefined⟩ [`swapcd`], page ⟨undefined⟩) and the `swapdd` opcode (see ⟨undefined⟩ [`swapdd`], page ⟨undefined⟩) 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.

7.3 How tables are found

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.

7.4 Deprecation of the logging system

As of version 2.6.0 `lou_logFile`, `lou_logPrint` and `lou_logEnd` are deprecated. They are replaced by a more powerful, abstract API consisting of `lou_registerLogCallback` and `lou_setLogLevel`.

Usage of `lou_logFile`, `lou_logPrint` and `lou_logEnd` is discouraged as they may not be part of future releases. Applications using Liblouis should implement their own logging system.

During the transitional phase, `lou_logPrint` is registered as default callback in `lou_registerLogCallback`. `lou_logPrint` is overwritten by the first call to `lou_registerLogCallback` and reattached when `NULL` is set as callback. Note that calling `lou_logPrint` directly will not cause an invocation of the registered callback.

7.5 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.

7.6 lou_translateString

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

This function takes a string of Unicode characters in `inbuf` and translates it into a string of characters in `outbuf`. Each 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 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. See ⟨undefined⟩ [How tables are found], page ⟨undefined⟩, for a description on how the tables are located in the file system. 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.

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 ⟨undefined⟩ [How to Write Translation Tables], page ⟨undefined⟩, 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 an array of `formtype` 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 element indicates the typeform of the corresponding character in the input buffer. The values and their meaning can be consulted in the `typeforms` enum in `liblouis.h`. 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 defined in `liblouis.h`. They are all powers of 2, so that a combined mode can be specified by adding up different values.

Note that the `mode` parameter is an integer, not a pointer to an integer.

A combination of the following mode flags can be used with the `lou_translateString` function:

`compbrlAtCursor`
: If this bit is set in the `mode` parameter the space-bounded characters containing the cursor will be translated in computer braille.

`compbrlLeftCursor`
: If this bit is set, only the characters to the left of the cursor will be in computer braille. This bit overrides `compbrlAtCursor`.

`dotsIO`
: When this bit is set, during forward translation, Liblouis will produce output as dot patterns. During back-translation Liblouis accepts input as dot patterns. Note that the produced dot patterns are affected if you have any `display` opcode (see ⟨undefined⟩ [`display`], page ⟨undefined⟩) defined in any of your tables.

`ucBrl`
: The `ucBrl` (Unicode Braille) bit is used by the functions `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.

`partialTrans`
: This flag specifies that back-translation input should be treated as an incomplete word. Rules that apply only for complete words or at the end of a word will not take effect. This is intended to be used when translating input typed on a braille keyboard to provide a rough idea to the user of the characters they are typing before the word is complete.

`noUndefined`
: Setting this bit disables the output of hexadecimal values when forward-translating undefined characters (characters that are not matched by any rule), and dot numbers when back-translating undefined braille patterns (braille patterns that are not matched by any rule). The default is for liblouis to output the hexadecimal value (as '\xhhhh') of an undefined character when forward-translating and the dot numbers (as \ddd/) of an undefined braille pattern when back-translating.

  When back translating input from a braille keyboard cell by cell, it is desirable to output characters as soon as they are produced. Similarly, when back translating contracted braille, it is desirable to provide a "guess" to the user of the characters they typed. To achieve this, liblouis needs to have the ability to produce no text when indicators (which don't produce a character by themselves) are not followed by another cell. This works automatically for indicators liblouis knows about such as capital sign, number sign, etc., but it does not work for indicators which are not (and cannot be) specifically defined as indicators. For example, in UEB, dots 4 5 6 alone produces the text "\456/". Setting the noUndefined mode suppresses this dot number output.

The function returns 1 if no errors were encountered[1] and 0 otherwise.

7.7 lou_translate

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

[1] When the output buffer is not big enough, `lou_translateString` returns a partial translation that is more or less accurate up until the returned `inlen`/`outlen`, and treats it as a successful translation, i.e. also returns 1.

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.

For a description of all other parameters, please see ⟨undefined⟩ [lou_translateString], page ⟨undefined⟩.

7.8 lou_backTranslateString

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

This is exactly the opposite of `lou_translateString`. `inbuf` is a string of Unicode characters representing braille. `outbuf` will contain a string of 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` specifies how the back-translation should be done.

By default, if a dot pattern in the input is undefined then its dot numbers will be included in the output (as \ddd/). This does not occur if the `noUndefined` mode is set; an undefined dot pattern simply produces no output.

The `partialTrans` mode specifies that the input should be treated as an incomplete word. That is, rules that apply only for complete words or at the end of a word will not take effect. This is intended to be used when translating input typed on a braille keyboard to provide a rough idea to the user of the characters they are typing before the word is complete.

7.9 lou_backTranslate

```
int lou_backTranslate(
  const char *tableList,
  const widechar *inbuf,
  int *inlen,
  widechar *outbuf,
  int *outlen,
  formtype *typeform,
```

```
  char *spacing,
  int *outputPos,
  int *inputPos,
  int *cursorPos,
  int mode);
```

This function is exactly the inverse of `lou_translate`.

7.10 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 liblouisutdml.

7.11 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.

7.12 lou_getTypeformForEmphClass

```
int lou_getTypeformForEmphClass (const char *tableList, const char *emphClass);
```

This function returns the typeform bit associated with the given emphasis class. If the emphasis class is undefined this function returns `0`. If errors are found error messages are logged to the log callback (see `lou_registerLogCallback`) and the return value is `0`. `tableList` is a list of names of table files separated by commas, as explained previ-

ously (see ⟨undefined⟩ [`tableList` parameter in `lou_translateString`], page ⟨undefined⟩). `emphClass` is the name of an emphasis class.

7.13 lou_dotsToChar

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

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.

Note that the `mode` parameter has no effect and is deprecated.

7.14 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.

7.15 lou_registerLogCallback

```
typedef void (*logcallback) (
  int level,
  const char *message);

void lou_registerLogCallback (
  logcallback callback);
```

This function can be used to register a custom logging callback. The callback must take two arguments, the log level and the message string. By default log messages are printed to stderr, or if a filename was specified with `lou_logFile` then messages are logged to that file. `lou_registerLogCallback` overrides the default callback. Passing `NULL` resets to the default callback.

7.16 lou_setLogLevel

```
typedef enum
{
  LOU_LOG_ALL = 0,
  LOU_LOG_DEBUG = 10000,
  LOU_LOG_INFO = 20000,
  LOU_LOG_WARN = 30000,
  LOU_LOG_ERROR = 40000,
  LOU_LOG_FATAL = 50000,
  LOU_LOG_OFF = 60000
} logLevels;
void lou_setLogLevel (
  logLevels level);
```

This function can be used to influence the amount of logging, from fatal error messages only to detailed debugging messages. Supported values are `LOU_LOG_DEBUG`, `LOU_LOG_INFO`, `LOU_LOG_WARN`, `LOU_LOG_ERROR`, `LOU_LOG_FATAL` and `LOU_LOG_OFF`. Enabling logging at a given level also enables logging at all higher levels. Setting the level to `LOU_LOG_OFF` disables logging. The default level is `LOU_LOG_INFO`.

7.17 lou_logFile (deprecated)

```
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 liblouisutdml. Any error messages generated will be printed to the file given in this call. The entire path name of the file must be given.

This function is deprecated. See ⟨undefined⟩ [Deprecation of the logging system], page ⟨undefined⟩.

7.18 lou_logPrint (deprecated)

```
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 liblouisutdml.

This function is deprecated. See ⟨undefined⟩ [Deprecation of the logging system], page ⟨undefined⟩.

7.19 lou_logEnd (deprecated)

```
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.

This function is deprecated. See ⟨undefined⟩ [Deprecation of the logging system], page ⟨undefined⟩.

7.20 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`.

7.21 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`.

7.22 lou_getTable

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

`tableList` is a list of names of table files separated by commas, as explained previously (see ⟨undefined⟩ [`tableList` parameter in `lou_translateString`], page ⟨undefined⟩). If no errors are found this function returns a pointer to the compiled table. If errors are found error messages are logged to the log callback (see `lou_registerLogCallback`). Errors result in a `NULL` pointer being returned.

7.23 lou_findTable

```
char *lou_findTable (const char *query);
```

This function can be used to find a table based on metadata. `query` is a string in the special ⟨undefined⟩ [Query Syntax], page ⟨undefined⟩. It is matched against ⟨undefined⟩ [Table Metadata], page ⟨undefined⟩, inside the tables that were previously indexed with ⟨undefined⟩ [lou_indexTables], page ⟨undefined⟩. Returns the file name of the best match. Returns `NULL` if the query is invalid or if no match can be found.

The match algorithm works as follows:

- For every table a match quotient with the query is computed. The table with the highest (positive) match quotient wins. If no table has a positive quotient, there is no match.
- A query is a list of features. Features defined first have a higher importance (have a higher impact on the final quotient) than features defined later.
- A feature that matches a metadata field in the table (keys equal and values equal, or both values absent) adds to the quotient.
- A feature that is undefined in the table (no field with that key) creates a medium penalty.

- A feature that is defined in the table (one or more metadata fields with that key) but does not match (no metadata field with the right value) creates the highest penalty.
- Every field in the table that has no corresponding feature in the query creates a very small penalty.

7.24 lou_indexTables

```
void lou_indexTables (const char **tables);
```

This function must be called prior to ⟨undefined⟩ [lou_findTable], page ⟨undefined⟩. It parses, analyzes and indexes all specified tables. `tables` must be an array of file names. Tables that contain invalid metadata are ignored.

7.25 lou_checkTable

```
int lou_checkTable (const char *tableList);
```

This function does the same as `lou_getTable` but does not return a pointer to the resulting table. It is to be preferred if only the validity of a table needs to be checked. `tableList` is a list of names of table files separated by commas, as explained previously (see ⟨undefined⟩ [`tableList` parameter in `lou_translateString`], page ⟨undefined⟩). If no errors are found this function returns a non-zero. If errors are found error messages are logged to the log callback (see `lou_registerLogCallback`) and the return value is `0`.

7.26 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`.

7.27 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.

7.28 lou_charSize

```
int lou_charSize ();
```

This function returns the size of `widechar` in bytes and can therefore be used to differentiate between 16-bit and 32bit-Unicode builds of liblouis.

7.29 Python bindings

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

Concept Index

(Index is nonexistent)

Opcode Index

(Index is nonexistent)

Function Index

(Index is nonexistent)

Program Index

(Index is nonexistent)