Liblouisutdml User's and Programmer's Manual

Release 2.11.0

by John J. Boyer

This manual is for liblouisutdml (version 2.11.0, 11 January 2022), an xml to Braille Translation Library.

Table of Contents

1 Introduction

liblouisutdml is a software component which can be incorporated into software packages to provide the capability of translating any file in the computer lingua franca xml format or plain text into properly transcribed braille. This includes translation into grade two, if desired, mathematical codes, etc. It also includes formatting according to a style sheet which can be modified by the user. The first program into which liblouisutdml has been incorporated is `file2brl`. This program will translate an xml or text file into an embosser-ready braille file. It is not necessary to know xml, because MSWord and other word processors can export files in this format. If the word processor has been used correctly `file2brl` will produce an excellent braille file.

Users who want to generate Braille using `file2brl` will be interested in Section 2.1 [Transcribing XML files with file2brl], page 2. Those who wish to change the output generated by liblouisutdml should read Chapter 3 [Customization Configuring liblouisutdml], page 5. If you encounter a type of xml file with which liblouisutdml is not familiar you can learn how to tell it how to process that file by reading Chapter 4 [Connecting with the xml Document], page 15. If you wish to implement a new braille mathematics code read Chapter 7 [Implementing Braille Mathematics Codes], page 36. Finally, computer programmers who wish to use liblouisutdml in their software can find the information they need in Chapter 8 [Programming with liblouisutdml], page 39.

You will also find it advantageous to be acquainted with the companion library liblouis, which is a braille translator and back-translator (see Section "Overview" in *Liblouis User's and Programmer's Manual*).

2 Transcribing Documents

2.1 Transcribing XML files with file2brl

At the moment, actual transcription with liblouisutdml is done with the command-line (or console) program `file2brl`. The line to type is:

```
file2brl [OPTIONS] [-f config-file] [infile] [outfile]
```

The brackets indicate that something is optional. You will see that nothing is required except the program name itself, `file2brl`. The various optional parts control how the program will behave, as follows:

`-h`
`--help` This option causes `file2brl` to print a help message describing usage and exit.

`-v`
`--version`
This option causes `file2brl` to display the version information and exit.

`-f configfile`
`--config-file configfile`
This specifies the configuration file which tells `file2brl` how to do the transcription. (It may be a list of file names separated by commas.) This file specifies such things as the number of cells per line, the number of lines per page, The translation tables to be used, how paragraphs and headings are to be formatted, etc. If this part of the command line is omitted, `file2brl` assumes that the configuration file is named `preferences.cfg`. If the configuration file name contains a pathname `file2brl` will consider this as a path on which to look for files that it needs (see Section 8.3 [Files and Paths], page 40). If no pathname is given the standard paths are searched and finally the current directory. To make `file2brl` search the current directory first, precede the file name with `./`.

`-b`
`--backward`
back-translate. The input file must be a braille file, such as `.brf`. The output file is a back-translation of this file. It may be in either plain-text or xhtml (html), according to the setting of `backFormat` in the `outputFormat` section of the configuration file. Html files will contain page numbers and emphasis. To get good html, the liblouis table must have the entry '`space \e 1b`' so that it will pass through escape characters. The `html.sem` file must also contain the line '`pagenum pagenum`'. Text output files simply have a blank line between paragraphs. Encoding of text files is controlled by the `outputEncoding` setting. Html files are always in UTF-8.

`-r`
`--reformat`
Reformat. The input file must be a braille file, such as `.brf`. The output is a braille file formatted according to the configuration file. It is advisable to set backFormat to html, since this will preserve print page numbers and emphasis.

This option can be useful for changing the line length and page length of a braille file, for example, from 40 to 32 cells. It is also an excellent way to check the accuracy of liblouis tables. The original page numbers at the tops and bottoms of pages are discarded, and new ones are generated.

`-T`

`--text` Consider the document to be a text file, even if it is xml or html.

`-t`

`--html` The document is an h(t)ml file, not xhtml. This option is useful with files downloaded from the Web in source form. Without it, the program will first try to parse the file as an xml document, producing lots of error messages. It will then try the html parser. With this option, it goes directly to the html parser. See also the `formatFor` configuration (see [formatFor setting], page 8) file setting, which enables you to format the braille output for viewing in a browser.

`-p`
`--poorly-formatted`
Poorly formatted input translation. Infile is any text file such as may have been obtained by extracting the text in a pdf file. The input file may also be an xml or html file which is so poorly formatted that better braille can be obtained by ignoring the formatting. `file2brl` tries to guess paragraph breaks. The output is generally reasonably formatted, that is, with reasonable paragraph breaks.

`-P`
`--paragraph-line`
Treat each block of text ending in a newline as a paragraph. If there are two newline characters a blank line will be inserted before the next paragraph.

`-Csetting=value`
`--config-setting setting=value`
This option enables you to specify configuration settings on the command line instead of changing the configuration file. You can use as many `-C` options as you wish. Any settings can be specified except those having to do with styles. See [Configuration Settings Index], page 45, for a list of available settings. These must be specified in configuration files. The settings may be in any order. They override any settings in `liblouisutdml.ini` or in the configuration file used by `file2brl`.

`-w`
`--writeable-path`
This option enables you to specify where the log file and other temporary files will be written.

`-l`
`--log-file`
This option will cause `file2brl` and liblouisutdml to print error messages to `file2brl.log` instead of stderr. The file will be in the current directory. This option is particularly useful if `file2brl` is called by a GUI script or Web application.

`infile` This is the name of the input file containing the material to be transcribed. The file may be either an xml file or a text file. The `-b`, `-r` and `-p` options discussed above provide for other types of files and processing. Typical xml files are those provided by `www.bookshare.org` or those derived from a word processor by saving in xml format. If a text file is used, paragraphs and headings should be separated by blank lines. In such a file there is no way to distinguish between paragraphs and headings, so they will all be formatted as paragraphs, as specified by the configuration file. However, if you want a blank line in the braille transcription, use two consecutive blank lines in the text file.

`outfile` This is the name of the output file. It will be transcribed as specified by the configuration file and the `-C` configuration settings. The following paragraphs provide more information on both the input and output files.

`file2brl` is set up so that it can be used in a "pipe". To do this, omit both infile and outfile. Input is then taken from the standard input unit.

The first file name encountered (a word not preceded by a minus sign) is taken to be the input file and the second to be the output file. If you wish input to be taken from stdin and still want to specify an output file, use one minus sign (‘`-`’) for the input file.

If only the program name is typed `file2brl` assumes that the configuration file is `preferences.cfg`, input is from the standard input unit, and output is to the standard output unit.

2.2 Transcribing Text Documents

See the previous section on using `file2brl`. This program recognizes text files automatically and transcribes them according to the information in the configuration files. Paragraphs must be separated with a blank line. If you want a blank line in the output use two blank lines.

2.3 Transcribing Poorly Formatted Documents

```
file2brl -p infile outfile
```

Some text documents, such as those derived from pdf files, and even some xml and html documents, are so poorly formatted that you can get better braille by ignoring whatever markup they contain. The `-p` option of `file2brl` does this. It ignores xml or html markup and uses heuristics to find the beginning of paragraphs. Its choices are usually good. Note that it does not work with rtf files.

2.4 Transcribing html Documents

```
file2brl -t infile outfile
```

The `-t` option prevents `file2brl` from trying to transcribe infile as an xml document. This will produce a lot of error messages. `file2brl` will then try the html parser. Note that xhtml documents are actually xml.

3 Customization: Configuring liblouisutdml

The operation of liblouisutdml is controlled by two types of files: semantic-action files and configuration files. The former are discussed in the section Connecting with the xml Document - Semantic-action Files (see Chapter 4 [Connecting with the xml Document - Semantic-Action Files], page 15). The latter are discussed in this section. A third type of file, braille translation tables, is discussed in the liblouis documentation (see Section "Overview" in *Liblouis User's and Programmer's Manual*).

Another section of the present document which may be of interest is Implementing Braille Mathematical Codes (see Chapter 7 [Implementing Braille Mathematics Codes], page 36).

Besides files, liblouisutdml can also be controlled by configuration strings, which are character strings in memory containing configuration settings separated by end-of-line characters. Such strings can be generated by the `-C` option on the `file2brl` command line, by the `configstring` and `configtweak` semantic actions, or by passing a string to the `lbu_initialize` function.

The information below applies to `file2brl` as much as to liblouisutdml.

Before discussing configuration files in detail it is worth noting that the application program has access to the information in the configuration files by calling the liblouisutdml function `lbu_initialize`. This function returns a pointer to a data structure containing the configuration information. The calling program must include the header file `louisutdml.h`. You do not need to call `lbu_initialize` unless you need the facilities which it provides.

A configuration file specification may contain more than one file name, separated by commas. liblouisutdml will process these files in sequence, merging the information they contain. The first file name may also contain a path. liblouisutdml will search for the files it needs first on this path. To make it search first the current directory precede the first file name with `./`. After the path, if any, has been evaluated, but before reading any of the files, liblouisutdml reads in a file called `liblouisutdml.ini`. This file can contain any configuration settings, but it usually contains only the minimum ones for liblouisutdml to operate properly. You may alter the values in the distribution `liblouisutdml.ini`, but you should not delete any settings. Do not specify `liblouisutdml.ini` as your configuration file. This will lead to error messages and program termination. If a configuration file read in later contains a particular setting name, the value specified simply replaces the one specified in `liblouisutdml.ini` or any previously read configuration file.

Originally, configuration files contained four main sections, `outputFormat`, `translation`, `xml` and `style`. The section names, except for `style` are now optional. In addition, a configuration file can contain an include entry. This causes the file named on that line to be read in at the point where the line occurs. The sections need not follow each other in any particular order, nor is the order of settings within each section important. The section names, except for `style` are optional. In this document and in the `liblouisutdml.ini` file, where section and setting names consist of more than one word, the first letter of each word following the initial one is capitalized. This is merely for readability. The case of the letters in these names is ignored by the program. Section and setting names may not contain spaces.

In addition to `liblouisutdml.ini` the distribution also contains a number of configuration files. The most important of these is `preferences.cfg`, which contains all possible

settings and a "default" value for each. You should use this file as a reference. It is the file read by the `file2brl` command-line interface program if no configuration file is given.

Here, then, is an explanation of each section and setting in the `preferences.cfg` file. When you look at this file you will see that the section names start at the left margin, while the settings are indented one tab stop. This is done for readability, it has no effect on the meaning of the lines. You will also see lines beginning with a number sign ('`#`'), which are comments. Blank lines can also be used anywhere in a configuration file. In general, a section name is a single word or combination of unspaced words. However, each style has a section of its own, so the word '`style`' is followed by a space then by the name of the style. Setting lines begin with the name of the setting, followed by at least one space or tab, followed by the value of the setting. A few settings have two values.

3.1 outputFormat

This section specifies the format of the output file (or string).

`cellsPerLine 40`
: The number of cells in a braille line.

`linesPerPage 25`
: The number of lines on a braille page

`interpoint no`
: Whether or not the output will be used to produce interpoint braille. This affects the placement of page numbers and may affect other things in the future. The only two values recognized are '`yes`' and '`no`'.

`lineEnd \r\n`
: This specifies the control characters to be placed at the end of each output line. These characters vary from one intended use of the output to another. Most embossers require the carriage-return and line-feed combination specified above. However, a braille display may work best with just one or the other. Any valid control characters can be specified.

`pageEnd \f`
: The control Character to be given at the end of a page. Here it is a forms-feed character, but it can be something else if deeded.

`fileEnd ^z`
: The control character to be placed at the end of the file, here a control-z.

`printPages yes`
: Whether or not to show print page numbers if they are given in the xml input. The two valid values are '`yes`' and '`no`'.

`braillePages yes`
: Whether or not to format the output into pages. Here the value is '`yes`', for use with an embosser. However the user of a braille display may wish to specify '`no`', so as not to be bothered with page numbers and forms feed characters. If no is specified the lines will still be of the length given in `cellsPerLine`, but the value of `linesPerPage` will be ignored.

`paragraphs yes`
: Whether or not to format the output into paragraphs, using appropriate styles. If '`no`' is specified, what would be a paragraph is output simply as one long line. Applications that wish to do their own formatting may specify '`no`'.

`beginningPageNumber 1`
: This is the number to be placed on the first Braille page if `braillePages` is yes. This is useful when producing multiple Braille volumes.

`printPageNumberAt top`
: If print page numbers are given in the xml input file they will be placed at the top of each braille page in the right-hand corner. If `pageSeparator` is set to '`yes`', a page separator line will also be produced on the Braille page where the print page break actually occurs. You may also specify '`bottom`' for this setting.

`braillePageNumberAt bottom`
: The braille page number will be placed in the bottom right-hand corner of each page. If `interpoint yes` has been specified only odd pages will receive page numbers. You may also specify '`top`' for this setting. If print page numbers and Braille page numbers are both placed at the top or bottom, they are rendered next to each other with a space in between.

`continuePages yes`
: Print page numbers can be prefixed with a letter (a, b, c, etc.) on continued pages. The two valid values are '`yes`' and '`no`'.

`pageSeparator yes`
: A page separator line (or page break indicator), a line of unspaced Braille dots 36, will be placed wherever a print page break occurs. No page separator lines are placed on the first or last line of a Braille page, and no page separator lines are shown when the new print page coincides with a new Braille page.

`pageSeparatorNumber yes`
: Show a page number at the far right margin of a page separator line. No space is left between the separator line and the first symbol of the page number.

`ignoreEmptyPages yes`
: An empty page occurs when a `pagenum` tag is immediately followed by another `pagenum` tag. By default, empty pages are completely ignored. If you specify '`no`' for this setting, a sequence of `pagenum` tags will lead to a *combined* print page number: the number of the first empty page is combined with that of the page on which text reappears, e.g. 5-7. If lettered continuation pages are required (see `continuePages`), they carry only the number of the page on which text reappears.

`printPageNumberRange no`
: By default, only the page number of the *first* print page on a Braille page is shown at the top or bottom. However, if `printPageNumberRange` is set to '`yes`', the *range* of print pages contained in the current Braille page is displayed. If the first page in this range is a continued print page, it is prefixed with a letter as usual (see `continuePages`).

`mergeUnnumberedPages yes`
: Page breaks without a page number can simply be ignored. This means that unnumbered print pages will be treated as if they were a part of the preceding page. You can also specify '`no`' for this setting.

`pageNumberTopSeparateLine yes`
: Whether or not to provide a separate line for page numbers when they are placed at the top of a Braille page. The two valid values are '`yes`' and '`no`'. A print page number range (see `printPageNumberRange`) at the top of a page is always displayed on a separate line.

`pageNumberBottomSeparateLine yes`
: Whether or not to provide a separate line for page numbers when they are placed at the bottom of a Braille page.

`hyphenate no`
: If '`yes`' is specified words will be hyphenated at the ends of lines if a hyphenation table is available. In contracted English Braille, hyphenation is not generally used, but it can save considerable space. The hyphenation table is specified as part of the table list in the `literaryTextTable` setting of the translation section.

`outputEncoding ascii8`
: This specifies that the output is to be in the form of 8-bit ASCII characters. This is generally used if the output is intended directly for a braille embosser or display. The other values of encoding are '`UTF8`', '`UTF16`' and '`UTF32`'. These are useful if the application will process the output further, such as for generating displays of braille dots on a screen.

`inputTextEncoding ascii8`
: This setting is used to specify the encoding of an input text file. The valid values are '`UTF8`' and '`ascii8`'.

`formatFor textDevice`
: This setting specifies the type of device the output is intended for. '`textDevice`' is any device that accepts plain text, including embossers. You can also specify '`browser`'. In this case the output will be formatted for viewing in a browser. If the input file contains links, they will be preserved and can be used in the normal way. The text will be translated into braille with the correct line length. Math and computer material will be translated appropriately. These files work well in lynx and Internet Explorer, not so well in elinks and Firefox (Before Jaws 10).

`backFormat plain`
: This setting specifies the format of back-translated files. '`Plain`' specifies plain-text, while '`html`' specifies xhtml. The latter is always encoded in UTF-8. Plain-text files can be encoded in ascii8, UTF-8 or UTF-16. Html is strongly recommended, since it will preserve print page numbering and emphasis.

`backLineLength 70`
: This setting specifies the length of lines in back-translated files, whether in plain-text or html. This is mainly for human readability. Lines may sometimes be somewhat longer.

`lineFill '`
: This setting defines the fill character that will be used before the page numbers in the table of contents for example. The default fill character is an apostrophe (dot 3).

3.2 translation

This section specifies the liblouis translation tables to be used for various purposes.

`literaryTextTable en-us-g2.ctb`
: The table used for producing literary braille. This may be either contracted or uncontracted.

`uncontractedTable en-us-g1.ctb`
: The table used for producing uncontracted or Grade One braille. This setting appears to be superfluous and may be eliminated in the future.

`compbrailleTable en-us-compbrl.ctb`
: The table used for producing large amounts of output in computer braille, such as computer programs. The computer braille table is usually combined with one of the two tables above.

`mathtextTable en-us-mathtext.ctb`
: This table specifies how the non-mathematical parts of math books are to be translated. In many cases it will be the same as literaryTextTable or uncontractedTable. For books translated with the Nemeth Code it is different, because this code requires modification of standard Grade Two.

`MathexpTable nemeth.ctb`
: This is the table used to translate mathematical expressions.

`editTable nemeth_edit.ctb`
: When the output includes both mathematics and text there may be errors where one type of translation directly follows another. The editTable removes these errors.

3.3 xml

This section provides various information for the processing of xml files.

`semanticFiles *\`
: nemeth.sem, This setting gives a list of semantic-action files. These files are read in the sequence given in the list. Here the first member of the list is an asterisk ('`*`'). This means that the corresponding file is to be named by taking the root element of the document and appending '`.sem`'. This asterisk member may occur anywhere in the list.

`xmlheader <?xml version='1.0' encoding='UTF8' standalone='yes'?>`
: This line gives the xml header to be added to strings produced by programs like `Mathtype` that lack one.

`entity nbsp ^1`
: This line defines an entity or substitution in an xml file. It is one of those that has two values. The first is the thing to be replaced, and the second is the replacement. As many entity lines as necessary can be used. The information they contain is added to the information provided by xmlHeader. In `liblouisutdml.ini` this line is commented out, because specifying it at this point would prevent the user from specifying his own xmlheader.

`internetAccess yes`
: The computer has an internet connection and liblouisutdml may obtain information necessary for the processing of this file from the Internet. If this setting is '`no`' liblouisutdml will not try to use the internet. The necessary information may, however, be provided on the local machine in the form of a `"dtd"` file.

`newEntries yes`
: liblouisutdml may create a new semantic-action file (beginning with `new_`) for a document with an unknown root element or a file (beginning with `appended_`) containing new entries for an existing semantic-action file. Both kinds of files are placed on the current directory. If this setting is '`no`' liblouisutdml will not create a file of new entries and if it encounters a document with an unknown root element it will issue an error message. Setting newEntries to '`no`' may be useful if users should not be bothered with the minutiae of semantic-action files.

3.4 style

The following sections all deal with styles. Each style has its own section. Style section names are unlike other section names in that they consist of the word style, followed by a space, followed by a style name. With some exceptions, styles are not hard-coded. The user may define any style desired, with any name except `document`, `para`, `heading1`, `heading2`, `heading3`, `heading4`, `contentsheader`, `contents1`, `contents2`, `contents3` and `contents4`. The first two are needed for basic formatting. The others are needed for the table of contents tool. The user must define settings for these styles as for any others. This is done in `liblouisutdml.ini`, which also contains definitions and settings for many other styles. The user can add styles at any time in her/his own configuration files.

Styles can be nested. That is, a document may contain a section of one style, and inside this may be a section of another style. For example, you might have styles named frontMatter, titlePage, dedication, contents, and so on. Your document might contain a section of style frontMatter. Inside this section might be subsections of styles titlePage, dedication, contents, and so on. Inside the titlePage section there might be other sections with styles heading1, para, centered, etc.

Your frontMatter style might also define the `"persistent"` style setting `braillePageNumberFormat roman`. This setting will apply to all the styles nested within frontMatter, unless they have a setting other than '`normal`', which is the default and means ordinary braille page numbers. However, the titlePage style might have

the setting `braillePageNumberFormat blank`. This will apply to all styles nested within it. When the titlePage section ends, the frontMatter setting '`roman`' will be restored. The '`braillePageNumberFormat`' setting is an example of a `"persistent"` style setting. Most settings apply only to the style for which they are declared.

Below are the settings for the predefined style names. The '`document`' style contains all possible settings. The others contain only settings that are different from the defaults.

3.4.1 style document

This is a predefined style name. All settings have their default values. The user must specify any other values. If a `"persistent"` style setting is specified, it will apply to the whole document.

`linesBefore 0`
: This setting gives the number of blank lines which should be left before the text to which this style applies. It is set to a non-zero value for some header styles.

`linesAfter 0`
: The number of blank lines which should be left after the text to which this style applies.

`leftMargin 0`
: The number of cells by which the left margin of all lines in the text should be indented. Used for hanging indents, among other things. This is a `"persistent"` setting, so by default all nested styles will inherit the setting.

`rightMargin 0`
: The equivalent of '`leftMargin`' for the right side of the page. This is also a persistent setting.

`firstLineIndent 0`
: The number of cells by which the first line is to be indented relative to leftMargin. firstLineIndent may be negative. If the result is less than 0 it will be set to 0. This setting is persistent.

`translate contracted`
: This setting is currently inactive. It may be used in the future. This setting tells how text in this style should be translated. Possible values are '`contracted`', '`uncontracted`', '`compbrl`', '`mathtext`' and '`mathexpr`'.

`skipNumberLines no`
: If this setting is '`yes`' the top and bottom lines on the page will be skipped if they contain braille or print page numbers. This is useful in some of the mathematical and graphical styles.

`format leftJustified`
: The format setting controls how the text in the style will be formatted. Valid values are '`leftJustified`', '`rightJustified`', '`centered`', '`computerCoded`', '`alignColumnsLeft`', '`alignColumnsRight`', and '`contents`'. The first three are self-explanatory. '`computerCoded`' is used for computer programs and similar material. The next two are used for tabular material. '`alignColumnsLeft`' causes the left ends of columns to be aligned. '`alignColumnsRight`' causes

the right ends of columns to be aligned. '`contents`' is used only in styles specifically intended for tables of contents. In the case of '`leftJustified`', '`rightJustified`' and '`centered`', nested styles inherit this setting by default.

`newPageBefore no`
: If this setting is '`yes`', the text will begin on a new page. This is useful for certain mathematical and graphical styles. Page numbers are handled properly.

`newPageAfter no`
: If this setting is '`yes`' any remaining space on the page after the material covered by this style is handled is left blank, except for page numbers.

`rightHandPage no`
: if this setting is '`yes`' and interpoint is yes the material covered by this style will start on a right-hand page. This may cause a left-hand page to be left blank except for page numbers. If interpoint is '`no`' this setting is equivalent to newPageBefore.

`braillePageNumberFormat normal`
: This setting specifies the format of braille page numbers. '`normal`' means ordinary Arabic numbers. '`roman`' means Roman numbers. '`p`' means to precede Arabic numbers with the letter "p" (for preliminary). Finally, '`blank`' causes the page number to be blank (no page numbers). This is a "persistent" style setting.

`dontSplit no`
: If this setting is '`yes`', the element is protected from being split across pages. This means that if a block of text doesn't fit on the current page, it will be placed at the beginning of the next one. This setting applies to the whole element, including children, so if nested styles specify other values for '`dontSplit`', these values will be ignored.

`keepWithNext no`
: If this setting is '`yes`', the element covered by this style is protected from being split across pages, and in addition it is kept together with the first line of text of the next sibling.

`orphanControl 0`
: With this setting you can control how many lines of text of an element must be printed at least at the bottom of a braille page. The default value is '`0`'. To have an effect, the setting must have a value of '`2`' or more.

3.4.2 style contentsheader

This style is used to specify where the table of contents should be placed and its title. The xml tag assigned to it in the semantic action file should be placed in the document where you want the table of contents, and it should contain the title of that table between its starting and ending markers.

```
linesBefore 1
linesAfter 1
format centered
```

3.4.3 style contents1

This style and the other contents styles are used for the table of contents and correspond to the ten heading levels (‘`contents5`’, ‘`contents6`’, ‘`contents7`’, ‘`contents8`’, ‘`contents9`’ and ‘`contents10`’ are not showed here).

```
firstLineIndent -2
leftMargin 2
format contents
```

3.4.4 style contents2

```
firstLineIndent -2
leftMargin 4
format contents
```

3.4.5 style contents3

```
firstLineIndent -2
leftMargin 6
format contents
```

3.4.6 style contents4

```
firstLineIndent -2
leftMargin 8
format contents
```

3.4.7 style heading1

This style is used for main headings, such as chapter titles.

```
linesBefore 1
center yes
linesAfter 1
```

3.4.8 style heading2

The first level of subheadings after the main heading.

```
linesBefore 1
firstLineIndent 4
```

3.4.9 style heading3

The third level of headings.

```
firstLineIndent 4
```

3.4.10 style heading4

The fourth level of headings. There are six more levels: ‘`heading5`’, ‘`heading6`’, ‘`heading7`’, ‘`heading8`’, ‘`heading9`’ and ‘`heading10`’.

```
firstLineIndent 4
```

3.4.11 style para

Paragraph. This is ordinary body text.

`firstLineIndent 2`

3.4.12 style boxline

Typically used to form the top and bottom lines of "boxed" material. The character must be chosen to produce the desired dot pattern on the embosser or display in use.

`topBoxline .`
: This should be set to the character you want to be used for the boxline which appears before the content.

`bottomBoxline .`
: This should be set to the character you want to be used for the boxline which appears after the content.

4 Connecting with the xml Document - Semantic-Action Files

4.1 Overview

When liblouisutdml (or `file2brl`) processes an xml document, it needs to be told how to use the information in that document to produce a properly translated and formatted braille document. These instructions are provided by a semantic-action file, called so because it explains the meaning, or semantics, of the various specifications in the xml document. To understand how this works, it is necessary to have a basic knowledge of the organization of an xml document.

An xml document is organized like a book, but with much finer detail. First there is the title of the whole book. Then there are various sections, such as author, copyright, table of contents, dedication, acknowledgments, preface, various chapters, bibliography, index, and so on. Each chapter may be divided into sections, and these in turn can be divided into subsections, subsubsections, etc. In a book the parts have names or titles distinguished by capitalization, type fonts, spacing, and so forth. In an xml document the names of the parts are enclosed in angle brackets (‘`<>`’). For example, if liblouisutdml encounters `<html>` at the beginning of a document, it knows it is dealing with a document that conforms to the standards of the extensible markup language (xhtml) - at least we hope it does. When you see a book, you know it’s a book. The computer can know only by being told. Something enclosed in angle brackets is called an "element" (more properly, a "tag") in xml parlance. (There may be more between the angle brackets than just the name of the element. More of this later). The first "element" in a document thus tells liblouisutdml what kind of document it is dealing with. This element is called the "root element" because the document is visualized as branching out from it like a tree. Some examples of root elements are `<html>`, `<math>`, `<book>`, `<dtbook>` and `<wordDocument>`. Whenever liblouisutdml encounters a root element that it doesn’t know about it creates a new file called a semantic-action file. The name of this file is formed by stripping the angle brackets from the root element, putting ‘`new_`’ in front of it and adding a period plus the letters ‘`sem`’. For example, ‘`new_myformat.sem`’. If you look in a directory containing semantic-action files you will see names like `html.sem`, `dtbook.sem`, `math.sem`, and so on. The "new" semantic-action files must be edited by a person and the prefix "new" removed to get an ordinary semantic-action file name.

Sometimes it is advantageous to preempt the creation of a semantic-action file for a new root element. For example, an article written according to the docbook specification may have the root element `<article>`. However, the specification itself has the root element `<book>`. In this case you can specify the `book.sem` file in the configuration file by writing, in the xml section,:

```
semanticFiles book.sem
```

You will note that this setting uses the plural of "file". This is because you can actually specify a list of file names separated by commas. You might want to do this to specify the semantic-action file for the particular braille mathematical code to be used. For example:

```
semanticFiles book.sem,ukmaths.sem
```

You can use an asterisk `*` to specify the semantic-action file corresponding to the root element of the document anywhere in the list.

As you will see in the next section, different braille style conventions and different braille mathematical codes may require different semantic-action files

liblouisutdml records the names of all elements found in the document in the semantic-action file. The document has a multitude of elements, which can be thought of as describing the headings of various parts of the document. One element is used to denote a chapter heading. Another is used to denote a paragraph, Still another to denote text in bold type, and so on. In other words, the elements take the place of the capitalization, changes in type font, spacing, etc. in a book. However, the computer still does not know what to do when it encounters an element. The semantic-action file tells it that.

Consider `html.sem`. A copy is included as part of this documentation with the name `example_html.sem` (see Appendix A [Example files], page 44). It may differ from the file that liblouisutdml is currently using. You will see that it begins with some lines about copyrights. Each line begins with a number sign (‘`#`’). This indicates that it is a `"comment"`, intended for the human reader and the computer should ignore it. Then there is a blank line. Finally, there are two other comments explaining that the file must be edited to get proper output. This is because a human being must tell the computer what to do with each element. The semantic files for common types of documents have already been edited, so you generally don't have to worry about this. But if you encounter a new type of document or wish to specify special handling for styles or mathematics you may have to edit the semantic-action file or send it to the maintainer for editing. In any case the rest of this section is essential for understanding how liblouisutdml handles documents and for making changes if the way it does so is not correct.

After another blank line you will see a table consisting of two, and sometimes three, columns. The first column contains a word which tells the computer to do something. For example, the first entry in the table is: ‘`include nemeth.sem`’. This tells liblouisutdml to include the information in the `nemeth.sem` file when it is deciphering an html (actually xhtml) document (it may be preferable to use the semanticFiles setting in the configuration file rather than an include).

The second row of the table is:

```
no hr
```

‘`hr`’ is an element with the angle brackets removed. It means nothing in itself. However, the first column contains the word ‘`no`’. This tells liblouisutdml `"no do"`, that is, do nothing. This is not strictly true, since liblouisutdml will sometimes insert a blank space so that words in text do not run together.

After a few more lines with ‘`no`’ in the first column, we see one that says:

```
softreturn br
```

This means that when the element `<br>` is encountered, liblouisutdml is to do a soft return, that is, start a new line without starting a new paragraph.

The next line says:

```
heading1 h1
```

This tells liblouisutdml that when it encounters the element `<h1>` it is to format the text which follows as a first-level braille heading, that is, the text will be centered and preceded and followed by blank lines. (You can change this by changing the definition of the heading1 style).

The next line says:

```
italicx em
```

This tells liblouisutdml that when it encounters the element `<em>` it is to enclose the text which follows in braille italic indicators. The '`x`' at the end of the semantic action name is there to prevent conflicts with names elsewhere in the software. Just where the italic indicators will be placed is controlled by the liblouis translation table in use.

The next line says:

```
skip style
```

This tells liblouis to simply skip ahead until it encounters the element `</style>`. Nothing in between will have any effect on the braille output. Note the slash ('`/`') before the '`style`'. This means the end of whatever the `<style>` element was referring to. Actually, it was referring to specifications of how things should be printed. If liblouisutdml had not been told to skip these specifications, the braille output would have contained a lot of gobledygook.

The next line says:

```
italicx strong
```

This tells liblouis to also use the italic braille indicators for the text between the `<strong>` and `</strong>` elements.

After a few more lines with '`no`' in the first column we come to the line:

```
document html
```

This tells liblouisutdml that everything between `<html>` and `</html>` is an entire document. `<html>` was the root element of this document, so this is logical.

After another '`no`' line we come to:

```
para p
```

liblouisutdml will consider everything between `<p>` and `</p>` to be a normal body text paragraph.

The next line is:

```
heading1 title
```

this causes the title of the document to also be treated as a braille level 1 heading.

Next we have the line:

```
list li
```

The xhtml `<li>` and `</li>` pair of elements is used to enclose an item in a list. liblouisutdml will format this with its own list style. That is, the first line will begin at the left margin and subsequent lines will be indented two cells.

Next we have:

```
table table
```

You will note that the names of actions and elements are often identical. This is because they are both mnemonic. In any case, this line tells liblouisutdml to format the table contained in the xhtml document according to the table formatting rules it has been given for braille output.

Next we have the line:

```
heading2 h2
```

This means that the text between `<h2>` and `</h2>` is to be formatted according to the Liblouisutdml style heading2. A blank line will be left before the heading and the first line will be indented four spaces.

After a few more lines we come to:

```
no table,cellpadding
```

Note the comma in the second column. This divides the column into two subcolumns. The first is the table element name. The second is called an "attribute" in xml. It gives further instructions about the material enclosed between the starting and ending "tags" of the element (`<table>` and `</table>`. Full information requires three subcolumns. The third is called the value and gives the actual information. The attribute is merely the name of the information.

Much further down we find:

```
no table,border,0
```

Here the element is table, the attribute is border and the value is 0. If liblouisutdml were to interpret this, it would mean that the table was to have a border of 0 width. It is not told to do so because tables in braille do not have borders.

Now let's look at the file which is included at the beginning of the `html.sem` file. This is `nemeth.sem`. As with `html.sem`, a copy is included in the appendix (see Appendix A [Example files], page 44), but it is not necessarily the one that liblouisutdml is currently using. It illustrates several more things about how liblouisutdml uses semantic-action files.

The first thing you will notice is that for quite a few lines the first and second columns are identical. This is because the MathML element and attribute names are part of a standard, and it was simplest to use the element names for the semantic actions as well. Most of these actions do not do anything and could be replaced with the `generic` semantic action. They are retained for backward compatibility.

The first line of real interest is:

```
math math
```

Every mathematical expression begins with the element `<math>` (which may have attributes and values), and ends with `</math>`. This is therefore the root element of a mathematical expression. However, mathematical expressions are usually part of a document, so it is not given the semantic action document. The math semantic action causes liblouisutdml to carry out special interpretation actions. These will become clearer as we continue to look at the `nemeth.sem` file. You will note that this line has three columns. The meaning of the third column is discussed below.

After another uninteresting line we come to two that illustrate several more facts about semantic-action files:

```
mfrac mfrac ^?,/,^#
mfrac mfrac,linethickness,0 ^(,^;%,^)
```

Like the math entry above, the first line has three columns. While the first two columns must always be present, the third column is optional. Here, it is also divided into subcolumns by commas. The element `<mfrac>` indicates a fraction. A fraction has two parts, a numerator and a denominator. In xml, we call these parts children of `<mfrac>`. They may be represented in various ways, which need not concern us here. What is of real importance is that the third column tells liblouisutdml to put the characters '`^?`' before the

numerator, '/' between the numerator and denominator, and '~#' after the denominator. Later on, liblouis will translate these characters into the proper representation of a fraction in the Nemeth Code of Braille Mathematics. (For other mathematical codes, see Chapter 7 [Implementing Braille Mathematics Codes], page 36).

The second line is of even greater interest. The first column is again 'mfrac', but this line is for binomial coefficient. The second column contains three subcolumns, an element name, an attribute name and an attribute value. The attribute linethickness specifies the thickness of the line separating the numerator and denominator. Here it is 0, so there is no line. This is how the binomial coefficient is represented in print. The third column tells how to represent it in braille. liblouisutdml will supply '~(', upper number, '~%', lower number, '~)' to liblouis, which will then produce the proper braille representation for the binomial coefficient.

Returning to the line for the math element, we see that the third column begins with a backslash followed by an asterisk. The backslash is an escape character which gives a special meaning to the character which follows it. Here the asterisk means that what follows is to be placed at the very end of the mathematical expression, no matter how complex it is.

For further discussion of how the third column is used see Chapter 7 [Implementing Braille Mathematics Codes], page 36. The third column is not limited to mathematics. It can be used to add characters to anything enclosed by an xml tag.

4.2 Semantic Actions in detail

Here is a complete list of the semantic actions which liblouisutdml recognizes. Some of them are also the names of styles. These are listed in the first table. For a discussion of these, see Chapter 3 [Customization Configuring liblouisutdml], page 5.

Generally the format of a semantic action is:

```
semanticAction elementSpecifier optionalArguments
```

`elementSpecifier` is the second-column value, which may be an element name, an element-attribute pair or an element-attribute-value triplet, separated by commas. This specifies where a semantic action is to be applied. If it is solely an element then the action is applied if this element is encountered. If it is an element-attribute pair then the action is applied if the given element also has the specified attribute. In the last case with an element-attribute-value triplet the action is only applied if the element has the specified attribute and the value of this attribute is equal to the specified value.

`contents1 elementSpecifier`
: Note that the `contents1`, etc. semantic actions are never assigned an actual `elementSpecifier`. There used internally by the table of contents generator. They should be assigned style settings, however.

`contents2 elementSpecifier`
`contents3 elementSpecifier`
`contents4 elementSpecifier`
`contentsheader elementSpecifier`
: This semantic action must be assigned an element specifier if used. See the discussion of it in Section 3.4 [style], page 10.

```
document elementSpecifier
heading1 elementSpecifier
heading2 elementSpecifier
heading3 elementSpecifier
heading4 elementSpecifier
para elementSpecifier
```

The following table explains each of the non-style semantic actions. In general, each one performs a particular function. If a third column is given, the subcolumns will be inserted in order before each branch of any subtree starting from `elementSpecifier`.

`blankline elementSpecifier`
: This semantic action causes a blank line to appear in the output wherever it may occur. It is useful for fine formatting independent of styles. `elementSpecifier` should be an empty element, that is, of the form `<elementSpecifier/>`. If it is not, any content which it may contain will be ignored.

`boldx elementSpecifier`
: Enclose the text which follows in braille bold indicators. The '`x`' at the end of the semantic action name is there to prevent conflicts with names elsewhere in the software. Just where the bold indicators will be placed is controlled by the liblouis translation table in use.

`chemistry elementSpecifier`
: When a module to handle chemical notation is ready, this semantic action will invoke it. The processing will be like that produced by the semantic action `math`.

`changetable elementSpecifier`
: This semantic action is used to change the active translation table. It can switch to a table for another language or to a table for computer braille in a mathematical expression, etc. `elementSpecifier` is in the form `element,attribute`. The document contains something like:

```
<span lang="en-us-g1.ctb">
This is uncontracted.
</span>
```

The specified table remains in effect from `<element attribute="tablename">` until `</element>`, no matter what is between the two. The previous table is then restored.

`compbrl elementSpecifier`
: The material between `elementSpecifier` and `/elementSpecifier` is translated as computer braille, if the liblouis table in use provides for it. Beginning and ending computer braille indicators are inserted if they are in the table.

`configfile elementSpecifier filename`
: The `configfile`, `configstring` and `configtweak` semantic actions enable the configuration of liblouisutdml to be changed according to the contents of the document being transcribed. `configfile` and `configstring` take effect during the document analysis phase performed by `examine_document.c`.

`configtweak` is effective during the transcription phase, performed by `transcribe_document.c` and the functions called in this module.

`elementSpecifier` is the usual second-column value, which may be an element name, an element-attribute pair or an element-attribute-value triplet, separated by commas. `filename` must be on one of the paths set in the `paths.c` module. The file may contain any configuration settings except those in the xml section. These would be ineffective, since the document has already been parsed.

`configstring elementSpecifier setting1=value1;setting2=value2;...`

Note that the `setting=value` pairs are separated by semicolons. Because the string may be longer than a screen line, you can use a backslash '`\`' followed immediately by a line ending '`\n`', to continue to another line. The string must not contain any blanks. Any setting which can be specified in a file read with configfile can be specified in `configstring`.

`configtweak elementSpecifier settings`

`configtweak` is identical to `configstring` except that it is called in the transcription phase. It can be used for things like changing translation tables. For example:

```
configtweak elementSpecifier literaryTextTable=fooTable;\
mathExprTable=barTable
```

`configtweak` is not a generalization of `changetable`. The latter changes the active table and applies to a subtree. `configtweak` remains in effect until changed by another `configtweak`.

`contracted elementSpecifier`

`footer elementSpecifier`

This semantic action is used to specify a footer which will be placed at the bottom of each page.

```
<elementSpecifier>This is a footer</elementSpecifier>
```

`generic elementSpecifier`

This is a general-purpose semantic action. If the third column is blank it does absolutely nothing. If the third column contains a string or subcolumns its contents are placed in the output according to the usual rules. That is, the first subcolumn is placed before the first branch of the subtree rooted at this node, the second is placed before the second branch, etc. If the last (or only) subcolumn begins with `\*` it is placed after the last branch, no matter how many branches there may be.

`graphic elementSpecifier`

When a module which can handle SVG graphics is ready this semantic action will invoke it.

`htmllink elementSpecifier`

This semantic action is used when the configuration file specifies `formatFor browser`. It sets up a link which the browser can follow.

`htmltarget elementSpecifier`

This semantic action establishes a target for a link in the same file when `formatFor browser` is specified in the configuration file.

`italicx elementSpecifier`

Enclose the text which follows in braille italic indicators. The '`x`' at the end of the semantic action name is there to prevent conflicts with names elsewhere in the software. Just where the italic indicators will be placed is controlled by the liblouis translation table in use.

`linespacing elementSpecifier digit`

This semantic action specifies the number of blank lines to be left between adjacent lines in the output. For example if the third column is '`1`', lines will be double-spaced. '`0`' specifies normal spacing. The number cannot be greater than '`3`'. `linespacing` remains in effect until another `linespacing` is encountered. It should be assigned to an empty element.

`maction elementSpecifier`

In the early stages of development I thought that a separate piece of code might be needed for each of the MathML element tags. It turned out, as noted elsewhere, that most of them could have been handled with the `generic` semantic action. They are retained for backward compatibility. Therefore, unless this is not the case or additional information is needed, they are simlly listed.

`maligngroup elementSpecifier`
`malignmark elementSpecifier`
`math elementSpecifier`

Every mathematical expression begins with the element `<elementSpecifier> math` (which may have attributes and values), and ends with `</elementSpecifier>` (`/math`). This is therefore the root element of a mathematical expression. However, mathematical expressions are usually part of a document, so it is not given the semantic action document. liblouisutdml will, however, handle files and strings which consist of nothing but a mathematical expression properly. The `math` semantic action causes liblouisutdml to carry out special interpretation actions.

`menclose elementSpecifier`
`merror elementSpecifier`
`mfenced elementSpecifier`
`mfrac elementSpecifier`
`mglyph elementSpecifier`
`mi elementSpecifier`
`mlabeledtr elementSpecifier`
`mmultiscripts elementSpecifier`
`mn elementSpecifier`
`mo elementSpecifier`
`mover elementSpecifier`
`mpadded elementSpecifier`
`mphantom elementSpecifier`
`mprescripts elementSpecifier`
`mroot elementSpecifier`

The MathML element `mroot` is actually given the semantic action `reverse`.

`mrow elementSpecifier`
: This can be important in implementing Math codes because it is often used to create visual groups, which may be significant for braille.

`ms elementSpecifier`
`mspace elementSpecifier`
: This element and its attributes can be helpful for determining spacing.

`msqrt elementSpecifier`
`mstyle elementSpecifier`
: This MathML element should usually have the semantic action `skip`.

`msub elementSpecifier`
`msubsup elementSpecifier`
`msup elementSpecifier`
`mtable elementSpecifier`
: The file `liblouisutdml.ini` defines the style `matrix`. The semantic-action files for math codes declare `mtable` to be `matrix`. Depending on the attributes of this element, it can be set to other styles, such as long division. The `matrix` style contains the setting `format alignColumnsLeft`.

`mtd elementSpecifier`
: This element specifies a column in a mathematical table. For the style `matrix` the third column of the entry in a semantic-action file must contain `\*|ec`. This indicates the end of the column. Other specifications using the liblouis `exactdots` feature may also be necessary.

`mtext elementSpecifier`
`mtr elementSpecifier`
: This element specifies a row in a mathematical table. The entry in a semantic-action file must contain `\*\er` in the third column for the `matrix` style, indicating the end of the row. Other things may also need to be specified using the liblouis `exactdots` feature. Note that rows are not declared as styles nested inside the `matrix` style. This is because the table must be considered as a whole.

`munder elementSpecifier`
`munderover elementSpecifier`
`music elementSpecifier`
: When a module which can interpret MusicML and produce braille music notation is ready this semantic action will invoke it.

`newpage elementSpecifier`
: This semantic action causes the rest of the current page to be left blank except for page numbers and footers. A new page is then begun. Like `blankline`, it is useful for fine formatting independent of styles.

`no elementSpecifier`
: Originally, this semantic action was intended to be the default and to do nothing when an `elementSpecifier` had no meaning for braille translation. Later it was found that it should insert a blank space if parts of the text would run together, so this is now its action.

`none elementSpecifier`
: This is a MathML element.

`notranslate elementSpecifier`
: Output the text between the start and end tags exactly as written. It will, however, be formatted with appropriate line breaks, page numbers etc. If you want to make sure that things appear on the same line separate them with an unbreakable space, '` `' or '` `'.

`pagenum elementSpecifier`
: The text between `<elementSpecifier>` and `</elementSpecifier>` is taken to be a print page number. If it does not begin with a digit the string `\_` is placed before it. It is then passed to liblouis for translation according to the active table. This table must contain an entry for translating `\_` into a letter sign or whatever else is wanted. This string is inserted so that roman page numbers will be handled properly. Unnumbered page breaks are indicated with an empty pagenum tag: `<elementSpecifier></elementSpecifier>`.

`reverse elementSpecifier`
: The branches of the subtree rooted at this node are reversed in order. This is used in handling roots, where the arguments in the translation are in reverse order to those in MathML. the MathML element `mroot` is declared with this semantic action

`righthandpage elementSpecifier`
: If `interpoint yes` has been specified in the configuration file, and the current page is a right-hand one, the rest of the page is skipped except for footer and page number. the following left-hand page is similarly skipped. Otherwise, the action is the same as `newpage`.

`runninghead elementSpecifier`
: This semantic action is used to specify a running header, such as a book title, to be placed at the top of each page. If the header is too long it will be truncated.

```
<elementSpecifier>liblouisutdml Manual</elementSpecifier>
```

`semantics elementSpecifier`
: This is a MathML action which seems to be irrelevant to braille translation.

`skip elementSpecifier`
: Skip ahead until encountering the element `</elementSpecifier>`. Nothing in between will have any effect on the braille output.

`softreturn elementSpecifier`
: Do a soft return, that is, start a new line without starting a new paragraph. `elementSpecifier` should be empty, for example, `<br/>`.

`uncontracted elementSpecifier`
: This semantic action seems superfluous and may be eliminated in the future.

`underlinex elementSpecifier`
: Enclose the text which follows in braille underline indicators.

4.3 Pseudo-actions

These actions affect the processing of semantic-action files. They are not connected with any tag in the document. They are executed when they are encountered in the processing of semantic-action files.

4.3.1 include

```
include filename
```

filename must be the name of a semantic action file. The file is compiled as though it were part of the file containing the `include` entry. Included files may include other files.

4.3.2 newentries

```
newentries no
```

The second column in this entry must contain '`no`'. Any new entries found in the document will be ignored. No '`appended_`' file will be produced. This affects only documents processed with this semantic-action file. The configuration setting `newEntries` affects all documents.

4.3.3 namespaces

```
namespaces dtb=http://www.daisy.org/z3986/2005/dtbook/
```

This pseudo-action is used to declare namespaces used in XPath expressions. (see Section 4.4 [Using XPath Expressions], page 25). The format is '`namespaces prefix1=url1,prefix2=url2,...`'. The list of namespaces may not contain blanks.

4.4 Using XPath Expressions

The second column of a semantic action may contain an XPath expression for matching nodes. When an XPath expression is to be used the second column should be of the format: '`&xpath(<expression>)`' where '`<expression>`' is the XPath expression to match nodes.

When constructing your XPath expressions you may wish to consider the following facts:

- The XPath expression may contain brackets but they must match
- liblouisutdml performs XPath matching from the document root, so you most likely want all XPath expressions to begin with double slash ('`//`').
- If the source document uses namespaces for the nodes you wish to match then you must define the namespace in the semantic action file (see Section 4.3.3 [namespaces], page 25) and then prefix the node with the namespace (e.g. for a namespace with alias '`xhtml`' and node with name '`p`', this would be '`xhtml:p`')
- You should be careful not to create XPath expressions which give overlapping node set results. When liblouisutdml finds a match for a node it assigns the semantic action to that node and this will not be changed subsequently. Note that this is unlike XSLT, where for each matcher a priority is calculated from the XPath expression while in liblouisutdml it is unpredictable which rule will win.
- XPath expressions take precedence over ordinary semantic-file entries.

As with other types of semantic actions, you may define arguments in the third column of a semantic action using XPath expressions.

```
para &xpath(//h4)
```

This example causes any element with the name '`h4`' to be given the semantic action `para`, no matter what other assignments may be made to it.

4.5 Using Macros

A semantic-action file permits only one action or style to be mentioned in the first column. Macros get around this limitation. They define a style and a series of semantic actions. A macro name can also be in the first column.

Macros somewhat resemble styles. They are defined in a configuration file and used in a semantic-action file. In a configuration file the definition is of the form

```
macro macroName macroBody
```

In a semantic-action file they are invoked as follows:

```
macroName element[,attribute[,value]] [parameters]
```

Here is an example of a macro definition.

```
macro useSpanish para,configtweak(literaryTextTable=es-Es-g2.ctb)
```

In a semantic-action file it might be used as follows.

```
useSpanish p,lang,ES
```

The meanings of the various parts of a macro are as follows:

`macroName`
: is a string of alphanumeric characters, of which the first must be a letter.

`macroBody`
: is a series of items separated by commas. It may not contain whitespace. If it is too long to fit on one line it may be continued by using a backslash ('\') followed by enter. It should not be more than 500 characters in length. Each item is one of the following:

  A style name
  : Only one style can be specified in a macro. This is due to the way in which styles are handled.

  A semantic action, optionally with parameters
  : The parameters are enclosed in parentheses. They consist of whatever is legal in the third column for the particular semantic action. If the parentheses are empty, that is, simply '`()`' the contents of the third column of the macro invocation itself are used as the parameters.

  '`pause`'
  : This is used after a style name so that text belonging to that style can be processed. Semantic actions can be specified between the style name and the word '`pause`'.

  '`endstyle`'
  : This causes the style to be terminated and the processing of any text which may have been entered into it to be completed. Semantic actions may be specified after the word '`endstyle`'.

If a style has been specified and neither '`pause`' nor '`endstyle`' has been used the style is terminated when the end of the macro has been reached, and the processing of text is completed. (When the end_macro function is called by the program.)

5 Special Features

5.1 Table of contents

A table of contents is produced for an xml file if the file contains a tag which has been defined with the `contentsheader` semantic action (see [`contentsheader`], page 19) and also tags for the `heading1`, `heading2`, `heading3` or `heading4` semantic actions (see [`heading1`], page 20). The table of contents will contain print and braille page numbers if these features have been enabled. A sequence of fill characters will be inserted before the page numbers, so that the latter are at the right margin. The fill character can be specified in a configuration file with the `lineFill` setting (see [`lineFill`], page 9). The default fill character is an apostrophe (dot 3).

Five new styles have been defined for the table of contents. The first is the `contentsheader` style (see [contentsheader style], page 12), which is used to specify where the table of contents should be placed and the title that should be given to it. In the latter respect it is much like a heading style. The others correspond to the four heading levels and are `contents1`, `contents2`, `contents3` and `contents4`. These styles are chosen as appropriate while the table of contents is being made. Do not declare them in a semantic-action file. See the `liblouisutdml.ini` file for the current default definitions of all these styles.

The table of contents will be placed where the xml tag is that you declared in the `contentsheader` semantic action (see [`contentsheader`], page 19). Its title will be whatever is inside that tag, formatted according to the definition of the `contentsheader` style. It begins on a new page. After it is completed the braille page number is reset to `beginningBraillePageNumber` and another new page is started. This means that the xml tag with the `contentsheader` semantic action should occur at the end of the information which you want to be at the head of the output, such as a title page, dedication, etc.

It is not necessary that an xml file contain a tag with the `contentsheader` semantic action. If the file contains headers you can obtain a table of contents by specifying `contents yes` in a configuration file or `-Ccontents=yes` on the command line of `file2brl`. In this case, the table of contents will appear at the beginning of the output. Pages will be numbered beginning with 1. When the table of contents is complete, the material in the file will start on a new page and the page number will be the value given in `beginningBraillePageNumber`.

The `contents1`, etc. styles all have the `format contents` setting. This is a variant of the `leftJustified` format. It has been necessary to change the way `firstLineIndent` is handled to accommodate multilevel lists. Up till now, if `firstLineIndent` was negative, the first line would start at the real left margin, regardless of the value of `leftMargin`. Now the value of `firstLineIndent` is simply added to `leftMargin`. This means that if it is negative it is really subtracted. For example, if `leftMargin` is 4 and `firstLineIndent` is -2 the first line will start in cell 2. If the result of adding these two values is negative it is set to 0.

5.2 Back-translation

```
file2brl -b infile outfile
```

infile must be a braille file. It can have either upper-case or lower-case letters, etc. outfile will contain the back-translation according to the configuration specifications. It can be in two formats according to the value of `backFormat`. '`ascii`' produces plain text output. The lines will generally correspond to the lines in the original braille file. '`html`' produces a file in xhtml format. This is recommended, since it preserves print page numbers, if present and some of the formatting of the original. It can also be loaded into a browser or word processor, which will format it for good readability. Note that for html format to work your liblouis table must contain the following line:

```
space \x001b 1b escape character
```

To perform the back-translation operation, `file2brl` uses the liblouisutdml function `lbu_backTranslateFile`.

5.3 Reformatting

```
file2brl -r infile outfile
```

As in the previous section, infile must be a braille file. It is back-translated and then forward-translated to produce a braille file in outfile which conforms to configuration specifications. It is useful for changing the line length and page length of a braille file. New braille page numbers will be generated if `braillePages yes` is specified. If `backFormat html` has been specified, print page numbers will be reproduced in the appropriate places. Some formatting may be lost.

5.4 Interlining

Interlining means printing the original text between the lines of translated braille. It requires special embossers or special methods. The present way in which liblouisutdml produces interlining relies on back-translation. However, it is inadequate for mathematics and depends too much on the quality of the liblouis tables. It is scheduled to be replaced, so you should not use it.

5.5 Browser-Friendly Output

```
file2brl infile outfile -CformatFor=browser
```

infile can be any of the file types accepted by `file2brl` (xml, html or text). If it contains html links or targets they will be formatted so that a browser can use them. This may be useful if a file contains internal links to different sections, such as its own table of contents. Text will be translated and formatted according to configuration specifications. If the file contains mathematics expressed as MathML it will be translated according to the mathematics code specified by the configuration. outfile should have the extension '`.html`'. It will actually be xhtml. The `-CformatFor=browser` part of the above example specifies a configuration setting, which, of course, can also be specified in a configuration file.

5.6 `CDATA` Sections

A `cdata` section may be given the semantic actions `skip`, `no` or `code`. In the first case, the data in the `cdata` section is ignored. In the second case, it is inserted into the output with no translation. In the third case it is translated into computer braille and inserted into the output. Any other semantic action has the same effect as `no`.

5.7 End notes

An endnote is defined as the link between a reference character in the body of the text and a description at the end of the output document.

5.7.1 Use of Endnotes

The position of the reference character in the body of the text is defined by the `noteref` semantic action (see [`noteref`], page 31), which is then linked to a `note` semantic action (see [`note`], page 31) (which can appear anywhere within the input file) with the same id. The content of `note` will appear at the end of the output document, along with its corresponding reference character and the page and line numbers of where the reference character appears in the text.

A heading for the first endnote page can be set using the `notesheader` semantic action (see [`notesheader`], page 31), and a small note can also be placed after this heading, but before the endnotes, by using the `notesdescription` semantic action (see [`notesdescription`], page 31).

The endnote page created will follow any formatting (page numbers, headers, footers) from the last page in the document.

5.7.2 Output

Let's look at an example. The following text in an input file

```
some text <mynoteref id="1">1</mynoteref> some text
...
<mynote id="1">Endnote Description</mynote>
```

will produce a reference to the end note in the output as follows (with usual representation of numbers):

```
some text 99#a some text
```

The '`99`' in ascii is used as the indicator for an endnote reference. And it will also produce an entry on the end notes page as follows:

```
#a p#g#b Endnote Description
```

with the usual representation of numbers and, in this example, the reference to the note, i.e. the reference character appearing on the 2nd line of the 7th page.

5.7.3 Configuration

`endnotes yes`
: Choose whether to use endnotes or not. Choosing '`no`' will ignore anything enclosed by all the semantic actions Section 5.7.5 [Semantic Actions], page 31.

5.7.4 Styles

The semantic actions `note`, `notesdescription`, and `notesheader` have corresponding styles.

`style note`
: Each endnote on the endnote page will use this style

`style notesheader`
: The style used by the title on the endnote page

`style notesdescription`
: The style used by the note just after the title on the endnote page

The `noteref` semantic action (see [`noteref`], page 31) will inherit its style.

5.7.5 Semantic Actions

`note elementSpecifier id`
: Defines the endnote displayed at the bottom of the file. The id attribute has to be unique, and be identical to the id attribute in a `noteref` in the file in order to be displayed, but the `note` action can appear before the corresponding `noteref` action if needed.

  Given the following definition in the semantic action file

  ```
  note    mynote,idref
  ```

  we could define an endnote in the input file as follows:

  ```
  <mynote idref="1">This is an endnote</mynote>
  ```

`noteref elementSpecifier id`
: Defines the position where the endnote reference character should be placed, as well as the endnote reference character in the endnote at the bottom of the file. The order that these appear in the input file determines the order that they appear in the endnote section.

  Given the following definition in the semantic action file

  ```
  noteref    mynoteref,idref
  ```

  we could place a reference to an endnote in the input file as follows:

  ```
  <mynoteref idref="1">*</mynoteref>
  ```

`notesheader elementSpecifier`
: The text enclosed defines the header to be placed at the top of the first endnote page. For multiple of these defined in the input, the last one is used.

  Given the following definition in the semantic action file

  ```
  notesheader    mynotesheader
  ```

  we could define the page header of the first endnotes page in the input file as follows:

  ```
  <mynotesheader>Endnote Page Header</mynotesheader>
  ```

`notesdescription elementSpecifier`
: Defines some text to be placed after the `notesheader` semantic action (see [`notesheader`], page 31), but before the rest of the endnotes. Again, for multiple definitions in the input file, the last one will be used.

  Given the following definition in the semantic action file

  ```
  notesdescription    mynotesdescription
  ```

  we could define some text to appear on the first endnotes page in the input file as follows:

  ```
  <mynotesdescription>Endnote Page Text</mynotesdescription>
  ```

5.7.6 Example

So, to define end notes we need an input file (see [endnotes.xml], page 32) containing end notes and references to them , end notes have to be enabled in the ini file or via command line options (see [endnotes.ini], page 32) and finally we have to specify which element in the input file constitutes an endnote and a reference to it by defining this in a semantic action file (see [endnotes.sem], page 32). Optionally we can also define styles for the notes, the header of the notes page and the descriptive text (see [endnote styles], page 32).

The following example shows the relevant excerpts in the input, sem and configuration files together with the resulting output (see [endnote output], page 33).

5.7.6.1 input.xml

```
<?xml version=""1.0" encoding="UTF-8"?>
<doc>
    <notesheader>not seen header</notesheader>
    <notesdescription>not seen description</notesdescription>

    <note id = "2">Endnote Description 2</note>

    <p>Foo<noteref id = "1">1</noteref>Bar.</p>
    <p>Foo<noteref id = "2">ref2</noteref>Bar.</p>

    <notesheader>Endnotes</notesheader>
    <notesdescription>Some descriptive text</notesdescription>

    <note id = "1">Endnote Description 1</note>
</doc>
```

5.7.6.2 liblouisutdml.ini

```
endnotes yes
```

5.7.6.3 endnotes.sem

```
note                note,id
noteref             noteref,id
notesheader         notesheader
notesdescription    notesdescription
```

5.7.6.4 styles.cfg

```
style note
    leftMargin 2
    firstLineIndent 2

style notesheader
    format centered

style notesdescription
    format leftJustified
```

5.7.6.5 output.txt

```
Foo 99#a Bar.
Foo 99ref#b Bar.
```

```
-page-
     Endnotes Header

Some descriptive text

#a p#a#a Endnote
  Description #a
ref#b p#a#b Endnote
  Description #b
-page-
```

6 Special Formats

6.1 Tables

Various methods of handling tables can be devised. One that is in current use requires the following lines in a semantic-action file:

```
list tr \*;
generic td \*;\s
```

The `list` style specifies that the first line should begin at the left margin and subsequent lines should be indented two spaces. The third column specifies that a semicolon should be placed at the very end of the row. The `generic` semantic action causes each column in the table to be followed by a semicolon and a space, as specified in the third column. your liblouis table must also contain the following line:

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

6.2 Reserving Space for Graphics

Your configuration files should contain lines like these:

```
style graphspace
rightHandPage yes
```

In your semantic-action file you must assign a tag to this style. Note that the semantic action `graphic` will invoke code to translate SVG graphics when this feature is developed. You can nest various styles within the '`graphspace`' style, such as a caption at the beginning. In particular, you should have another invocation of '`graphspace`' at the end to skip to a new page, or the next right-hand page if you are using interpoint.

6.3 Displayed Text

Conventions for setting off a block of text from the rest vary. you may wish to use the `quotation` style or devise a style of your own.

6.4 Displayed Mathematics

Again, conventions vary. you can define your own style for this purpose and invoke it according to the attributes of the `math` tag.

6.5 Spatial Layouts in Mathematics

This is also known as 2d mathematics. It spreads out complex fractions and other materials for easier viewing. It is being developed based on the specifications of MathML 3.

6.6 Arithmetic Examples

This is another format that is being developed using MathML 3. It is difficult in earlier versions.

6.7 Poetry

`liblouisutdml.ini` defines two styles which can be used to format poetry, as follows:

```
style stanza
linesBefore 1
linesAfter 1
style line
leftMargin 2
firstLineIndent -2
```

Your document might then contain the following from Samuel Taylor Coleridge's "Rime of the Ancient Mariner":

```
<stanza>
<line>He holds him with his glittering eye</line>
<line>The wedding guest stands still</line>
<line>And listens like a three-years' child.</line>
<line>He has no force nor will.</line>
</stanza>
```

Note that when stanzas follow each other liblouisutdml will produce only one blank line between them, not two.

6.8 Dividing a Book Into Volumes

Details are still under development. However, this much can be said. First, obtain a table of contents for the whole book. This requires that your configuration files have the following settings:

```
contents yes
braillePages yes
```

This will tell you the approximate braille pages on which things will be placed in the finished product. You can then calculate the number of pages required for each chapter and how many chapters will fit in a volume of your preferred size. From the point of view of the braille reader, it is desirable to avoid splitting chapters between volumes.

At this point you will probably have to edit the source xml file to indicate the beginning and end of volumes. You can define a liblouisutdml style called '`volume`' and assign appropriate xml tags to it in a semantic-action file. Within the volume style you can nest a title page, chapters, etc. A volume table of contents is still under development.

7 Implementing Braille Mathematics Codes

Much information useful in implementing braille mathematical codes is given in the sections on styles and on semantic actions, especially in the discussion of MathML semantic actions. The chapter on Special Formats also contains much useful information.

The Nemeth Code of Braille Mathematical and Science Notation, BAUK maths and Marburg Maths have been implemented. the Nemeth code was the first and uses an implementation which is now obsolete. The discussion below will concentrate on the implementation of BAUK Maths.

Four tables are used to translate xml documents containing a mixture of text and mathematics. They can be found in the subdirectory `lbu_files` of the liblouisutdml directory and in the `tables` subdirectory of the liblouis distribution. First, the semantic-action file `ukmaths.sem` is used to interpret the mathematical portions of the xml document (The text portions are interpreted by another semantic-action file which will not be discussed here). After the math and text have been interpreted, two liblouis tables, `ukmaths.ctb` and `en-us-g2.ctb` are used to translate them. The latter table may be replaced by another table at the user's discretion. Each piece of mathematics or text is translated separately and the pieces are strung together with blanks between them. This results in inaccuracies where mathematics meets text. The fourth table, also a liblouis table, is used to remove these inaccuracies. It is called `ukmaths_edit.ctb`, and it does things like removing the multi-purpose indicator before a blank, inserting the punctuation indicator before a punctuation mark following a math expression, and removing extra spaces. This table may need editing if a different text translation table is used.

The general format and use of semantic-action files were discussed in the section see Chapter 4 [Connecting with the xml Document - Semantic-Action Files], page 15. In this section we shall concentrate on the optional third column, which is used a lot in `ukmaths.sem`. While the first two columns can be generated by liblouisutdml but must be edited by a person, the third column must always be provided by a human.

As previously stated, the third column tells liblouisutdml what characters to insert to inform liblouis how to translate the math expression. In fact, you can tell liblouis exactly what dots to insert. This relies on the liblouis opcode `exactdots`. If you look at the file `example_ukmaths.ctb` you will see lines like the following:

```
exactdots @126
exactdots @345
exactdots @123456
```

This opcode has only a string operand. liblouis assumes that the characters following the at sign are its dot pattern.

In your semantic-action file you might have lines like:

```
mfenced mfenced @126,@345
mfenced mfenced,open,{ @246,@135
mover mover ,@4-346,@12456
```

By using this approach you do not have to remember which characters will produce the desired dots in a particular liblouis table or on a particular output device.

Sometimes an element or tag can have an indeterminate number of children. This is true of `<math>` itself. Yet, it may be necessary to place some characters after the very last element. Let us look at the `<math>` entry.

```
math math \eb,\*\ee
```

First let us discuss escape sequences starting with a backslash. These are basically the same as in liblouis. The sequence '`\e`' is shorthand for the escape character, which would otherwise be represented by '`\x001b`'. The beginning of a math expression is denoted by an escape character followed by the letter b and the end by an escape character followed by the letter '`e`'. This enables the editing table to do such things as drop the baseline indicator at the end of a math expression and insert a number sign at the beginning, if needed.

Not found in liblouis is the sequence '`\*`'. This means to put what follows after the very last child of the math element, no matter how many there are.

As another example consider:

```
mtd mtd \*\ec
```

`mtd` is the MathML tag for a table column. There may be many children of this tag. The entry says to put an escape character (hex 1b), plus the letter '`c`', after the very last of them.

As a final example consider:

```
mtr mtr ^.^\,^(,\*^.^\,^)\er
```

`mtr` is the MathML tag for a row in a table, in this case a matrix. Each row in a matrix must begin with the dot pattern '`46-6-12356`' and end with the dot pattern '`46-6-12456`'. As usual a caret is placed before the corresponding characters. Since dot 6 is a comma, it must be escaped. This is done by placing a backslash before the comma. There are two subcolumns. the first contains the characters to be placed at the beginning of each row. The second starts with '`\*`', signifying that the characters following it are to be placed at the end of everything in this row. A subcolumn starting with '`\*`' must be the last (or only) subcolumn.

Here this last subcolumn ends with an escape character and the letter `r`, signifying the end of a row.

So much for the semantic action file. Even though the characters in the third column were chosen to correspond with nemeth characters, they may not have to be changed for other math codes. liblouis can replace them with anything needed.

This brings us to a consideration of the two tables used by liblouis to translate mathematics texts. The first, `en-mathtext.ctb` is used to translate text appearing outside math expressions. It is necessary because the Nemeth code requires modifications of Grade 2 braille. Other math codes may not have this requirement.

The table actually used to translate mathematics is `nemeth.ctb`. It includes two other tables, `chardfs.cti` and `nemethdefs.cti`. The first gives ordinary character definitions and is included by all the other tables. Note however, that the unbreakable space, '`\x00a0`', is translated by dot 9. This is used before and after the equal sign and other symbols in `nemeth.ctb`. The second table contains character definitions for special math symbols, most of which are Unicode characters greater than '`\x00ff`'. The Greek letters are here. So are symbols like the integral sign.

Most of the entries in `nemeth.ctb` should be familiar from other tables. The unfamiliar ones follow the comments '`# Semantic pairs`' and '`# pass2 corrections`'. The first simply replace characters preceded by a caret with the character itself. The second make adjustments in the code generated directly from the `nemeth.sem` file. The pass2 opcode is discussed in the liblouis documentation (see Section "Overview" in *Liblouis User's and Programmer's Manual*). Here are some comments on a few of the entries in `nemeth.ctb`.

```
pass2 @1456-1456 @6-1456
```

Replaces double start-fraction indicators with the start complex fraction indicator.

```
pass2 @3456-3456 @6-3456
```

Replaces double end-fraction indicators with the end-complex-fraction indicator.

```
pass2 @56[$d1-5]@5 *
```

Removes the subscript and baseline indicators from numeric subscripts.

```
pass2 @5-9 @9
```

Removes the baseline or multipurpose indicator before an unbreakable space generated by the translation of an equal sign, etc.

```
pass2 @45-3-5 @3
```

Replaces a superscript apostrophe with a simple prime symbol.

```
pass2 @9[]$d @3456
```

Puts a number sign before a digit preceded by a blank.

```
pass2 @9-0 @9
```

Removes a space following an unbreakable space.

We now come to the fourth and last table used for math translation, the editing table, `nemeth_edit.ctb`. As explained at the beginning, this table is used to remove inaccuracies where math translation butts up against text translation. For example, the Nemeth code puts numbers in the lower part of the cell. However, punctuation marks are also in the lower part of the cell. So Nemeth puts a punctuation indicator, dots '`456`', in front of any lower-cell punctuation that immediately follows a mathematical expression. If this occurs inside Mathml it is handled by `nemeth.ctb`. However, a MathML expression is often followed by a punctuation mark which is the first part of text. liblouisutdml puts a blank between math and text, but this can result in a mathematical expression followed by a blank and then, say, a period, dots '`256`'. `nemeth_edit.ctb` replaces the blank with the punctuation indicator.

When you look at `nemeth_edit.ctb` you will see that it begins with an include of `chardefs.cti`. Most of the entries are ordinary, but some are interesting. for example,

```
always "\s 0
```

replaces the baseline or multipurpose indicator followed by a space with just a space.

8 Programming with liblouisutdml

8.1 License

8.2 Overview

liblouisutdml is an "extensible renderer", designed to translate a wide variety of xml and text documents into braille, but with a special emphasis on technical material. The overall operation of liblouisutdml is controlled by a configuration file. The way in which a particular type of xml document is to be rendered is specified by a semantic-action file for that document type. Braille translation is done by the liblouis braille translation and back-translation library (see Section "Overview" in *Liblouis User's and Programmer's Manual*). Its operation, in turn is controlled by translation table files. All these files are plain text and can be created and edited in any text editor. Configuration settings can also be specified on the command line of the console-mode transcription program `file2brl`.

The general operation of liblouisutdml is as follows. It uses the libxml2 library to construct a parse tree of the xml document. After the parse tree is constructed, a function called `examine_document` looks it over and determines whether math translation tables, etc. are needed. `examine_document` also constructs a prototype semantic-action file, if one does not exist already. It may also construct another file containing entries not found in an existing file. When it is finished, another function, called `transcribe_document`, does the actual braille transcription. It calls `transcribe_math` to handle MathML subtrees, `transcribe_chemistry` for chemical formula subtrees, `transcribe_graphic` for SVG graphics, etc. Entities are translated to Unicode, if they are not already. Sequences of symbols indicate superscripts, return to the baseline, subscripts, start and end of fractions, etc. The Braille translator and back-translator library liblouis is used to do the braille translation.

The `transcribe_math` function works in conjunction with the latest version of liblouis and a special math translation table to transcribe most mathematical expressions into good braille mathematical Code.

The functions which are not ready for use at the moment, such as `transcribe_chemistry`, are only skeletons. However, I hope that `transcribe_graphics` can be

expanded in the near future to use the graphics capability of the Tiger tactile graphics embossers.

The latest versions of liblouisutdml and liblouis can be downloaded from `www.liblouis.org`. This site also contains links to a mailing list and to project pages on github.com. Note that liblouisutdml will only work with the latest version of liblouis.

liblouisutdml can be compiled to use either 16-bit or 32-bit Unicode internally. This is inherited from liblouis, so liblouis must be compiled first and then liblouisutdml. Wherever 16 bits are mentioned in this document, read 32 if you have compiled the library for 32 bits.

8.3 Files and Paths

liblouisutdml uses three kinds of files, configuration files, semantic-action files, and liblouis translation tables. The first two are discussed elsewhere in this documentation. liblouis translation tables are discussed in the liblouis documentation (see Section "Overview" in *Liblouis User's and Programmer's Manual*) which is distributed with liblouis.

Note that liblouisutdml also generates some files, all of which are placed in the current directory. These files are new prototype semantic-action files, additions to old semantic-action files, temporary files, and log files. The first two can be used to extend the capability of liblouisutdml to process xml documents. The latter two are useful for debugging.

liblouisutdml determines the paths on which it will search for files at run time, as part of its initialization. First, if the first file in a configuration file list includes a path, liblouisutdml will search first on this path. The path may be either absolute or relative. Only the first filename in a configuration file list may have a path. Next, in Windows, liblouisutdml determines the path to itself. this is the second path on which it will look for files. The liblouis `tables` directory and the liblouisutdml `lbu_files` directory are relative to this path. In Unix systems, including the Mac,, these directories are absolute paths determined at compile time. liblouisutdml searches first the `tables` directory and then the `lbu_files` directory. Finally, it establishes the current directory as the final path to be searched. If you wish the current directory to be the first path searched, prefix the first configuration file name with '`./`' for Unix or '`.\`' for Windows.

Paths are handled in the `paths.c` module. This contains the function `set_paths`, which is called from `readconfig.c` and in turn calls `addPath` in the `paths.c` module.

8.4 lbu_version

```
char *lbu_version (void)
```

This function returns a pointer to a character string containing the version of liblouisutdml. Other information such as the release date and perhaps notable changes may be added later.

8.5 lbu_initialize

```
void * lbu_initialize (
const char *configFilelist,
const char *logFileName,
const char *settingsString)
```

This function initializes the libxml2 library, processes `liblouisutdml.ini` and configuration settings given in the configuration files given in `configFilelist`. This is a list of configuration file names separated by commas. If the first character is a comma it is taken to be a string containing configuration settings and is processed like the `settingsString` string. if the parameter `settingsString` is not `NULL` it is processed last. Such a string must conform to the format of a configuration file. Newlines should be represented with ASCII 10. If `logfilename` is not `null`, a log file is produced on the current directory. If it is `null` any messages are printed on stderr. The function returns a pointer to the `UserData` structure. This pointer is `void` and must be cast to `(UserData *)` in the calling program. To access the information in this structure you must include `louisutdml.h`. This function is used by `file2brl`.

8.6 lbu_translateString

```
int lbu_translateString (
const char *configfilelist,
char * inbuf,
widechar *outbuf,
int *outlen,
unsigned int mode)
```

This function takes a well-formed xml expression in `inbuf` and translates it into a string of 16-bit (or 32-bit if this has been specified in liblouis) braille characters in `outbuf`. The xml expression must be immediately followed by a zero or null byte. Leading whitespace is ignored. If it does not then begin with the characters '`<?xml`' an xml header is added. If it does not begin with '`<`' it is assumed to be a text string and is translated accordingly. The header is specified by the `xmlHeader` line in the configuration file. If no such line is present, a default header specifying UTF-8 encoding is used. The `mode` parameter specifies whether you want the library to be initialized. If it is 0 everything is reset, the `liblouisutdml.ini` file is processed and the configuration file and/or string (see previous section) are processed. If `mode` is 1 liblouisutdml simply prepares to handle a new document. For more on the `mode` parameter see the next section.

Which 16-bit character in `outbuf` represents which dot pattern is indicated in the liblouis translation tables. The `configfilelist` parameter points to a configuration file or string. Among other things, this file specifies translation tables. It is these tables which control just how the translation is made, whether in Grade 2, Grade 1, the Nemeth Code of Braille Mathematics or something else.

Note that the `*outlen` parameter is a pointer to an integer. When the function is called, this integer contains the maximum output length. When it returns, it is set to the actual length used. The function returns 1 if no errors were encountered and a negative number if a complete translation could not be done.

8.7 lbu_translateFile

```
int lbu_translateFile (
char *configfilelist,
char *inputFileName,
char *outputFileName,
```

```
unsigned int mode)
```

This function accepts a well-formed xml document in `inputFilename` and produces a braille translation in `outputFilename`. As for `lbu_translateString`, the `mode` parameter specifies whether the library is to be initialized with new configuration information or simply prepared to handle a new document. In addition, the `mode` parameter can specify that a document is in html, not xhtml. `liblouisutdml.h` contains an enumeration type with the values `dontInit` and `htmlDoc`. These can be combined with an or ('`|`') operator. The input file is assumed to be encoded in UTF-8, unless otherwise specified in the xml header. The encoding of the output file may be UTF-8, UTF-16, UTF-32 or Ascii-8. This is specified by the `outputEncoding` line in the configuration file, `configfilelist`. The function returns 1 if the translation was successful.

8.8 lbu_translateTextFile

```
int lbu_translateTextFile (
char *configfilelist,
char *inputFileName,
char *outputFileName,
unsigned int mode)
```

This function accepts a text file in `inputFilename` and produces a braille translation in `outputFilename`. The input file is assumed to be encoded in Ascii8. However, utf-8 can be specified with the configuration setting `inputTextEncoding utf8`. Blank lines indicate the divisions between paragraphs. Two blank lines cause a blank line between paragraphs (or headers). The output file may be in UTF-8, UTF-16, or Ascii8, as specified by the `outputEncoding` line in the configuration file, `configfilelist`. As for `lbu_translateString`, the `mode` parameter specifies whether complete initialization is to be done or simply initialization for a new document.

8.9 lbu_backTranslateFile

```
int lbu_backTranslateFile (
char *configfilelist,
char *inputFileName,
char *outputFileName,
unsigned int mode)
```

This function accepts a braille file in `inputFilename` and produces a back-translation in `outputFilename`. The input file is assumed to be encoded in Ascii8. The output file is in either plain text or html, according to the setting of `backFormat` in the configuration file. Html files are encoded in UTF8. In plain-text, blank lines are inserted between paragraphs. The output file may be in UTF-8, UTF-16, or Ascii8, as specified by the `outputEncoding` line in the configuration file, `configfilelist`. The mode parameter specifies whether or not the library is to be initialized with new configuration information, as described in the section on `lbu_translateString` (see Section 8.6 [lbu_translateString], page 41).

8.10 lbu_free

```
void lbu_free (void)
```

This function should be called at the end of the application to free all memory allocated by liblouisutdml and liblouis. If you wish to change configuration files during your application, use a `mode` parameter of 0 on the function call using the new configuration information. This will call the `lbu_free` function automatically.

Appendix A Example files

This manual refers to files to files contained in liblouisutdml and liblouis. You might want to look up these files in the source distribution (located under `lbu_files/`) or online in the source code repository (`https://github.com/liblouis/liblouisutdml`) and use them as a reference or to study how things are done. In particular the following files are mentioned:

`liblouisutdml.ini`
: The main initialization file (`https://github.com/liblouis/liblouisutdml/blob/master/lbu_files/liblouisutdml.ini`) that contains the minimal settings for liblouisutdml operation.

`preferences.cfg`
: The preferences.cfg (`https://github.com/liblouis/liblouisutdml/blob/master/lbu_files/preferences.cfg`) file contains all possible configuration settings, with sample values, where appropriate. It is used by the `file2brl` command-line interface if no configuration file is given.

`html.sem`
: An example of a semantic action file to handle html (`https://github.com/liblouis/liblouisutdml/blob/master/lbu_files/html.sem`).

`nemeth.sem`
: An example of a semantic action file to handle math (`https://github.com/liblouis/liblouisutdml/blob/master/lbu_files/nemeth.sem`).

`ukmaths.sem`
: An example of a semantic action file to handle math (`https://github.com/liblouis/liblouisutdml/blob/master/lbu_files/ukmaths.sem`).

`ukmaths.ctb`
: The liblouis UK Maths table for mathematics (`https://github.com/liblouis/liblouis/blob/master/tables/ukmaths.ctb`).

`ukmaths_edit.ctb`
: The liblouis table for post-translation editing of UK Maths (`https://github.com/liblouis/liblouis/blob/master/tables/ukmaths_edit.ctb`).

Configuration Settings Index

U

X

Semantic Action Index

R

S

U

Style Index

Function Index

Program Index

Concept Index