## TEX4ht: LATEX to Web publishing

Michal Hoftich

**Abstract**

The article gives overview of the current state of development of TEX4ht, a conversion system for (LA)TEX to HTML, XML, and more. It introduces `make4ht`, a build system for TEX4ht as well as basic ways how to configure TEX4ht.

### 1 Overview of the conversion process

TEX4ht is a system for conversion of TEX documents to various output formats, most notably HTML and *OpenDocument Format*, supported by word processors such as Microsoft Word or LibreOffice Writer. An overview of the system is depicted in figure 1.

The package `tex4ht.sty` starts the conversion process. The document preamble is loaded as usual, but it keeps track of all loaded files. It loads special configuration files for any packages used that are supported by TEX4ht at the beginning of the document. These configuration files are named as the configured file with extension `.4ht`. They may fix clashes between the configured package and TEX4ht, but most notably the package commands are patched to insert special marks to the DVI file, so-called hooks.

After the package configuration, another type of `.4ht` files are loaded. These populate inserted hooks with tags in the selected output format. In the last step before processing of the document contents, a `.cfg` provided by the user can configure the hooks with custom tags. Compilation of the document then continues as usual, resulting in a special DVI file.

The generated DVI file is then processed with the `tex4ht` command. This command creates output files, converts input encodings to UTF-8, and creates two auxiliary files: an `.idv` file, a special DVI file that contains pages to be converted to images, which can be the contents of the LATEX picture environment or complex mathematics; second, an `.lg` file with a list of output files, CSS instructions, and instructions for compiling individual pages in the `.idv` file to images.

The last step in the compilation chain is the `t4ht` program. It processes the `.lg` file and extracts the CSS instructions, converts the images in the `.idv` file, and may call various external commands.

### 2 Supporting scripts

Because the entire conversion process consists of several consecutive steps, we use scripts to make this process easier. The TEX4ht distribution contains several such scripts. They differ in the supported output format, TEX engine used, and options passed to the underlying commands. The most commonly used script is `htlatex`, which uses the PDFTEX engine with LATEX and produces HTML output.

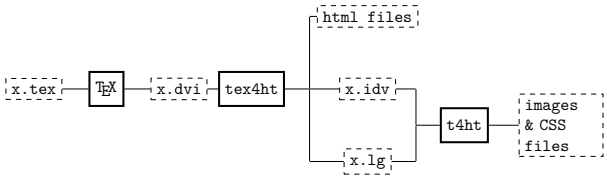


**Figure 1**: TEX4ht process overview

Each of the scripts loads the TEX4ht package without needing to specify it in the document, and options from the command line are passed to the package as well to `tex4ht` and `t4ht` commands.

For example, the following command can be used to request the output in XHTML format in the UTF-8 encoding:

```
htlatex file.tex "xhtml,charset=utf-8" \
                 "-utf8 -cunihtf"
```

However, these scripts are not flexible; each time, they execute a three-time compilation of the TEX document. This ensures the correct structure of hypertext links and tables, as they require multiple compilations to function properly, processing the document repeatedly with `tex4ht` and `t4ht`.

For example, if the document contains a bibliography or glossary that is created by external programs, it is necessary to first call `htlatex`, then the desired program, and then `htlatex` again. In the case of larger documents, compilation time may thus be relatively long.

Passing options to the underlying commands is also quite difficult.

New build scripts have been created for these reasons. My first project that attempted to simplify the TEX4ht compilation process was `tex4ebook`. It added support for e-books, specifically *ePub*, *ePub3* and *mobi* formats. It added support for use of command line switches and build files written in Lua.

The main difference between `tex4ebook` and TEX4ht is the third compilation step. The `t4ht` command is used only to create a CSS file. Image conversion and execution of the external commands is controlled by `tex4ebook` itself. In addition, thanks to the build file support, it is possible to execute commands between individual TEX compilations, for example, to execute an index processor or `bibtex` after the first compilation.

Translation by the author from the original in *Zpravodaj* 2018/1–4, pp. 11–21, for the BachoTEX 2019 proceedings.

The library that added the build support provided features useful also for other output formats than e-books. It was extracted as a standalone tool and the `make4ht` build system is now a recommended tool for use of TEX4ht.

## 3 The `make4ht` build system

`make4ht` enables creation of build scripts in the Lua language. It supports execution of arbitrary commands during the conversion, post-processing of the generated files and defining commands for image conversion. Using the so-called modes, it is possible to influence the order of compilation using switches directly from the command line. For example, the basic script used by `make4ht` supports a draft mode which runs only one compilation of the document instead of the usual three. This can be used to significantly speed up the compilation.

Currently, only LATEX is supported; plain TEX support is possible, but it is more complicated and ConTEXt is not supported at all. In the following text we will focus only on LATEX.

`make4ht` supports a number of switches and options that affect the progress of compiling and processing of the output files. `make4ht` can be launched as follows:

```
make4ht ⟨switches for make4ht⟩ file.tex \
  "⟨options for tex4ht.sty⟩" \
  "⟨switches for tex4ht⟩" \
  "⟨switches for t4ht⟩" \
  "⟨switches for TEX⟩"
```

This complicated list is a result of the way `htlatex` works: it needs to pass options for all components involved in compilation. In most cases, fortunately, there is no need to use all the options. Most of the properties that `tex4ht` and `t4ht` provide can be requested using the `make4ht` switches.

### 3.1 `make4ht` command line switches

Every command line switch that `make4ht` supports has a short and long version. In addition, short switches can be combined. For example, the following two commands are identical:

```
make4ht --lua --utf8 --mode draft filename.tex
make4ht -lum draft filename.tex
```

This command uses LuaLATEX for the compilation, which will be executed only one once, due to `draft` mode. The resulting document will be in UTF-8 text encoding. The default output format for `make4ht` is HTML5 (`htlatex`'s default is HTML4).

In addition to these `--lua`, `--utf8`, and `--mode` switches, there are a number of other useful switches:

`--config (-c)` configuration file for TEX4ht, allowing tags inserted into output files to be changed.

`--build-file (-e)` select a build file.

`--output-dir (-d)` the directory where the output files will be copied.

`--shell-escape (-s)` pass the `-shell-escape` option to LATEX, enabling execution of external commands.

`--xetex (-x)` compile the document with XELATEX.

`--format (-f)` select the output format.

There are other switches, but the above are the most commonly useful.

### 3.2 Output formats and extensions

TEX4ht supports a wide range of XML-based formats, from XHTML, through ODT to DocBook and TEI.

The `--format` switch for `make4ht` supports the formats `html5`, `xhtml`, `odt`, `tei` and `docbook` (the format names must be specified in lowercase). The default format is `html5`.

Formats can be selected also using the `tex4ht.sty` option:

```
make4ht filename.tex "docbook"
```

However, the advantage of `--format` is that it can fix some common issues for the particular formats. It can also load extensions. Extensions allow us to influence the compilation without having to use a build script. The list of extensions to be used can be written after the format name. They can be enabled using the plus character, and disabled with the minus character.[1] For example, the following command uses the HTMLTidy command to fix some common errors in the generated HTML file:

```
make4ht -f html5+tidy simple-example.tex
```

The following extensions are available:

`latexmk_build` Use the `latexmk` build system to compile the document. This will take care of calling external commands, for example, to create a bibliography.

`tidy` Clean HTML output with the `tidy` command.

`dvisvgm_hashes` efficient generation of images using the `dvisvgm` command. It can use multiple processors and only creates images that have been changed or created since the last compilation. This can make compilation noticeably faster.

`common_filters`, `common_domfilters` Clean the document using filters. Filters will be discussed later in the article.

`mathjaxnode` Convert MathML math code into special HTML using *MathJax Node Page*.[2] This

[1] Extensions can be enabled in a `make4ht` configuration file, so disabling them from the command line can be useful.

[2] `github.com/pkra/mathjax-node-page`

produces mathematics that can be viewed in web browsers without MathML support. The rendering of the result doesn't need JavaScript, which results in much faster display of the document compared to standard MathJax.

`staticsite` Create a document usable with static page generators such as *Jekyll*.[3] These are useful for creating blogs or more complex websites.

### 3.3 Configuration files for `make4ht`

`make4ht` supports build scripts in the Lua language. They can be used to call external commands, to pass parameters to an executed command, to apply filters to the output files, to affect the image conversion, or to configure extensions.

The `.make4ht` configuration file is a special build script that is loaded automatically and should contain only general configurations shared between documents. In contrast, normal build files may contain configurations useful only for the current document. The configuration file can be located in the directory of the current document or in its parent directories.

This can be useful, for example, for maintaining a blog, with each document in its own directory. In the parent directory, a configuration file ensures proper processing. Here's a small example:

```
filter_settings "staticsite" {
  site_root = "output"
}
Make:enable_extension("common_domfilters")
if mode=="publish" then
  Make:enable_extension("staticsite")
  Make:htlatex {}
end
```

This configuration file sets the option `site_root` for the `staticsite` extension using the command `filter_settings`. This command can be used to set options for both filters and extensions. The name of the filter or extension is separated from the command by a space, followed by another space-separated field, where options can be set.

The next command is `Make:enable_extension`, which enables the extension. In this case the extension `common_domfilters` is used in every compilation, but the `staticsite` extension is used only in *publish* mode. In this mode it is also necessary to use the `Make:htlatex{}` to require at least one LaTeX compilation.

Now we can run `make4ht` in publish mode:

```
make4ht -um publish simple-example.tex
```

The `output` directory will be created if it does not already exist; HTML and CSS files will be copied here. The static site generator must be configured to look for files here and it needs to be executed manually; the extension doesn't do that.

The resulting HTML looks something like this:

```
---
time: 1544811015
date: '2018-12-14 18:10:47'
title: 'sample'
styles:
- '2018-12-14-simple-example.css'
meta:
- charset: 'utf-8'
---
<p>Sample document</p>
```

The document header enclosed between the two `---` lines contains variables in the YAML format extracted from the HTML file. Only the contents of the document body remains in the document; the old header is stripped off. The static generator can then create a page based on the template and the variables in the YAML header.

This was just a basic example. Filters and extensions have much more extensive configurable options, all of which are described in the `make4ht` documentation.[4]

### 3.4 Build files

In the compilation scripts it is possible to use the same procedures as in the configuration file, but focused on the particular compiled document. The following code shows use of the DOM filters. These take advantage of the *LuaXML*[5] library. It supports processing XML files using the Document Object Model (DOM) interface. This makes it easy to navigate, edit, create or delete elements.

The use of DOM filters is shown in the following example for LuaLaTeX:

```
\documentclass{article}
\begin{document}
Test {\itshape háčků}
\end{document}
```

Because of a known error in processing the DVI file with the `tex4ht` command, each accented character in the generated HTML file will be placed in a separate `<span>` element:

```
<!--l. 4--><p class="noindent" >Test
<span class="rm-lmri-10">h</span><span
class="rm-lmri-10">á</span><span
class="rm-lmri-10">čk</span><span
class="rm-lmri-10">ů</span> </p>
```

The following build file removes this by using the built-in `joincharacters` DOM filter. In addition, it

[3] `jekyllrb.com`

[4] `ctan.org/pkg/make4ht`

[5] `ctan.org/pkg/luaxml`

changes the value of the `class` attribute for all `<p>` elements to `mypar`, just to show how to work with the DOM interface:

```
local domfilter = require("make4ht-domfilter")

local function domsample(dom)
  -- the following code will process
  -- all <p> elements
  for _, par in
    ipairs(dom:query_selector("p")) do
    -- set the "class" attribute
    par:set_attribute("class", "mypar")
  end
  return dom
end

local process = domfilter({
  "joincharacters",
  domsample
})
Make:match("html$", process)
```

The script uses the standard Lua `require` function to load the `make4ht-domfilter` library. This creates a `domfilter` function that takes a list of DOM filters to execute as a parameter.

Each call to the `domfilter` function creates another function with a chain of filters specified in a table. Parameters in the fields can be either the name of an existing DOM filter, or a function defined in the file.

The filter chain can then be used in the function `Make:match`. This takes a filename pattern to match files for which the filters should be executed, and the filter chain.

The `process` function will run on each file whose filename ends with `html` in this case.

The resulting HTML file does not contain the extra `<span>` elements and the `<p>` element has a class attribute value of `mypar`:

```
<!-- l. 3 --><p class='mypar'>
Test <span class='rm-lmri-10'>háčků</span>
</p>
```

Here is another example, of a more complex build file with external command execution and configuration of image generation:

```
Make:add("biber", "biber ${input}")
Make:htlatex {}
Make:biber {}
Make:htlatex {}
Make:image("png$",
  "dvipng -bg Transparent -T tight "
  .. "-o ${output} -pp ${page} ${source}")
Make:match("html$",
  "tidy -m -utf8 -asxhtml -q -i ${filename}")
```

The `Make:add` function defines a new usable command, `biber` in this case. The second parameter is a formatting string, which may contain `${...}` variable templates, which are in turn replaced by parameters set by `make4ht`. Here, the `${input}` will be replaced with the input file name.

The newly added command can then be used as `Make:`⟨*command*⟩, like the built-in commands. Additional variables may be set in the table passed as the argument.

The `Make:htlatex` command is built in and requires one execution of LaTeX with TeX4ht active.

The `Make:image` command configures the image conversion. Three variables are available: `page` contains the page number of the image in the DVI file, `output` is the name of the output image, and `source` is the name of the `.idv` file.

The use of the `Make:match` command was shown in the previous example, but it may also contain a string with the command to be executed. The `filename` variable contains the name of the generated file currently being processed.

## 4 TeX4ht configuration

Output format tags embedded in a document are fully configurable via several mechanisms. The easiest way is to use the `tex4ht.sty` package options, a more advanced choice is to use a custom configuration file, and the most powerful option is to use `.4ht` files.

When a TeX file is compiled using `make4ht` or another TeX4ht script, the `tex4ht.sty` package is loaded before the document itself. Package options are obtained from the compilation script arguments. As a result, it is not necessary to explicitly load the `tex4ht.sty` package in the document.

The TeX file loading mechanism is modified to register each loaded file with TeX4ht. For some packages, TeX4ht has code to simply stop it from being loaded, or to immediately override some macros. This is necessary for packages that are incompatible with TeX4ht, such as `fontspec`.

After execution of the document preamble, the configuration files for the packages detected during processing are loaded. These files are named as the base filename of the configured package, extended with `.4ht`. Their main function is to insert configurable macros, called hooks, into the commands provided by the package. In general, it is better not to redefine macros, only to patch them with the commands TeX4ht provides for this purpose. This is enough in most cases.

Output format configuration files are loaded after the package configuration files are processed. These define the contents of the hooks. Besides

inserting output format tags, the hooks can contain any valid TEX commands.

### 4.1 `tex4ht.sty` options

Many configurations are conditional, that is, executed only in the presence of a particular option being given for `tex4ht.sty`. Each output format configuration file can test any option, which means that there is no restriction on the list of possible options; each output format can support a different set of options.

As mentioned above, it is neither necessary nor desirable to load `tex4ht.sty` directly in the document, so it is possible to pass the options in other ways. The easiest way is to use the compilation script argument. This is always the argument following the document name. For example, here we specify the two options `mathml` and `mathjax`.

```
make4ht file.tex "mathml,mathjax"
```

Another way to pass options is to use `\Preamble` command in the private configuration file. We'll show this in the next section.

As mentioned above, the list of options is open-ended, but let's now look at some current options regarding mathematical outputs in HTML. The default configuration for mathematical environments produces a blend of rich text and images for more complex math, if it cannot be easily created with HTML elements. Often this output doesn't look good. As an alternative, it is possible to use images for all math content. This can be achieved by using the `pic-m` options for inline mathematics and `pic-`⟨*environment*⟩ for mathematical environments. For example, the `pic-align` option will make images for all `align` environments.

By default, images are created in the PNG bitmap format. Higher quality can be achieved using the SVG vector format. This can be specified with the `svg` option.

The TEX4ht documentation is unfortunately somewhat spartan. With the `info` option, much useful information about the available configurations can be found in the `.log` file after the compilation of a document.

The options listed in the example above, `mathml` and `mathjax`, provide the best quality output for mathematical content. The MathML markup language, requested by the first option, encodes the mathematical information, but its support in Web browsers is poor. The second option requests the *MathJax* library, which can render the MathML output in all browsers with JavaScript support.

The `mathjax` option used without `mathml` completely turns off compilation of math by TEX4ht; all math content remains in the HTML document as raw LATEX macros. MathJax then processes the document and renders the math in the correct way. The disadvantage of this method is that MathJax does not support all packages and user commands; it needs special configuration in these cases. Emulation of some complex macros may not even be possible.

### 4.2 Private configuration file

The private configuration file can be used to insert custom content into the configuration hooks. This file has a special structure:

⟨*preamble definitions*⟩ ...
`\Preamble{`⟨*tex4ht.sty options*⟩`}`
... ⟨*normal configurations*⟩ ...
`\begin{document}`
... ⟨*configuration for HTML head*⟩
`\EndPreamble`

The three commands shown here must be always included in this configuration file: `\Preamble`, `\begin{document}` and `\EndPreamble`. The configuration file name can be passed to `make4ht` using the switch `--config` (or `-c`), like this:

```
make4ht -c myconfig.cfg file.tex
```

The full path to the configuration file must be used if it is not placed in the current directory.

There are several configuration commands. The most important are `\Configure` for common configurations, `\ConfigureEnv` for configuration of LATEX environments, and `\ConfigureList` for the configuration of the list environments.

The `\HCode` command is used for insertion of the output format tags. The `\Hnewline` command inserts a newline in the output document. And the `\Css` command writes content to the CSS file.

The following example configures the hooks for the `\textit` command to insert the `<em>` element.

```
\Configure{textit}
  {\HCode{<em>}\NoFonts}
  {\EndNoFonts\HCode{</em>}}
```

The `\Configure` command takes a variable number of arguments. It depends on the hooks' definition how many arguments are needed. The first argument is always the name of the configuration; following arguments then put the code in the hooks. Typically, a configuration requires two hooks: the first places code before the start of the command, the second after it is done. This is the case for the `textit` example above. The configuration name may match the name of the configured command, but this is not always the case. The package `.4ht` file may choose the configuration names arbitrarily.

The \NoFonts command used above disables inserting formatting elements for fonts when processing a DVI file. TEX4ht automatically creates basic formatting for font changes. This makes it possible to create a document with basic formatting even for unsupported commands, but it is not desirable when the command is configured using custom HTML elements.

Correct paragraph handling is difficult, and TEX4ht sometimes puts paragraph tags in undesired places. This applies primarily to environment configurations that can contain several paragraphs and yet enclose their entire content in one element. It may happen that the starting paragraph mark is placed before the beginning of this element, but it should be placed right after that. The \IgnorePar command can prevent the insertion of a tag for the next paragraph. \EndP inserts a closing tag for the previous paragraph. There are more commands to work with paragraphs, but these are the most important.

To illustrate this issue, the following example uses the hypothetical rightaligned environment:

```
\ConfigureEnv{rightaligned}
  {\HCode{<section class="right">}}
  {\HCode{</section>}}
  {}
  {}
```

The \ConfigureEnv command expects five parameters. The first is name of the environment to configure. The contents of the second parameter are inserted at the beginning of the environment, and the contents of the third at the end of the environment. The other two parameters are used only if the configured environment is based on a list. In most cases they may be left blank. The HTML code created by the configuration above will look something like the following:

```
<p class="indent" ><section class="right">
...
</p><p class="indent></section>
```

As described above, this code is invalid. The terminating tag for the <p> element is placed at the wrong nesting level. The invalid code can cause the DOM filters and other post-processing tools expecting well-formed XML files to fail, so this situation must be avoided.

The correct configuration is somewhat more complicated:

```
\ConfigureEnv{rightaligned}
  {\ifvmode\IgnorePar\fi\EndP
   \HCode{<section class="right">}\par}
  {\ifvmode\IgnorePar\fi\EndP
   \HCode{</section>}}
  {}
  {}
```

In this case the insertion of tags for paragraphs is controlled, resulting in a correctly nested structure:

```
<section class="right">
<!--l. 9--><p class="indent" >
...
</p></section>
```

Another feature is the conversion of part of a document to an image. This can be requested using the commands \Picture* or \Picture+. The difference between these is that the first processes its content as a vertical box, and the second does not. The content between any of these commands and the closing \EndPicture is converted to an image.

The following example creates an image for the text contained in the topicture environment:

```
\documentclass{article}
\newenvironment{topicture}{\bfseries}{}
\begin{document}
\begin{topicture}
  Contents of this environment
  will be converted as an image.
\end{topicture}
\end{document}
```

The TEX4ht configuration for the topicture environment uses \Picture*:

```
\ConfigureEnv{topicture}
  {\Picture*{}}
  {\EndPicture}
  {}
  {}
```

The resulting document will contain an image of the text contained in the topicture environment as it was typeset in the DVI file.

## 5 Conclusion

The TEX4ht configuration options are extensive. We have touched only the basics in this article, but it should be enough to solve many basic issues that users might face. We omitted examples of how to add configurations for a new LATEX package; we hope to address this topic in a future article.

The system is easier and more efficient to use than in the past, thanks to the make4ht build system.

The new documentation for TEX4ht is being developed with financial support by CSTUG. It will describe the most useful user configurations, as well as technical details of the system.

◇ Michal Hoftich
Charles University, Faculty of Education
michal.hoftich (at) pedf dot cuni dot cz
https://www.kodymirus.cz