GNU Texinfo `texi2any` Output Customization

for GNU Texinfo version 7.2, 21 October 2024

This manual is for GNU Texinfo `texi2any` program output adaptation using customization files (version 7.2, 21 October 2024).

Short Contents

Table of Contents

1 Overview

> **Warning:** All of this information, with the exception of command-line options and search directories associated with command line options (see Chapter 2 [Loading Init Files], page 2), may become obsolete in a future Texinfo release. Right now, the "API" described in this chapter is immature, so we must keep open the possibility of incompatible, possibly major, changes. Of course we try to avoid incompatible changes, but it is not a promise.

This manual describes how to customize the `texi2any` HTML output. Although some of the features here can technically be used with other output formats, it's not especially useful to do so, so we'll write the documentation as if HTML were the target format. Most of the customizations are only available for HTML.

The conversion of Texinfo to HTML is done in two steps. After reading command-line options and init files, input Texinfo code is parsed into a Texinfo tree and information is gathered on the document structure. This first step can only be customized to a certain extent, by using the command-line options and setting customization variables (see Section "Global Customization Variables" in *Texinfo*). The Texinfo tree describes a Texinfo document in a structured way which makes it easy to go through the tree and format @-commands and other containers.

The second step is the *conversion* step done in a converter. The HTML converter takes a Texinfo tree as input and transforms it to HTML. The code that is used to go through the tree cannot be customized, but the conversion of tree elements can be fully customized.

2 Loading Initialization Files and Search Paths

> **Warning:** The `texi2any-config.pm` file related paths and even the use of `texi2any-config.pm` files is not definitive.

You can write so-called *initialization files*, or *init files* for short, to modify almost every aspect of output formatting. The program loads init files named `texi2any-config.pm` each time it is run. The directories lookup is based on the XDG Base Directory Specification (`https://specifications.freedesktop.org/basedir-spec/latest/`) with defaults based on installation directories. The `texi2any-config.pm` files are looked for in the following directories:

datadir`/texi2any/`
: (where *datadir* is the system data directory specified at compile-time, e.g., `/usr/local/share`)

`XDG_CONFIG_DIRS/texi2any`
: (for each directory in the `:` delimited `XDG_CONFIG_DIRS` environment variable, in reverse order)

sysconfdir`/xdg/texi2any/`
: (where *sysconfdir* is specified at compile time, e.g., `/usr/local/etc`)

`~/.config/texi2any/`
: (where `~` is the current user's home directory, only if `XDG_CONFIG_HOME` is not set)

`XDG_CONFIG_HOME/texi2any`
: (with `XDG_CONFIG_HOME` an environment variable)

`./.texi2any/`
: (under the current directory)

`./`
: (the current directory)

All `texi2any-config.pm` files found are loaded, in the above order. Thus, `./texi2any-config.pm` can override entries in, say, *datadir*`/texi2any/texi2any-config.pm`.

However, the most common way to load an initialization file path is with the `--init-file` option, explicitly specifying the file path to be loaded. If the initialization file path contains directories, it is loaded if found. Otherwise, if the file path is a simple file name, the following directories are searched, in the following order by default. Only the first file found is used:

1. The current directory `./`;
2. `./.texi2any/` under the current directory;
3. `XDG_CONFIG_HOME/texi2any` if the `XDG_CONFIG_HOME` environment is set, otherwise `~/.config/texi2any/` where `~` is the current user's home directory;
4. *sysconfdir*`/xdg/texi2any/` where *sysconfdir* is the system configuration directory specified at compile-time, e.g., `/usr/local/etc`;
5. if the environment variable `XDG_CONFIG_DIRS` is set, *directory*`/texi2any` for each *directory* in the `:` delimited `XDG_CONFIG_DIRS` environment variable value;

6. *datadir*`/texi2any/` Where *datadir* is the system data directory specified at compile time, e.g., `/usr/local/share`;
7. `./.texinfo/init/` under the current directory;
8. `XDG_CONFIG_HOME/texinfo/init` if the `XDG_CONFIG_HOME` environment is set, otherwise `~/.config/texinfo/init/` where `~` is the current user's home directory;
9. *sysconfdir*`/xdg/texinfo/init/` with *sysconfdir* as above;
10. if the environment variable `XDG_CONFIG_DIRS` is set, *directory*`/texinfo/init` for each *directory* in the `:` delimited `XDG_CONFIG_DIRS` environment variable value;
11. *datadir*`/texinfo/init/` with *datadir* as above.
12. *datadir*`/texinfo/ext/` with *datadir* as above.

The *datadir*`/texinfo/ext/` directory contains the init files directly loaded from `texi2any` code. When loaded from `texi2any` code directly, init files are only searched for in that directory, being considered as part of the program and not as user customization. Since the directory is also in the list of directories searched for init files loaded by the `--init-file` option, those init files can also be loaded as regular user specified init files.

Additional directories may be prepended to the list with the `--conf-dir` option (see Section "Invoking `texi2any`" in *Texinfo*).

3 Init File Basics

Init files are written in Perl, and by convention have extension `.init` or `.pm`. Several init files are included in the Texinfo distribution, and can serve as a good model for writing your own. Another example is the `Texinfo::Convert::HTML` module which implements almost all the Texinfo HTML function described in this manual for the conversion to HTML[1]. In `Texinfo::Convert::HTML` the API may not be followed strictly for performance reasons, in that case there should always be a '`API info:`' comment which shows what the API conformant code should be. The Licenses conditions of the diverse files used as example should be taken into account when reusing code.

3.1 Init File Namespace

Initialization file are loaded from the main program in the `Texinfo::Config` namespace. This means that the namespace of the main program and the namespace of initialization files are distinct, which minimizes the chance of a name clash.

It is possible to start init files with:

```
package Texinfo::Config;
```

It is not required, but it may help some debugging tools determine in which namespace the code is run.

In the `Texinfo::Config` namespace, the functions names beginning with '`texinfo_`', '`GNUT_`' and '`_GNUT_`' are reserved. User defined functions in init files should never begin with those prefixes.

The HTML converter is not available directly in the init files namespace, instead it is passed to functions defined in init files that are registered as functions to be called from the converter. See Chapter 6 [User Defined Functions], page 20.

3.2 Getting Build Constants

Some constants are set independently of the output format for a Texinfo build. They are available through `Texinfo::Common::get_build_constant`:

$value = `Texinfo::Common::get_build_constant` (*$name*) [Function]

Retrieve build constant *$name* value.

Defined build constants:

[1] The `Texinfo::Convert::HTML` module also implements the HTML converter which go through the tree and call user defined functions.

`PACKAGE`
`PACKAGE_CONFIG`
`PACKAGE_AND_VERSION`
`PACKAGE_AND_VERSION_CONFIG`
`PACKAGE_NAME`
`PACKAGE_NAME_CONFIG`
`PACKAGE_VERSION`
`PACKAGE_VERSION_CONFIG`
`PACKAGE_URL`
`PACKAGE_URL_CONFIG`

Texinfo package name and versions. Values of build constants without '`_CONFIG`' appended are set by configure. Similar customization variables exist with the same value set in the default case from the main program, with values that can be modified.

The values of the build constants with '`_CONFIG`' appended are duplicate of the values of the build constants without '`_CONFIG`'[2].

3.3 Managing Customization Variables

The basic operations on customization variables are to set and retrieve their values.

The customization variables also valid in the main program out of the HTML converter are handled differently if their associated values are strings or arrays. Conversely, customization variables only relevant for the conversion phase set in the main program are always set by associating a string, an array reference or a hash references to customization variables in the same way.

This section describes customization variables set in the main program. These variables are in general passed to converters. It is also possible to set customization variables in the converters only, not in the main program. This is explained later on (see Section 7.2 [Conversion Customization Variables], page 21).

3.3.1 Setting Main Program String Variables

To set the value of a string customization variable from an initialization file, use `texinfo_set_from_init_file`:

`texinfo_set_from_init_file` (*$variable_name, $variable_value*) [Function]

$variable_name is a string containing the name of the variable you want to set, and *$variable_value* is the value to which you want to set it. *$variable_value* may be '`undef`'.

For example,

```
texinfo_set_from_init_file('documentlanguage', 'fr');
```

overrides the `@documentlanguage` from the document. It would be overridden by `--document-language` on the command line. Another example:

```
texinfo_set_from_init_file('SPLIT', 'chapter');
```

[2] They are set to correspond to macro set in the C code. In the C code there are no macros with the names without '`_CONFIG`' appended as they would clash with the names of the customization options.

overrides the default splitting of the document. It would be overridden by `--split` on the command line.

A final example:

```
texinfo_set_from_init_file('NO_CSS', 1);
```

overrides the default value for `NO_CSS`. It would be overridden by `--set-init-variable NO_CSS=1` on the command line.

Setting the output format cannot be done by setting the customization variable `TEXINFO_OUTPUT_FORMAT`. This customization variable sets the output format in the main program, but not from init files as additional code needs to be run. Instead, call the `texinfo_set_format_from_init_file` function:

`texinfo_set_format_from_init_file` (*$output_format*) [Function]
: *$output_format* is the output format; sets the output format, without overriding formats set from the command line.

Any output format can be set, but since only HTML can be customized, the main use of `texinfo_set_format_from_init_file` is to set the format to '`html`', such that HTML is generated instead of Info in the default case.

For the customization variables associated with @-commands, see Section "Customization Variables for @-Commands" in *Texinfo*. For the customization variables associated with command line options, see Section "Customization Variables and Options" in *Texinfo*.

3.3.2 Modifying Main Program Array Variables

> **Warning:** The main program customization variables associated with arrays are not documented.

Customization variables for the main program associated with an array of values are handled differently. You can use `texinfo_add_to_option_list` to add values to the array and `texinfo_remove_from_option_list` to remove values from the array associated with the customization variable:

`texinfo_add_to_option_list` (*$variable_name*, *$variable_values_array_reference*, *$prepend*) [Function]

`texinfo_remove_from_option_list` (*$variable_name*, *$variable_values_array_reference*) [Function]

$variable_name is the name of the variable; the values in the array reference *$variable_values_array_reference* are added to the list associated with the variable with `texinfo_add_to_option_list`, and removed with `texinfo_remove_from_option_list`.

If the optional argument of `texinfo_add_to_option_list` *$prepend* is set, the values are prepended instead of being appended.

3.3.3 Setting Converter Variables in Main Program

Array and hash references customization variables values relevant in converters only (not in main program, but in the HTML converter) can be set through the main program in init files. These variables cannot be set on the command-line. They are documented in

the customization documentation, not in the main Texinfo manual. You set such arrays or hashes references with `texinfo_set_from_init_file`. For example:

```
my @SECTION_BUTTONS =
  (
   \&singular_banner,
   'Back', 'Forward',   'FastBack', 'FastForward',
   'Up', 'Top', 'Contents', 'Index', 'About'
  );

texinfo_set_from_init_file ('SECTION_BUTTONS', \@SECTION_BUTTONS);
```

3.3.4 Getting Main Program Variables Values

To get the value of a variable, the function is `texinfo_get_conf`:

`texinfo_get_conf` (*$variable_name*) [Function]
: *$variable_name* is the name of the variable; its value (possibly `undef`) is returned.

For example:

```
if (texinfo_get_conf('footnotestyle') eq 'separate') { ... }
```

3.4 Init File Loading Error Reporting

If an error or a warning should be emitted when loading an init file, before the conversion, use `texinfo_register_init_loading_error` for an error and `texinfo_register_init_loading_warning` for a warning.

`texinfo_register_init_loading_error` (*$message*) [Function]
`texinfo_register_init_loading_warning` (*$message*) [Function]
: Cause an error message or a warning message based on *$message* to be output, taking into account options related to error reporting such as `--force` or `--no-warn`.

Errors or warning emitted from user defined functions should use the converter (see Section 7.1 [Error Reporting in User Defined Functions], page 21).

4 Simple formatting customization

Some change in output formatting can be specified with simple code, not very different from simple textual configuration information.

4.1 Init File Expansion Contexts: Normal, Preformatted, Code, String, Math

Output formatting simple customization needs to be specified especially for different formatting contexts. There are five expansion contexts of interest:

normal context
: Paragraphs, index entries, tables, ...

preformatted context
: When spaces between words are kept. For example, within the `@display` (see Section "`@display`" in *Texinfo*) and `@example` environments (see Section "`@example`" in *Texinfo*), and in menu comments. The preformatted regions are usually rendered using `<pre>` elements in HTML.

code context
: When quotes and minus are kept. In particular `---`, ``` `` ``` and other similar constructs are not converted to dash and quote special characters. For example, in `@code` or `@option` commands (see Section "Useful Highlighting" in *Texinfo*).

math context
: Math (see Section "`@math`" in *Texinfo*). Code or preformatted specifications are often used for math too. In those cases, there is no way to separately specify the formatting in math context.

string context
: When rendering strings without formatting elements, for example in titles. The string context allows for limited formatting, typically without any element when producing HTML or XML, so the value can be used in an attribute. XML entities can be used in strings.

It is worth mentioning that in some cases, in particular for file names, plain text can also be used in conversion. There is no associated context in the converter, so the conversion to plain text is usually performed by converting a Texinfo elements tree outside of the main conversion flow.

4.2 Simple Customization for Commands Without Arguments

These commands include those whose names are a single nonletter character, such as `@@`, and those with a normal alphabetic name but whose braces should be empty, such as `@TeX{}` and `@AA{}`.

To change the formatting of a command, the functions is `texinfo_register_no_arg_command_formatting`:

`texinfo_register_no_arg_command_formatting` [Function]
(*$command_name, $context, $text, $html_element, $translated_string_converted, $translated_string_to_convert*)

$command_name is the @-command name, without the leading @. *$context* is '`normal`', '`preformatted`' or '`string`'. There is no separate math context, '`preformatted`' should be used for math context. See Section 4.1 [Init File Expansion Contexts], page 8. If *$context* is `undef`, the '`normal`' context is assumed.

The remaining arguments determine the formatting. If *$text* is set, the corresponding text is output when the @-command is formatted. *$text* can contain HTML elements if needed. If *$html_element* is set, the text is enclosed between the *$html_element* element opening and the element closing. If *$translated_string_converted* is set, the corresponding text is translated when the document language changes and used as text. *$translated_string_converted* should already be HTML. If *$translated_string_to_convert* is set, the corresponding text is translated when the document language changes and converted from Texinfo code to HTML. Since the conversion is done in the appropriate context, *$translated_string_to_convert* should only be set for the '`normal`' context. See Section "Texinfo::Translations METHODS" in `texi2any_internals`.

It is not required to set values for all the contexts. If preformatted context output is not set, normal context output is used. If string context output is not set, preformatted context output is used.

For example, if you want `&shy;` to be output for `@-` in normal, preformatted (and math) and string context, call

```
texinfo_register_no_arg_command_formatting('-', undef, '&shy;');
```

If you want "`<small>...</small>`" to be output for `@enddots` in normal context and `...` to be output in other contexts, call

```
texinfo_register_no_arg_command_formatting('enddots',
                                           'normal', '...', 'small');
texinfo_register_no_arg_command_formatting('enddots',
                                           'preformatted', '...');
```

If you want "`error--&gt;`" to be used for `@error` in every context, with a translation when the document language changes, call

```
texinfo_register_no_arg_command_formatting('error', undef, undef, undef,
                                           'error--&gt;');
```

If you want "`is the` **same** `as`" to be used for `@equiv`, translated when the document language changes, and converted from Texinfo to HTML in the context of the translation, call

```
texinfo_register_no_arg_command_formatting('equiv', undef, undef, undef,
                                    undef, 'is the @strong{same} as');
```

See Section 15.2 [Translated Strings Customization], page 63, for customization of translated strings.

4.3 Simple Customization for Simple Commands with Braces

4.3.1 Customization of Commands Converting to Uppercase

Formatting of `@sc` may involve uppercasing the argument. The specification of @-command argument uppercasing can be changed with `texinfo_register_upper_case_command`:

`texinfo_register_upper_case_command` (*$command_name, $value*) [Function]
$command_name is the @-command name, without the leading @. *$value* sets or unsets uppercasing of argument.

For example, to prevent `@sc` argument from being uppercased and set `@var` argument to be uppercased:

```
texinfo_register_upper_case_command('sc', 0);
texinfo_register_upper_case_command('var', 1);
```

4.3.2 Simple Output Customization for Simple Commands with Braces

You can change the formatting of the output produced by "indicator" and font commands (e.g., `@code`, `@t`), and other simple commands with arguments (e.g., `@asis`, `@clicksequence`, `@sup`, `@verb`) with `texinfo_register_style_command_formatting`:

`texinfo_register_style_command_formatting` (*$command_name, $html_element, $in_quotes, $context*) [Function]
$command_name is the @-command name, without the leading @. *$context* is '`normal`' or '`preformatted`'. There is no separate math context, '`preformatted`' should be used for math context. See Section 4.1 [Init File Expansion Contexts], page 8. There is no string context either, as in string context simple formatting without the need for per command information should be sufficient. If *$context* is `undef`, the '`normal`' context is assumed.

If *$html_element* is set, the argument is enclosed between the *$html_element* element opening and the element closing. If *$in_quotes* is true, the result is enclosed in quotes associated with customization variables `OPEN_QUOTE_SYMBOL` and `CLOSE_QUOTE_SYMBOL` (see Section "Customization of HTML Code Inserted" in *Texinfo*).

If *$html_element* is undefined and *$in_quotes* is not set, the formatted argument is output as is.

For example, to set `@sansserif{argument}` to be formatted as `<code>`*argument*`</code>` in normal and preformatted context, with quotes in preformatted context, use:

```
texinfo_register_style_command_formatting('sansserif', 'code', 0,
                                          'normal');
texinfo_register_style_command_formatting('sansserif', 'code', 1,
                                          'preformatted');
```

To output the formatted argument of `@t` as is:

```
foreach my $context ('normal', 'preformatted') {
  texinfo_register_style_command_formatting ('t', undef,
                                             undef, $context);
}
```

4.4 Simple Customization of Accent Commands

You can modify the formatting of accent commands such as `@'`, `@ringaccent` or `@dotless` by selecting HTML general features, to output numeric entities or characters (see Section "HTML Features Customization" in *Texinfo*).

You can also change how accented commands are converted to named entities. The accent named entities are obtained by combining a letter to be accented, such as '`e`' and a postfix based on the accent command name, for example '`acute`' for the acute accent `@'`. For example, '`@'e`' is converted to the '`&eacute;`' named entity in the default case.

The association of accent @-command and named entity postfix and the list of letters that can be prepended can be changed with `texinfo_register_accent_command_formatting`:

`texinfo_register_accent_command_formatting` [Function]
(*$accent_command_name, $entity_postfix, $letters*)

$accent_command_name is a Texinfo accent formatting @-command name, *$entity_postfix* is a string corresponding to the accent command that is postpended to the letter accent argument. *$letters* is a string listing the letters that can be combined with the *$entity_postfix*. If *$entity_postfix* or *$letters* is an empty string, numeric entities are used instead of named entities.

For example, with the following code, `@dotless{i}` should be converted to `&inodot;`, and `@dotless{j}` to `&jnodot;`. Other letters than '`i`' and '`j`' in argument of `@dotless` should not be combined into a named entity with that example.

```
texinfo_register_accent_command_formatting('dotless', 'nodot', 'ij');
```

4.5 Simple Customization of Containers

Texinfo tree elements that are not text container nor directly associated with an @-command can have information set on their formatting. The first piece of information is whether their contents should be considered in code context (see Section 4.1 [Init File Expansion Contexts], page 8). The other piece of information is the type of preformatted environment they are, analogous with the @-command names of `@example` or `@display`[1].

The function used is `texinfo_register_type_format_info`:

`texinfo_register_type_format_info` (*$type, $code_type,* [Function]
$pre_class_type)

$type is the type of the container. If *$code_type* is set, the container contents are assumed to be in code context. See Section 4.1 [Init File Expansion Contexts], page 8. If *$pre_class_type* is set, the HTML `<pre>` elements class attribute are based on *$pre_class_type*, if there are such HTML elements output for preformatted content of *$type* containers.

For example, to set content appearing in-between node links in `@menu`, which is in the `menu_comment` container type to be formatted in a code context, with preformatted type '`menu-between`', use:

```
texinfo_register_type_format_info('menu_comment', 1, 'menu-between');
```

[1] Note that setting the type of preformatted environment does not make sure that there are preformatted containers used for the formatting of their contents instead of paragraph containers, since this is determined in the very first step of parsing the Texinfo code, which cannot be customized.

See Section "Texinfo::Parser Types of container elements" in `texi2any_internals`, for a description of container types.

5 Simple headers customizations

Headers and footers with a navigation panel are output in the default case. You can already customize the overall formatting and appearance of headers and navigation panels with customization variables (see Section "Customization of Navigation and Headers" in *Texinfo*).

You can specify more precisely the labels and links output in navigation panels with simple code. To explain how navigation panels are customized, we first describe how the document is organized and which directions are available as the directions is the basis for navigation panel customization.

5.1 Output Units

We will call the main unit of output documents a document output *unit*. An output unit's association with output files is determined by the split options (see Section "Splitting Output" in *Texinfo*). This section describes precisely how these output units work, with details for customization.

The output units are:

Normal output units
: These are composed of normal sections and nodes. Usually a node is associated with a following sectioning command, while a sectioning command is associated with a previous node; they both together make up the output unit. Either the node or the sectioning command is considered to be the main element component, depending on the values of the customization variables `USE_NODES` (see Section "HTML Output Structure Customization" in *Texinfo*).

  For example, when generating Info, the nodes are the main elements; when generating HTML, either case may happen (see Section "Two Paths" in *Texinfo*).

Top output unit
: The top output unit is the highest output unit in the document structure. If the document has an `@top` section (see Section "`@top` Command" in *Texinfo*), it is the output unit associated with that section; otherwise, it is the output unit associated with the document's `@node Top` (see Section "The Top Node" in *Texinfo*). If there is no `@node Top`, the first output unit in the document is the top output unit. The Top output unit is also a normal output unit.

Miscellaneous output units
: The remaining output units are associated with different files if the document is split, and also if `MONOLITHIC` is not set (see Section "HTML Output Structure Customization" in *Texinfo*). There are four such miscellaneous output units, also called special output units:

  1. Table of contents
  2. Short table of contents, also called Overview
  3. Footnotes page
  4. About page

More details:

- The *Table of contents* should only be formatted if `@contents` is present in the document.
- Similarly the *Short table of contents* should only appear if `@shortcontents` or `@summarycontents` is present.
- The customization variables `contents` and `shortcontents` may be set to trigger the output of the respective output units.
- If `CONTENTS_OUTPUT_LOCATION` is set to '`separate_element`', the *Table of contents* and *Short table of contents* output units are separate (see Section 16.3 [Contents and Short Table of Contents Customization], page 69). Otherwise the *Table of contents* and *Short table of contents* are directly included within the document, at locations depending on the specific `CONTENTS_OUTPUT_LOCATION` value (see Section "HTML Output Structure Customization" in *Texinfo*.
- When generating HTML, the *Footnotes page* should only be present if the footnotes appear on a separate page (see Section "Footnote Styles" in *Texinfo*). However, a footnote output unit is present if the document is not split.
- The *About page* shouldn't be present for documents consisting of only one sectioning element, or for monolithic documents without navigation information, or if `DO_ABOUT` is not set.

It is common not to have anything but normal output units, especially in case of monolithic output.

A last type of output units exist, "virtual" output units corresponding to directions to external manual nodes. They are not part of the output, but can be used in directions. They are often referred to as *external node units* or *external manual units*. These units do not exist in the document output, therefore there are no functions called to format those output units. They can appear in directions formatting (see Section 17.1 [Navigation Panel Button Formatting], page 73).

5.2 Directions

A variety of data items, called *output units directions*, are associated with output units. They may be used in the formatting functions, and/or associated with a button (see Section 5.4 [Simple Navigation Panel Customization], page 17).

Each output unit direction has a name and a reference to the unit they point to, when such an unit exists. The unit is either a global unit (for example, the Top output unit) or relative to the current output unit (for example, the next output unit). Such relative output units are determined with respect to the document structure defined by the section structuring commands (`@chapter`, `@unnumbered`...) or by the nodes if the node pointers are specified on `@node` lines or in menus (see Section "Two Paths" in *Texinfo*).

Here is the list of global output units directions:

' ' An empty button.

Top Top output unit.

First First output unit in reading order.

Last Last output unit in reading order.

About About (help) page.

Contents Table of contents.

Overview Overview: short table of contents.

Footnotes Corresponds to the `Footnotes` output unit (see Section 5.1 [Output Units], page 13).

Index The first output unit with `@printindex`.

Here is the list of relative output units directions:

This The current output unit.

Forward Next output unit in reading order.

Back Previous output unit in reading order.

FastForward
Next chapter output unit.

FastBack Beginning of this chapter, or previous chapter if the output unit corresponds to a chapter.

Next Next section output unit at the same level.

Prev Previous section output unit at the same level.

Up Up section.

NodeNext Next node output unit.

NodeForward
Next node output unit in node reading order.

NodeBack Previous node output unit in node reading order.

NodePrev Previous node output unit.

NodeUp Up node output unit.

Relative direction are each associated with a variant, with '`FirstInFile`' prepended, which points to the direction relative to the first output unit in file. For example, `FirstInFileNodeNext` is the next node output unit relative to the first output unit in file. The '`FirstInFile`' directions are usually used in footers.

5.2.1 Output Unit Direction Information Type

The output unit directions also have types of information associated, which are in general set dynamically depending on the current output unit, for instance on the output unit whose navigation panel is being formatted:

`href` A string that can be used as an href attribute linking to the output unit corresponding to the direction.

`string` A string representing the direction that can be used in context where only entities are available (attributes). See Section 4.1 [Init File Expansion Contexts], page 8.

`text` A string representing the direction to be used in contexts with HTML elements (preformatted and normal contexts). See Section 4.1 [Init File Expansion Contexts], page 8.

`node` Same as `text`, but selecting the node associated with the output unit direction in priority.

`section` Same as `text`, but selecting the sectioning command associated with the output unit direction in priority.

`text` and `string` also have a variant with '`_nonumber`' prepended, such as `text_nonumber` without sectioning command number in the representation.

5.2.2 Direction Strings

Directions have strings associated, corresponding to their names, description or specific HTML keywords:

`accesskey`
Direction `accesskey` attribute used in navigation.

`button` Direction short name typically used for buttons.

`description`
Description of the direction.

`example` Section number corresponding to the example used in the About special element text.

`rel` Direction `rel` attribute used in navigation.

`text` Direction text in a few words.

'`button`', '`description`' and '`text`' are translated based on the document language.

The `FirstInFile` direction variants are associated with the same strings as the direction they are prepended to (see [`FirstInFile` direction variant], page 15). For example `FirstInFileNodeNext` is associated with the same strings as `NodeNext`.

5.3 Direction Strings Customization

The direction strings are used in the default case in navigation panel formatting, therefore replacing their values is a way to customize headers. They are also used in the About special unit formatting (see Section 16.4 [About Special Output Unit Customization], page 71). The direction strings can be customized with `texinfo_register_direction_string_info`:

`texinfo_register_direction_string_info` (*$direction, $type, $converted_string, $string_to_convert, $context*) [Function]

$direction is a direction (see Section 5.2 [Directions], page 14), *$type* is the type of string (see Section 5.2.2 [Direction Strings], page 16). The other arguments are optional. *$context* is '`normal`' or '`string`'. See Section 4.1 [Init File Expansion Contexts], page 8. If $context is `undef`, the '`normal`' context is assumed.

$converted_string is the string, already converted to HTML that is used for the specified context. If the '`normal`' context *$converted_string* only is specified, the same string will be used for the '`string`' context.

Alternatively, *$string_to_convert* can be specified to set the string to the corresponding Texinfo code after translation and conversion to HTML. In that case, the context is ignored, as it will be set at the time of the conversion.

$string_to_convert is ignored for special strings that do not need to be translated and cannot contain Texinfo @-commands ('`accesskey`', '`rel`' and '`example`'). *$string_to_convert* is also ignored if *$converted_string* is set for any context.

For example, to set the '`Up`' '`button`' to be translated as '`Higher`', use:

```
texinfo_register_direction_string_info('Up', 'button', undef, 'Higher');
```

5.4 Simple Navigation Panel Customization

The *navigation panel* is the line of links (and labels) that typically appears at the top of each node, so that users can easily get to the next node, the table of contents, and so on. It can be customized extensively.

You can set the `ICONS` customization variable to use icons for the navigation panel. Setting `ICONS` is necessary but not sufficient to get icons for direction buttons since no button image is specified in the default case. The `ACTIVE_ICONS` and `PASSIVE_ICONS` customization variables need to be set in addition:

`ACTIVE_ICONS`
`PASSIVE_ICONS`
: Hash references with output unit directions as key (see Section 5.2 [Directions], page 14) and button image icons as values. `ACTIVE_ICONS` is used for directions actually linking to an output unit, and `PASSIVE_ICONS` are used if there is no output unit to link to. The button images are interpreted as URLs.

Several arrays and hashes enable even more precise control over the navigation panel buttons and their display. They can be set as customization variables with `texinfo_set_from_init_file`. See Section 3.3.1 [Setting Main Program String Variables], page 5.

The following customization variables arrays determine the buttons present in the various navigation panels:

`SECTION_BUTTONS`
: Specifies the navigation panel buttons present at the beginning of sectioning elements in the case of section navigation being enabled or if split at nodes. Specifies the navigation panel buttons present at the page header if split at section and there is no section navigation.

`SECTION_FOOTER_BUTTONS`
`CHAPTER_FOOTER_BUTTONS`
`NODE_FOOTER_BUTTONS`
: These arrays specify the navigation panel buttons present in the page footer when the output is split at sections, chapters or nodes, respectively.

`CHAPTER_BUTTONS`
: Specifies the buttons appearing at the page header if split at chapters and there is no section navigation.

`MISC_BUTTONS`
: Specifies the buttons appearing at the beginning of special output units and, if the output is split, at the end of such units.

`LINKS_BUTTONS`
: Used for `<link>` elements if they are output in the HTML `<head>`.

`TOP_BUTTONS`
`TOP_FOOTER_BUTTONS`
: Specifies the buttons used in the top output unit (see Section 5.1 [Output Units], page 13). `TOP_BUTTONS` buttons are used for the header and `TOP_FOOTER_BUTTONS` are used for the footer.

Each array specifies which buttons are included, and how they are displayed. Each array item is associated with a button of the navigation panel from left to right. The meaning of the array item values is the following:

string with an output unit direction
: If icons are not used, the button is a link to the corresponding output unit whose text is the `text` direction string (see Section 5.2.2 [Direction Strings], page 16), surrounded by '`[`' and '`]`'. If the direction is ' ', the '`[`' and '`]`' are omitted.

  If icons are used, the button is an image whose file is determined by the value associated with the output unit direction in the `ACTIVE_ICONS` variable hash if the link leads to an output unit, or in the `PASSIVE_ICONS` variable hash if there is no output unit to link to. If there is a link to the output unit, the icon links to that output unit. The button name and button description are given as HTML attributes to have a textual description of the icon. The corresponding strings correspond to the `button` direction string for the button name and the `description` for a more detailed description (see Section 5.2.2 [Direction Strings], page 16).

function reference
: The functions is called with one arguments, the converter object. The function should return two scalars, the button text and a boolean set if a delimiter is desired before the button.

scalar reference
: The scalar value is printed.

array reference of length 2
: Here, the first array item should be a an output unit direction. The text of the link depends on the second array item.

  reference to a text string
  : A link to the output unit associated with the output unit direction is generated with the corresponding text used for the text link.

reference to a function
: The functions is called with three arguments, the converter object, the output unit direction and the source Texinfo tree element (possibly undef). The function should return two scalars, the button text and a boolean set if a delimiter is desired before the button.

No delimiter is printed before the first button. Leading space buttons mixed with directions not found may be omitted of the navigation panel output.

If the customization variable `USE_ACCESSKEY` is set, the `accesskey` attribute is used in navigation. The `accesskey` direction string is then used for the `accesskey` attributes (see Section 5.2.2 [Direction Strings], page 16).

Similarly, if the `USE_REL_REV` customization variable is set, the `rel` attribute is used in navigation. In that case the `rel` direction string is used for the `rel` attribute (see Section 5.2.2 [Direction Strings], page 16).

6 User Defined Functions

Getting beyond the customization described previously requires writing some functions and registering those functions such that they are called for the conversion. This allows dynamic redefinition of functions used to produce output.

6.1 User Defined Functions are Registered

User defined functions are always passed as a code reference to a registering function, together with a string describing what the function formats. In the following made up example, `my_formatting_function` is passed as a function reference `\&my_formatting_function` to the fictitious registering function `texinfo_register_some_formatting`, with the string specifying the formatting done by the function being '`format_thing`':

```
sub my_formatting_function {
  my $arg1 = shift;
  my $arg2 = shift;
  # prepare $formatted_text
  ...
  return $formatted_text;
}

texinfo_register_some_formatting ('format_thing', \&my_formatting_function);
```

As such functions are defined by a reference name associated with a string we will always use the string in function prototypes. For the function arguments we will use `\@array` to indicate a reference to an array (a.k.a. list, in Perl terminology), `\%hash` for a reference to a hash and `\&function` for a reference on a function.

To illustrate these conventions, here is the prototype for the function associated with '`format_thing`':

$text format_thing ($arg1, \@arg2) [Function Reference]

A function reference associated with '`format_thing`' has a first argument *$arg1*, a second argument a reference to an array *\@arg2*, and returns the formatted text *$text*.

6.2 Converter Object and Conversion Functions

The first argument of most, if not all user defined function is a converter object. This object gives access to methods to get information on the conversion context and to methods useful for the conversion, both as an HTML converter and as a generic `Texinfo::Convert::Converter` (see Section "Texinfo::Convert::Converter Helper methods" in `texi2any_internals`). The converter can be used for error reporting by using `Texinfo::Convert::Converter` methods. See Section 7.1 [Error Reporting in User Defined Functions], page 21, on error reporting. The converter can also be used for in-document strings translation. See Chapter 15 [Translations in Output and Customization], page 62, on translated strings in output.

7 Error Reporting, Customization and Paths Manipulation with Converter

Some tasks common to all the user-defined functions related to error reporting, customization variables handling and paths and URL manipulation are described in this chapter.

7.1 Error Reporting in User Defined Functions

To report an error or a warning in a user defined function, use the methods of `Texinfo::Convert::Converter` through a converter object (see Section 6.2 [Converter Object and Conversion Functions], page 20).

To report a warning or an error not specific of an element conversion, use `converter_document_warn` or `converter_document_error`:

$converter*`->converter_document_error` (*$text, $continuation) [Function]
$converter*`->converter_document_warn` (*$text, $continuation) [Function]
Register a document-wide error or warning. *$text* is the error or warning message.

The optional *$continuation* argument, if set, conveys that the message is a continuation of the previous registered message.

To report a warning or an error in element conversion, use `converter_line_warn` or `converter_line_error`

$converter*`->converter_line_error` (*$text, \%location_info, $continuation) [Function]
$converter*`->converter_line_warn` (*$text, \%location_info, $continuation) [Function]
Register a warning or an error. *$text* is the text of the error or warning. The *\%location_info* holds the information on the error or warning location. The *\%location_info* reference on hash may be obtained from Texinfo elements `source_info` keys.

The optional *$continuation* argument, if set, conveys that the message is a continuation of the previous registered message.

Note that registering an error does not stop the processing of the Texinfo tree.

See Section "Texinfo::Convert::Converter Registering error and warning messages" in `texi2any_internals` for the converter module documentation on errors and warning messages registration.

7.2 Setting and Getting Conversion Customization Variables

The customization variables values set during the conversion process may be different from the main program customization variables. The general rule is that variables set in the main program, in particular from init files, are passed to the converter. Some variables, however, only appear in the converter. Some variables are also set in the converter based on the main program customization variables. Finally, some variables should be set or reset during conversion, in particular when converting the tree representing the Texinfo document, when expanding the tree element corresponding to @-commands associated with customization variables (see Section "Customization Variables for @-Commands" in *Texinfo*).

The functions described here should be used in user defined functions, but should not be used out of functions. Conversely, the similar functions used to set customization variables from init files without a converter should not be used in functions, but should be used out of functions in init files (see Section 3.3 [Managing Customization Variables], page 5).

To get the value of a variable in a converter `$converter`, the function is `get_conf`:

$result = *$converter*`->get_conf (`*$variable_name*`)` [Function]
$variable_name is the name of the variable; its value in the converter *$converter* (possibly `undef`) is returned.

For example:

```
my $footnotestyle = $converter->get_conf('footnotestyle');
```

To set a variable in a converter `$converter`, the function is `set_conf`:

$converter*`->set_conf (`*$variable_name, $variable_value`)` [Function]
$variable_name is the name of the variable; its value in the converter *$converter* is set to *$variable_value*. The *$variable_name* value will not be overidden if it was set from the command line or from an init file.

For example:

```
$converter->set_conf('footnotestyle', 'separate');
```

Some customization variables, in particular those associated with @-commands, can be reset to the value they had before starting the conversion. For example, they are reset in order to obtain their value before the conversion. Thet are also reset to the value they had before starting the conversion when their value at the end of the preambule or at the end of the document is needed, but there are no @-commands at those locations in the Texinfo manual. If a value set by `set_conf` is intended to be found when the customization variable value is reset, `set_conf` should be called early. For example, when called from a user-defined function called at different stage, it should be called in the '`setup`' stage (see Chapter 9 [Init File Calling at Different Stages], page 28).

The values set in converter with `set_conf` will not override command-line set customization variables, nor variables set early in init files. This is the expected behaviour, in particular when the values are set from the document. In the rare cases when overriding the customization would be needed, the `force_conf` function can be used:

$converter*`->force_conf (`*$variable_name, $variable_value`)` [Function]
$variable_name is the name of the variable; its value in the converter *$converter* is set to *$variable_value*, overriding any previous value.

7.3 Encoding and Decoding File Path Strings

7.3.1 Encoding File Path Strings

In general, the strings in the customization functions are character strings. For most purposes, this is right, and the encoding in output files is taken care of by the converter. Operations on directories and file names, however, such as the creation of a directory or the opening of a file require binary strings.

To encode file names consistently with file name encoding used in the conversion to HTML, there is a function `encoded_output_file_name`:

(*$encoded_name*, *$encoding*) = *$converter*->encoded_output_file_name (*$character_string_name*) [Function]

Encode *$character_string_name* in the same way as other file name are encoded in the converter. Use OUTPUT_FILE_NAME_ENCODING value for the file name encoding if set. Otherwise, if DOC_ENCODING_FOR_OUTPUT_FILE_NAME is set the input Texinfo document encoding is used, if unset, the default, files names are encoded using the current locale (see Section "Global Customization Variables" in *Texinfo*). Return the encoded name and the encoding used to encode the name.

There is also a similar function for the input file names encoding, encoded_input_file_name, which uses INPUT_FILE_NAME_ENCODING and DOC_ENCODING_FOR_INPUT_FILE_NAME and is less likely to be useful.

When calling external commands, the command line arguments should also be encoded. To do similarly with other codes, the customization variable MESSAGE_ENCODING should be used. Already encoded file names may be used. For example

```
use Encode qw(encode);

....

my ($encoded_file_path, $encoding)
  = $converter->encoded_output_file_name($file_name);

my $fh = open($encoded_file_path);

.....

my $call_start = "command --set '$action' ";
my $encoding = $converter->get_conf('MESSAGE_ENCODING');
if (defined($encoding)) {
  $encoded_call_start = encode($encoding, $call_start);
} else {
  $encoded_call_start = $call_start;
}
my $encoded_call = $encoded_call_start . $encoded_file_path;
my $call = $call_start . $file_name;
if (system($encoded_call)) {
 $converter->document_error($converter,
     sprintf(__("command did not succeed: %s"),
            $call));
}
```

7.3.2 Decoding File Path Strings

The binary strings that could be accessed correspond to the customization variables strings or arrays INCLUDE_DIRECTORIES, CSS_FILES, MACRO_EXPAND and INTERNAL_LINKS. If they need to be decoded into character strings, for example to appear in error messages, it is possible to use the COMMAND_LINE_ENCODING customization variable value as encoding name to mimic how the decoding of these strings from the command line is done in the

main program and in the converters. See Section "Global Customization Variables" in *Texinfo*.

For example:

```
my $macro_expand_fname = $converter->get_conf('MACRO_EXPAND');
my $encoding = $converter->get_conf('COMMAND_LINE_ENCODING');
if (defined($encoding)) {
  $macro_expand_fname = Encode::decode($encoding, $macro_expand_fname);
}
```

More information on Perl and encodings in perlunifaq (`https://perldoc.perl.org/perlunifaq`).

7.4 Protection of URLs

URLs need to be "percent-encoded" to protect non-ASCII characters, spaces and other ASCII characters. Percent-encoding also allows to have characters be interpreted as part of a path and not as characters with a special role in URLs. For example, '`?`' has a special role in URLs as it starts a query string. To have it considered as part of a file path, instead of a marker of the beginning of a query, it needs to be percent encoded.

Convenience functions are provided for URL protection. To protect a whole URL, in which characters with a special role in URL are left as is, use `url_protect_url_text`. To protect file path in URL, including characters with a special role in URLs, use `url_protect_file_text`.

$protected_url = *$converter*`->url_protect_url_text(`*$input_string*`)` [Function]

Percent-encode *$input_string*, leaving as is all the characters with a special role in URLs, such as '`:`', '`/`', '`?`', '`&`', '`#`' or '`%`' (and a few other). HTML reserved characters and form feeds protected are also protected as entities (see Section 11.2 [`format_protect_text`], page 41). This is typically used on complete URLs pointing to diverse internet resources, such as the `@url` URL argument.

for example

```
return $converter->html_attribute_class('a', ['myurl'])
 .' href="'.$converter->url_protect_url_text($url)."\">$text</a>";
```

$protected_path = *$converter*`->url_protect_file_text(`*$input_string*`)` [Function]

Percent-encode *$input_string* leaving as is character appearing in file paths only, such as '`/`', '`.`', '`-`' or '`_`'. All the other characters that can be percent-protected are protected, including characters with a special role in URLs. For example, '`?`', '`&`' and '`%`' are percent-protected. HTML reserved characters and form feeds protected are also protected as entities (see Section 11.2 [`format_protect_text`], page 41). This is typically used on file names corresponding to actual files, used in the path portion of an URL, such as the image file path in `@image`.

For example

```
$converter->html_attribute_class('img', [$cmdname])
   . ' src="'.$converter->url_protect_file_text($image_file)."\");
```

8 Customizing Output-Related Names

It is possible to control both output file names and target identifiers in detail.

User defined functions customizing file names and targets are registered with `texinfo_register_file_id_setting_function`:

`texinfo_register_file_id_setting_function` (*$customized*, *\&handler*) [Function]

$customized is a string describing what the function should set. *\&handler* should be a reference on the user defined function. The different functions that can be registered have different arguments and return values.

The different possibilities for the customized information are explained in the next sections.

For example:

```
sub my_node_file_name($$$) {
  my ($converter, $element, $filename) = @_;
  # ....
  return $node_file_name
}

texinfo_register_file_id_setting_function('node_file_name',
                                          \&my_node_file_name);
```

8.1 Customizing Output File Names

You can specify the output file or directory, intermediate directories and file extension with customization variables (see Section "File Names and Links Customization for HTML" in *Texinfo*).

Two function references registered with `texinfo_register_file_id_setting_function` enable further customization. The first, `node_file_name` is used to customize the nodes files names.

$node_file `node_file_name` (*$converter*, *\%node_element*, *$file_name*) [Function Reference]

$converter* is a converter object. *\%node_element* is the Texinfo tree element corresponding to the `@node`. *$file_name is the node file name that has been already set. The function should return the node file name (*$node_file*).

The other function reference, `unit_file_name`, is used to customize the file names associated with each normal output unit (see Section 5.1 [Output Units], page 13).

(*$file*, *$path*) `unit_file_name` (*$converter*, *\%output_unit*, *$file_name*, *$file_path*) [Function Reference]

$converter* is a converter object. *\%output_unit* is the output unit. *$file_name is the file name that has been already set. *$file_path* is the file path that has been already set. *$file_path* is '`undef`' if the file is relative to the output directory, which is the case if the output is split. The function should return the file name for the output

unit, *$file*, and the file path for the output unit, *$path*, which should be '`undef`' if the file path is to be constructed by putting *$file* in the destination directory.

In the user defined functions, the information that an output unit is associated with `@top` or `@node Top` may be determined with:

```
$converter->unit_is_top_output_unit(\%output_unit);
```

The information on tree elements may be interesting for those functions (see Section 10.4.4 [Texinfo Tree Elements in User Defined Functions], page 37). The `extra` key `associated_section` of a node element and `associated_node` of a sectioning command element may also be useful.

The file name associated with a sectioning command is set together with the target, and is described in the next section.

8.2 Customizing Output Target Names

Similar to file names, so-called target and id names may be set. The *id* is placed where the item is located, while the *target* is used to construct references to that item. The id and target are the same. A function used to set both target and file name is also described here.

The following function reference is for target items (nodes, anchors, floats), including for external manuals:

$target `label_target_name (`*`$converter, $normalized, \%element, $default_target`*`)` [Function Reference]

$converter is a converter object. *$normalized* is the normalized node name, *\%element* is a reference on a Texinfo tree command label element whose contents correspond to the node name. *$default_target* is the target that has been already set. The function should return the target (*$target*).

The element corresponding to the label can be found with `label_command` if the label corresponds to an internal reference (see Section 12.1.4 [Target Tree Element Link], page 47):

```
my $element;
$element = $converter->label_command($normalized)
  if (defined($normalized));
```

For sectioning commands, in addition to the sectioning command target, targets for the sectioning command in table of contents and in short table of contents are needed. The following function reference is for sectioning command related target and file name:

(*$target, $target_contents, $target_shortcontents, $file*) `sectioning_command_target_name (`*`$converter, \%section_element, $default_target, $default_target_contents, $default_target_shortcontents, $file_name`*`)` [Function Reference]

$converter is a converter object. *\%section_element* is the Texinfo element corresponding to the sectioning command.

$default_target, *$default_target_contents* and *$default_target_shortcontents* are the targets that have been already set for the sectioning element and the sectioning element in table of contents and in short table of contents. *$file_name* is the file name that has been already set.

The function should return the *$target*, *$target_contents* and *$target_shortcontents* sectioning element target and sectioning element in table of contents and in short table of contents targets, and the file name for the sectioning element (*$file*).

8.3 Customizing External Node Output Names

In the default case references to external nodes are set as described in the Texinfo manual (see Section "HTML Xref" in *Texinfo*). You can specify external node manuals URLs in cross-references customization files (see Section "HTML Xref Configuration" in *Texinfo*). You can also set a base directory, the Top node file target, the extension and other overall references to external nodes formatting with customization variables (see Section "File Names and Links Customization for HTML" in *Texinfo*).

If the external reference is not already ignored because of `IGNORE_REF_TO_TOP_NODE_UP`, two function references give full control over the external node target output names, with `external_target_split_name` if the external target is considered to be split, and `external_target_non_split_name` if the external target is non split.

(*$target*, *$host_directory*, *$file_name*) [Function Reference]
external_target_split_name(*$converter*, *$normalized*, *\%element*, *$default_target*, *$default_host_directory*, *$default_file_name*)

$converter is a converter object. *$normalized* is the normalized node name, *\%element* is a reference on an element containing information on the external node.

$default_target, *$default_host_directory* and *$default_file_name* are the target, host and directory URL part and file name URL part that have been already set.

The function should return the *$target*, *$host_directory* and *$file_name* URL parts.

(*$target*, *$host_directory_file*) [Function Reference]
external_target_non_split_name(*$converter*, *$normalized*, *\%element*, *$default_target*, *$default_host_directory_file*)

$converter is a converter object. *$normalized* is the normalized node name, *\%element* is a reference on an element containing information on the external node.

$default_target is the target and *$default_host_directory_file* is the host and file name part of the URL that have been already set.

The function should return the *$target* and *$host_directory_file* URL parts.

8.4 Customizing Special Elements Output Names

For special output units file and target (see Section 5.1 [Output Units], page 13), the function reference is:

(*$target*, *$file*) special_unit_target_file_name [Function Reference]
(*$converter*, *\%output_unit*, *$default_target*, *$file_name*)

$converter* is a converter object. *\%output_unit* is the special output unit hash. *$default_target is the target that has been already set, and *$file_name* is the file name that has been already set. The function should return the *$target* and *$file*.

To determine the variety of the special output unit processed, the output unit `special_unit_variety` hash key can be used. See Table 16.1.

9 Init File Calling at Different Stages

Arbitrary user-defined functions may be called during conversion. This could be used, for example, to initialize variables and collect some @-commands text, and doing clean-up after the Texinfo tree conversion.

There are four stages for user defined functions:

`setup` Called right after completing main program customization information with converter specific customization information, but before anything else is done, including collecting the output files names and registering the customization variables pre-conversion values.

`structure`
Called after setting and determining information on CSS, output files and directories, document structure and associated directions, file names, labels and links for nodes, sectioning commands, special output units, footnotes and index entries.

`init` Called after some gathering of global information on the document, such as titles, copying comment and document description, which require some conversion of Texinfo, right before the main output processing. At that point most of the information available from the converter is set (see Section 10.2 [Converter General Information], page 30).

`finish` Called after output generation is finished.

The function used to register a user defined functions is `texinfo_register_handler`:

`texinfo_register_handler` (*$stage*, *\&handler*, *$priority*) [Function]
$stage* is one of the stages described just above. *\&handler* is a reference on the user defined function. *$priority is an optional priority class.

To determine the order of user defined functions calls, the priority classes are sorted, and within a priority class the order is the order of calling `texinfo_register_handler`.

The call of the user defined functions is:

$status `stage_handler` (*$converter*, *$document*, *$stage*) [Function Reference]
$converter is a converter object. *$document* is the Texinfo parsed `Texinfo::Document` document. *$stage* is the current stage.

If *$status* is not 0 it means that an error occured. If *$status* is positive, the user defined functions should have registered an error or warning message, for example with `document_error` (see Section 7.1 [Error Reporting in User Defined Functions], page 21). If *$status* is negative, the converter will emit a non specific error message. If the *$status* is lower than `-HANDLER_FATAL_ERROR_LEVEL` or higher than `HANDLER_FATAL_ERROR_LEVEL`, the processing stops immediately. Default value for `HANDLER_FATAL_ERROR_LEVEL` is 100.

10 Formatting HTML Output

HTML output formatting in user-defined code should take into account the conversion context, can access converter information and use converter functions to convert Perl Texinfo trees. There are also several conventions and constraints that user defined code should abide to when formatting HTML, in order to comply with customization option values, avoid modifying structures that should not be modified, and also to have information correctly registered in the converter.

Formatting of HTML output should be used in formatting functions (see Chapter 11 [Customization and Use of Formatting Functions], page 40), tree elements conversion functions (see Chapter 12 [Tree Element Conversion Customization], page 43) and output units conversion functions (see Chapter 13 [Output Units Conversion Functions], page 58) described later on. Tree elements and output units conversion functions can also be used to output HTML, how to retrieve the corresponding function references and call those functions is also described with the functions customization.

10.1 Specific HTML Constructs Formatting Functions

A few specific HTML constructs should be formatted using particular functions: elements with classes, "void elements" without end tag and non breaking spaces.

10.1.1 Formatting HTML Element with Classes

Opening an HTML element with one or more classes should always be done through `html_attribute_class`:

$element_open = *$converter*`->html_attribute_class` (*$html_element*, `\@classes`) [Function]

Formats the beginning of an HTML element *$html_element*. `\@classes` is the list of classes for this element. The element opening returned does not include the end of element symbol '`>`' such that it is possible to add more attributes.

If the HTML element is `span`, an empty string is returned if there is also no attribute.

If `NO_CSS` is set, no attribute is set for the element. Otherwise a `class` attribute is set based on `\@classes` . If `INLINE_CSS_STYLE` is set, a CSS style attribute based on CSS element class rules is also added (see Section "HTML CSS" in *Texinfo*). Otherwise the information that the element class was seen is registered by the converter.

Examples of use:

```
my $open = $converter->html_attribute_class('span', ['math-arg']);
my $arg_result = $open.'>'.$arg.'</span>'
  if ($open ne '');

my $result = $converter->html_attribute_class('em', [$cmdname, 'jax_p'])
   . '>' . $arg_result . '</em>';
```

10.1.2 Closing Lone HTML Element

HTML elements with an opening element, but no closing element, such as `<img>` or `<link>`, also called *void elements* should be closed by calling `close_html_lone_element`:

$html_element = *$converter*->close_html_lone_element (*$unclosed_element*) [Function]

Close the *$unclosed_element*, which can contain attributes, by prepending '>' or '/>' depending on the USE_XML_SYNTAX customization variable value (see Section "HTML Features Customization" in *Texinfo*).

Examples of use:

```
$description = $converter->close_html_lone_element(
    "<meta name=\"description\" content=\"$description\"");
```

10.1.3 Substituting Non Breaking Space

A non-breaking code should be inserted using the non_breaking_space information, taken from the general information (see Section 10.2 [Converter General Information], page 30), using get_info:

```
my $non_breaking_space = $converter->get_info('non_breaking_space');
my $result = '<tr><td>' .$non_breaking_space. '</tr></td>';
```

In that case, there is nothing more to do.

If a   can directly appear in formatted code, however, the corresponding text should be in a call to substitute_html_non_breaking_space, to take into account OUTPUT_CHARACTERS and USE_NUMERIC_ENTITY customization variables:

$substituted_text = *$converter*->substitute_html_non_breaking_space (*$formatted_text*) [Function]

Substitute   according to customization variables values.

See Section "HTML Features Customization" in *Texinfo* for OUTPUT_CHARACTERS and USE_NUMERIC_ENTITY description.

10.2 Converter General Information

Some general information is available from the converter. This information should not change during conversion.

To determine if an output format such as 'html' or 'tex' is expanded (see Section "Conditional Commands" in *Texinfo*), use is_format_expanded:

$is_format_expanded = *$converter*->is_format_expanded (*$format*) [Function]

Return true if format *$format* is expanded, according to command-line and init file information.

The main method to get information from the converter is get_info:

$info = *$converter*->get_info (*$item*) [Function]

Return information on *$item*.

The available information is about:

`copying_comment`
: Text appearing in `@copying` with all the Texinfo commands put into comments (see Section "`@copying`" in *Texinfo*).

`destination_directory`
: Destination directory for the output files. It is common to use that string in directory or file paths with functions requiring binary strings. In that case the character string needs to be encoded. See Section 7.3.1 [Encoding File Path Strings], page 22.

`document`
: The `Texinfo::Document` parsed Texinfo document being converted. Some information relevant for conversion is available from the document using function accessors:

  `floats_information`
  : Information on floats. See Section "Texinfo::Document::floats_information" in `texi2any_internals`.

  `global_commands_information`
  : Global commands information. See Section "Texinfo::Document::global_commands_information" in `texi2any_internals`.

  `global_information`
  : Diverse information. See Section "Texinfo::Document::global_information" in `texi2any_internals`.

  `indices_information`
  : Information about defined indices, merged indices and index entries. See Section "Texinfo::Document::indices_information" in `texi2any_internals`.

  `sections_list`
  : List of the sectioning commands in the document.

  See Section "Texinfo::Document Getting document information" in `texi2any_internals` on information available from the document.

`document_name`
: Base name of the document. It is common to use that string in in directory or file paths with functions requiring binary strings. In that case the character string needs to be encoded. See Section 7.3.1 [Encoding File Path Strings], page 22.

`documentdescription_string`
: `@documentdescription` argument converted in a string context (see Section "`@documentdescription`" in *Texinfo*). See Section 4.1 [Init File Expansion Contexts], page 8.

`expanded_formats`
: Information on output formats such as '`html`' or '`tex`' expansion (see Section "Conditional Commands" in *Texinfo*). An hash reference with format names as

key and a true value as value if the format is expanded, according to command-line and init file information.

`expanded_formats` information should be consistent with `is_format_expanded` call result (see [`is_format_expanded`], page 30).

`jslicenses`
: An hash reference with categories of javascript used in the document as keys. The corresponding values are also hashes with file names as keys and with array references as values. The array references contain information on each of the file licences, with content

  1. licence name
  2. license URL
  3. file name or source of file

`line_break_element`
: HTML line break element, based on '`<br>`', also taking into account `USE_XML_SYNTAX` customization variable value.

`non_breaking_space`
: Non breaking space, can be '` `', but also a non breaking space character or the corresponding numeric entity based on `OUTPUT_CHARACTERS` and `USE_NUMERIC_ENTITY` customization variables values. See Section 10.1.3 [Substituting Non Breaking Space], page 30.

`paragraph_symbol`
: Paragraph symbol, can be '`&para;`', but also the corresponding numeric entity or encoded character based on `OUTPUT_CHARACTERS` and `USE_NUMERIC_ENTITY` customization variables values. See Section "HTML Features Customization" in *Texinfo*.

`title_string`
`title_tree`
`simpletitle_tree`
`simpletitle_command_name`
: Some information is deduced from the title commands: *simpletitle* reflects `@settitle` vs. `@shorttitlepage`, and *title* is constructed by trying all the title-related commands, including `@top` and `@titlefont`, in the top element.

  `title_tree` is a Texinfo tree corresponding to the title, `title_string` is the result of the conversion in a string context (see Section 4.1 [Init File Expansion Contexts], page 8). `simpletitle_tree` is a Texinfo tree corresponding to the simpletitle, and `simpletitle_command_name` is the @-command name used for the simpletitle, without the leading @.

`title_titlepage`
: The formatted title, possibly based on `@titlepage`, or on `simpletitle_tree` and similar information, depending on `SHOW_TITLE` and `USE_TITLEPAGE_FOR_TITLE` customization variables in the default case (see Section "HTML Output Structure Customization" in *Texinfo*).

See Section 19.2.1 [Customization of CSS Rules Imports and Selectors], page 79, for an explanation on getting information on CSS.

10.3 Getting Conversion Context

Some dynamically generated information should be used from the converter, in particular the expansion context (see Section 4.1 [Init File Expansion Contexts], page 8).

10.3.1 Conversion in String Context

Conversion and formatting functions should check if in string context to avoid using HTML elements in formatting when in string context. See Section 4.1 [Init File Expansion Contexts], page 8.

To determine if in string context, the functions is `in_string`:

$in_string = *$converter*`->in_string ()` [Function]
: Return true if in string context.

Example of use:

```
if ($converter->in_string()) {
  return "$mail_string ($text)";
} else {
  return $converter->html_attribute_class('a', [$cmdname])
                    ." href=\"mailto:$mail_string\">$text</a>";
}
```

10.3.2 Conversion in Preformatted Context

Conversion and formatting functions should test if in preformatted context to convert accordingly. See Section 4.1 [Init File Expansion Contexts], page 8.

To determine if in preformatted context, the functions is `in_preformatted_context`:

$in_preformatted = *$converter*`->in_preformatted_context ()` [Function]
: Return true if in preformatted context.

Another function tells if within a preformatted command:

$inside_preformatted = *$converter*`->inside_preformatted ()` [Function]
: Return true if within a preformatted block command such as `@preformatted`, `@format`.

  It is not exactly the same as preformatted context, for instance menu comments are in preformatted context even if not in a preformatted block command.

If in preformatted context, it is possible to get preformatted @-commands and preformatted types nesting with `preformatted_classes_stack`:

\@*preformatted_nesting* = *$converter*`->preformatted_classes_stack ()` [Function]
: Returns an reference on an array containing the block preformatted @-commands such as `@example`, `@display` or `@menu` names without the leading @ and the HTML attribute class preformatted container names, in order of appearance.

The `%Texinfo::Commands::preformatted_code_commands` hash can be used to determine if a preformatted command is to be formatted as code (see Section "Texinfo::Commands %preformatted_code_commands" in `texi2any_internals`).

```
my $pre_classes = $converter->preformatted_classes_stack();
foreach my $pre_class (@$pre_classes) {
  if ($Texinfo::Commands::preformatted_code_commands{$pre_class}) {
    $result = '<var>' .$result. '</var>';
    last;
  }
}
```

See Section 4.5 [Simple Customization of Containers], page 11, on customizing containers preformatted class.

10.3.3 Other Dynamic Information

To get the current output unit being converted, use `current_output_unit`:

$output_unit = *$converter*`->current_output_unit` () [Function]
: Return the output unit being converted, or `undef` if there is no output unit.

To get the file name of the current output unit being converted, use `current_filename`:

$filename = *$converter*`->current_filename` () [Function]
: Return the file name of the current output unit being converted.

To get the text filling and alignement context, determined by `@flushleft` or `@center`, use `in_align`:

$align_context = *$converter*`->in_align` () [Function]
: If the alignment context is the default alignement context, return `undef`. Otherwise, returns the command name of the alignment context.

To determine if the conversion is in a context converted multiple times, use `in_multiple_conversions`:

$multiple_conversion = *$converter*`->in_multiple_conversions` () [Function]
: Return true if the Texinfo tree being converted is converted multiple times and the current conversion is not the main conversion.

  For example, return true if a node name is converted as part of a direction string formating in a navigation panel, which is not the main expansion of the `@node`. The main `@node` element expansion occurs where the @-command is located.

To determine if the a mutiple expansion context is set, use `in_multi_expanded`:

$multi_expanded_context_information = *$converter*`->in_multi_expanded` () [Function]
: Return a string representing the multiple expanded context, or `undef` if not in a multiple expanded context.

  A multiple expanded context implies to be in multiple conversions. However, it is possible to be in multiple conversions without being in a multiple expanded context, as a multiple expanded context needs to be set explicitly, and is not always needed.

To get the current paragraph and preformatted number, use `paragraph_number` or `preformatted_number`:

$number = *$converter*->paragraph_number () [Function]
$number = *$converter*->preformatted_number () [Function]
Return the current paragraph or preformatted container number in the current formatting context.

To get the topmost block @-command being converted, use `top_block_command`:

$command_name = *$converter*->top_block_command () [Function]
Return the most recent block @-command seen in the current formatting context.

10.4 Converting Texinfo Trees

In some cases, it may be needed to convert a Texinfo tree rooted at any element. There is no reason to do that often, as the converter already goes through the tree calling functions to convert the elements, but it can be interesting in some cases.

This is, for example, often needed if a translated Texinfo tree is setup (see Section 15.1 [Internationalization of Strings Function], page 62). For example, here a Texinfo tree is returned by the `cdt` call, based on the translation of the '`No value for @strong{{item}}`' string, and converted to HTML with `convert_tree`:

```
my $tree = $converter->cdt('No value for @strong{{item}}',
                           {'item' => $text_element});
my $no_value_converted_output = $converter->convert_tree($tree);
```

10.4.1 Texinfo Tree Conversion Functions

The `convert_tree` function converts a Texinfo tree rooted at any element:

$converted_text = *$converter*->convert_tree (*%element*, *$explanation*) [Function]
%element is a Texinfo tree element. *$explanation* is optional, it is a string explaining why the function was called, to help in case of debugging. The function returns *%element* converted.

`convert_tree` is suitable when the conversion is in the flow of the Texinfo tree conversion. Sometime, it is better to ignore the formatting context of the main conversion, for example for the formatting of a caption, or the formatting of footnotes texts. Another special case is the case of tree elements being converted more than once, even if in the flow of the Texinfo tree conversion, for example if there are multiple `@insertcopying` in a document. A last special case arise, with formatting done in advance or out of the main conversion. This is the case, in practice, for sectioning commands or node commands which may be formatted as directions in navigation panels, menus or indices, may appear more than once in the document and be converted more than once, if language changes, for example.

For such cases, the function is `convert_tree_new_formatting_context` which sets the context appropriately. `convert_tree_new_formatting_context` ultimately calls `convert_tree`.

$converted_text = [Function]
$converter->convert_tree_new_formatting_context (*%element*, *$context*, *$multiple_pass*, *$global_context*, *$block_command_name*)

%element is a Texinfo tree element. *$context* describes the new context setup to format out of the main conversion flow. *$multiple_pass* is an optional string that marks that the conversion is done more than once. It should be unique and suitable for inclusion in targets and identifiers. *$global_context* is an optional string that marks that the formatting may be done in advance, and can be redone. *$block_command_name* is an optional block command name that is used to initialized the new context. It can be useful, in particular, to propagate the topmost block command in the new context.

The function returns *%element* converted, setting the conversion context according to the arguments.

See Section 10.4.2 [Setting the Context for Conversion], page 36, on how to set a specific context for a Texinfo tree conversion.

10.4.2 Setting the Context for Conversion

Special container types are recognized by the converter and can be used to convert a Texinfo tree in a specific context. Those types cannot appear in a regular Texinfo tree. In general they should be the type of tree root elements setup by the user.

The types are:

`_code` In this container, the conversion is done in a code context See Section 4.1 [Init File Expansion Contexts], page 8. For a container tree element.

`_converted`
The text of this text element is considered to be already formatted.

`_string` In this container, the conversion is done in a string context. See Section 4.1 [Init File Expansion Contexts], page 8. For a container tree element.

These contexts are typically used together with converter conversion functions (see Section 10.4.1 [Texinfo Tree Conversion Functions], page 35). For example:

```
my @contents = @{$element->{'contents'}};
push @contents, {'text' => ' <code>HTML</code> text ',
                 'type' => '_converted'};
my $result = $converter->convert_tree({'type' => '_code',
                                       'contents' => \@contents });
```

10.4.3 Conversion to Plain Text

The conversion to plain text can be achieved by using the `Texinfo::Text` converter `convert_to_text` function (see Section "Texinfo::Convert::Text" in `texi2any_internals`).

`convert_to_text` requires a conversion options argument to determine how the conversion to text should be done, specifying, for instance, the encoding or the formatting context. Such options are available in `$converter->{'convert_text_options'}`.

For example, to convert the Texinfo tree element *$element* to plain text:

```
my $plain_text = Texinfo::Convert::Text::convert_to_text($element,
                                    $converter->{'convert_text_options'});
```

Conversion to plain text is often used for strings that are to be formatted in code context. Code context can be set and reset by using `Texinfo::Convert::Text::set_options_code` and `Texinfo::Convert::Text::reset_options_code`:

```
Texinfo::Convert::Text::set_options_code(
                $converter->{'convert_text_options'});
my $code_string = Texinfo::Convert::Text::convert_to_text($element,
                                $converter->{'convert_text_options'});
Texinfo::Convert::Text::reset_options_code(
                $converter->{'convert_text_options'});
```

If encoded characters should be used irrespective of the specified document encoding, a possibility relevant, in general, for file names, `Texinfo::Convert::Text::set_options_encoding_if_not_ascii` should be called before the conversion and the original options should be reset afterwards by calling `Texinfo::Convert::Text::reset_options_encoding`:

```
Texinfo::Convert::Text::set_options_code(
                $converter->{'convert_text_options'});
Texinfo::Convert::Text::set_options_encoding_if_not_ascii($converter,
                                 $converter->{'convert_text_options'});
my $file_name = Texinfo::Convert::Text::convert_to_text($element,
                                $converter->{'convert_text_options'});
Texinfo::Convert::Text::reset_options_code(
                $converter->{'convert_text_options'});
Texinfo::Convert::Text::reset_options_encoding(
                $converter->{'convert_text_options'});
```

10.4.4 Texinfo Tree Elements in User Defined Functions

Many user defined functions used for formatting have Texinfo tree elements as arguments. The user defined code should never modify the tree elements. It is possible to reuse Texinfo tree elements information, but with a copy. For example, the following is correct:

```
my @contents = @{$element->{'contents'}};
push @contents, {'text' => ' my added text'};
my $result = $converter->convert_tree({'cmdname' => 'strong',
                                    'contents' => \@contents });
```

The following is incorrect:

```
push @{$element->{'contents'}}, {'text' => ' my added text'};
```

Nodes and sectioning elements hold information on the document structure (see Section "Texinfo::Structuring METHODS" in `texi2any_internals`). For example, the following keys of the `extra` sectioning elements hash can be interesting in several user-defined formatting and conversion functions:

`section_childs`
: For sectioning commands elements. The children of the sectioning element in the sectioning tree.

`section_level`
: The level of the section, taking into account `@raisesections` and `@lowersections`. Level 0 corresponds to `@top` or `@part` and level 1 to `@chapter` level sectioning commands. See Section "Raise/lower sections" in *Texinfo*.

Detailed information on the tree elements is available in the Texinfo Parser documentation, in particular a list of types and of information in the elements `extra` hash (see Section "Texinfo::Parser TEXINFO TREE" in `texi2any_internals`).

10.4.5 Output Units in User Defined Functions

Some information is available only in output units. Finding an output unit and using the information associated with the output unit hash reference keys may be needed in user-defined conversion functions.

Both normal and special output units (see Section 5.1 [Output Units], page 13) can be obtained as output units conversion function arguments (see Chapter 13 [Output Units Conversion Functions], page 58). The current output unit being processed is also available as `$converter->current_output_unit()` (see Section 10.3.3 [Other Dynamic Information], page 34). Root command (`@node` or sectioning command) `associated_unit` key value points to the associated output unit. Lastly `get_element_root_command_element` may be used to get the the output unit associated with an element (see [`get_element_root_command_element`], page 50).

The following keys of output unit hashes can be interesting:

`unit_type`
: `unit` for normal output units, `special_unit` for special units and `external_node_unit` for external nodes virtual units corresponding to references to external manuals.

`unit_command`
: For normal output units, points to the associated `@node` or sectioning @-command depending on which of nodes or sectioning commands are the main components of output units. See Section 5.1 [Output Units], page 13. The corresponding sectioning and `@node` @-command elements have an `associated_unit` key directly in their hash that points to the associated output unit.

  For special units, points to a "virtual" tree element with type `special_unit_element` associated with the special element, that does not appear in the Texinfo tree but can be used as a target for directions to the special unit. This element has an `associated_unit` key that points to the associated output unit.

  for references to external manuals virtual units, points to the tree element corresponding to the external manual and node label.

`unit_contents`
: Array reference on tree elements associated with the output unit.

`unit_filename`
: The associated file name.

`unit_directions`
: Hash with `next` and `prev` for the next and previous output units in document order.

`special_unit_variety`
: The variety of the special output unit. For special units only. See Table 16.1.

see Section "Texinfo::Structuring METHODS" in `texi2any_internals` for more on document structure information held by output units.

11 Customization and Use of Formatting Functions

Full customization of output is achieved with replacing default formatting functions with user defined functions. There are two broad classes of functions, the *conversion* functions used for output units and elements of the Texinfo tree, and other *formatting* functions with diverse purposes, including formatting that are not based on tree elements (for example beginning and end of file formatting). *Conversion* functions are described in the next chapters.

This chapter describes how *formatting* functions are registered and basic formatting functions that can be used in diverse situations. More specific formatting functions are described later on together with information on specific output customization.

Most formatting functions are specific, with specific arguments, and a specific item formatted. They can be called for HTML formatting and may also be customized.

User defined functions associated with the formatting of special output units body (see Section 5.1 [Output Units], page 13) can be considered as formatting functions, but are registered especially (see Section 16.5 [Special Unit Body Formatting Functions], page 71).

The formatting functions are often called from functions that can be replaced by user-defined functions, therefore these functions may not be called if the replacement functions do not keep a similar operation.

11.1 Registering Specific Formating Functions

User defined formatting functions are registered with `texinfo_register_formatting_function`:

`texinfo_register_formatting_function` (*$formatted*, *\&handler*) [Function]
: *$formatted* is a string describing the formatting function. *\&handler* is the user defined function reference.

To call a formatting function from user defined code, the function reference should first be retrieved using `formatting_function`:

\&formatting_function = *$converter*`->formatting_function` (*$formatted*) [Function]
: *$formatted* is a string describing the formatting function. Returns the associated formatting function reference.

It is possible to have access to the default formatting function reference. The function used is:

\&default_formatting_function = *$converter*`->default_formatting_function` (*$formatted*) [Function]
: *$formatted* is a string describing the formatting function. Returns the default formatting function reference.

The string that should be used to register or call each of the formatting functions and the call of the formatting functions are documented in the following sections of the manual, depending on where they are relevant.

11.2 Basic Formatting Customization

The following formatting functions references handle basic formatting and are called from diverse formatting and conversion functions. See Section 11.1 [Registering Specific Formating Functions], page 40, for information on how to register and get the functions references.

All the functions take a converter object as their first argument.

`format_comment`

`$text format_comment ($converter, $input_text)` [Function Reference]

Return *$input_text* in a comment.

See Section "Texinfo::Convert::Converter::xml_comment" in `texi2any_internals`.

`format_heading_text`

`$text format_heading_text ($converter, $command_name, \@classes, $input_text, $level, $id, \%element, $target)` [Function Reference]

Returns a heading formatted using *$input_text* as heading text, *$level* as heading level, *\@classes* for a class attribute. *$command_name* gives an information on the @-command the heading is associated with and can be `undef`, for instance for special output units headings.

$id* is an optional identifier, and *\%element* is an optional Texinfo tree element associated with the heading. *$target is the id of the element this heading is referring to.

In the default case, if the *$target* or *$id* are specified, a copiable anchor will be generated and injected into the heading. In the case both are specified, *$id* is preferred over *$target*, as it is closer to the element the user sees the anchor on.

This function reference can be called for `@node` and sectioning commands, heading commands, special output units and title @-commands.

A formatted headings is, in the default case, like `<h2>$input_text</h2>` for a *$level* 2 heading.

`format_program_string`

`$text format_program_string ($converter)` [Function Reference]

This function reference should return the formatted program string.

`format_protect_text`

`$text format_protect_text ($converter, $input_text)` [Function Reference]

Return *$input_text* with HTML reserved characters and form feeds protected.

For performance reasons, this function reference may not be called everywhere text is protected. For those cases, the calling function

should also be redefined to call `&{$converter->formatting_function('format_protect_text')}(...)` instead of another function[1].

See Section "Texinfo::Convert::Converter::xml_protect_text" in `texi2any_internals`.

`format_separate_anchor`
: This function reference is called if there is not another HTML element to add an identifier attribute to.

  $text `format_separate_anchor` (*$converter*, *$id*, *$class*) [Function Reference]
  : *id* is the identifier. *$class* is an optional class to be used in an HTML class attribute.

    Return an anchor with identifier *$id*.

For example, a separate anchor with an id built from a counter could be obtained with:

```
$counter++;
my $anchor_id = 'anchor_id_' . $counter;
my $anchor_with_counter
  = &{$converter->formatting_function('format_separate_anchor')}(
                                $converter, $anchor_id, 'myanchor_class');
```

The default function used for separate anchors can be replaced by a user-defined anchor formatting function using a `<p>` element with:

```
sub my_format_separate_anchor($$;$)
{
  my $converter = shift;
  my $id = shift;
  my $class = shift;

  return $converter->html_attribute_class('p', [$class])." id=\"$id\"></p>";
}

texinfo_register_formatting_function('format_separate_anchor',
                                     \&my_format_separate_anchor);
```

[1] The function called is actually the function referenced as `$converter->formatting_function('format_protect_text')` in the default case, but it is called directly to avoid an indirection

12 Tree Element Conversion Customization

Customization of tree elements associated with @-commands is done with different functions than those used for other tree elements, for instance containers with a type and tree elements holding text.

There are two main functions for each element command or type, one called when the element is first encountered, and the other called after formatting the contents of the element. The actual conversion is usually done after formatting the contents of the element, but it may sometime be necessary to have some code run when the element is first encountered.

For @-commands with both a command name and a type, the type is used as selector for the formating function for `def_line`, `definfoenclose_command` and `index_entry_command` types.

12.1 Command Tree Element Conversion

All the command elements can have a conversion function and an opening function that can be registered to be called by the converter. Some commands also require more specific information and functions for their formatting.

12.1.1 Command Tree Element Conversion Functions

User defined functions called for an @-command element conversion, after arguments and contents have been formatted, are registered with `texinfo_register_command_formatting`:

`texinfo_register_command_formatting` (*$command_name*, *\&handler*) [Function]

$command_name is an @-command name, without the leading @. *\&handler* is the user defined function reference.

The call of the user defined functions is:

$text `command_conversion` (*$converter*, *$command_name*, *\%element*, *\@args*, *$content*) [Function Reference]

$converter is a converter object. *$command_name* is the @-command name without the @. *\%element* is the Texinfo element.

\@args, if defined, is a reference on the formatted arguments of the @-command. Each of the array items correspond to each of the @-command argument. Each array item is either `undef` if the argument is empty, or a hash reference, with keys corresponding to possible argument formatting contexts:

`normal` Argument formatted in a normal context

`monospace`
: Argument formatted in a context where spaces are kept as is, as well as quotes and minus characters, for instance in '`--`' and '```'. Both in preformatted and code context. See Section 4.1 [Init File Expansion Contexts], page 8.

`monospacestring`
: Same as monospace, but in addition in string context. See Section 4.1 [Init File Expansion Contexts], page 8.

`monospacetext`
: Same as monospace, but in addition the argument is converted to plain text.

`filenametext`
: Same as monospacetext, but in addition the document encoding is used to convert accented letters and special insertion @-commands to plain text independently of customization variables.

`raw`
: Text is kept as is, special HTML characters are not protected. Appears only as `@inlineraw` second argument.

`string`
: In string context. See Section 4.1 [Init File Expansion Contexts], page 8.

`arg_tree`
: The Texinfo tree element corresponding to the argument. See Section 10.4.4 [Texinfo Tree Elements in User Defined Functions], page 37.

`url`
: Similar with filenametext. The difference is that UTF-8 encoding is always used for the conversion of accented and special insertion @-commands to plain text. This is best for percent encoding of URLs, which should always be produced from UTF-8 encoded strings.

The formatted arguments contexts depend on the @-command, there could be none, for `@footnote` argument which is not directly converted where the footnote command is, or multiple, for example for the fourth argument of `@image` which is both available as '`normal`' and '`string`'. See Table 12.1, for the converted arguments contexts. @-commands not specified in the table have their arguments in '`normal`' context.

For example, `$args->[0]->{'normal'}` is the first argument converted in normal context. It should be present for most @-commands with arguments, but not for all, for example `@anchor` argument is only available as `monospacestring`.

$content is the @-command formatted contents. It corresponds to the contents of block @-commands, and to Texinfo code following `@node`, sectioning commands, `@tab` and `@item` in `@enumerate` and `@itemize`. *$content* can be `undef` or the empty string.

The *$text* returned is the result of the @-command conversion.

`@abbr`	normal	normal, string			
`@acronym`	normal	normal, string			
`@anchor`	monospacestring				
`@email`	url, monospacestring	normal			
`@footnote`					
`@image`	monospacestring, filenametext, url	filenametext	filenametext	normal, string	filenametext
`@inforef`	monospace	normal	filenametext		
`@inlinefmt`	monospacetext	normal			
`@inlinefmtifelse`	monospacetext	normal	normal		
`@inlineifclear`	monospacetext	normal			
`@inlineifset`	monospacetext	normal			
`@inlineraw`	monospacetext	raw			
`@item`					
`@itemx`					
`@link`	monospace	normal	filenametext		
`@printindex`					
`@pxref`	monospace	normal	normal	filenametext	normal
`@ref`	monospace	normal	normal	filenametext	normal
`@sp`					
`@uref`	url, monospacestring	normal	normal		
`@url`	url, monospacestring	normal	normal		
`@value`	monospacestring				
`@xref`	monospace	normal	normal	filenametext	normal

Table 12.1: HTML command arguments formatting contexts in conversion function argument

To call a conversion function from user defined code, the function reference should first be retrieved using `command_conversion`:

`\&command_conversion = $converter->command_conversion ($command_name)` [Function]

$command_name is the @-command name without the @. Returns the conversion function reference for *$command_name*, or '`undef`' if there is none, which should only be the case for @-commands ignored in HTML not defined by the user.

for example, to call the conversion function for the `@tab` @-command, passing arguments that may correspond to another @-command:

```
&{$converter->command_conversion('tab')}($converter, $cmdname,
                                         $command, $args, $content);
```

It is possible to have access to the default conversion function reference. The function used is:

`\&default_command_conversion = $converter->default_command_conversion ($command_name)` [Function]

$command_name is the @-command name without the @. Returns the default conversion function reference for *$command_name*, or '`undef`' if there is none, which should only be the case for @-commands ignored in HTML.

12.1.2 Command Tree Element Opening Functions

User defined functions called when an @-command element is first encountered are registered with `texinfo_register_command_opening`. In general the possibility to call code at the @-command opening is not used much, as the HTML formatting is in general done when the content appearing in the comand is formatted. In the default conversion functions, this function is used for `@quotation`, to register prependended text to be output with the following inline container, usually a paragraph. This is described in detail with the inline containers formatting (see Section 12.2.4 [Inline Text Containers Formatting], page 55).

`texinfo_register_command_opening ($command_name, \&handler)` [Function]

$command_name is an @-command name, with the leading @. *\&handler* is the user defined function reference.

The call of the user defined functions is:

$text `command_open ($converter, $command_name, \%element)` [Function Reference]

$converter is a converter object. *$command_name* is the @-command name without the @. *\%element* is the Texinfo element.

The *$text* returned is prepended to the formatting of the @-command.

It is possible to have access to the default opening function reference. The function used is:

`\&default_command_open = $converter->default_command_open ($command_name)` [Function]

$command_name is the @-command name without the @. Returns the default opening function reference for *$command_name*, or '`undef`' if there is none.

12.1.3 Heading Commands Formatting

You can change the heading commands formatting by setting customization variables. In particular, you can change the navigation information output in headers associated with heading commands by selecting a different type of navigation (see Section "HTML Output Structure Customization" in *Texinfo*), by changing the links formatting (see Section "File Names and Links Customization for HTML" in *Texinfo*), the navigation panels formatting (see Section "Customization of Navigation and Headers" in *Texinfo*) and the heading levels (see Section "Specific Customization of HTML Formatting" in *Texinfo*).

`@node` and sectioning commands default conversion function call `format_heading_text` (see Section 11.2 [Basic Formatting Customization], page 41) and `format_element_header`

(see Section 17.3 [Element Header and Footer Formatting], page 75). The `@node` and sectioning elements are formatted like any other elements associated with @-commands. The corresponding function references can therefore be replaced by user defined functions for a precise control of conversion (See Section 12.1.1 [Command Tree Element Conversion Functions], page 43).

In the default formatting, when a sectioning command is encountered, a `<div>` element is opened for the extent of the sectioning command including its children sectioning commands. This extent need to be closed at different places, for instance when another sectioning command is reached, at the end of a file, or at the end of the document.

The user defined formatting function should take care of registering and closing opened section levels. In the default code, registering is done in the sectioning commands conversion function only.

The function for registering opened section extent is `register_opened_section_level`:

$converter*`->register_opened_section_level (`*$filename, $level, $closing_text`)` [Function]

$filename is the filename the section belongs to. You could use `$converter->current_filename()` for *$filename*. *$level* is the sectioning command level. It is typically obtained with `section->{'extra'}->{'section_level'}` (see Section 10.4.4 [Texinfo Tree Elements in User Defined Functions], page 37). *$closing_text* is the text that should be output when the section level *$level* is closed.

The function for closing registered section extents is `close_registered_sections_level`:

\@closing_texts `=` *$converter*`->close_registered_sections_level (`*$filename, $level*`)` [Function]

$filename is the filename the closed sections belong to. You could use `$converter->current_filename()` for *$filename*. *$level* is the sectioning command level. Opened section are closed down to section level *$level*. The closing texts are returned in the *\@closing_texts* array reference in order.

Example of use:

```
my $level = $opening_section->{'extra'}->{'section_level'};
my $closed_strings
   = $converter->close_registered_sections_level(
                    $converter->current_filename(), $level);
$result .= join('', @{$closed_strings});

# ....

$converter->register_opened_section_level(
                  $converter->current_filename(), $level, "</div>\n");
```

12.1.4 Target Tree Element Link

User-defined code in functions replacing default conversion functions for `@xref` and similar @-commands output, `@menu`, `@node`, sectioning commands, `@printindex` and

`@listoffloats` formatting requires links, labels and files information, mainly to output hyperlinks to *target commands*.

Target @-commands are @-commands that are associated with an identifier and can be linked to. They corresponds first to @-commands with unique identifier used as labels, `@node`, `@anchor` and `@float`. Sectioning commands, index entries, footnotes are also associated with targets. The "virtual" elements associated with special output units are also associated with targets leading to each special unit (see Section 10.4.5 [Output Units in User Defined Functions], page 38).

To get the unique Texinfo tree element corresponding to a label, use `label_command`:

`\%element = $converter->label_command ($label)` [Function]

Return the element in the tree that *$label* refers to.

For example:

```
my $node_element = $converter->label_command('Some-node_003f');
```

Labels are available for target @-commands and for @-command referring to target elements such as `@xref` or `@menu` entry node argument as the extra '`normalized`' element key.

For example, the first `@xref` argument '`normalized`' element key may be used to get the node or anchor element it points to:

```
my $arg_node = $xref_tree_element->{'args'}->[0];
if ($arg_node and $arg_node->{'extra'}
    and defined($arg_node->{'extra'}->{'normalized'})) {
  my $target_node
    = $converter->label_command($arg_node->{'extra'}->{'normalized'});
}
```

Tree elements not associated with a label are obtained each differently. For example, elements associated with index entries can be obtained using the Texinfo parsed document index entries with `$converter->get_info('document')->indices_information()` (see Section 10.2 [Converter General Information], page 30), or through sorted indices information (see Section 12.1.5 [Specific Formatting for Indices], page 50). Footnote elements can be obtained through the `@footnote` conversion function, and can also be passed to footnote formatting functions (see Section 16.2 [Customizing Footnotes], page 67). Floats elements in `@listoffloats` can be obtained from `$converter->get_info('document')->floats_information()` (see Section 10.2 [Converter General Information], page 30).

To get the identifier, file name and href of tree elements that may be used as link target, use `command_id`, `command_filename` and `command_href`:

$identifier = *$converter*`->command_id (`*\%target_element*`)` [Function]

Returns the id specific of the *\%target_element* tree element.

$file_name = *$converter*`->command_filename (`*\%target_element*`)` [Function]

Returns the file name of the *\%target_element* tree element.

$href = *$converter*->command_href (\%*target_element*, *$source_filename*, *$source_command*, *$specified_target*) [Function]

Return string for linking to \%*target_element* with `<a href>` or `undef` if not found or empty. *$source_filename* is the file the link comes from. If not set, the current file name is used. *$source_command* is an optional argument, the @-command the link comes from. It is only used for messages. *$specified_target* is an optional identifier that overrides the target identifier if set.

To get the text of tree elements that may be used as link description, use `command_text`:

$result = *$converter*->command_text (\%*target_element*, *$type*) [Function]

Return the information to be used for a hyperlink to \%*target_element*. The information returned depends on *$type*:

text Return text.

string Return text in string context. See Section 4.1 [Init File Expansion Contexts], page 8.

Using those functions, a target element hyperlink can be constructed as:

```
my $target_text = $converter->command_text($target_element);
my $target_href = $converter->command_href($target_element);
my $hyperlink = "<a href=\"$target_href\">$target_text</a>";
```

To get a Texinfo tree of elements that may be used as link description, use `command_tree`:

$result = *$converter*->command_tree (\%*target_element*, *$no_number*) [Function]

Return the a Texinfo tree to be used for a hyperlink to \%*target_element*. If *$no_number* is set, return a Texinfo elements tree representing text without a chapter number being included.

To obtain the top level command element associated with a footnote, float or index entry target element, use `command_root_element_command`:

\%*top_level_element* = *$converter*->command_root_element_command (\%*target_element*) [Function]

Return the top level element, either a `@node` or a sectioning element \%*target_element* is in.

This is used in indices formatting to link to index entry associated sectioning element in addition to linking to the index entry location. For example:

```
my $entry_root_link = '';
my $associated_command
 =  $converter->command_root_element_command($target_element);
if ($associated_command) {
```

```
  my $associated_command_href
    = $converter->command_href($associated_command);
  my $associated_command_text
    = $converter->command_text($associated_command);

  if (defined($associated_command_href)) {
    $entry_root_link
      = "<a href=\"$associated_command_href\">"
          ."$associated_command_text</a>";
} elsif (defined($associated_command_text)) {
  $entry_root_link = $associated_command_text;
}

my $formatted_entry = "<td><tr>$hyperlink</tr>"
                  ."<tr>$entry_root_link</tr></td>\n";
```

To get the node element associated with a target element, use `command_node`:

`\%node_element = $converter->command_node (\%target_element)` [Function]

Return the node element associated with *\%target_element*.

For elements that are not target elements, use `get_element_root_command_element` to get the root level Perl tree element and the output unit containing the element:

`\%top_level_element, \%output_unit = $converter->get_element_root_command_element (\%element)` [Function]

Return the top level element and output unit a Texinfo tree *\%element* is in. Both the top level element and the output unit may be undefined, depending on how the converter is called and on the Texinfo tree. The top level element returned is also determined by the customization variable `USE_NODES`. If `USE_NODES` is set the `@node` is preferred, otherwise the sectioning command is preferred.

For example to get the `@node` or sectioning root command tree element containing a `@printindex` element tree and the associated identifier for the formatting of the `@printindex` Texinfo tree element:

```
my ($output_unit, $root_command)
 = $converter->get_element_root_command_element($printindex_element);
my $index_element_id = $converter->command_id($root_command);
```

12.1.5 Specific Formatting for Indices

Index formatting customization is achieved through registering a conversion function for `@printindex` (see Section 12.1.1 [Command Tree Element Conversion Functions], page 43). The Texinfo parsed document index entries information directly obtained from the Texinfo manual parsing is available through `$converter->get_info('document')->indices_information()` (see Section 10.2 [Converter General Information], page 30). Sorted index entries, which are usually used for index formatting are available through `get_converter_indices_sorted_by_letter`:

`\%sorted_index_entries =` [Function]
`$converter->get_converter_indices_sorted_by_letter ()`

Returns index entries sorted by index and letter. This function should be called each time sorted indices are needed, in case the sorting depends on the `@documentlanguage` value. See Section "Texinfo::Convert::Converter::get_converter_indices_sorted_by_letter" in `texi2any_internals`.

12.1.6 Image Formatting

Image `@image` command formatting is customized by registering a conversion function for `@image` (see Section 12.1.1 [Command Tree Element Conversion Functions], page 43). To get the location of an image file, it could be useful to use `html_image_file_location_name`:

`($image_file, $image_basefile, $image_extension,` [Function]
`$image_path, $image_path_encoding) =`
`$converter->html_image_file_location_name ($command_name, \%element, \@args)`

$command_name, *\%element* and *\@args* should be the arguments of an `@image` @-command formatting (see Section 12.1.1 [Command Tree Element Conversion Functions], page 43).

The return values gives information on the image file if found, or fallback values. *$image_file* is the relative image file name. It is the file name used in formatting of the `@image` command in the default case. *$image_basefile* is the base file name of the image, without extension, corresponding to the `@image` @-command first argument. *$image_extension* is the image file extension (without a leading dot). *$image_path* is the path to the actual image file, `undef` if no file was found. *$image_path* is returned as a binary string, the other strings returned are character strings. *$image_path_encoding* is the encoding used to encode the image path to a binary string.

12.2 Type Tree Element Conversion

All the containers and text Texinfo tree elements not handled with command tree elements have a *type* associated. As for commands tree elements, they can have an opening function and a conversion function registered for a type and used. Some types may need more specific information too.

For tree elements that contain text, a `text` type is used to select the formatting functions, irrespective of the actual type of such a tree element. The `text` type does not exist in actual Texinfo tree elements.

12.2.1 Type Tree Element Conversion Functions

User defined functions called for the conversion of an element without @-command with text or a container type are registered with `texinfo_register_type_formatting`. For containers, the user defined function is called after conversion of the content.

`texinfo_register_type_formatting` (*$type, \&handler*) [Function]

$type is the element type. *\&handler* is the user defined function reference.

The call of the user defined functions is:

$text type_conversion (*$converter*, *$type*, *\%element*, *$content*) [Function Reference]

$converter is a converter object. *$type* is the element type. *\%element* is the Texinfo element. *$content* is text for elements associated with text, or the formatted contents for other elements. *$content* can be `undef` or the empty string.

The *$text* returned is the result of the element conversion.

To call a conversion function from user defined code, the function reference should first be retrieved using `type_conversion`:

\&type_conversion = *$converter*->type_conversion (*$type*) [Function]

$type is the element type. Returns the conversion function reference for *$type*, or '`undef`' if there is none, which should only be the case for types ignored in HTML not defined by the user.

It is possible to have access to the default conversion function reference. The function used is:

\&default_type_conversion = *$converter*->default_type_conversion (*$type*) [Function]

$type is the element type. Returns the default conversion function reference for *$type*, or '`undef`' if there is none, which should only be the case for types ignored in HTML.

Here is an example of paragraph formatting that prepends some HTML code to each paragraph and formats in code context (see Section 10.4.2 [Setting the Context for Conversion], page 36). It also shows how string context can be taken into account.

```
sub my_tree_element_convert_paragraph_type($$$$)
{
  my $converter = shift;
  my $type = shift;
  my $element = shift;
  my $content = shift;

  $content = '' if (!defined($content));

  if ($converter->in_string()) {
    return $content;
  }

  my @contents = {$element->{'contents'}};
  push @contents, {'text' => ' <code>HTML</code> text ',
                   'type' => '_converted'};
  my $result = $converter->convert_tree({'type' => '_code',
                                         'contents' => \@contents });
  return "<p>".$result."</p>";
}

texinfo_register_type_formatting('paragraph',
                          \&my_tree_element_convert_paragraph_type);
```

12.2.2 Type Tree Element Opening Functions

User defined functions called when an element without @-command with a container type is first encountered are registered with `texinfo_register_type_opening`:

`texinfo_register_type_opening` (*$type*, *\&handler*) [Function]
$type is the element type. *\&handler* is the user defined function reference.

The call of the user defined functions is:

$text `type_open` (*$converter*, *$type*, *\%element*) [Function Reference]
$converter is a converter object. *$type* is the element type. *\%element* is the Texinfo element.

The *$text* returned is prepended to the formatting of the type container.

It is possible to have access to the default opening function reference. The function used is:

\&default_type_open = *$converter*`->default_type_open` (*$type*) [Function]
$command_name is the element type. Returns the default opening function reference for *$type*, or '`undef`' if there is none.

In the default conversion functions, this function is not often used, conversion is in general done after the elements inside of the type container have been formatted. This function is defined for inline text container elements to get text to prepend to their content (see Section 12.2.4 [Inline Text Containers Formatting], page 55).

12.2.3 Text Tree Elements Conversion

Tree elements holding text are converted by the function reference registered for the `text` type conversion irrespective of the actual tree element type. For example, a tree element with type `spaces_before_paragraph` and text and a tree element without type but with text are both converted by the function reference registered for `text`.

The definition and registration of a conversion function for all the tree elements holding text should be along:

```
sub my_convert_text($$$)
{
  my $converter = shift;
  my $type = shift;
  my $element = shift;
  my $text = shift;

  # ...

  $text = uc($text) if ($converter->in_upper_case());

  # ...
}

texinfo_register_type_formatting ('text', \&my_convert_text);
```

The *$type* free conversion function argument is the actual type of the converted element (can be `undef`).

Formatting of text requires to use informative functions on specific contexts only relevant for text. User defined functions should convert the text according to the context.

Each context is associated with a function:

code

`$in_code = $converter->in_code ()` [Function]
Return true if in code context. See Section 4.1 [Init File Expansion Contexts], page 8.

math

`$in_math = $converter->in_math ()` [Function]
Return true if in math context. See Section 4.1 [Init File Expansion Contexts], page 8.

raw

`$in_raw = $converter->in_raw ()` [Function]
Return true if in raw format, in `@inlineraw` or in `@html`. In such a context, text should be kept as is and special HTML characters should not be protected.

verbatim

`$in_verbatim = $converter->in_verbatim ()` [Function]
Return true if in verbatim context, corresponding to `@verb` and `@verbatim`. In general, HTML characters should be protected in this context.

upper-case

`$in_upper_case = $converter->in_upper_case ()` [Function]
Return true if in upper-case context, corresponding to `@sc`.

non-breakable space

`$in_non_breakable_space =` [Function]
`$converter->in_non_breakable_space ()`
Return true if in context where line breaks are forbidden, corresponding to `@w`.

space protected

`$in_space_protected =` [Function]
`$converter->in_space_protected ()`
Return true if in context where space and newline characters are kept, corresponding to `@verb`.

12.2.4 Inline Text Containers Paragraph and Preformatted Formatting

Text is mainly output in two *inline* text containers, `paragraph` for text in paragraph and `preformatted` for text in preformatted environments. The Texinfo code parsing makes sure that it is the case, to simplify conversion to formats which allow text only in specific environments such as HTML.

Formatted text may also be prepared based on information from Texinfo elements tree while out of the inline containers. For example, `@quotation` argument should in general be prepended to the first paragraph in `@quotation`, caption number is also typically prependended to the caption. For that case, functions allow to register pending inline formatted content, and get the content to be prepended in inline text containers.

Pending formatted content text is registered with `register_pending_formatted_inline_content`:

$converter`->register_pending_formatted_inline_content` [Function]
(*$category*, *$content*)

$content is the formatted content to be registered and output in the next inline container. *$category* is a indicator of the source of the formatted inline content, mostly used to cancel registered content if no inline container was seen.

For example

```
my $quotation_arg_to_prepend
  = $converter->convert_text($quotation_arg_element);
$converter->register_pending_formatted_inline_content('quotation',
                                    $formatted_quotation_arg_to_prepend);
```

Pending formatted content can (and should) be cancelled when it is known that there is no suitable inline container to be used to output the text. The function is `cancel_pending_formatted_inline_content`:

$cancelled_content = [Function]
$converter`->cancel_pending_formatted_inline_content`
(*$category*)

Cancel the first *$category* pending formatted content text found. Returns `undef` if nothing was cancelled, and the cancelled content otherwise.

Pending formatted content is gathered by calling `get_pending_formatted_inline_content`. In the default case, this is done in inline containers opening code (see Section 12.2.2 [Type Tree Element Opening Functions], page 53).

$content = [Function]
$converter`->get_pending_formatted_inline_content` ()

Returns the concatenated pending content.

The inline containers get the content when they are opened, but are converted after the formatting of their contents. Two additional functions allow to associate pending content to an element, `associate_pending_formatted_inline_content`, and get the associated content, `get_associated_formatted_inline_content`. `associate_pending_formatted_inline_content` is normally called in inline container opening code, right

after `get_pending_formatted_inline_content`, while `get_associated_formatted_inline_content` is called in the inline container conversion function (see Section 12.2.1 [Type Tree Element Conversion Functions], page 51).

$converter`->associate_pending_formatted_inline_content` [Function]
(*%element*, *$content*)
Associate *$content* to the Texinfo tree element *%element*.

$content = [Function]
$converter`->get_associated_formatted_inline_content`
(*%element*)
Get *$content* associated with the Texinfo tree element *%element*.

Here is some inline container formatting code showing how those functions are used, with the paragraph type element formatting example completed:

```
sub _open_inline_container_type($$$)
{
  my $self = shift;
  my $type = shift;
  my $element = shift;

  my $pending_formatted = $self->get_pending_formatted_inline_content();

  if (defined($pending_formatted)) {
    $self->associate_pending_formatted_inline_content($element,
                                                      $pending_formatted);
  }
  return '';
}

sub my_final_convert_paragraph_type($$$$)
{
  my $converter = shift;
  my $type = shift;
  my $element = shift;
  my $content = shift;

  $content = '' if (!defined($content));

  my $prepended
    = $converter->get_associated_formatted_inline_content($element);
  if ($converter->in_string()) {
    return $prepended.$content;
  }
```

```
  my @contents = {$element->{'contents'}};
  push @contents, {'text' => ' <code>HTML</code> text ',
                   'type' => '_converted'};
  my $result = $converter->convert_tree({'type' => '_code',
                                         'contents' => \@contents });
  return "<p>".$prepended.$result."</p>";
}

texinfo_register_type_formatting('paragraph',
                                 \&my_final_convert_paragraph_type);
```

13 Output Units Conversion Functions

Output units formatting function are setup and used similarly as for tree container types (see Section 12.2.1 [Type Tree Element Conversion Functions], page 51). The output unit types correspond to the `unit_type` key values of output unit hashes (see [Unit Type], page 38).

User defined functions called for the conversion of an output unit are registered with `texinfo_register_output_unit_formatting`. The user defined function is called after conversion of the content.

`texinfo_register_output_unit_formatting` (*$unit_type*, *\&handler*) [Function]

$unit_type is the output unit type. *\&handler* is the user defined function reference.

The call of the user defined functions is:

$text `output_unit_conversion` (*$converter*, *$unit_type*, *\%output_unit*, *$content*) [Function Reference]

$converter is a converter object. *$unit_type* is the output unit type. *\%output_unit* is the output unit. *$content* the formatted contents. *$content* can be `undef` or the empty string.

The *$text* returned is the result of the output unit conversion.

To call a conversion function from user defined code, the function reference should first be retrieved using `type_conversion`:

\&output_unit_conversion = *$converter*`->output_unit_conversion` (*$unit_type*) [Function]

$unit_type is the output unit type. Returns the conversion function reference for *$unit_type*, or '`undef`' if there is none.

It is possible to have access to the default conversion function reference. The function used is:

\&default_output_unit_conversion = *$converter*`->default_output_unit_conversion` (*$unit_type*) [Function]

$unit_type is the output unit type. Returns the default conversion function reference for *$unit_type*, or '`undef`' if there is none.

Nomal output units with output unit type `unit` default conversion involves calling the formatting reference `format_element_footer` (see Section 17.3 [Element Header and Footer Formatting], page 75).

Special units conversion is achieved by calling `special_unit_body_formatting` (see Section 16.5 [Special Unit Body Formatting Functions], page 71), `format_navigation_header` (see Section 17.2 [Navigation Panel and Navigation Header Formatting], page 74), `format_heading_text` (see Section 11.2 [Basic Formatting Customization], page 41) and `format_element_footer` (see Section 17.3 [Element Header and Footer Formatting], page 75). Special units type is `special_unit`.

14 Shared Conversion State

For advanced customization, it is often necessary to pass information during conversion between different formatting functions or between different calls of the same function. An interface is provided for information shared among formatting functions, called *shared conversion state*. Each data piece in the *shared conversion state* is associated with an @-command name, has a name, and a list of selectors.

This interface is often useful for the formatting of paragraph and preformatted containers and @-commands such as `@abbr`, `@footnote`, `@node`, sectioning commands, `@quotation` and `@float`.

It is required to use that interface when sharing information with the default formatting functions. Every type of function can use shared state, formatting functions (see Chapter 11 [Customization and Use of Formatting Functions], page 40), tree elements (see Chapter 12 [Tree Element Conversion Customization], page 43) and output units conversion functions (see Chapter 13 [Output Units Conversion Functions], page 58).

14.1 Define, Get and Set Shared Conversion State

Four types for selectors and value are currently considered:

`string` A string.

`integer` An integer

`element` A Texinfo tree element.

`index_entry`.
: An index entry reference as appearing in index data structures. See Section "Texinfo::Document index_entries" in `texi2any_internals`.

New shared infomation is defined with `define_shared_conversion_state`:

$converter*`->define_shared_conversion_state (`*$cmdname, $name, \@specification`)` [Function]
: *$cmdname* is an @-command name, without leading @. *name* is the name associated with the data. The `top` command name is conventionally used if there is no natural association with another @-command. *\@specification* array reference specifies the types of the selectors and the type of the value as strings. The last string of the array specifies the type of the value. The preceding strings specify the number and types of selectors[1].

  For example, `['integer', 'element', 'string']` specifies a '`string`' type for the value, and two selectors, the first with '`integer`' type, and the second with '`element`' type. `['integer']` specifies an integer for the value and no selector.

For example, the following defines a '`color`' shared conversion state formally associated with `@quotation`, with an integer value and a string selector.

```
$converter->define_shared_conversion_state ('quotation', 'color',
```

[1] The number of strings in the specification is the only information actually used, to determine the number of selectors. However, it is advised to use the specified types for consistency and compatibility with future changes.

```
                                                ['string', 'integer']);
```

The association with an @-command is provided for a semantic classification of shared conversion information, but has no practical effect. In particular, nothing prevents using shared conversion state information associated with an @-command in the formatting of another @-command.

The function `get_shared_conversion_state` is used to get information:

$value = *$converter*`->get_shared_conversion_state` (*$cmdname*, *$name*, [*$selector ...*]) [Function]

Return the reference *$value* associated with *$cmdname* and *$name*. The number of selectors given in argument should match the number of selectors in the definition (possibly none).

For example, continuing with the '`color`' shared information data defined above, with one selector variable:

```
my $color_number
   = $converter->get_shared_conversion_state('quotation',
                                             'color', 'purple1');
```

The function `set_shared_conversion_state` is used to set shared information:

$converter*`->define_shared_conversion_state` (*$cmdname, *$name*, [*$selector ...*], *$value*) [Function]

Sets *$value* associated with *$cmdname* and *$name*. The number of selectors given in argument should match the number of selectors in the definition (possible none).

For example:

```
$converter->set_shared_conversion_state('quotation', 'color',
                                        'special_black', 42);
```

The converter is used to hold the information, but does not use nor write.

14.2 Shared Conversion State in Default Formatting

The following shared conversion state information is defined in the default formatting functions:

Command	Name	Selectors	Value
`abbr`	`explained_commands`	string (first argument)	string
`acronym`	`explained_commands`	string (first argument)	string
`footnote`	`footnote_number`		integer
`footnote`	`footnote_id_numbers`	string (footnote id)	integer
`listoffloats`	`formatted_listoffloats`	string (normalized argument)	integer
`menu`	`html_menu_entry_index`		integer
`printindex`	`formatted_index_entries`	index_entry (index entry hash)	integer
`top`	`in_skipped_node_top`		integer
`nodedescription`	`formatted_nodedescriptions`	element (`@nodedescription` tree element)	integer

These shared information data correspond to:

`explained_commands`
: Associate the explanation given in a previous `@abbr` or `@acronym` second argument to first `@abbr` or `@acronym` arguments.

`footnote_number`
: The current number of footnotes formatted in document.

`footnote_id_numbers`
: Associate a footnote identifier, typically used in hyperlink reference, to the number of time the corresponding footnote was formatted. More than one corresponds to very rare cases, for instance a footnote in `@copying` and multiple `@insertcopying`.

`formatted_listoffloats`
: Associate a list of float type to the number of time it was formatted.

`html_menu_entry_index`
: Current number of menu entries in a menu. Reset to 0 at `@menu` beginning.

`formatted_index_entries`
: Associate an index entry to the number of time it was formatted.

`in_skipped_node_top`
: Set to 1 in a `Top` node being skipped, in case `NO_TOP_NODE_OUTPUT` is set.

`formatted_nodedescriptions`
: Associate a `@nodedescription` tree element to the number of time it was formatted.

15 Translations in Output and Customization

Translated strings can be specified in customization functions, for @-commands without arguments (see Section 4.2 [Simple Customization for Commands Without Arguments], page 8), for direction strings (see Section 5.3 [Direction Strings Customization], page 16) and for specific elements headings such as footnotes, contents or about (see Section 16.1 [Special Units Information Customization], page 66). Translated strings can also be inserted in the output in user-defined customization functions, by using specific functions for internationalization of strings, `cdt`, `cdt_string` or `pcdt` (see Section "Texinfo::Convert::Converter Translations in output documents" in `texi2any_internals`).

It is possible to customize the translated strings, in order to change the translations of the strings translated in the default case. If new translated strings are added, it is even required to use translated strings customization to add translations for the added strings.

See Section "Internationalization of Document Strings" in *Texinfo* for additional information on the default case.

15.1 Internationalization of Strings Function

The subroutines `cdt`, `cdt_string` or `pcdt`, are used for translated strings:

$translated_tree = *$converter*->cdt (*$string*, *\%variables_hash*, *$translation_context*) [Function]

$translated_string = *$converter*->cdt_string (*$string*, *\%variables_hash*, *$translation_context*) [Function]

$translated_tree = *$converter*->pcdt (*$translation_context*, *$string*, *\%variables_hash*) [Function]

$string* is the string to be translated, *\%variables_hash* is a hash reference holding the variable parts of the translated string. *$translation_context is an optional translation context that limits the search of the translated string to that context (see Section "Contexts" in *GNU gettext tools*).

The result returned is a Perl Texinfo tree for `cdt` and `pcdt` and a string for `cdt_string`. With `cdt_string` the substitutions may only be strings.

If called as `pcdt`, *$translation_context* is not optional and is the first argument.

With `cdt` and `pcdt`, when the string is expanded as Texinfo, and converted to a Texinfo tree in Perl, the arguments are substituted; for example, '`{arg_name}`' is replaced by the corresponding actual argument, which should be a Texinfo tree element. With `cdt_string`, the string should already be converted, the arguments are substituted as strings; for example '`{arg_name}`' is replaced by the corresponding actual argument, which should be a string.

In the following example, '`{date}`', '`{program_homepage}`' and '`{program}`' are the arguments of the string. Since they are used in `@uref`, their order in the formatted output depends on the formatting and is not predictable. '`{date}`', '`{program_homepage}`' and '`{program}`' are substituted after the expansion, which means that they should already be Texinfo tree elements.

```
$converter->cdt('Generated @emph{@today{}} using '
 .'@uref{{homepage}, @emph{{program}}}.',
```

```
        { 'homepage' => { 'text' => $converter->get_conf('PACKAGE_URL') },
          'program' => { 'text' => $converter->get_conf('PROGRAM') }});
```

An example of combining conversion with translation:

```
$converter->convert_tree($converter->cdt(
       '{explained_string} ({explanation})',
       {'explained_string' => {'type' => '_converted',
                               'text' => $result},
        'explanation' => {'type' => '_converted',
                          'text' => $explanation_result}}),
                         "convert explained $cmdname");
```

In the default case, the functions from the `Texinfo::Translations` module are used for translated strings through converter functions. It is possible to use user-defined functions instead as seen next. See Section "Texinfo::Translations" in `texi2any_internals` for more on `Texinfo::Translations`.

In `texi2any` code, `cdt` and `cdt_string` are also used to mark translated strings for tools extracting translatable strings to produce template files. `pcdt` is used to mark translated string with a translation context associated.

15.2 Translated Strings Customization

To customize strings translations, register the `format_translate_message` function reference:

$translated_string `format_translate_message` [Function Reference]
(*$converter*, *$string*, *$lang*, *$translation_context*)

$string is the string to be translated, *$lang* is the language. *$translation_context* is an optional translation context.

The result returned should be the translated string. The result returned may also be '`undef`', in which case the translation is done as if the function reference had not been defined.

See Section 15.1 [Internationalization of Strings Function], page 62, for more information on strings translations function arguments.

This function reference is not set in the default case, in the default case `translate_string` from the `Texinfo::Translations` module is called (see Section 15.1 [Internationalization of Strings Function], page 62). See Section 11.1 [Registering Specific Formating Functions], page 40, for information on how to register and get the function reference.

Here is an example with new translated strings added and definition of `format_translate_message` to translate the strings:

```
texinfo_register_no_arg_command_formatting('error', undef, undef,
                                                  undef, 'error--&gt;');
my %translations = (
 'fr' => {
           'error--&gt;' => {'' => 'erreur--&gt;',},
           # ...
         },
```

```
 'de' => {
           'error--&gt;' => {'' => 'Fehler--&gt;',},
           # ...
         }
 # ...
);

sub my_format_translate_message($$$;$)
{
  my ($self, $string, $lang, $translation_context) = @_;
  $translation_context = '' if (!defined($translation_context));
  if (exists($translations{$lang})
      and exists($translations{$lang}->{$string})
      and exists($translations{$lang}->{$string}
                                  ->{$translation_context})) {
    my $translation = $translations{$lang}->{$string}
                                      ->{$translation_context};
    return $translation;
  }
  return undef;
}

texinfo_register_formatting_function('format_translate_message',
                                  \&my_format_translate_message);
```

15.3 Translation Contexts

Translation contexts may be set to avoid ambiguities for translated strings, in particular when the strings are short (see Section "Contexts" in *GNU* `gettext` *utilities*). Translation contexts are set for translated direction strings (see Section 5.2.2 [Direction Strings], page 16) and for special output units headings (see Section 16.1 [Special Units Information Customization], page 66).

For direction strings, the translation context is based on the direction name (see Section 5.2 [Directions], page 14), with '`direction`' prepended and another string prepended, depending on the type of string:

`button` '`button label`' is prepended

`description`
: '`description`' is prepended

`text` '`string`' is prepended

For example, the `Top` direction `button` direction string translation context is '`Top direction button label`'.

As an exception, the `This` direction has '`(current section)`' prepended to have a more explicit translation context. The `This` direction `text` direction string translation context is thus '`This (current section) direction string`'.

For special output unit headings, the translation context is obtained by prepending '`section heading`' to the special output unit variety (see Table 16.1). For example, the `footnotes` special output unit variety heading translation context is '`footnotes section heading`'.

Here is an example, which could be used with a similar function registered as in the example above (see [New translated strings example], page 63):

```
texinfo_register_direction_string_info('Forward', 'text', undef,
                                       'Forward');
texinfo_register_special_unit_info('heading', 'contents',
                                'The @emph{Table of Contents}');

my %translations = (
  'fr' => {
         'The @emph{Table of Contents}' => {'contents section heading'
                      => '@result{} La @emph{Table des mati@`eres}',},
         'Forward' => {'Forward direction string'
                      => 'Vers l\'avant @result{}',},
          }
   ...
);
```

Other translated strings may also be associated with translation contexts. The translation template file `po_document/texinfo_document.pot` in the source directory of Texinfo contains the translated strings appearing in all the output formats.

16 Customizing Footnotes, Tables of Contents and About

Some customization is specific for different special output units content formatting, especially when the formatting is not done in a separate output unit (see Section 5.1 [Output Units], page 13), but some customization is relevant for all the special units. The formatting of special units bodies is handled the same for all the special units, when formatted as separate units.

To specify a special unit in those contexts, the special units varieties are used, as described in Table 16.1.

Special Unit	**Special Unit Variety**
Table of contents	`contents`
Short table of contents	`shortcontents`
Footnotes	`footnotes`
About	`about`

Table 16.1: Association of special elements names with their special element variety

The variety of special elements is the special unit hash value associated with the `special_unit_variety` key.

To get information on the special output unit variety associated with an @-command command name, use `command_name_special_unit_information`:

```
($special_unit_variety, \%output_unit, $class_base,          [Function]
        $output_unit_direction) =
        $converter->command_name_special_unit_information
        ($command_name)
```

$command_name is an @-command name without the leading @. If the *$command_name* is not associated with a special output unit, returns `undef`. Otherwise, return the *$special_unit_variety* (see Table 16.1), the *\%output_unit* output unit, a *$class_base* string for HTML class attribute and the *$output_unit_direction* direction corresponding to that special elements (see Section 5.2 [Directions], page 14).

In the current setup, special output units are associated with `@contents`, `@shortcontents` and `@summarycontents` and with `@footnote`.

16.1 Special Units Information Customization

To customize special output units formatting, a simple possibility is to change the information associated with the special output units.

The following items common to all the special units may be customized:

`class` String for special element HTML class attributes.

`direction`
: Direction corresponding to the special element. See Section 5.2 [Directions], page 14.

`heading` Special element heading Texinfo code.

`heading_tree`
: Special element heading Texinfo tree.

`order`
: Index determining the sorting order of special elements.

`file_string`
: File string portion prepended to the special element file names, such as '`_toc`'.

`target`
: A string representing the target of the special element, typically used as id attribute and in href attribute.

The heading string is set with `heading`, and should be a Texinfo code string. `heading_tree` cannot be set directly, but can be retrieved. It is determined from `heading` after translation and conversion to a Texinfo tree.

To set the information, use `texinfo_register_special_unit_info` in an init file:

`texinfo_register_special_unit_info` (*$item_type, $special_unit_variety, $value*) [Function]

Set *$item_type* information for the special unit variety *$special_unit_variety* to *$value*. *$value* may be '`undef`', or an empty string, but only `heading` and `target` should be set to that value as a non-empty value is needed for the other items for formatting.

To get the list of varieties, use `get_special_unit_info_varieties`:

$list = *$converter*`->get_special_unit_info_varieties` (*$item_type*) [Function]

$item_type is the type of information to be retrieved as described above. The list of the special units varieties with information for the *$item_type* is returned.

To retrieve the information for formatting, use `special_unit_info`:

$list_or_value = *$converter*`->special_unit_info` (*$item_type, $special_unit_variety*) [Function]

$item_type is the type of information to be retrieved as described above. *$special_unit_variety* is a special unit variety, the corresponding value is returned.

The value returned is translated and converted to a Texinfo tree for '`heading_tree`'.

16.2 Customizing Footnotes

In the default case footnotes are numbered. If `NUMBER_FOOTNOTES` is set to 0, a '`*`' is used instead, or the `NO_NUMBER_FOOTNOTE_SYMBOL` customization variable value, if set.

Redefinition of `@footnote` conversion reference and footnote formatting references is needed for further customization.

`@footnote` @-commands appearing in the Texinfo elements tree are converted like any other elements associated with @-commands (see Section 12.1.1 [Command Tree Element Conversion Functions], page 43). It is therefore possible to redefine their formatting by registering a user defined function.

To pass information on footnotes between the conversion function processing the `@footnote` command at the location they appear in the document and the functions formatting their argument elsewhere, two functions are available: `register_footnote` to be called where they appear in the document, and `get_pending_footnotes` to be called where they are formatted.

$converter->register_footnote (*%element*, *$footnote_id*, *$foot_in_doc_id*, *$number_in_doc*, *$footnote_location_filename*, *$multi_expanded_region*) [Function]

%element is the footnote Texinfo tree element. *$footnote_id* is the identifier for the location where the footnote arguments are expanded. *$foot_in_doc_id* is the identifier for the location where the footnote appears in the document. *$number_in_doc* is the number of the footnote in the document. *$footnote_location_filename* is the filename of the output unit of the footnote in the document. If the footnote appears in a region that is expanded multiple times, the information on the expansion is *$multi_expanded_region* (see Section 10.3.3 [Other Dynamic Information], page 34).

`register_footnote` is normally called in the `@footnote` @-command conversion function reference. The default conversion function also call `command_href` to link to the location where the footnote text will be expanded (see Section 12.1.4 [Target Tree Element Link], page 47).

\@pending_footnotes_information = *$converter*->get_pending_footnotes () [Function]

Returns in *\@pending_footnotes_information* the information gathered in `register_footnote`. Each of the array reference element in *\@pending_footnotes_information* is an array reference containing the arguments of `register_footnote` in the same order.

The formatting of footnotes content is done by the `format_footnotes_sequence` formatting reference (see Section 11.1 [Registering Specific Formating Functions], page 40):

$footnotes_sequence format_footnotes_sequence (*$converter*) [Function Reference]

Formats and returns the footnotes that need to be formatted. This function normally calls `get_pending_footnotes`. The default function also calls `footnote_location_href` to link to the location in the document where the footnote appeared, and the `format_single_footnote` formatting function to format a single footnote.

The formatting of one footnote is done by the `format_single_footnote` formatting reference:

$footnote format_single_footnote (*$converter*, *\%element*, *$footnote_id*, *$number_in_doc*, *$footnote_location_href*, *$footnote_mark*) [Function Reference]

Formats and returns a single footnote. *\%element* is the footnote Texinfo tree element. *$footnote_id* is the identifier for the location where the footnote arguments are expanded. *$number_in_doc* is the number of the footnote in the document. *$footnote_location_href* is the href that links to the footnote location in the main document. *$footnote_mark* is the footnote number or mark.

If footnotes are in a separate output unit (see Section 5.1 [Output Units], page 13), the default footnote special output unit body formatting function calls `format_footnotes_sequence` (see Section 16.5 [Special Unit Body Formatting Functions], page 71).

If the footnotes are not in a separate output unit, or there is no separate unit because there is only one output unit or no output unit, the `format_footnotes_segment` formatting reference is called when pending footnotes need to be formatted. This function reference can be replaced by a user defined function.

$footnotes_segment `format_footnotes_segment` (*$converter*) [Function Reference]

Returns the footnotes formatted. In the default case, the function reference calls `format_footnotes_sequence` and also sets up a header with `format_heading_text` (see Section 11.2 [Basic Formatting Customization], page 41), using the customization variables `FOOTNOTE_END_HEADER_LEVEL` and the special `footnotes` element `heading` information (see Section 16.1 [Special Units Information Customization], page 66).

To get the id of a footnote in the main document, use `footnote_location_target`:

$target = *$converter*`->footnote_location_target` (*\%footnote_element*) [Function]

Return the id for the location of the footnote *\%footnote_element* in the main document (where the footnote number or symbol appears).

To get an href to link to a footnote location in the main document, use `footnote_location_href`:

$href = *$converter*`->footnote_location_href` (*\%footnote_element*, *$source_filename*, *$specified_target*, *$target_filename*) [Function]

Return string for linking to *\%footnote_element* location in the main document with `<a href>`. *$source_filename* is the file the link comes from. If not set, the current file name is used. *$specified_target* is an optional identifier that overrides the target identifier if set. *$target_filename* is an optional file name that overrides the file name href part if set.

See Section 12.1.4 [Target Tree Element Link], page 47, to get link information for the location where footnote text is output.

16.3 Contents and Short Table of Contents Customization

You can set the customization variable `CONTENTS_OUTPUT_LOCATION` to determine where the table of contents and short table of content are output in the document (see Section "HTML Output Structure Customization" in *Texinfo*):

'`after_top`'
: The tables of contents are output at the end of the `@top` section, to have the main location for navigation in the whole document early on. This is in line with `FORMAT_MENU` set to '`sectiontoc`' with sectioning command being used in HTML for navigation rather than menus. This is the default.

'`inline`'
: The tables of content are output where the corresponding @-command, for example `@contents`, is set. This behavior is consistent with `texi2dvi`.

'`separate_element`'
: The tables of contents are output in separate output units, either at the end of the document if the output is unsplit or in separate files if not. This makes sense when menus are used for navigation with `FORMAT_MENU` set to '`menu`'.

'`after_title`'
: The tables of contents are merged into the title material, which in turn is not output by default; see Section 19.1 [HTML Title Page Customization], page 79.

You can set other customization variables to modify table of contents links formatting (see Section "File Names and Links Customization for HTML" in *Texinfo*) and change the HTML code inserted before and after the tables of contents (see Section "Customization of HTML Code Inserted" in *Texinfo*).

Finally, the following function reference provides even more control over the table of contents and short table of contents formatting reference:

$toc_result `format_contents` (*$converter, $command_name, \%element, $filename*) [Function Reference]
: *$command_name* is the @-command name without leading @, should be '`contents`', '`shortcontents`' or '`summarycontents`'. *\%element* is optional. It corresponds to the *$command_name* Texinfo tree element, but it is only set if `format_contents` is called from a Texinfo tree element conversion, and not as a special element body formatting. *$filename* is optional and should correspond to the filename where the formatting happens, for links.

  In the default function, structuring information is used to format the table of contents (see Section 10.2 [Converter General Information], page 30), and `command_contents_href` and `command_href` (see Section 12.1.4 [Target Tree Element Link], page 47) are used for links. If *$filename* is unset, the current file name is used, using `$converter->current_filename()`.

  Return formatted table of contents or short table of contents.

If contents are in a separate output unit (see Section 5.1 [Output Units], page 13), the default contents and shortcontents special element body formatting function calls `format_contents` (see Section 16.5 [Special Unit Body Formatting Functions], page 71). Otherwise, `format_contents` is called in the conversion of heading @-command, in title page formatting, and in `@contents` conversion function, depending on the `CONTENTS_OUTPUT_LOCATION` value.

To get id and link href of sectioning commands in table of contents and short table of contents, use `command_contents_target` and `command_contents_href`:

$target = *$converter*`->command_contents_target` (*\%sectioning_element, $contents_or_shortcontents*) [Function]
: Returns the id for the location of *\%sectioning_element* sectioning element in the table of contents, if *$contents_or_shortcontents* is '`contents`', or in the short table of contents, if *$contents_or_shortcontents* is set to '`shortcontents`' or '`summarycontents`'.

$href = *$converter*->command_contents_href (*%sectioning_element*, *$contents_or_shortcontents*, *$source_filename*) [Function]

Return string for linking to the *\%sectioning_element* sectioning element location in the table of contents, if *$contents_or_shortcontents* is '`contents`' or in the short table of contents, if *$contents_or_shortcontents* is set to '`shortcontents`' or '`summarycontents`'. *$source_filename* is the file the link comes from. If not set, the current file name is used. Returns `undef` if no string is found or the string is empty.

16.4 About Special Output Unit Customization

The default About output unit has an explanation of the buttons used in the document, controlled by `SECTION_BUTTONS`. The formatting of this is influenced by the `text`, `description` and `example` direction strings (see Section 5.2.2 [Direction Strings], page 16) and by `ACTIVE_ICONS` (see Section 5.4 [Simple Navigation Panel Customization], page 17).

`PROGRAM_NAME_IN_ABOUT` can also be used to change the beginning of the About output unit formatting.

If the above is not enough and you want to control exactly the formatting of the about unit, the `about` special output unit body reference function may be overridden (see Section 16.5 [Special Unit Body Formatting Functions], page 71).

16.5 Special Unit Body Formatting Functions

In addition to the formatting possibilities available with the default special output units formatting presented previously, it is also possible to control completely how a separate special output unit is formatted.

To register body formating user defined functions for special output units (see Section 5.1 [Output Units], page 13), the special output units varieties are used, as described in Table 16.1. Special element body formatting user defined functions are registered with `texinfo_register_formatting_special_unit_body`:

`texinfo_register_formatting_special_unit_body` (*$special_unit_variety*, *\&handler*) [Function]

$special_unit_variety is the element variety (see Table 16.1). *\&handler* is the user defined function reference.

The call of the user defined functions is:

$text special_unit_body (*$converter*, *$special_unit_variety*, *\%special_unit*) [Function Reference]

$converter is a converter object. *$special_unit_variety* is the unit variety. *\%special_unit* is the special output unit.

The *$text* returned is the formatted special output unit body.

To call a special output unit body formatting function from user defined code, the function reference should first be retrieved using `special_unit_body_formatting`:

`\&special_unit_body_formatting =` [Function]
`$converter->special_unit_body_formatting ($special_unit_variety)`

$special_unit_variety is the special output unit variety. Returns the conversion function reference for *$variety*, or '`undef`' if there is none, which should not happen for the special output units described in this manual.

For example:

```
my $footnotes_element_body
 = &{$converter->special_unit_body_formatting('footnotes')}(
                                    $converter, 'footnotes', $element);
```

It is possible to have access to the default conversion function reference. The function used is:

`\&default_special_unit_body_formatting =` [Function]
`$converter->defaults_special_unit_body_formatting ($special_unit_variety)`

$special_unit_variety is the special output unit variety. Returns the default conversion function reference for *$special_unit_variety*, or `undef` if there is none, which should not happen for the special output units described in this manual.

See Section 16.2 [Customizing Footnotes], page 67, for more on footnotes formatting. See Section 16.3 [Contents and Short Table of Contents Customization], page 69, for more on the `contents` and `shortcontents` formatting. See Section 16.4 [About Special Output Unit Customization], page 71, for more on the `about` special output unit body formatting.

17 Customizing HTML Footers, Headers and Navigation Panels

`texi2any` provides for customization of the HTML page headers, footers, and navigation panel. (These are unrelated to the headings and "footings" produced in TEX output; see Section "Page Headings" in *Texinfo*.)

In the event that your needs are not met by setting customization variables (see Section "Customization of Navigation and Headers" in *Texinfo*) and changing the navigation buttons (see Section 5.4 [Simple Navigation Panel Customization], page 17), you can completely control the formatting of navigation panels by redefining function references. See Section 11.1 [Registering Specific Formating Functions], page 40, for information on how to register the function references.

In a nutshell, element header and footer formatting function determines the button directions list to use and calls navigation header formatting. The navigation header formatting adds some formatting if needed, but mostly calls the navigation panel formatting. The navigation panel can call buttons formatting.

All the formatting functions take a converter object as first argument.

17.1 Navigation Panel Button Formatting

The function reference `format_button` does the formatting of one button, corresponding, in general, to a link to a direction:

$formatted_button `format_button (`*$converter, $button, $source_command*`)` [Function Reference]

$button holds the specification of the button (see [Buttons Display], page 18). *$source_command* is an optional argument, the @-command the link comes from.

Returns the formatted result in *$formatted_button*.

The buttons images can be formatted with `format_button_icon_img` (see below).

Simple navigation panel customization (see Section 5.4 [Simple Navigation Panel Customization], page 17), `USE_ACCESSKEY`, `USE_REL_REV` and direction strings customization (see Section 5.3 [Direction Strings Customization], page 16) can be relevant for the formatting of a button.

To get direction strings typically used for formatting of buttons or hyperrefs leading to that direction, use `direction_string`:

$string `= `*$converter*`->direction_string (`*$direction, $string_type, $context*`)` [Function]

Retrieve the *$direction* (see Section 5.2 [Directions], page 14) string of type *$string_type* (see Section 5.2.2 [Direction Strings], page 16). *$context* is '`normal`' or '`string`'. See Section 4.1 [Init File Expansion Contexts], page 8. If *$context* is `undef`, the '`normal`' context is assumed. The string will be translated if needed. May return `undef`.

To get the Texinfo special output unit associated with a special output unit direction, such as '`About`' or '`Contents`', as well as output unit associated with other global directions, such as '`Top`' or '`Index`', use use `global_direction_unit`:

`\%output_unit = $converter->global_direction_unit ($direction)` [Function]

Return the output unit associated with direction *$direction*, or `undef` if the direction is not a global output unit direction nor a special output unit direction or the associated special output unit is not output.

To get link information for relative and global directions, use `from_element_direction`:

`$result = $converter->from_element_direction ($direction, $type, $source_element, $source_filename, $source_command)` [Function]

Return a string for linking to *$direction*, or the information to be used for a hyperlink to *$direction*, depending on *$type*. The possible values for *$type* are described in Section 5.2.1 [Output Unit Direction Information Type], page 15.

$source_element is the output unit the link comes from. If not set, the current output unit is used. *$source_filename* is the file the link comes from. If not set, the current file name is used. *$source_command* is an optional argument, the @-command the link comes from. It is only used for messages.

`format_button_icon_img` formatting function can be redefined. In the default case, it is called for an active direction, if `ICONS` is set, when formatting a navigation panel button.

`$text format_button_icon_img ($converter, $button, $icon, $name)` [Function Reference]

$button is a button name, typically obtained from the `button` direction string (see Section 5.2.2 [Direction Strings], page 16). *$icon* is an image file name to be used as icon. *$name* is the direction heading, typically formatted in string context. See Section 4.1 [Init File Expansion Contexts], page 8.

Returns a formatted icon image.

See Section 5.2 [Directions], page 14, for the list of directions.

17.2 Navigation Panel and Navigation Header Formatting

The overall display of navigation panels is controlled via this function reference, `format_navigation_header`:

`$navigation_text format_navigation_header ($converter, \@buttons, $command_name, \%element)` [Function Reference]

\@buttons is an array reference holding the specification of the buttons for the navigation panel (see Section 5.4 [Simple Navigation Panel Customization], page 17). *\%element* is the element in which the navigation header is formatted. *$command_name* is the associated command (sectioning command or `@node`). It may be `undef` for special output units.

Returns the formatted navigation header and panel. The navigation panel itself can be formatted with a call to `format_navigation_panel`.

The customization variable `VERTICAL_HEAD_NAVIGATION` should be relevant (see Section "Customization of Navigation and Headers" in *Texinfo*).

The navigation panel display is controlled via `format_navigation_panel`:

$navigation_text `format_navigation_panel` [Function Reference]
(*$converter*, *\@buttons*, *$command_name*, *\%element*, *$vertical*)

\@buttons is an array reference holding the specification of the buttons for that navigation panel. *\%element* is the element in which the navigation header is formatted. *$command_name* is the associated command (sectioning command or `@node`). It may be `undef` for special elements. *$vertical* is true if the navigation panel should be vertical.

Returns the formatted navigation panel in *$navigation_text*. The buttons in the navigation panel can be formatted with a call to `format_button` (see [`format_button`], page 73).

17.3 Element Header and Footer Formatting

By default, the function associated with `format_element_header` formats the header and navigation panel of an output unit.

$formatted_header `format_element_header` [Function Reference]
(*$converter*, *$command_name*, *\%element*, *\%output_unit*)

\%element is the element in which the navigation header is formatted (sectioning command, `@node` or special output unit). *$command_name* is the associated command name. It may be `undef` for special output units. *\%output_unit* is the associated output unit (see Section 10.4.4 [Texinfo Tree Elements in User Defined Functions], page 37).

Returns the formatted navigation header and panel.

In the default code, the function reference select a buttons list (see Section 5.4 [Simple Navigation Panel Customization], page 17). The navigation header can then be formatted with a call to `format_navigation_header` (see [`format_navigation_header`], page 74). It is also possible to format directly the navigation panel, depending on customization variables values and location in file.

Similarly, the function associated with `format_element_footer` formats the footer and navigation panel of a output unit.

$formatted_footer `format_element_footer` [Function Reference]
(*$converter*, *$output_unit_type*, *\%output_unit*, *$content*, *$command*)

\%output_unit is the output unit in which the navigation footer is formatted. *$output_unit_type* is the associated type. *$content* is the formatted element content. *$content* can be `undef`. *$command* is an optional argument, the @-command associated with the *\%output_unit*.

Returns the formatted navigation footer and panel.

In the default code, the function reference select a buttons list (see Section 5.4 [Simple Navigation Panel Customization], page 17). The navigation header can then be formatted with a call to `format_navigation_header` (see [`format_navigation_header`], page 74).

Many customization variables have an effect on the footer formatting, such as `SPLIT` (see Section "HTML Splitting" in *Texinfo*), customization variables used for the customization of headers such as `HEADERS` or `WORDS_IN_PAGE` (see Section "Customization

of Navigation and Headers" in *Texinfo*) and customization variables setting inserted HTML code such as `DEFAULT_RULE` (see Section "Customization of HTML Code Inserted" in *Texinfo*).

To select the list of buttons for header and footer formatting, it may be handy to be able to determine if the output unit being formatted is the Top output unit. To determine if a output unit is associated with the top output unit, use `unit_is_top_output_unit`:

$is_top_output_unit = *$converter*`->unit_is_top_output_unit` (`\%output_unit`) [Function]

Returns true if the *\%output_unit* output unit is the Top output unit (see Section 5.1 [Output Units], page 13) and is either associated with the `@top` sectioning command or with the `Top` `@node`.

17.4 Element Counters in Files

The position of the output unit being formatted in its file or the total number of elements output to a file is interesting for navigation header and footer formatting, for instance to format end of files, decide which type navigation header or footer is needed and whether a rule should be output.

To get information on tree elements unit counter in files, use `count_elements_in_filename`:

$count = *$converter*`->count_elements_in_filename` (*$specification*, *$file_name*) [Function]

Return output unit counter for *$file_name*, or `undef` if the counter does not exist. The counter returned depends on *$specification*:

current Return the number of output units associated with *$file_name* having already been processed.

remaining Return the number of output units associated with *$file_name* that remains to be processed.

total Return the total number of output units associated with the file.

For example, to get the total number of output units associated with the file of a node element:

```
my $file_name = $converter->command_filename($node_element);
my $number = $converter->count_elements_in_filename('total',
                                                    $file_name);
```

18 Beginning and Ending Files

The end of file (footers) formatting function reference is called from the converter after all the output units in the file have been converted. The beginning of file (headers) formatting function reference is called right after the footers formatting function reference.

See Section 11.1 [Registering Specific Formating Functions], page 40, for information on how to register and get the functions references.

18.1 Customizing HTML File Beginning

You can set the variable `DOCTYPE` to replace the default. The `DOCTYPE` is output at the very beginning of each output file.

You can define the variable `EXTRA_HEAD` to add text within the `<head>` HTML element. Similarly, the value of `AFTER_BODY_OPEN` is added just after `<body>` is output. These variables are empty by default.

The `<body>` element attributes may be set by defining the customization variable `BODY_ELEMENT_ATTRIBUTES`.

By default, the encoding name from `OUTPUT_ENCODING_NAME` is used. If this variable is not defined, it is automatically determined.

A date is output in the header if `DATE_IN_HEADER` is set.

The description from `@documentdescription` (or a value set as a customization variable) is used in the header (see Section "`@documentdescription`" in *Texinfo*).

`<link>` elements are used in the header if `USE_LINKS` is set, in which case `LINKS_BUTTONS` determines which links are used and the `rel` direction string (see Section 5.2.2 [Direction Strings], page 16) determines the link type associated with the `rel` attribute. See Section 5.4 [Simple Navigation Panel Customization], page 17.

You can set `HTML_ROOT_ELEMENT_ATTRIBUTES` to add attributes to the `<html>` element.

If `SECTION_NAME_IN_TITLE` is set, the sectioning command argument is used in the `<title>` HTML element instead of the `@node` argument.

You can also set a JavaScript browsing interface with customization variables (see Section "JavaScript Interface and Licenses" in *Texinfo*). See Section "Customization of Navigation and Headers" in *Texinfo* for more information on customization variables in the main Texinfo manual. See Section "Customization of HTML Code Inserted" in *Texinfo* for more on insertion of HTML code in output.

The following function references give full control over the page header formatting done at the top of each HTML output file.

$file_begin `format_begin_file (`*$converter, $filename, \%output_unit*`)` [Function Reference]

$filename is the name of the file output. *\%output_unit* is the first output unit of the file. This function should print the page header, in HTML, including the `<body>` element.

18.2 Customizing HTML File End

You can define the variable `PRE_BODY_CLOSE` to add text just before the HTML `</body>` element. Nothing is added by default. If `PROGRAM_NAME_IN_FOOTER` is set, the date and name of the program that generated the output are output in the footer.

By default, the JavaScript license web labels page is formatted and output at the end of file (see Section "JavaScript Interface and Licenses" in *Texinfo*).

The `format_end_file` function reference give full control over the page footer formatting done at the bottom of each HTML output file.

$file_end `format_end_file` (*$converter*, *$filename*, *\%output_unit*) [Function Reference]

$filename is the name of the file output. *\%output_unit* is the last output unit of the file. This function should print the page footer, including the `</body>` element.

18.3 Associating Information to an Output File

To be able to retrieve information associated with the current file, in general for the file begin or end formatting, `register_file_information` can be used to associate integer information, and `get_file_information` to retrieve that information.

$converter*`->register_file_information` (*$key, *$value*) [Function]

Associate the current output file name file to the key *$key*, itself associated with the integer value *$value*.

$value = *$converter*`->get_file_information` (*$key*, *$file_name*) [Function]

Return the value associated with the key *$key* and file name *$file_name*.

By default, this interface is used to get '`mathjax`' file information registered when converting math @-commands to insert references to MathJax scripts in file beginning (see Section "MathJax scripts" in *Texinfo*) and license information in end of files (see Section "JavaScript Interface and Licenses" in *Texinfo*).

19 Titlepage, CSS and Redirection Files

19.1 HTML Title Page Customization

If `SHOW_TITLE` is not set, no title is output. `SHOW_TITLE` is '`undef`' in the default case. If '`undef`', `SHOW_TITLE` is set if `NO_TOP_NODE_OUTPUT` is set. The "title page" is used to format the HTML title if `USE_TITLEPAGE_FOR_TITLE` is set, otherwise the `simpletitle` is used. `USE_TITLEPAGE_FOR_TITLE` is set in the default case. See Section "HTML Output Structure Customization" in *Texinfo*.

The following functions references provides full control on the title and "title page" formatting:

$title_titlepage `format_title_titlepage` (*$converter*) [Function Reference]

Returns the formatted title or "title page" text.

In the default case, return nothing if `SHOW_TITLE` is not set, return the output of `format_titlepage` if `USE_TITLEPAGE_FOR_TITLE` is set, and otherwise output a simple title based on `simpletitle`.

$title_page `format_titlepage` (*$converter*) [Function Reference]

Returns the formatted "title page" text.

In the default case, the `@titlepage` is used if found in global information, otherwise `simpletitle` is used (see Section 10.2 [Converter General Information], page 30).

19.2 CSS Customization

CSS in HTML output can already be modified with command line options and customization variables (see Section "HTML CSS" in *Texinfo*). More control of the generated CSS is available through functions.

19.2.1 Customization of CSS Rules, Imports and Selectors

Information on static CSS data used in conversion and more direct control over rules, CSS imports and selectors is available through functions. The information is about CSS rules lines and CSS import lines obtained from parsing `--css-include=`*`file`* files, as described in Section "HTML CSS" in *Texinfo*, and CSS style rules associated with HTML elements and class attributes used in the conversion to HTML. The CSS style rules selectors are, classically, *`element.class`* strings with *element* an HTML element and *class* an attribute class associated with that element.

The function used are `css_get_info` and `css_get_selector_style` to get information and `css_add_info` and `css_set_selector_style` to modify:

$converter*`->css_get_info` (*$specification) [Function]
$converter*`->css_get_selector_style` (*$selector) [Function]
$converter*`->css_add_info` (*$specification, $css_info) [Function]
$converter*`->css_set_selector_style` (*$selector, $css_style) [Function]

Those functions can only be used on a converter *$converter*, from functions registered and called with a converter. *$specification* is `'rules'` to get information on or set

information for CSS rules lines and `'imports'` to get information on or set information for CSS import lines. Any other value for *$specification* corresponds to HTML elements and class attributes selectors, and can be used to get the list of selectors.

With `css_get_info`, array references corresponding to *$specification* are returned.

`css_get_selector_style` returns the CSS style corresponding to the HTML element and class attribute selector *$selector*, or `undef` if not found.

With `css_add_info`, *$css_info* is an additional entry added to CSS rules lines if *$specification* is set to `'rules'` or an additional entry added to CSS import lines if *$specification* is set to `'imports'`.

With `css_set_selector_style`, *$selector* is a CSS rule selector and the associated style rule is set to *$css_style*.

Some examples of use:

```
my @all_included_rules = $converter->css_get_info('rules');
my $all_default_selector_styles = $converter->css_get_info('styles');
my $titlefont_header_style = $converter->css_get_selector_style('h1.titlefont');

$converter->css_set_selector_style('h1.titlefont', 'text-align:center');
$converter->css_add_info('imports', "\@import \"special.css\";\n");
```

Note that the CSS selectors and associated style rules that can be accessed and modified will not necessarily end up in the HTML output. They are output only if the HTML element and class corresponding to a selector is seen in the document. See Section 19.2.2 [Customizing the CSS Lines], page 80.

The simplest way to modify CSS rules would be to use a function registered for the '`structure`' stage:

```
sub my_function_set_some_css {
  my $converter = shift;

  $converter->css_set_selector_style('h1.titlefont',
                                     'text-align:center');
  # ... calls to  $converter->css_add_info();
}

texinfo_register_handler('structure', \&my_function_set_some_css);
```

19.2.2 Customizing the CSS Lines

The CSS *element.class* that appeared in a file, gathered through `html_attribute_class` calls (see Section 10.1.1 [Formatting HTML Element with Classes], page 29) are available through the `html_get_css_elements_classes` function:

`\@css_element_classes =` [Function]
$converter*`->html_get_css_elements_classes (`*$file_name`)`

Returns a reference on an array containing `element.class` pairs of elements and classes appearing in *$file_name*.

It is possible to change completely how CSS lines are generated by redefining the following function reference:

$css_lines `format_css_lines` (*$converter*, *$file_name*) [Function Reference]

This function returns the CSS lines and `<script>` HTML element for *$file_name*.

In the default case, the function reference uses `CSS_REFS` corresponding to command-line `--css-ref`, `html_get_css_elements_classes`, `css_get_info` and `css_element_class_rule` (see Section 19.2.1 [Customization of CSS Rules Imports and Selectors], page 79) to determine the CSS lines.

19.3 Customizing Node Redirection Pages

Node redirection pages are output if `NODE_FILES` is set (see Section "Invoking `texi2any`" in *Texinfo*).

It is possible to change completely how node redirection pages are generated by redefining the following function reference:

$node_redirection_file_content `format_node_redirection_page` (*$converter*, *\%element*) [Function Reference]

\%element is a node element needing a redirection page. A redirection page is needed if the node file name is not the file name expected for HTML cross manual references (see Section "HTML Xref" in *Texinfo*).

Returns the content of the node redirection file.

Appendix A Specific Functions for Specific Elements

Links on Texinfo Perl modules functions or descriptions of functions that can be used for specific elements formatting:

`@today` See Section "Texinfo::Convert::Utils::expand_today" in `texi2any_internals`.

`@verbatiminclude`
: See Section "Texinfo::Convert::Utils::expand_verbatiminclude" in `texi2any_internals`.

`@def*` @-commands
: See Section "Texinfo::Convert::Utils::definition_arguments_content" in `texi2any_internals`. See Section "Texinfo::Convert::Utils::definition_category_tree" in `texi2any_internals`.

`@float` See Section "Texinfo::Convert::Converter::float_name_caption" in `texi2any_internals`. Can be called as `$converter->float_name_caption`.

accent @-commands
: See Section "Texinfo::Convert::Converter::xml_accent" in `texi2any_internals`. Can be called as `$converter->xml_accent`.

  See Section "Texinfo::Convert::Converter::xml_numeric_entity_accent" in `texi2any_internals`.

  See Section "Texinfo::Convert::Converter::convert_accents" in `texi2any_internals`.

text element
: See Section "Texinfo::Convert::Converter::xml_format_text_with_numeric_entities" in `texi2any_internals`. Can be called as `$converter->xml_format_text_with_numeric_entities`.

`@item` in `@table` and similar @-commands
: See Section "Texinfo::Convert::Converter::table_item_content_tree" in `texi2any_internals`. Can be called as `$converter->table_item_content_tree`.

`@*index` `@subentry`
: See Section "Texinfo::Convert::Converter::comma_index_subentries_tree" in `texi2any_internals`. Can be called as `$converter->comma_index_subentries_tree`.

global informative commands (`@contents`, `@footnotestyle` ...)
: See Section "Texinfo::Common::set_informative_command_value" in `texi2any_internals`.

heading commands, such as `@subheading`
: See Section "Texinfo::Common::section_level" in `texi2any_internals`. This function would work for sectioning commands too, but for sectioning commands, `section->{'extra'}->{'section_level'}` can also be used. See Section 10.4.4 [Texinfo Tree Elements in User Defined Functions], page 37.

sectioning commands

See Section "Texinfo::Structuring::section_level_adjusted_command_name" in `texi2any_internals`.

`@itemize` `@itemize` normally have an @-command as argument. If, instead, the argument is some Texinfo code, `html_convert_css_string_for_list_mark` can be used to convert that argument to text usable in CSS style specifications.

`$text_for_css` `=` [Function]
`$converter``->html_convert_css_string_for_list_mark (\%element, $explanation)`

\%element is the Texinfo element that is converted to CSS text. In general, it is `$itemize->{'args'}->[0]`, with `$itemize` an `@itemize` Texinfo tree element. *$explanation* is an optional string describing what is being done that can be useful for debugging.

Returns *\%element* formatted as text suitable for CSS.

The `Texinfo::Convert::NodeNameNormalization` converter, used for normalization of labels, exports functions that can be used on Texinfo elements trees to obtain strings that are unique and can be used in attributes. See Section "Texinfo::Convert::NodeNameNormalization" in `texi2any_internals`.

Appendix B Functions Index

$

C

E

F

L

N

O

S

T

U

Appendix C Variables Index

U

Appendix D General Index

L

M

N

O

P

R

S

T

U