LinuxDoc+Emacs+Ispell−HOWTO

Table of Contents

LinuxDoc+Emacs+Ispell−HOWTO

Author: Philippe MARTIN (feloy@wanadoo.fr)
Translator: Sébastien Blondeel (Sebastien.Blondeel@lifl.fr)

v0.4, 27 February 1998

This document is aimed at writers and translators of Linux HOWTOs or any other paper for the Linux Documentation Project. It gives them hints at using tools including Emacs and Ispell.

5.Ispell

6.Dirty Tricks

Appendix

7.An `insert−sgml−header` function

1.Preamble

1.1 Copyright

1.2 Credits

Special thanks go to Sébastien Blondeel, who is a nasty bugger and asked me so much about Emacs setup. His clever questions have allowed me to understand it better and pass the knowledge to you through this document.

1.3 Comments

Do not hesitate to tell me any thing you think will help make this document better. I will examine your critics thoroughly.
Do not hesitate as well to ask me any questions related to topics discussed here. I will be more than happy to answer them, as they may help me further improve this document.

Translator note: If the English is ugly, well then that goes to me!

1.4 Versions

This paper is about the following versions:

- Sgml−tools version 0.99,
- Emacs version 19.34,
- Ispell version 3.1,
- All Emacs libraries referred to in this document are distributed with the above Emacs version, apart from `iso−sgml`, which is distributed with XEmacs, and `psgml`, which is a stand−alone library.

2.Introduction

2.1 SGML

Standard Generalised Mark−up Language, or **SGML**, is a language to define document types.

For instance, one may define the document type *recipe*, with a first part presenting the ingredients, a second part introducing the accessories, a third part giving step by step instructions for baking the cake, and a nice final picture to show the outcome of it all.

This is called a *Document Type Definition*. It does not define what the final product will look like, it only defines what it may contain.

To use the same example again, I'm sure that upon reading my idea of a recipe, you recognised yours, or your favourite cook's. Nevertheless, they actually look different: mine have a picture in the upper left corner of the bathroom cupboard, and the ingredients list can be found in the back garden, between the swimming pool and the barbecue. Yours?

Thanks to this standard definition, one can write a document, without taking into account what it will look like in the end to the reader.

2.2 The `LinuxDoc` Type Definition

This type is used to write, as you might have guessed, documents related to Linux.

Such documents are generally built as follows: they start with a title followed by the name of the author, and the version number and date. Then comes the abstract (so you don't have to browse through it before realizing it isn't what you were looking for after all), then the contents which show the structure so that those in a rush can go directly to the part they want to read.

Then comes a list of chapters, sections, paragraphs. Among these, one can insert bits of programs, change the font to emphasise a word or a sentence, insert lists, refer to another part of the document, etc.

To write such a document, you just need to specify at the right time the title, the author, the date, and the document version, the chapters and sections, say when a list is to be inserted, what its elements are etc.

2.3 `SGML−Tools`

SGML−Tools will turn the specification of a document into the final result in the form you prefer. If you want it in your personal library, you will choose *PostScript*. If you want to share it with the world through the Web, it will be *HTML*. If you can't help it and must read it under Windows, you can turn it into *RTF* to be able to read it with any word processor. Or maybe use all three formats to accommodate your changing moods.

SGML−Tools are available via anonymous FTP at ftp://ftp.lip6.fr/pub/sgml−tools/

3.Your first document.

3.1 From a text document

If you want to turn a text document into SGML to port it to other formats, this is the way to go:

1. Add the following lines at the very beginning:

   ```
   <!doctype linuxdoc system>
    <article>
     <title>Title Goes Here</title>
     <author>
      name of author, author's e−mail, etc.
     </author>
     <date>
      version and date
     </date>
   ```
2. If you describe briefly the contents of the document in the beginning, surround that paragraph with the `<abstract>` and `</abstract>` tags.
3. Then insert the `<toc>` tag, which stands for *Table Of Contents*.
4. At the beginning of each new chapter, replace the line giving the number and title of the chapter with:

   ```
   <sect>The Title Of The Chapter
   ```

   and add the `</sect>` tag at the end of the chapter.

Note : You don't have to put the chapter number, this is done automatically.

5. Proceed in the same way for sections. You need to delete their numbers and tag their titles with `<sect1>` and they end with `</sect1>`.
6. You can also define as many as 4 levels of nesting in the sections, using `<sectn>` and `</sectn>` where `n`= 2, 3, or 4 in a similar way.
7. In the beginning of each paragraph, insert the `<p>` tag.
8. If you need to emphasise some parts, tag them with `<it>` and `</it>` (*italics*), `<bf>` and `</bf>` (**bold face**), or `<tt>` and `</tt>` (`typewriter style`).
9. To insert a list like the following one:

```
 This is a four lines list:

  - first line goes here
  - second line comes next
  - yet another one
  - that's it.
```

you must replace it with:

```
 This is a four lines list:
<itemize>
<item>first line goes here
<item>second line come next
<item>yet another one
<item>that's it.
</itemize>
```

10. When a whole block is a part of a program, or something else that needs to stick out:

```
<verb>
 10 REM Oh my God what's this?
 20 REM I thought this had long disappeared!
 30 PRINT "I am back to";
 40 PRINT "save the world."
 50 INPUT "From whom, do you reckon? ",M$
 60 IF M$="Bill" THEN PRINT "Thou art wise.":GOTO PARADISE
 70 ELSE PRINT "You ain't got a clue...":GOTO RICHMOND
 </verb>
```

11. Thus far, your SGML formating skills are fairly decent. If you want to refine your document, you may have a look at the user's guide for **SGML−Tools**, which gives more details about the **LinuxDoc** document type.

4.Configuring `Emacs`

4.1 Accented Characters

If you want to write documents in French or in any other western European language, you will need 8−bit characters. This is how to set Emacs up to tell it to accept such characters.

The displaying of 8−bit characters

To let Emacs display 8−bit characters, you will need the following lines in your `.emacs` file:

```
(standard-display-european 1)
(load-library "iso-syntax")
```

If you are using Emacs on a terminal which has no 8−bit support, you can use the `iso-ascii` library (`(load-library "iso-ascii")`), which tells Emacs to display such characters to its best approximation.

The typing of 8−bit characters

If your keyboard allows you to enter accented characters, no problem. If not here are some remedies:

The `iso-acc` library

The Emacs `iso-acc` library will let you type 8−bit characters from a 7−bit keyboard.

To use it, insert the following in your `.emacs` file:

```
(load-library "iso-acc")
```

Then, upon running Emacs and opening the file you need to edit, type `Meta-x iso-accents-mode`.

You can then enter the **é** of the French word *café* typing `'` then `e`. More generally, you will type an accented character typing the accent first, then the letter to accent (upper or lower case). The following are the accents you may use:

- `'` : Acute
- `` ` `` : Grave
- `^` : Circumflex
- `"` : Dieresis
- `~` : Tilde, cedilla, and other particular cases (cf iso−acc.el).
- `/` : To bar a letter, etc.

If you need one of these characters and not an accented letter, type a space next to it. For instance, to type *l'éléphant*, type `l ' spc ' e l ' e ...`

You will find all the possible combinations in the `iso-acc.el` file.

The Meta key

Some terminals will let you type 8−bit characters with the Meta (or Alt) key. For example, pressing **Meta−i** will get you the **é** character.

But Emacs reserved the Meta key for other uses, and I know of no library which lets you use it for accented characters.

This is a solution:

```
(global-set-key "\ei" '(lambda () (interactive) (insert ?\351)))
                   _                                       ___
```

Such a line, if inserted in your `.emacs` file, will let you type **é** using the **Meta−i** keystroke. You can redefine in such a way the combinations you need if you replace **i** with the right key and **351** with the right code (the code being taken from the ISO−8859−1 character set).

Warning! Some local modes may redefine such key combinations.

The displaying of 8−bit SGML characters

Under SGML, you can type accented characters with macros. For example, the **é** key is **é**. Generally, the applications that need to read SGML can read 8−bit characters and there is no need to use these macros. But some may not be able to do so. Given that there is a way to solve this problem, it would be a waste to let these crash.

The `iso-sgml` library will let you type accented characters under Emacs, like always, but upon saving your file to the disk, it will turn these 8−bit characters into their SGML equivalent.

It is therefore easy, thanks to this library, to type and reread your document under Emacs, and you can be sure a non 8−bit clean application will accept you document.

To use this library, you just need to add the following lines to your `.emacs` file:

```
(setq sgml-mode-hook
'(lambda () "Defaults for SGML mode."
 (load-library "iso-sgml")))
```

4.2 SGML mode

Upon loading a file with the **.sgml** extension, Emacs enters the **sgml mode** automatically. If it doesn't, you can tell it to do so manually by typing `Meta-x sgml-mode`, or automatically by adding the following lines to your `.emacs` file:

```
(setq auto-mode-alist
(append '(("\.sgml$"  . sgml-mode))
          auto-mode-alist))
```

This mode will let you choose how to insert 8−bit characters for example. With `Meta-x sgml-name-8bit-mode` (or the menu item *SGML/Toggle 8−bit insertion*), you can choose to type 8−bit characters as is, or in SGML form, i.e. in the form **&...;**.

It will as well let you hide or show SGML tags, with `Meta-x sgml-tags-invisible` (or the menu item *SGML/Toggle Tag Visibility*).

4.3 PSGML mode

PSGML mode helps a lot to edit SGML documents with Emacs.

The psgml−linuxdoc documentation explains how to install this mode and use it with *LinuxDoc*.

4.4 Miscellaneous

auto−fill mode

In the normal mode, when you type a paragraph and get to the end of the line, you must use the Return key yourself to get to the next line, or else your line goes on through the whole paragraph. When you use Return to get to the next line, you get a paragraph with ragged right margins.

If you let some lines go beyond a reasonable width, you won't be able to see them with some editors.

The **auto−fill** mode automates this boring task: when you go further than a certain column (the 70th by default), you are automatically taken to the next line.

This is how to use this mode, and set the width of your lines to 80:

```
(setq sgml-mode-hook
      '(lambda () "Defaults for SGML mode."
          (auto-fill-mode)
          (setq fill-column 80)))
```

5.Ispell

If you want to spell−check your document from within Emacs, you may use the **Ispell** package and its Emacs mode.

5.1 Choosing your default dictionaries

You can set up Emacs so that upon loading a file, it chooses automatically which dictionaries to use (you can use several). The first one, certainly the most important, is the main dictionary, distributed with Ispell. You can choose among several languages. The second one is your personal dictionary, where Ispell will insert words it couldn't find in the main dictionary but you told it to remember.

If you wish to use as a default dictionary the French dictionary that comes with Ispell, and if you wish to use the file `.ispell-dico-perso` in your home directory as a personal dictionary, insert the following lines in your `.emacs` file:

```
(setq sgml-mode-hook
'(lambda () "Defauts for SGML mode."
(setq ispell-personal-dictionary "~/.ispell-dico-perso")
(ispell-change-dictionary "francais")
))
```

5.2 Selecting special dictionaries for certain files

You may have a little problem if you do not spell−check documents in the same language at all times. If you translate documents, it is very likely that you swap languages (and dictionaries) very often.

I don't know of any Lisp way of selecting, either automatically, or with a single mouse click, the main and personal dictionaries associated to the language currently being used. (If you do, please tell me!)

However, it is possible to indicate, at the end of the file, which dictionaries you want to use for the current file (and only this one). It suffices to add them as commentaries, so that Ispell can read them upon launching a spell−check:

```
<!-- Local IspellDict: english -->
<!-- Local IspellPersDict: ~/emacs/.ispell-english -->
```

If you have previously defined, in your `.emacs` file, that your default dictionaries are the French dictionaries, then you can add these lines in the end of any file written in English.

5.3 Spell−checking your document

To spell−check the whole of your document, use, from anywhere in the document the `Meta−x ispell−buffer` command. You may as well only run the checking on a region in your document:

- Mark the beginning of the region with `Ctrl−Spc` (mark−set−command),
- Go to the end of the region to check,
- type `Meta−x ispell−region`.

Emacs then runs Ispell. Upon meeting an unknown word, this one shows you said word (usually highlighted) and prompts you for a key:

- **spc** accepts the word, this time only,
- **i** accepts the word and inserts it in your personal dictionary,
- **a** accepts the word for this session,
- **A** accepts the word for this file, and inserts it in the local file dictionary
- **r** allows you to correct the word by hand
- **R** allows you to correct all the occurrences of the misspelled word,
- **x** stops the checking, and puts the cursor back in place,
- **X** stops the checking and leaves the cursor where it is, letting you correct your file; you will be able to continue the spell−checking later if you type `Meta−x ispell−continue`,
- **?** gives you online help.

If ispell finds one or several words close to the unknown one, it will show them in a little window, each one of them preceded by a digit. Just type this digit to replace the misspelled word with the corresponding word.

5.4 Personal dictionary versus local file dictionary

The **i** key will let you insert a word in your personal dictionary, whereas **A** will let you insert a word in the local file dictionary.

The local file dictionary is a sequence of words inserted at the end of the file, as comments, reread by Ispell each time it is run on the file. This way, you can accept some words, acceptable in this file, but not necessarily acceptable in other files.

As far as I am concerned, I think it is better that the personal dictionary be reserved for words the main dictionary doesn't know but which belong to the language (like hyphenated words), plus some common words like proper nouns or others (like *Linux*), if they don't look too much like a real word of the main dictionary; adding too many words in the personal dictionary, such as first names, may be dangerous, because they may look like a word of the language (one can imagine Ispell being mystified on the following: `*When the going gets tof, the tof get going*

Tof is a French abbreviation for the first name *Christophe*.

'!).

5.5 Typing spell−checking

Ispell can spell−check your file while you're typing. You need to use **ispell−minor−mode** for this. To start it or stop it, type `Meta−x ispell−minor−mode`. Ispell will *beep* you each time you type a word it doesn't know.

If those *beeps* hassle you (or your roommate is taking a nap), you can replace those annoying *beeps* with a flash on the screen, with the command `Meta−x set−variable RET visible−bell RET t RET`. You can add the following line in your `.emacs` and silence Emacs forever:

```
(setq visible-bell t)
```

6.Dirty Tricks

6.1 Inserting a header automatically

Emacs allows you to *hook* some actions to any event (opening of a file, saving, running a new mode, etc).

The **autoinsert** library uses this feature: when you open a new file under Emacs, this library inserts, according to the type of the file, a *standard* header.

In our case, this *standard* header could well be the part declaring the document type (LinuxDoc), the title, the author, and the date.

I will describe here two ways to insert such a header. You could insert a template file containing the information to insert, or you could run an **elisp** routine.

by inserting a file

You must first tell Emacs to run the `auto−insert` when opening a file, then to read the **autoinsert** library which declares the `auto−insert−alist` list which we need to change. This list defines the header to insert for each file type. By default, the file to insert must be in the `~/insert/` directory, but it is possible to redefine the `auto−insert−directory` variable if you want to put it somewhere else.

Add the following lines to your `.emacs` file to insert the `~/emacs/sgml-insert.sgml` file each time you open a new SGML file:

```
(add-hook 'find-file-hooks 'auto-insert)
(load-library "autoinsert")
(setq auto-insert-directory "~/emacs/")
(setq auto-insert-alist
      (append '((sgml-mode .  "sgml-insert.sgml"))
              auto-insert-alist))
```

You can then write in the `~/emacs/sgml-insert.sgml` file your customised header, then re−run Emacs and open some `foobar.sgml` file. Emacs should ask you to confirm the automatic insertion, and if you answer yes, insert your header.

by running a routine

This works like before, but instead of setting the `auto-insert-alist` to a file to insert, you need to set it to a function to execute. This is how to proceed, taking for granted you want to write this function in a file named `~/emacs/sgml-header.el`. (there's no need to burden your `.emacs` file with such a function, as it may turn out to be quite long):

```
(add-hook 'find-file-hooks 'auto-insert)
(load-library "autoinsert")
(add-to-list 'load-path "~/emacs")
(load-library "sgml-header")
(setq auto-insert-alist
      (append '(((sgml-mode .  "SGML Mode") . insert-sgml-header))
              auto-insert-alist))
```

You will find in appendix an example of `insert-sgml-header` function.

7.An `insert-sgml-header` function

This function will let the user insert a customised header for a Linux Documentation Project document in a file. It can be called automatically when one opens a new file, or explicitly, by the user.

This function prompts the user, through the *mini−buffer*, for some pieces of information, some of which are necessary, some of which are not.

First comes the title. If none is given, the function returns immediately, and inserts nothing. Then comes the date, the author, his e−mail and home page (these last two are optional).

Then comes a request for the name of the translator. If there is none, just type *Return*, and no further prompting about a hypothetical translator will be done. If there is one, you are asked for his e−mail and home page (optional as well).

This function then prints your request to the current buffer, including of course all the information you typed in a set up form, and including as well the tags which will serve for the abstract and the first chapter. It finally puts the cursor in the place where the abstract needs to be typed.

```
(defun insert-sgml-header ()
  "Inserts the header for a LinuxDoc document"
  (interactive)
  (let (title author email home translator email-translator home-translator date
             starting-point)
    (setq title (read-from-minibuffer "Title: "))
    (if (> (length title) 0)
        (progn
          (setq date (read-from-minibuffer "Date: ")
                author (read-from-minibuffer "Author: ")
                email (read-from-minibuffer "Author e-mail: ")
                home (read-from-minibuffer "Author home page: http://")
                translator (read-from-minibuffer "Translator: "))
          (insert "<!doctype linuxdoc system>\n<article>\n<title>")
          (insert title)
          (insert "</title>\n<author>\nAuthor: ") (insert author) (insert "<newline>\n")
          (if (> (length email) 0)
              (progn
                (insert "<htmlurl url=\"mailto:")
                (insert email) (insert "\" name=\"") (insert email)
                (insert "\"><newline>\n")))
          (if (> (length home) 0)
              (progn
                (insert "<htmlurl url=\"http://")
                (insert home) (insert "\" name=\"") (insert home)
                (insert "\">\n<newline>")))
          (if (> (length translator) 0)
              (progn
                (setq email-translator (read-from-minibuffer "Translator e-mail: ")
                      home-translator (read-from-minibuffer "Translator home page: http://"))
                (insert "Translator : ")
                (insert translator)
                (insert "<newline>\n")
                (if (> (length email-translator) 0)
                    (progn
                      (insert "<htmlurl url=\"mailto:")
                      (insert email-translator) (insert "\" name=\"")
                      (insert email-translator)
                      (insert "\"><newline>\n")))
                (if (> (length home-translator) 0)
                    (progn
                      (insert "<htmlurl url=\"http://")
                      (insert home-translator) (insert "\" name=\"")
                      (insert home-translator)
                      (insert "\"><newline>\n")))))
          (insert "</author>\n<date>\n")
          (insert date)
          (insert "\n</date>\n\n<abstract>\n")
          (setq point-beginning (point))
          (insert "\n</abstract>\n<toc>\n\n<sect>\n<p>\n\n\n</sect>\n\n</article>\n")
          (goto-char point-beginning)
          ))))
```