

(empty left first page)

LibreOffice & ZmL plain source text working and comparing and AsciiDoc

Dr. Hartmut Schorrig

www.vishia.org

2024-10-09

LibreOffice odt content is held parallel and also editable and convertible in a plain text, the Format is named ZmL (Z markup Language). Also working with AsciiDoc is supported.

Table of Contents

1 Approaches.....	4
2 Some decisions how to write a technical documentation.....	10
3 Z markup Language.....	23
4 Handling Writing and Converting.....	42
5 Implementation.....	46

1 Approaches

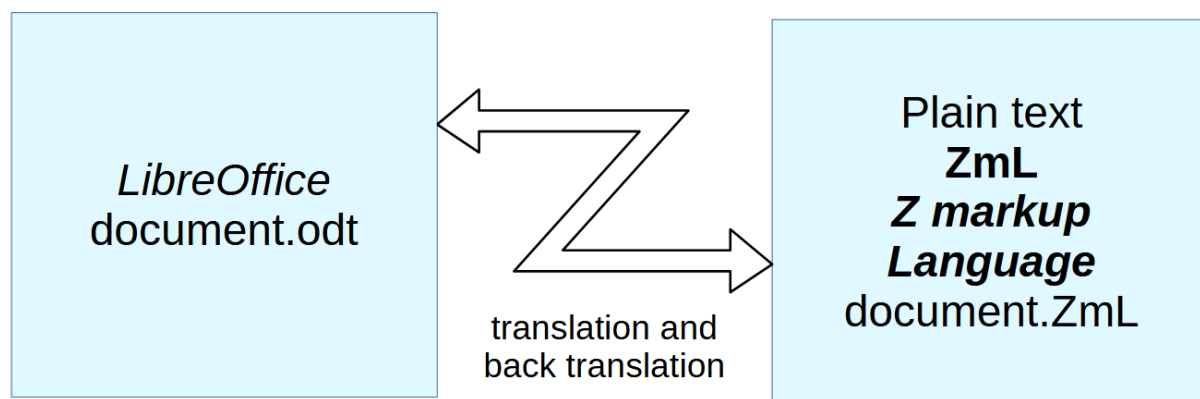


Figure 1: LOffcZmLOverview.png

There are two approaches to work with documents:

- Using an Office tool like LibreOffice, working with “*what you see is what you get*”
- Working with a textual markup language such as Latex or Markdown or AsciiDoc, generate the document immediately as possible and look for outfit.

On simple small documents for daily using an Office tool with wysiwig (“*what you see is what you get*”) is of course the best choice. But for large technical or commercial documents often using a plain text markup language is familiar and has advantages.

The here offered tool converts between LibreOffice (its `content.xml`) and a specific textual markup language. Additional features are for example

- Supplementing correct HTML links to Javadoc generated parts (or also Doxygen possible)
- Insertion of parts of code original from sources
- Divide the document in more independent parts while working, and join the Document at least, or also built different documents with same selected content. It’s a simple content management system.

The advantage working in the plain text is: “*What you see is what you have*”. Conversion between both sources can be done currently while working. Change one side, generate the other side, continue working on the other side. Change the page layout, insert images in LibreOffice. Adjust links in the plain text. Change code snippets in the plain text.

Why a specific markup language: Because the common known candidates don’t support the necessities. The Z markup Language is well structured (better than AsciiDoc) and simple (better than LaTeX)..

The name “ZmL” may mean the *last ultimate* Markup language. Unfortunately “VML” is already in use for “Vector Markup Language” and “Yet another ...” is used for YAML. I have had start my software live with the Z80 processor, this Z is in the brain.

https://vishia.org/LibreOffc/html/Videos_LOffcZmL.html

Table of Contents

1 Approaches.....	4
1.1 LibreOffice beside the plain text of content.....	5
1.2 Why another markup format instead and beside AsciiDoc?.....	6
1.3 Using only indirect styles.....	7
1.4 What you see is what you have.....	8
1.5 Working in the document in LibreOffice and similar in ZmL.....	8
1.6 Software docu including generated docu and source code.....	9

1.1 LibreOffice beside the plain text of content

LibreOffice and (for example) AsciiDoc are two very different approaches to write (technical) documentation. Both have advantages and disadvantages. One intention to use AsciiDoc and LibreOffice parallel for the same document is: LibreOffice has the disadvantage that “*what you see is **what you have***” is not true. It follows the known approach “*What you see is what you get*”, but some stuff is hidden which should be more obviously – The advantage of AsciiDoc as also all other textual markup languages is: You see what you have. For example specific formats (styles) with its names, exact written relative link, etc. AsciiDoc is a source format, it is a plain text without hidden stuff.

But AsciiDoc has unfortunately a specific 'grown' syntax and cannot present all necessities of a well documentation. Therefore, a slightly different way was gone: AsciiDoc is not used, instead a specific here defined markup format is used as counterpart for LibreOffice presentation. AsciiDoc is generated ready to use for HTML output generation too. But editing in the plain text should be done with the specific ZmL markup format.

The substantial approach for using LibreOffice additional to the textual markup language is: It is proper for page oriented documents. Such documents can show

technical things in a standard-two-page view on the current familiar large monitors, inclusively well positioned figures as explanation. It is better than the linear scrolling HTML view. But both may be necessary, a documentation should be available in both formats.

The only generate PDF from for example AsciiDoc or familiar also from LaTeX was not satisfying, the PDF converter is not proper able to control, in my mind. Editing the documentation in both formats, markup and in LibreOffice is an advantage. For “*What you see is what you **have***” - content use the plain text markup, for appearance of the PDF document view use LibreOffice. Plain text markup format has also the advantage of comparability to older versions.

This tool converts LibreOffice.odt files to ZmL PlainText.zml (and also to AsciiDoc.adoc) and from ZmL back again to LibreOffice.odt.

A side effect is: On back generation of LibreOffice from the plain text, a lot of junk which may be grown in the XML data is removed. This junk comes internally from used and remove again direct formatted text parts. But in conclusion, all exclusive some special direct formatting information are removed. They are not supported and not desired. See chapter **1.3 Using only indirect styles** page 7.

1.2 Why another markup format instead and beside AsciiDoc?

The basically idea is, that LibreOffice is supplemented with a Markup language in plain textual form to see all “*What you see is what you have*” with all internals. There are several markup formats, see [https://en.wikipedia.org/wiki/Markup language](https://en.wikipedia.org/wiki/Markup_language) or also in German: <https://de.wikipedia.org/wiki/Auszeichnungssprache>. There are some considerations to the markup language:

- Procedural markup: The markup contains statements how to print or render. Latex is for example partially a procedural markup. The principle is: Say what to do with the following text. For example Take an italic font with given name, then continue rendering.
- Descriptive markup: The markup describes the properties of parts of the text near the text itself or including the text. If the properties can be semantically oriented. It means, a text part is marked as “Quotation” because it is a quotation. Using an italic font is controlled by the style. The most known descriptive markup is **HTML** (*Hyper Text Markup Language*). The styles are placed in the associated CSS script (Cascading Style Sheet).

Also some proven markup languages exist. Latex should be known, also MD (*MarkDown*

<https://en.wikipedia.org/wiki/Markdown>).

AsciiDoc is frequently used for software documentation, also because of the advantage, that AsciiDoc can immediately include code snippets from sources in the documentation. But exactly this is solved a little bit abbreviating, see chapter **3.7.5 Include of code snippets from sources** page **29**. But nevertheless AsciiDoc is selected firstly, also because AsciiDoc was and is frequently used by me beside LibreOffice in the past.

LibreOffice is internally also stored as markup language, it is named “representational markup”. It means outside the user see the presentation (*wysiwyg*), inside all data are contained similar as a descriptive markup. This are the internal structure definition in the xml files `content.xml` and `style.xml` inside an `libreOffice.odt` file.

AsciiDoc is by itself a little bit confuse in selecting formatting text. That’s why some discussions and also adaptations are made here.

To get a proper plain text editable markup format, which follows the structure of LibreOffice or a really proper text system, and it is also simple and widely compatible to AsciiDoc, an own markup system, named ZmL (Vishia Markup Language) is developed and used. See chapter **3 Z markup Language** on page **22**

1.3 Using only indirect styles

Writing a document with any office tool, you are inclined to use the simple direct formatting possibilities. Make a paragraph a little bit lesser meaningful, oh use italic font with a little bit lesser size, looks nice. All the office tools supports both, indirect and direct formatting. What is indirect formatting: Using style sheets. With style sheets, you can associate to any paragraph or text area a meaning. Not immediately the output format. The style sheets give the text also a semantic. A `Quotation` as style marks a text first as quotation, not “*print italic*”. Then you can set proper outfits to all used styles, and the whole text is proper changed if the necessary outfit should be changed.

If you want to express an additional info, lesser meaningful, you can use a style `AddInfo`. Or if you want to express a snippet from code, you can use a style `codeJava` or on another part `codeC++` instead assign a monospace font with any size and maybe a back color. It is similar a semantic label to this part of text, it defines what it is.

Then you can set the style in the *Styles* side bar, define it, or use it from another document, and give it a proper outfit, the desired font, appearance etc. Then the appearance of that text parts are all equal in the text, and that is it, what is necessary.

Using indirect styles is a very old and proven technology. It is present in Word for

DOS since the 1990th, present of course also in LibreOffice since beginning, present for HTML in the CSS style sheets, and such text writer systems as Latex and also AsciiDoc works exclusively with indirect styles.

You can see the very familiar *italic* or *bold* also as indirect styles with this names, or also translate it to the indirect character styles `Quotation` and `Emphasis` which has a semantic meaning.

Direct styles complicates a document. All office tools support direct styles, because there are a lot of users, which do not write important documents, they write for there own and does not know the indirect style usage and advantage. But all professional document writer should only use indirect styles.

In (all) Office tools because of too much direct styles which were used, then deleted etc. there is a lot of data in the document which are meanwhile nonsense. On translation from LibreOffice to the plain markup text ZmL and also AsciiDoc this nonsense direct style entries but also all other direct formatting styles are ignored, except a few important ones. Translated back to LibreOffice this direct styles are removed. It is a cleanup process which may be important on dealing with large documents.

1.4 What you see is what you have

This is a very important saying, but not in all brain. The all known “*What you see is what you get*” instead is very known and it hides the view to “***what you have***”.

For example, links in the document. What you see in the text is: The text to the link, not the link itself. If you open the link (*Insert - Hyperlink*), then you do not see the real used link in LibreOffice, you see the absolute file position. (It is a RFC, Request for change, internally in LibreOffice [Bug 128216](#) to see also the relative path.

If you want to change some more relative paths in its start point, because your directory tree is a little bit changed, then in the plain text source file `Plaintext.vml.adoc` you can relative simple use *search and replace* to gather and change all. In

LibreOffice you must painstaking open each link, with the mouse, think what is happen etc. pp. And if you have your result-pdf, you may get bad surprises — LibreOffice may work exact. But you may make some mistakes while the painstaking work, then do it again.

Just holding the sources in both forms, as a `LibreOffice.odt` and also as `Plaintext.vml.adoc` with the same content, you can do the work where it can be done better. Improve your page layout and format text content in LibreOffice, and correct links, sections, Overview over chapters, images in the `Plaintext.vml.adoc`. After finish work in one file, you should only start the conversion to the other one, which needs about one second.

1.5 Working in the document in LibreOffice and similar in ZmL

The advantage of both tools can only be used if you can work in both for editing.

For that, a converter is provided, which converts either `LibreOffice.odt` File format to `Plaintext.vml.adoc` and vice versa. Then after converting you have both, can look and further work with both. But if you edit one of them, you should convert to the other format to use both and can edit furthermore with both.

In `Plaintext.vml.adoc` you have the advantage to compare simply the files, to see what is changed. In that manner also

an editing by mistake of an older version can be fixed. But you should look anyway to have both file formats in the currently version.

Of course, if you can use both editing approaches, and converting the documents, you can only use an intersection set of capabilities of both formats. But this intersection set has enough capabilities.

This intersection set is discussed in chapter ***2.4 Using a real small set of format styles and less direct formatting.***

1.6 Software docu including generated docu and source code

An important application for documentation is the software, how does it works, but also explaining the concepts. For that, automatic generated documentation from software tools should be used to ensure correctness (immediately generated from the same software sources which are used for the software application). Software develop-

ment tools have capabilities to generate docu.

But this docu is not satisfying to understand concepts, understand why it is implemented in that or that form. The software documentation should be supplemented by handwritten documentation. This is the task for LibreOffice.

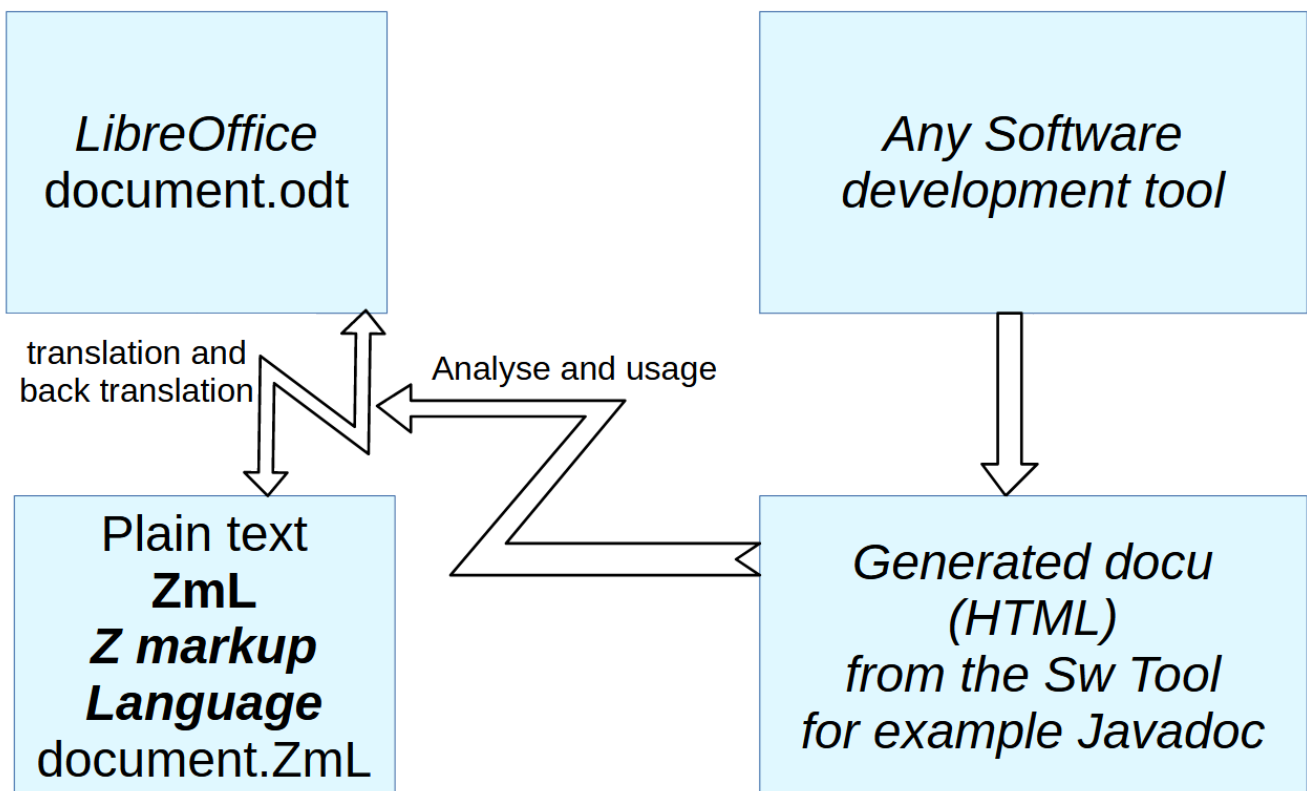


Figure 2: LoffcZmlJavadoc.png

Specifically for Javadoc (but adaptable also for other sources, using the concept) some features are introduced in the ZmL /

LibreOffice translation tool to work with the sophisticated link labels (targets in the generated Javadoc-HTML).

2 Some decisions how to write a technical documentation

Table of Contents

2 Some decisions how to write a technical documentation.....	10
2.1 Using double page view in book mode.....	10
2.2 Writing style in columns for each (sub) chapter.....	11
2.3 Manual column or page breaks and positions of images.....	11
2.3.1 Page breaks and reserve space on page end.....	11
2.3.2 How to insert a page or column break.....	11
2.3.3 Position of images.....	13
2.4 Using a real small set of format styles and less direct formatting.....	14
2.4.1 Is a free styled document design proper?.....	14
2.4.2 List appearances.....	14
2.4.3 Code snippets also possible to include from sources.....	15
2.4.4 Character styles.....	16
2.5 Character set and special characters.....	17
2.6 Internal links, bookmarks.....	18
2.7 External links to Javadoc local files and the internet.....	19
2.7.1 Software documentation.....	19
2.7.2 Relative local links and supplement www link with same path.....	19
2.7.3 Supplement argument types of intern operation links (anchors in html).....	20
2.8 Exchange and maintain the styles of the document.....	21

2.1 Using double page view in book mode

The advantage of a PDF view instead HTML is beside the fact, that the printed version is equal the screen version: You can have more overview. Using a double page view is usual possible on normal (1980x1080) and larger monitors. If you use additional the 'non continues page view' then you can scroll with full pages, the pages have always a fix orientation as reading a book. Then it is interesting that repeated presentations starts always on the same position on page, for example tables with technical information always after some explanations on top of the page, or better on top on the left page. Then you can very more better search with your eyes for information, which's designation is currently not in your brain (not possible to use ctrl-F for searching, you does not know how it's named in the moment, but you know related information about). That is human thinking approach.

For that it is important to regard the rule since Gutenberg's time: The **even page is left**, and the **odd page is right**. The **title** should be **page 1 on right side**.

But the title page is page 1. How does it work in LibreOffice: The style `Tit1e` has the following setting:

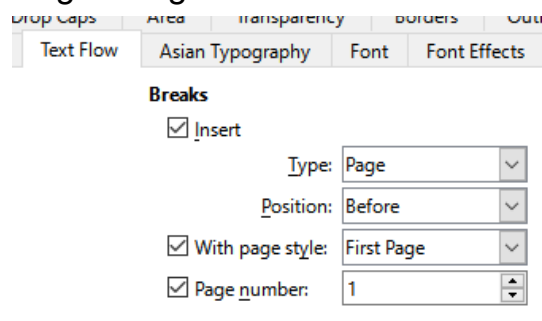


Figure 3: TitlePage1.png

But unfortunately some PDF viewer especially on browsers does not support this "book mode". That's why also this document is converted to PDF in two kinds: In book mode and in "stupid browser left side mode" with an fist empty page

2.2 Writing style in columns for each (sub) chapter

The first what should be obvious for this document is: It is written in columns. Reading in columns has the advantage, that the eye of the reader can capture the text in a vertical movement. Because the lines are not too long. You can fast capture the content, for example while searching a catchword. This style was familiar in the old years of printed documents, for example in encyclopedias. but it was forgotten in a time of html browser for first small screens. Now here the idea is recovered.

And a second advantage is: The column width is also proper for reading a pdf on a smart phone.

But in difference to the familiar column style in news papers, the columns does not go over the whole page from top to down, they are regular broken on each new chapter title, and also on an image or figure which needs the page width.

LibreOffice does support writing in columns, but for editing some times a little bit difficult to handle. But editing parallel in `vm1.adoc` it is a little bit more supported.

2.3 Manual column or page breaks and positions of images

2.3.1 Page breaks and reserve space on page end

The reason or intention for page breaks is: You want to present a closed content on two pages side by side. Whereas in book view mode or for the printed document the left page has the even number and the right page has the odd one. This is important. Normally pdf viewer should be able to set for this mode.

This means also, a new chapter should start on top of an even page to have two pages side by side for the overview. But in opposite of this rule, it is often

recommended to start a new main chapter on right side in a book, the same side as the title was written. This is a little bit contradictory. Never you should start an important new chapter near the end of the right (odd) page, that is stupid. Means, insert manually a page break before.

Using page breaks on proper positions in the text helps also that the page disposition is not sensitive to confuse on each small text changes. You have some space before page breaks.

2.3.2 How to insert a page or column break

Traditionally it seems to be proper to insert manually a page break with Menu “Insert – Page break” or “Insert – More Breaks”, and then select “Column break”. But effectively, the property of breaking the page is a property of the paragraph format. It means following the idea of direct formatting, to to the “Format – Paragraph”, then select the tab “Text flow”, then check the “Insert” box with Type: “Page” or “Column” and use Position: “Before”, that is proper. Then you get a column or page break before this paragraph. And this is usual what you want. The same is done internally if you use the Menu “Insert – Page break”, but the paragraph is split on the cursor position, some times unexpected. Using the “Format – Paragraph” is more simple.

In the plain text ZmL presentation the page or column break is written in a line before the paragraph as

```
<:pageBreak.>
```

This forces on back conversion to LibreOffice exact the above described behavior, a paragraph style with Insert Page break or column break before. It is compatible and obviously.

To simplify a page break in the current text, you should use the style `TextPg` Or `TextCol` instead `Text`. This is also done automatically by back conversion.

2.3.3 Position of images

Traditional often images are positioned as possible, depending of the page formatting. For example Latex has its own free style to positioning images on a proper position in meaning of the Latex rendering, not in meaning of the user.

But familiar in html (often used for technical documentation), images are always exact positioned in the text flow. For technical documentation this may be important to have the images closed to its explanation. Look her for example right side, there is the image spoken above.

That's why this converting system between LibreOffice and the plain ZmL text, which has a natural closed relationship between text and image because of their sequence in the plain text presentation, uses a simple presentation of images in LibreOffice: **Images are always bounded to a paragraph which contains the image caption.** The position is left or right bounded, but with 0 distance to the paragraph. If you move unintentionally the image in LibreOffice, go to its properties (right mouse) and entry 0.0 for its position, and the image should be proper again.

But there may be sometimes a problem: If the LibreOffice rendering for the page would insert a page break or column break at an inappropriate place near the image, then you should insert a manual page break or column break before the paragraph to which the image is bound.

The figure right above is inserted with this shown style `ImgCaptionTextCol`. It forces a column break before, as also see in the image.

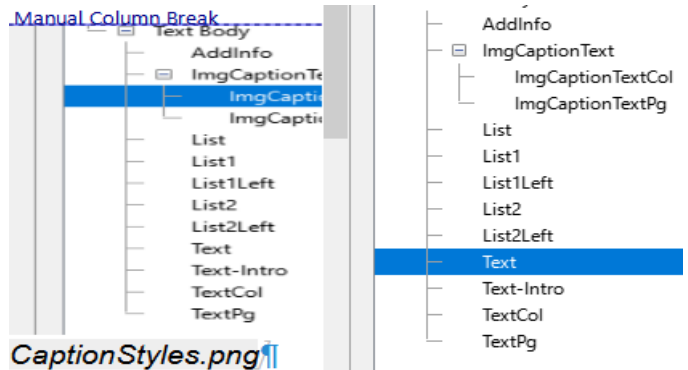


Figure 4: ImageCaptionStyles.png

But you should taken care about the new column, it should be more filled that the column before, elsewhere the rendering may fill the columns in an equal kind and inserts an unexpected new column here.

This handling should be done usual in the LibreOffice presentation. Then you see the image close to the text in the plain text ZmL presentation too. If you edit their, be careful with page breaks and column breaks, which are proper syntactically able to see in the plain text ZmL, and all is proper.

The numbering of the images is done by the ZmL to LibreOffice converting, not from LibreOffice itself. See chapter **3.8 Images** page **31**, because the `ImgCaptionText...` cannot support numbering. The behavior of tables is similar to that of images Make sure that the page and column breaks are proper.

LibreOffice has a little bit trouble if images are shifted with the mouse. This is a concession to the user who wants to have free mouse positioning, but exact this breaks the relationship between text and images. A second problem is handle image captions in a text box, with two sources of positioning errors.

2.4 Using a real small set of format styles and less direct formatting

Think about the proven rule “less is more”.

2.4.1 Is a free styled document design proper?

If you are concentrate to text writing with an office tool, you may be triggered to use a lot of nice styles for free possibilities for your design.

But also for AsciiDoc, html, most of other Markup sources, the style of the currently text is only one. It has not a specific style dedication in the ZmL (and also not in the

other markup languages), it's only text. The converter from ZmL to LibreOffice takes the Paragraph style `Text` (not `Text Body`) This style should be used and defined in LibreOffice. The rule “*less is more*” produce a more relaxed design, concentrate to content, not to unnecessary “*nice to have*” but real not useful appearances.

2.4.2 List appearances

For Lists, usual the given list styles are used, which have the possibility to select different bullets etc. But especially the bullets can be also part of the text itself. This opens the opportunity to use a context related bullet, which can be also a specific text. “*Numbered list*” with an auto incremented number in different styles are proper for common articles. But for exact presentation of technical things, the number should be related to the text, should not be automatically incremented and hence changed if a new list item is added. That's why using a numbered list is not recommended in my mind. Write the bullet appearance by yourself.

All markup languages supports a numbered and an unnumbered list, also ZmL by using the familiar from AsciiDoc known writing style

* `list item`

- `list item` using this appearance of the list style in LibreOffice as non numbered list. You can adapt it.

But it may be recommended not using a list style for lists, instead a specific paragraph style with the necessary indentation, and write the bullet manually. This styles can be defined with any specific user name in your

document, but the recommended style names are:

- `List1`: The bullet point is manually set in LibreOffice, usual copy via clipboard. It is contained in the standard UTF coded character set. In the ZmL it is mapped to `**` proper able to read and write.

Use a simple tab left side to continue with a next paragraph to this list item.

The indentation here is 24 pt, but the first line has -18 pt (6 pt from left) in this document, it is able to adapt. Use a tab character after the bullet.

- a) `List2`: And this is a manual written ‘*bullet*’ which can be used in the further text as manual link (hint, marker). The deeper indented list has here 48 pt from left, and the first line -18 pt, which is 30 pt from left.

- `List1Left`: This is a list which is proper for shortage of space for the line in a column. The indentation is 0, but the first line is 6 pt and the tab is placed on 24 pt.

- b) `List2Left`: But also this kind of List exists for the space saving writing style.

`List1`, `List2`, `List1Left` and `List2Left` are paragraph style names in this document. copy or create this styles, but use this

names. In this kind some more specific paragraph styles can be used. In ZmL there are written in form:

```
<List1>\**\t <:cStyle:List1.> The bullet ...
```

The format style is given first in the paragraph line. The bullet can be coded in UTF-8 in the text or used with the given subscription.

\t with one following space presents the tabulator character. The specific text style is written with <:cStyle:List1.>, whereby cStyle

2.4.3 Code snippets also possible to include from sources

See also the ZmL definitions in **3.7 Code snippets** page 28

As also able to see above, Code snippets are often used in a technical software documentation. The important feature is: The lines shouldn't be wrapped.

That's why the source text should be limited in text line width. For readability this is usual proper, because this code snippets are only snippets for illustration, and not the complete code ready to copy. For such use the original source files. Read and edit the sources in the proper IDE (Integrated Development Environment) to work with it!

But for that, of course, the code snippets should have, as often usual, a monospace font. And also in the paragraph style the check box *"Do not add spaces between paragraphs of the same style"* should be activated. Which Code paragraph styles should be given: This depends of your requirements. You can define it by your own. The default LibreOffice paragraph style `code` should be used for common. But for specific languages some styles below Code in the Hierarchy should be given:

- `CodeCmd`: for common command lines
- `CodeScript` for common scripts. The designation is short.
- `CodeCpp` or `CodeJava` for programming languages

is the character style name in LibreOffice, see next.

For using AsciiDoc this is translated to:

```
[.List1]
• [cStyle]`List1`: The bullet point
```

The possibility of `[.List1]` in AsciiDoc creates in HTML:

```
<div class="paragraph List1"><p>...</p></div>
```

It means it is presented by a so named division, which has a specific paragraph appearance controlled by the CSS script (Cascade Style Sheets for HTML).

- `CodeVMU` and `CodeAdoc` especially for this document.

The font styles may be identically, or different in bold or italic, for your own. Usual the background color should be selected for recognizing the language. This is not so proper but acceptable for a white/gray/black printed document, but proper for a pdf viewer. Use a pastel color for the background.

In AsciiDoc code blocks are written as:

```
[Source:cpp]
----
float y = sqrtf(y); //Copied from source
return y;          //(1)
----
```

In ZmL the code snippets are presented as

```
<:Code:Cpp>
float y = sqrtf(y); //Copied from source
return y;          //<:cM:(ret).>
<.Code>
```

This appears as:

```
float y = sqrtf(y); //Copied from source
return y;          //(ret)
```

ZmL allows designations of specific characters introduced with the back slash, especially `\\` for the back slash itself, as also in other texts. It supports also character styles, which are written as `<:style:content.>`. Hence the code blocks can also be styled. Especially marker, here the `(ret)` helps for explaining the code in the text.

Including sources:

But it is an effort to copy source content to the documentation and correct them, if the sources are changed. Often this will be forgotten. AsciiDoc supports copying the content from referenced sources during HTML generation in the HTML, as very interesting feature for software documentation. Also, in HTML this code is shown in a sub window with a horizontal slider. But this is never usable in a printed document where LibreOffice is source of, and it's also not possible in LibreOffice, its only for HTML view.

In VmL and LibreOffice there is a similar way. A ZmL script can contain:

```
<:Code:Java>
<:include:../path/to/src.c::label::43.>
<.Code>
```

this is similar as in AsciiDoc where it is written:

```
----
include:../path/to/src.c[tag=label]
----
```

2.4.4 Character styles

The usual used bold, italic, underlined, and also subscript and superscript character style dedications are all direct styles. If you press *ctrl+M* (“*Format – Clear Direct Formatting*”) which may be sometimes necessary, this designations are removed. Instead you should always use indirect character styles for that. All of this possibilities are given with the indirect styles.

But for compatibility and fast writing using the known hot keys as *ctrl+I* for Italic, the detected direct styles in a LibreOffice document are automatically translated to the necessary indirect style in the ZmL plain text. While back conversion to LibreOffice you get the indirect formatting in LibreOffice for further working. You should know that, you have not effort, and you have the possibility to change the

But VmL has additional features, see **3.7 Code snippets28** page, also for markers. And it shorts the line (because the sub window with slider is not applicable).

But, of course, the sources should be prepared for this including:

- The lines should contain the important expressive content in the left part of the line till the expected truncate position. Note that the code snippets in the documentation are not source code ready to copy, it is only for orientation in the sources.
- The sources need marker where areas for copying begins and ends. It means they should be written regarding documentation necessities. But this may be a problem if different teams are responsible for the sources and documentation. Last is often adjusted afterwards, but the sources should not be changed if the software version is finished. Fine tuning may be blocked.

For the last point there are some possibilities for markers and omitted lines in the ZmL include line. See **3.7 Code snippets28** page.

appearance for all marked texts in a unique kind.

The per default used character styles for the replacement of the direct styles are:

- **Quotation** (*Standard style in LibreOffice*) instead italic direct style, `<:q:text.>` in ZmL, `__text__` in AsciiDoc.
- **Strong Emphasis** (**Standard style in LibreOffice**) instead bold direct style, `<:s:text.>` in ZmL, `**text**` in AsciiDoc.
- **Emphasis** (**Standard style in LibreOffice**) instead italic bold direct style. This is the standard appearance of this style, `<:E:text.>` in ZmL, `__**text**__` in AsciiDoc.
- **Subscript** for Indices (user defined style in LibreOffice) instead subscript direct

style, `<:1:text.>` in ZmL, `~text~` in AsciiDoc.

- **Superscript** for ^{Indices} (user defined style in LibreOffice) instead superscript direct style, `<:2:text.>` in ZmL, `^text^` in AsciiDoc.

For this character styles which should only influence this given properties of the text, and not the font size etc. LibreOffice works exact, but it hidden its exact working and it is error-sensible. What's happen: If you change the character style and you do not entry a new font or fount size, all is ok. The font and its size is derived from the paragraph style. But if you change the font for this character styles, it is changed. You can never revert it to "derived font". This is a problem of LibreOffice, should be fixed. But it is able to fix by manually handling of the internal `styles.xml` respectively replace a

changed `styles.xml` by a proper one from another document, see **2.8 Exchange and maintain the styles of the document** page 20

Additional you should have all code fonts and backgrounds which are existing as Code paragraph styles also as character styles. This allows refer to code snippets with the same writing style also in your currently texts. This styles should start all with "c", it is necessary for the ZmL conversion. After the "c" for some standard code styles only one character should be written. This is proper (but not necessary) for ZmL. The character styles in ZmL are written generally as `<:style:text.>`. For example it is proper readable and writable: `<:cc:float var; // C/++.>` which appears in LibreOffice as `float var; // C/++;` or `<cM:(M).>` for a marker in a source which appears as `(M)`.

2.5 Character set and special characters

LibreOffice works with all available characters defined in the UTF character set. The ZmL file and its editor uses UTF-8. That is free.

But some characters especially which are not proper readable and writable in the ZmL

file are replaced by transliteration, see chapter **3.12 Transliteration of specific characters** page 40. Also storing the ZmL in Standard 7 bit ASCII is supported but not recommended.

2.6 Internal links, bookmarks

All chapter title should have proper mnemonic bookmarks. They are used for chapter references. The bookmarks or labels are written in the ZmL plain text in form

```
=== chapter title <#chapterLabel>
```

The concept of chapter labels is used in similar as also in AsciiDoc, there written as

```
[chapterLabel]
=== chapter title
```

The `chapterLabel` is generated also to the HTML document as anchor usable in the URL. In LibreOffice it is a bookmark.

References to internal bookmarks are written in ZmL in form `<:@ref:#ChapterLabel:3.8 Title>` but the Title and its number is not used for back translation to LibreOffice, it is automatically created there. It is an information in ZmL after translation from LibreOffice.

Links to the file system should be used in LibreOffice generally as relative links. They are proper able to see and editable in ZmL, for example if the relations in the relative paths are a little bit tuned. This is one of the important advantage of ZmL. See chapter 3.x TODO.

2.7 External links to Javadoc local files and the internet

This is a special approach for documentation of Java programs, but it can be applied similar also to other programming languages. The topic is: The software should be documented. Yea.

2.7.1 Software documentation

But how? There are three levels:

- The inner level is the software itself. Should lines be commented? Clear programming says “no”, users who study old software says “yes please, why this statement ...?”. The problem on immediately software documentation is: It is often not maintained if the software is changed. That’s why some people speak about “clear programming” and require that the identifier, the names of variables, classes, operations are proper. That is right. But without any comment ...?

- In Java it’s familiar to have a documentation also in the source, for each element of a class (variables, operations) and for the classes. This documentation should be, and is written in a common understandable style, and from this information in the source a usual HTML document is generated. This is Javadoc. It’s nice, it is familiar since many decades. For other languages often Doxygen is used in the similar kind. Also from some tools (UML) some docu is automatic generated.

The Javadoc itself may be satisfying, is it? It is satisfying to explain usage of a sub class or a specific operation. Is it not satisfying explaining a tool written in Java?

- The next level is, explain how the tool in Java works. How the software works. This may need graphical overview, maybe using UML tools, or at least an explanation about functions and operations from the user’s view. This is the minimum.

And now it is nice to have to link from this explaining document to the Javadoc generated stuff, because both supplements

each other. Javadoc contains the exact description of an operations, or of a class, all what this does or contains, but it does not explain how and why to use. For that the here written LibreOffice may be responsible to. See also chapter 6 Internals page 48 in this document. This may be an example how to explain the software.

And now the request for the documentation:

2.7.2 Relative local links and supplement www link with same path

- First, it should be possible to work without internet. Yes! Presumed the Javadoc from your own sources is side by side to the document, or the Javadoc from a used source can be downloaded one time locally via zip, unzipped and places side by side to the downloaded pdf document. Then you can work without internet. Later, for a user, links to the internet should be also present, but optional, see following.

If you open a pdf document in the browser, not downloaded, it is also sometimes possible to open a relative link in the document without problems if the destination of this relative link is on the same location in the internet.

But sometimes, the relative linked destination, for Javadoc or other, is not available. That’s why I **place a relative link and a link to the proper internet location side by side** in form: See [WriteOdt.main\(...\)](#) ([www](#)). Both refer the same content, local and in internet. The local link may contain the class and operation, as shown here, the www does not need its repetition.

3.11.2 Relative local links and supplement www link with same path How does it work – see 38page

2.7.3 Supplement argument types of intern operation links (anchors in html)

The problem is, that the link to an operation in Javadoc with a lot of arguments is sophisticated. And if you change only one argument type, or add one argument more, while software development, the link gets faulty. You should search and adapt it painful and expensively in your documentation.

This work can be done automatically and is done by the ZmL / LibreOffice translator: The destination HTML file is read, all existing anchors are detected. With them an internal index container is created with

the key: Name of the operation, and value: complete anchor. The problem of the overridden operations (different operations with the same name, but different argument types) is regarded in that kind, that non unique operation names are detected. Only unique operation names are supported by this simple replacement. But that is often sufficient. Overridden operations should not be used too extensive for simple approaches, only if the operations are really similar and only distinguished by argument details.

3.11.3 Supplement argument types of intern operation links (anchors in html)

How does it work – see page .

2.8 Exchange and maintain the styles of the document

It is planned that also the styles.xml inside the odg document should be convert to a plain text presentation and back again. If this is done, also the styles can be edited in the plain text and compare for versions.

Problem is that also too much styles are presented in the style list, sometimes irritating, sometimes useless styles, which can be set to "Hidden Styles" (cannot be removed).

The second problem is, some properties of styles are really derived, but they are not shown as derived. And this is an important thing.

It is possible to open the odg document as zip, copy the `styles.xml`, view it, edit it if you are familiar with the XML content, replace it. But make a save copy before and check whether all is ok after.

(empty page)

3 Z markup Language

Table of Contents

3 Z markup Language.....	23
3.1 Basic Considerations.....	23
3.1.1 Plain source text.....	23
3.1.2 Comment lines.....	24
3.1.3 Section, chapter and paragraph structure near AsciiDoc.....	24
3.1.4 Text structure (syntax) similar AsciiDoc but other designations.....	24
3.2 Syntax overview by examples.....	25
3.2.1 Third level chapter.....	26
3.3 Chapter designation and content.....	27
3.4 Writing style of paragraphs.....	27
3.5 Sections.....	27
3.6 Lists.....	27
3.7 Code snippets.....	28
3.7.1 Syntax and styles.....	28
3.7.2 Shortened code lines.....	28
3.7.3 Lines can have character styles.....	28
3.7.4 Code lines can contain special characters.....	28
3.7.5 Include of code snippets from sources.....	29
3.7.6 Include code snippets with labels and off/on in include line, Syntax of the code include line.....	30
3.8 Images.....	31
3.8.1 Some remarks to size of images.....	32
3.9 Possibility of include and dispersion.....	34
3.10 Cross references inside the document, how to deal with interrelated documents... 35	
3.10.1 Use a proper name for bookmark labels.....	35
3.10.2 How to write a reference.....	35
3.10.3 Cross reference to other documents of the same suite.....	36
3.11 Hyperlinks and Hyperlink anchor completion.....	38
3.11.1 Simple Hyperlinks with or without target to the internet or locally.....	38
3.11.2 Relative local links and supplement www link with same path.....	38
3.11.3 Supplement argument types of intern operation links (anchors in html).....	38
3.12 Transliteration of specific characters.....	40
3.13 Using Character styles, semantic text span.....	41

3.1 Basic Considerations

3.1.1 Plain source text

The important basic consideration is: It should be based on plain source text. The encoding should be UTF-8 to support also rarely letter also in LibreOffice without transliteration. But for exception situations (using an old editor) also US-ASCII or ISO-8859-x (8 bit width character coding) should be possible.

Some special non visible characters should be transliterated, see list in chapter **3.12 Transliteration of specific characters** page 40

There are only a few character sequences which controls the structure. Outside of paragraph texts there are more possibilities, see **3.1.3 Section, chapter and paragraph structure near AsciiDoc** Inside a consecutive text (in a paragraph) only `<:xxx.>` and the transliteration with `\x` is used. This prevents confusion with text parts as in AsciiDoc, the known AsciiDoc's `pass:[text]` for a non interpreted text and its also confusing abbreviation `+text+` is not necessary. See **3.1.4 Text structure (syntax) similar AsciiDoc but other designations** page 23.

3.1.2 Comment lines

```
// Comment
<:Comment:marker>
  block comment
<:Comment:marker>
```

A line starting with `//` is ignored, but not inside a code block. This is also true inside lines of a paragraph. The comment line does not break the paragraph block.

The block comment can be used also to disable blocks of text. Nesting is allowed (TODO?).

Comment lines cannot be transferred to the odt document and back again. That's way they are not really able to use. It is only sensible, if the vml code comes from

another source. Then some things may be seen in this comment lines.

3.1.3 Section, chapter and paragraph structure near AsciiDoc

```
== chapter title <:@ref:#label.>

<:p:style>
paragraph one line per sentence
or broken inside.

Next paragraph in standard style.

* A list
** With sub items

<:Section:style>

paragraph in section, maybe in columns.

=== sub chapter in section <:#label2.>

<.Section>
```

Sections are parts of the document containing paragraphs and also complete chapters, which have a specific format. Especially this is used for writing in columns. Also have a specific background color for parts of the document is possible.

The chapter and paragraph structure is basically similar in AsciiDoc, Mark Down, Wikipedia text. Here the basically chapter and paragraph structure of AsciiDoc is used, with some specifics. It means:

3.1.4 Text structure (syntax) similar AsciiDoc but other designations

Inside a paragraph text and also all other texts (list items, chapter title etc) normal text is written as is. The UTF-8 coding allows using also rarely specific characters. Only specific character are transcribed, which are not able to show in normal text coding. That are for example the non breaking space, UTF-16 coding `\u00A0`, written as `\` in ZmL. See chapter **3.12 Transliteration of specific characters** page 40

All character designations uses character styles of LibreOffice. But there are some shortcuts for the standards, see chapter **3.13 Using Character styles, semantic text span** at page **41**. The general solution is writing in the text:

```
...text text1
```

There is only this one control sequence `<: .>`. This is very more simple and obviously than the many specific

designations in AsciiDoc and some other markup language which can conflict with the normal text. Hence the known AsciiDoc's `pass:[text]` for a non interpreted text (with special designation as to write) and its also confusing abbreviation `+text+` is not necessary. To write a itself in the non styled text in ZmL you should transcribe it with `<\:.` and `.\>`, for example to explain ZmL itself. Also the `\` can expressed by `\\`. No more is necessary.

3.2 Syntax overview by examples

First note that only indirect formatting is supported. The styles used should all well defined in the LibreOffice file. In VML their names are used.

Second, there are only a few set of control character strings. All control strings have the form `<:>`. The character sequence `<:` in the current text is written as `<\: .`

Additionally there are some specific character substitutions which are maybe not available or invisible in the plain editor's character set, such as `\--` for a long dash or `\+`, `\-` for breaking possibilities (see table right page).

The basic outfit of the ZmL text is similar AsciiDoc, with the advantage using a AsciiDoc editor (for example in Eclipse) to write texts. See following example VmL text:

```
<:p:Title.>Title of the document

<::TOC-1.>Table of Contents

==== 1.1.1 Third level chapter <:#Label1.>

<::Section: Column2>
From here the page as two columns.
The style <:cStyle:column2.> is not a LibreOffice indirect style, it's a specific style.

Paragraph in format-style <:cStyle:text.> as currently text, all lines is one paragraph as in
other Markup languages.
Here character styles are possible, for example <:cJ:inline code snippet from Java language.>
or just
<:cC:#define ClanguageMakro.> or also \'<:I:Quotation\''>, <:E:Emphasis.>, <:S:Strong
Emphasis.>

<:p:List1.>\**\t This is a list with left bullet, but not with the list styles, it is a simple
paragraph style.

* This is a standard list, translation depending on conversion options.

<.Section>
<:Code:C>
void function(int argument) { // <:cM:(1).> This is a marker
    int y; // the lines should be short enough to prevent line break,
    printf("xxx"); // but the lines are not joined as in paragraphs
}<.Code>

You can have references to chapter <:@ref:#Label1:1.1 Second level chapter.>, page <:@page:3.>
or
also to an external link <:@link:https://vishia.org/index.html::Webpage vishia.org.>.
You can insert an image:

<:@image:../img/L0-AsciiDoc/L0ffcZmLOverview.png :: id=__Img_L0ffcZmLOverview.png_1 ::
title=Figure 2: L0ffcZmLOverview.png :: style=ImageCenter :: size=8.0cm*2.52cm ::
px=1435*454 :: DPI = 456.>

<::Section: Column2>
```

This appears as:

...Title of the document (***formatted as title***)

3.2.1 Third level chapter

From here the page as two columns. The style `column2` is not a LibreOffice indirect style, it's a specific style.

Paragraph in format-style `text` as currently `text`, all lines is one paragraph as in other Markup languages. Here character styles are possible, for example `inline code snippet from Java language` or just `#define`

```
void function(int argument) { // (1) This is a marker
    int y; // the lines should be short enough to prevent line break,
    printf("xxx"); // but the lines are not joined as in paragraphs
```

You can have references to chapter **3.2.1 Third level chapter**, page **26** or also to an external link [Webpage vishia.org](http://Webpage.vishia.org). You can insert an image:

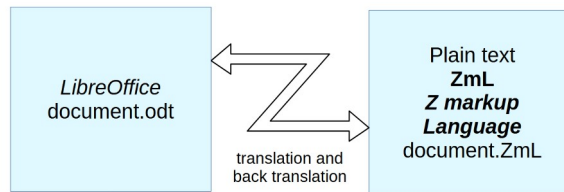


Figure 5: LOfcZmLOverview.png

List of character replacement:

`<\`: replacement for `<`: to prevent confusion

`<.\`: replacement for `<.`

`.\>`: replacement for `.\>`

`\---` — full width dash (FF0D)

`\--` – width hyphen - (2013)

`\` Non breaking space (00A0)

`\:` Small space width as dot (2008)

`\|` No width optional break (200D)

`\~` Soft hyphen (00AD)

`\-` Non breaking hyphen(2011)

`\+` Word joiner, no break here (2060)

`\n` line break (000A)

`\t` Tabulator (0009)

`_` Double underliner (2017)

`\<<` « left pointing guillemet (00AB)

`\>>` » right pointing guillemet (00BB)

`\<` < single left angle quotation (2039)

`\>` > single right angle quot (203A)

`\"` “ Left quotation language specific (en:2018, ge: 201E)

`\''` ” Right quotation language specific (en:201C, ge: 2018)

`\, ,` „ Low left quotation (201E)

`\,` , low single quotation mark (201A)

`\'` ‘ high single quotation language specific (en: 201B)

`\^` ’ right side single quotation (2019)

`*` • small Bullet point (2022)

`**` ● Bullet point (25CF)

`*>` ▶ Triangle bullet point (2023)

3.3 Chapter designation and content

A chapter title line starts with

```
=== chapter title <:#chapterLabel>.
```

The number of `===` describes the deepness of the chapter. But other than in AsciiDoc a

label for the chapter is given on end in form `<:#label.>`.

A chapter can contain paragraphs, lists, code snippets, images, tables.

3.4 Writing style of paragraphs

A paragraph starts with a new line with an empty line before. All lines below which are not empty and do not start with `*` (for List items) are part of the paragraph. A line separator between is ignored. More as that: It is recommended to write each one sentence in a new line, and also a part of a sentence on a long sentence. The plain source text should not have the necessary of wrapping lines in editor.

A paragraphs can have a specific style. In AsciiDoc this is able to express with `[style]` before the paragraph, builds a `<div class = style>` in HTML. Instead in ZmL it is designated also before the paragraph with `<:p:style.>`.

3.5 Sections

In LibreOffice unfortunately there is no indirect style for sections (2024-06). Hence some sensible section styles are defined in

the ZmL itself, as virtual (de facto, non formal) indirect style. Sections are enclosed by `<:Section:style>` and `<.Section>`.

3.6 Lists

The items of a list starts after a `*` or more `**` also without empty line between, but recommended write an empty line before each list item. The list itself is not specific dedicated. The items builds the list.

3.7 Code snippets

See also **2.4.3 Code snippets also possible to include from sources** page 14

3.7.1 Syntax and styles

Code snippets are written as:

```
<:Code:Language>
Code lines
<.Code>
```

For the `Language` the styles of the `odt` file must have the proper `CodeLanguage` style as paragraph style. Ones of the recommended language styles are

- `Code` for common unspecific code
- `CodeCmd` for operation system commands, file designation etc.
- `CodeScript` for scripts
- `CodeCfg` if you have configuration files to describe.
- `CodeCpp` for C and C++ code
- `CodeJava` for Java code

It is recommended that you should use a mono spaced font in the correspond `odt` file. The paragraph style should not have spaces above and below the paragraph respectively no space between paragraphs of the same style.

3.7.2 Shortened code lines

The code lines should be short enough, so that they are not broken. Usual, the code is only a illustration of part of the explanation, and not a source of copy the code. Hence shorten of lines may be possible. But shorten lines should be marked as such:

```
<:Code:Cpp>
void operationXy( float x, DataType data, ...
    data.x = x;           // This is com...
<.Code>
```

The code snippet above is an example. Because writing in columns, here only 45 character have place, but should be enough for this short presentation. Use the full width (approximately 94 character on A4 page, writing width 18.5 cm, with 10 pt font size) for elaborately code.

3.7.3 Lines can have character styles

This can be used especially for example to show markers as in the example:

```
x += this->b; //a special code (2)
```

Now you can use the same marker:

(2) It adds this value.

to explain the code line. In ZmL this is:

```
<:Code:Cpp>
    x += this->b; //a special code <:cM:(2).>
<.Code>
```

3.7.4 Code lines can contain special characters

This is the reason why a single backslash in the ZmL source code should be written with two backslashes. A simple quotation mark remains a simple quotation mark with ASCII = `0x22`, important for Strings in source code. Whereas the specific left and right quotation marks should be written with `\` and `\'`:

A code line with `\` and "text"

is in ZmL:

```
<:Code:Script>
A code line with \\ and "text"
<.Code>
```

3.7.5 Include of code snippets from sources

It is possible to include code snippets immediately from sources via link (recommended as relative link). This is done if the `odt` file is generated from the given `vm1` file. The next text is the example how this appears in the documentation:

```
include:..\java\org\vishia\odt\readOdt\
WriteOdt.java::main::43

public static void main ( String[] sAr...
  try {
    int exitCode = smain(sArgs, Sy...exit
    System.exit(exitCode);
  } catch (Exception e) {          ...exc
  .....
    System.exit(255);              ...
  }
} //
```

This shows a snippet from the named file. In the real Java source file there are some marker written as:

Java source where the code snippet comes from:

```
public static void main ( ...<:main.>
  try {
    int exitCode = smain(sA...//<:@exit.>
    System.exit(exitCode);
  } catch (Exception e) // <:@exc.>
    System.err.println( ...//<:-main.>
    e.printStackTrace(System.err);
    System.exit(255); //<:+main.>
  }
} //<:main.>
```

The ZmL script contains:

```
<:Code:Java>
<:include:..\java\org\vishia\odt\readOdt\
WriteOdt.java::main::43.>
<.Code>
```

whereby the `<:include...` line is one longer till the ending `.>`. It means, this `<:include...` should be a part of the code block.

How is it controlled which part of the source should be presented?

In the source code, usually in the comment, there should be a tag in the line where the code snippet begins. That is the `<:main.>` in the Java source. The `main` is given as tag

name for the code snippet also in the `<:include...` line in the ZmL script. The source code line with this tag is included as first, but exclusively the tag itself, which should be written right side on the end of the line in comment. If the include should start with the next line after this tag, write instead `<:~main.>`. `main` is the tag name for this example. If you write `$` for the tag name, then the whole file is included. But you can also use `<:-$.>` `<:+$.>` to omit lines, and `<:$.>` and `<:~$.>` to end including.

The end of including the snippet is marked with `<:main.>` respectively `<:~main.>` if this line should not be included also. The 'next' and 'not' variants may be interesting if the source for the snippet cannot have end line comments, as for example Windows batch files. Note, in AsciiDoc you need for the adequate feature always an extra line for the "tag" which inflates code lines. In AsciiDoc `tag::name[]` should be written in one line before start including, and `end::name[]` to end the including block exclusively this tagged line.

If some lines should be omitted, to shorten the documentation, then `<:-main.>` can be written to stop on this line, and `<:+main.>` to continue. On the stop line always an ellipsis line `...` is written for documentation.

Last not least the source code can contain markers. That should be written in the source code as `<:@marker.>` as shown for `<:@exit.>` and `<:@exc.>` in the source example. They appears in the documentation and can be repeated in the text to explain. This is used in this document see 5.1.3 Macro =>ZmL and script `-callOdt2ZmL.bat` and `.callOdt2ZmL.sh` page 42.

But there is another interesting feature. You can write also stop and start including and labels in the `<:include:...`, see next page.

3.7.6 Include code snippets with labels and off/on in include line, Syntax of the code include line

If you look to chapter **5.1.2** *callZmL2odt.bat and callZmL2odt.sh* at page **48** you have an example how to use it. The problem here is: A Windows batch file is given. In a Windows batch file there is no possibility for end line comments, only lines can be commented beginning with `REM` or `::`. Hence it is not possible to write any marker in a active line, nor it is possible to write tags with start and omit in lines which are not comments.

The syntax of the `<:include...` line in ZmL is:

```
<:include:PATH :: TAG [ ::MAXLINELEN [ ::{ LINENR : [ -
| + | MARKER ] ? , } ]].>
```

The syntax in ZBNF writing style (similar EBNF, see www.vishia.org/docuZBNF/sfZbnfMain_en.html) is unique: `[...]` are options, `[... | ...]` is an optional choice, where one option should be selected. `{ ... ? ... }` is a repetition, where the part after `?` is the repeat condition.

As you see, the `MAXLINELEN` is optional, and also the whole third part with `LINENR` and `MARKER`.

MAXLINELEN: This is a numeric value and should be proper to the maximal number of character in the code line. If a mono spaced font is used, this should be determined. But it depends from the font size, from margins and especially from the choice, writing in columns or not. It means if you want to be variable in the documentation, you should look how many character can a code line have, take the minimum. Do not be frivolous in changing font sizes at will. Think about proper text layouts.

The part after the `MAXLINELEN` can contain any number of *line numbers* `LINENR`, `MARKER` and `+` or `-`, with the comma `,` as repetition and the colon `:` as separator between number and `MARKER` or `+` and `-` instead the `MARKER`. The `LINENR` counts from 1 starting with the first line =1 presented in the code

Another intension may be, you cannot change the code with this documentation stuff. Only one tag may be admissible to find out the correct position for the snippet. This can occur if another developer is responsible for the code. You can only agree about the one tag, but not about some markers and omitted lines. Not because your contributor is evil, no, only because it is effort to checkout and commit sources during the documentation phase.

snipping. It means from the line containing the `<::TAG.>` or the next line after a `<::~~TAG.>` or the first line as 1 of a file if tag is `$`. Omitted lines with `<:-TAG.>` till `<:+TAG.>` are countered, the source is determining.

The `-` or `+` instead the `MARKER` switches the code snippet off or on to omit lines, adequate the `<:-TAG.>` till `<:+TAG.>`

The `MARKER` is written on end of the numbered line.

The problem for line countering is only, if the source is changed, the line numbers may need to adjust. But the impact is seen in the documentation and can be adjusted only in the documentation (in the ZmL file) without changing sources again. Because the line numbering starts with the begin of the snippet in the source, only changes inside this snippet range in the source are effective.

See the documentation in the chapter **5.1.2** *callZmL2odt.bat and callZmL2odt.sh* at page **48**, compare it with the ZmL file of this documentation in look in the batch file, also in this working tree to have a proper example. And try by yourself.

3.8 Images

Images are always bounded to a paragraph which contains the image caption. The position is left or right bounded, but with 0 distance to the paragraph. If you move unintentionally the image in LibreOffice, go to its properties (right mouse) and entry 0.0 for its position, and the image should be proper again. On Translation ZmL to LibreOffice this positions are 0. See also **2.3 Manual column or page breaks and positions of images** page 11

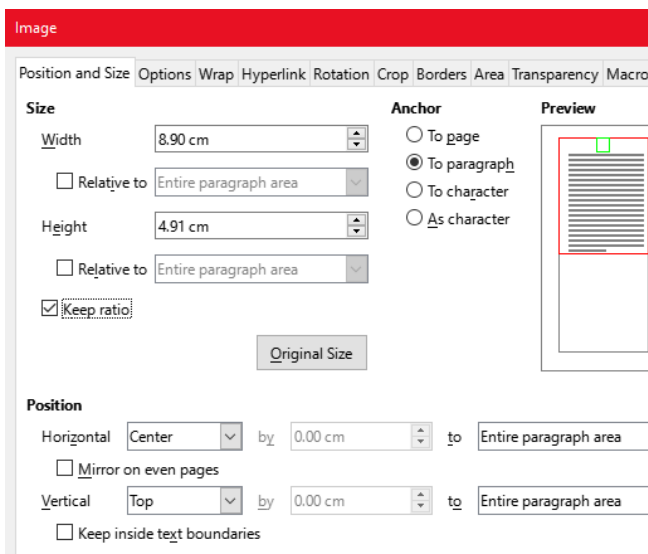


Figure 6: ImagePosSize.png

The writing style in ZmL for images is:

```
<:@image:../img/dir/ImagePosSize.png::
id=__Img_ImagePosSize.png ::
title=Figure 1: ImagePosSize.png ::
style=ImageCenter :: size=8.5cm*7.26cm ::
px=512*437 :: DPI = 153.>
```

For translation ZmL to LibreOffice only the size information is relevant. But the height can be removed, the image is resized with its ratio automatically during translation.

The title builds the content of the paragraph for the image caption, where the image is bounded to. The style of this paragraph is always either `ImgCaptionText` or also `mgCaptionTextPg` or `mgCaptionTextCol` depending from a

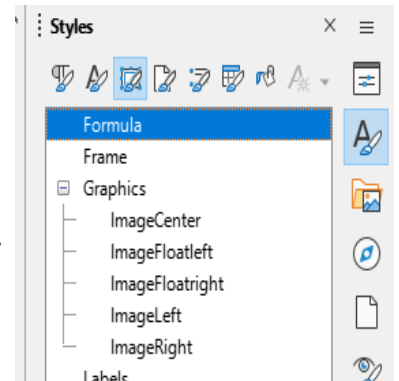
```
<:columnBreak.>
```

or a `<:pageBreak.>` before the image line separated with an empty line.

The given style `style=ImageCenter` determines the used style for the image itself.

Figure 7: ImageStyles.png

The right image has the style `ImageFloatLeft`, and that's why the text floats left of the image as seen here.



In profession, the saying "less is more" is important. Only a few scopes for design is really enough. That is for example a right side image flowing on left side with text (as usual in Wikipedia), an image positioned left side and flowing with text right side, or a central or left or right aligned non flowing image. The borders are not a point of discussion, borders should always the same, for example 2 mm or 0.08 inch. That suggest, using an indirect style also for images in LibreOffice and remove all direct styles.

The preferred styles in LibreOffice for images are:

- `Img`: A central image between paragraphs.
- `ImgRight`: A right side image between paragraphs.
- `ImgfloatLeft`: A right side image as part of a paragraph, floated left side with the paragraph's text.
- `ImgLeft`: A left side image between paragraphs.
- `ImgfloatRight`: A left side image as part of a paragraph, floated right side with the paragraph's text.
- `ImgChar`: An image inside of a line of the text of a paragraph, usual a small image.
- `ImgfloatChar`: An image inside of the text of a paragraph, the lines above are left and

right of the image, the base line is broken by (contains) the image.

No more is necessary.

The following syntax is used for images in ZmL:

```
<:@image:PATH/TO/IMAGE ::
title=CAPTION ::
style=STYLE ::
size=Xcm*Ycm ::
px=PX::PY ::
DPI=DPI
```

All arguments are optional, except the PATH/TO/IMAGE: Line breaks are optional after the ::.

- `IPATH/TO/IMAGE`: This should be recommended a relative path to the image starting from the odt document folder. You can use in MS-Windows a symbolic directory link created with `mklink /J NAME PATH` or also a symbolic linked directory in Linux/Unix to a little bit remote existing image directory tree to reach images with a simple link. Using an absolute path is strongly not recommended, because then, you cannot copy your files to another computer with a non exact equal directory tree structure. Also links inside the odt document are possible but not recommended.
- `title=TEXT`: A title or caption for each image should be recommended.
- `style=STYLENAME`: This should be one of the named indirect styles for the image positioning.
- `size=xSize*ySize`: The image size should be usual given in the measurements of the document, not in pixel. See **3.8.1 Some remarks to size of images**. Write for example `size=9.87cm*3.14cm` or `size=123pt*87pt`. If this parameter is not given but `: DPI=..` is given, then the size is calculated by this values.
- `px=xPixel*yPixel`: This value is used only if the image is not available as file while translation. Elsewhere the pixel size is read from the image file and write to ZmL

as information. Note: Till now only png images are used.

3.8.1 Some remarks to size of images

For a printed document and also for pdf and inside LibreOffice the resolution of pixel depends on the output capability. It is not related to the pixel size of the given image file. The printer or render in pdf and inside LibreOffice adapts the pixel of the image to the pixel of the used output. For that also anti-aliasing algorithm are usual used. That's why the pixel size does not play a role for the size of the image. It may be interesting only for the resolution or quality of the image.

The size is determined by the size on the output device or related to the paper format. It is named in following text as "printed size". That is either a value in cm, inches, pt or pica as usual units for that. Only this size is used also internally for LibreOffice.

But, sometimes the pixel size should be used to determine the printed size, if the image is changed or if the document is written newly, maybe to show one pixel of the image exactly by 1, 2 or 4 pixel in the printed output, maybe to have a relation to the image pixel size, or maybe also to prevent some aliasing effects on bad rendering.

That's why you can give the image size also in pixel with a related DPI ("*dot per inch*") resolution. If the printed size as `size=...` is not given, then this printed size value is calculated by translation to LibreOffice.

For translation from `LibreOffice.odt` to `Plaintext.vml.adoc` the pixel size is gotten from the image file (if it exists in the given link), and the DPI value is calculated with the given print size. The DPI value may be an interesting information. With this information you can tune the size of the image in your `vml.adoc` file, for example tuning the DPI value, together with

removing the information to force new calculation of the size.

For example you see the following line:

```
<:image:...
:: size=5.6cm*4.3cm :: px=1024::768 ::
DPI=464*453 .>
```

Then you see, you have a fine resolution, because the image is small with a high number of pixel, but you see also that the image is a little bit biased. The reason may be, the image was change in pixel size, but not in the document. If you change this line to

```
<:image:...
:: px=1024::768 :: DPI=450 .>
```

The you force new calculation of the size in cm, whereby the size will be a little bit greater, because of reduced DPI. But now (empty page)

the bias is removed, the original width and height relation is mapped. And last not least for a printing output with 150 DPI exact three image pixel are used to build the print pixel. On next generation from LibreOffice you will get the line

```
<:image:...
:: size=5.78cm*4.34cm :: px=1024::768 ::
DPI=450 .>
```

which is the real size now.

If the size is given in the

```
<:image:...
:: size=10cm*5cm .>
```

then this given printing size value is used, independent of the image pixel size. The additional pixel size and DPI value is ignored then.

3.9 Possibility of include and dispersion

The idea is, one PDF document, hence one odt document is mapped to more as one `source.vml` files, and vice versa a odt document or a PDF document is generated from more as one dedicated `sourceXy.vml`. The advantage is, the source files will be smaller and better to overview, whereas the end publishing PDF may be one large document. Another important advantage is: It is possible to produce a proper document only from some designated parts of sources, and produce another document with the same sources in another combination, for specific issues and usages. It supports a content management system.

```
<:@include:sourcepart.vml.>
```

This statement includes the named file, but only from the first chapter title, not with the title / content information before. This allows generate an extra document with this only one same source with an adequate specific title page.

This first chapter should start with a label

```
== the chapter <:#__PART_sourcepart.>
```

This allows the conversion back from odt to ZmL to the extra `sourcepart.vml` file. More as that: The given vml file is read, this chapter is searched, and only this chapter till the next chapter with the same level, or just the end of the document is replaced.

The feature of this partially replacement can be used in a more complex way:

```
<:@include:sourcepart.vml#LabelXy.>
```

This statement includes only from this named `sourcepart.vml` file the chapter with the given `labelXy`:

```
== the chapter <:#__PART_sourcepart#LabelXy.>
```

The designation in the long form `#__PART_sourcepart#LabelXy` is necessary for back conversion from odt. This label is stored as bookmark in LibreOffice. All chapter with a Bookmark starting with `#__PART_` are written in an extra file, whose name follows. If a `#LabelXy` is detected in this chapter label, only this chapter is replaced by the generated content. If no additional label is given, the first (and usual one) chapter is replaces, only the title information are preserved, and this chapter gets the label `#__PART_sourcepart` as described above.

The cross referencing bookmark handling should be clarified, because in the whole large document some internal links to other chapters can be given, which cannot be fulfilled i a smaller document only with specific parts.

The answer of that is: The ZmL file contains the `<:@ref:#bookmark.>` though the reference is not in the same `source.vml` and maybe also not in the same `LibreOffc.odt` file after generation.. On generation from vml to odt

On conversion vml to odt to vml a file will be written with `NAME.label.txt`, with the same name as the generated output file. This file contains one each line per bookmark:

```
#label 1.2 chapter title @34
```

If some files are generated, there is a sum of `NAMEXy.label.txt` files.

The file name is the file where this destination are contained in.

The bookmark - label is searched in all

For back generation odt to vml

3.10 Cross references inside the document, how to deal with interrelated documents

LibreOffice uses *Bookmarks* for References inside the documents. Without user activity this bookmarks will be created automatically if necessary (for references, for table of contents) and have the form (example) `__RefNumPara__5173_3943018604`. This numbers are not usable for the ZmL presentation. But LibreOffice supports also manual set Bookmarks with any text, which can be referenced:

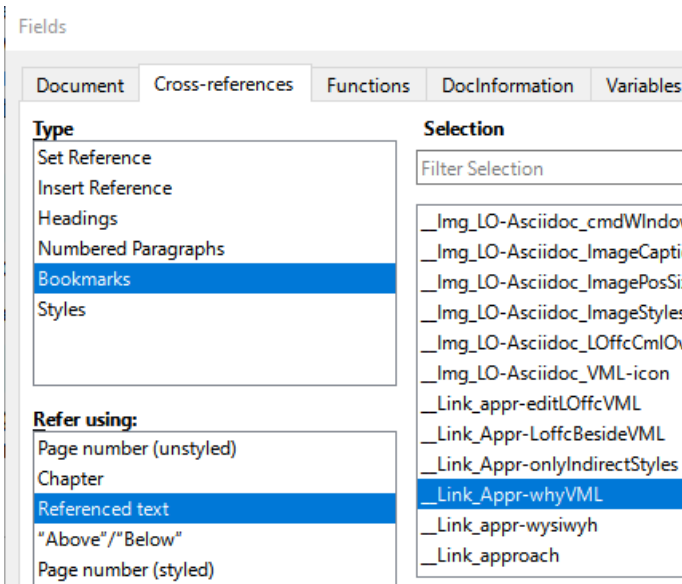


Figure 8: DialogCrossReferences-1.png

Markup languages have usual manual determined labels for references, such as HTML as “anchor” ``, or AsciiDoc as

```
[label]
=== chapter in AsciiDoc
```

Exact this approach is also used for ZmL. You should write to associate a bookmark label to a chapter via

```
=== chapter title ZmL <:#label.>
```

3.10.1 Use a proper name for bookmark labels

Because it is necessary to dedicate the bookmark labels by your own, it is in your response. Generally the bookmarks are local for the document. But because of the **3.9 Possibility of include and dispersion**

page **34**, the labels should regard the adequate correspond documents.

It is recommended that labels should start with the same text of the main topic (chapter level 1), following by text parts for sub chapters, as:

```
==== Title <:#Main-sub1-sub2.>
```

But for that if you change your chapter structure, you should adapt the name of your labels. Such refactoring may be sometimes necessary.

Changing a given label means, you should adapt all usages. This is not hard to do in the ZmL plain text – you can use search ‘n replace over some files. But if you have published files, and the bookmark labels are also used for links from extern, it is problematically. As compatible solution you can offer two bookmarks for one chapter:

```
==== Title <:#Main-sub1-sub2.> <:#label0ld.>
```

If a chapter was referenced which has no label in the ZmL till now, an automatic label as `<:#$Label_12.>` will be inserted. Search and replace such labels on further working in ZmL editing.

3.10.2 How to write a reference

In LibreOffice writer the reference to another chapter of the same document can either be done as usual, with Cross Reference to “Headings” as also as shown in **Figure 8: DialogCrossReferences-1.png** to manual given Bookmarks. In the second kind you should know your bookmarks, because the Header texts are not displayed. But both is compatible. On creating ZmL the bookmark to a header in form `__RefHeading:12345` is replaced by the given known manual bookmark. In both case the ZmL is written as:

```
...text <:@ref:#label:1.2 Title.> text...
```

For back conversion ZmL to odt the part after the label (here `1.2 Title`) is ignored.

This is only an information in ZmL to know what happen. It means if the chapter title is changed, the current given chapter title is used for the next odt to ZmL conversion.

If the page is given to a label (proper for a printed document, or also for pdf), the page should be placed **after** the referenced text, for example in form page 12, where the 12 is the inserted “Page number (unstyled)”. For odt to ZmL conversion the following information is generated:

```
<:@ref:#label:1.2 Title.> page <#@page:77.>
```

The 77 is only a place holder without meaning. (in an older version the real number was output here, but the disadvantage is, that the ZmL source file is changed at many positions only because moved page numbers). The back conversion ZmL to odt uses the label from the last `<:@#ref:label.>` also for the page as label. Backward compatible CR (*Change Request*): write either `<:@page.>` or `<:@page:label.>`

3.10.3 Cross reference to other documents of the same suite

Follow **3.9 Possibility of include and dispersion** page 34. If you produce a large document from more ZmL files, the bookmarks should be all internally. If you produce more documents, one from one ZmL file, then the **same** bookmarks plays the role. It means, bookmarks are used in the same kind as internal references as also for references in the other documents of the same suite. But of course, for a link from one document to the other of the same suite, a link, not an internal reference is necessary for odt and PDF, and also HTML.

How does it work, what to do?

First, the related documents should be in the same folder. This folder should be given with the argument

```
-rlinkhtml:../html/*.html
-rlinkpdf:../pdf/*.pdf
```

The both paths are examples, but meaningful examples. Regarding the generated user-ready documents: They are PDF and HTML. They may be stored in the named path. A side by side PDF document is reached with `theOther.pdf` as also as `../pdf/theOther.pdf` if both are contained in the pdf folder. It means you can track the link from the generated document, and also from the odt if the `../odt` directory is beside the `../pdf` and `../html`. This directories may organized as symbolic link to the original location, because the odt is anywhere in a source working tree. For the directory tree,

as example, see also **4.1 Directory tree structure in the working area 42**.

Alternatively you can use

```
-alinkhtml:https://www.my-page/html/*.html
-alinkpdf:https://www.my-page/pdf/*.pdf
```

or also as absolute path possible in a local network. This is sensible if the referenced documents are only able to find there, as absolute location.

Then, the ZmL to odt conversion uses this path to create a hyperlink to the external document, instead the internal reference.

But how to decide which label is external, and how to get more information? This is done during odt to ZmL conversion. On any conversion from one given odt file its internal bookmarks are written in a file proper to the command line argument:

```
-labels:*.Labels.txt
```

The `*` is replaced by the name of the converted document. Hence you have the files

```
MyDocument.odt
MyDocument.vml
MyDocument.Labels.txt
```

side by side. Now the ZmL to odt conversion reads all this files exclusively the own file (with the own name), and hence it gets an internal list of all labels which are existing in the other, related documents. The `*.Labels.txt` contains the label (bookmark), the chapter title and the page number. The name of the

file.Labes.txt is also gathered in the internal list of external labels. So the link can be completed with the name of the external file and the label as internal anchor (target). With this information a reference to a related document is written as:

Example:

See also [html Internals LibreOffcZMarkup.pdf](#): **1.4** *WriteOdt* on page **3**.

This is a link to the related document. The link is created in ZmL writing only:

```
See also <:@ref:#internal-WriteOdt.> on page <:@page.>.
```

Which label should be used, how the label is named: This is determinable by looking either in the other odt file, the name of the bookmark of the chapter, or also in the generated and given `OtherDocu.Labels.txt` file. This is a one time effort. If you do not change the bookmark labels (see **3.10.1 Use a proper name for bookmark labels** page **353.10.1 Use a proper name for bookmark labels**), then it is done for all time. If you change your label texts, you should search and replace all occurrences, which is also a possible not to high effort, as described in .

The information about page, chapter title and number is gotten from the file `Internals_LibreOffcVMarkup.Labels.txt` which

- (empty page)

was written on last conversion from this odt file to ZmL. But note, **the information about chapter title and page are only given if the odt file contains anywhere a Table of contents where this chapter is referenced**. This should be a matter of course.

The generated information in the odt contains a complete link to the given HTML file, with the internal label (anchor), as also the reference to the given PDF file as local or global as hyperlink. For PDF unfortunately there is no common standard concept for internal label selection. Hence the user can select by its own the given chapter or page. But if you read and have opened the documents anyway, you should be familiar with the table of contents in PDF, often as tree left side, and you can find fast the related chapter and also the surrounding chapters. The chapter title and page number is marked with the character style `Reference` which may be conspicuous.

On back conversion from odt to ZmL this given information in odt are translated to a ZmL link:

```
See also <:@ref:#internal-WriteOdt: Internals_LibreOffcVMarkup.pdf: 1.2 WriteOdt.> on page <:@page.>.
```

The information in the `<:ref:...: After the label till .>` are optional, not used for ZmL to odt conversion, but they are an important information while reading the ZmL file.

3.11 Hyperlinks and Hyperlink anchor completion

3.11.1 Simple Hyperlinks with or without target to the internet or locally

Hyperlinks are written in ZmL as

```
<:@link:../relative/path/to/file.html::hyperlink text.>
```

or also as absolute or internet link:

```
<:@link:https://theWebPage.org/path/to/file.html::hyperlink text.>
```

The link can also have a target or anchor designation in the known style from HTML or also used in LibreOffice in the link dialog:

```
<:@link:../relative/path/to/file.html#targetLabel::hyperlink text to the target.>
```

```
<:@link:https://theWebPage.org/path/to/file.html#targetLabel::hyperlink text to the target.>
```

This links are converted in `LibreOffice.odt` in this adequate kind. Unfortunately the targets does not work in LibreOffice for relative links (Document links). They only work for the links to the internet. It is a bad feature which complicates proper software documentation.

3.11.2 Relative local links and supplement www link with same path

As mentioned in [2.7 External links to Javadoc local files and the internet](#) page 18 But sometimes, the relative linked destination, for Javadoc or other, is not available. That's why I **place a relative link and a link to the proper internet location side by side** in form: See [WriteOdt.main\(...\)](#) ([www](#)). Both refer the same content, local and in internet. The local link may contain the class and operation, as shown here, the www does not need its repetition.

As argument on translation you need:

```
-www:https:myWeb/dir
```

In the ZmL you can write the link to the adequate operation as:

```
<:@link:../../docuSrcJava_vishiaLibreOffc/org/vishia/odt/readOdt/WriteOdt.html::WriteOdt.>
```

```
(<:@link:https://vishia.org/LibreOffc/...::www.>
```

Then the ellipse in the given www link `/...` is replaced by the String after all back path

characters `/` and `..` or `.`, the start of the real used path.

For back translation from odt to ZmL it is tested whether the www link starts with the given argument `-www:https:myWeb/dir`. If it is so, then also the rest of the path is compared as stored in the content.xml in the odt file. This depends of course from handling in LibreOffice. If they are equal, the ellipse is created again for VmL. On translation in both direction it works anyway.

3.11.3 Supplement argument types of intern operation links (anchors in html)

The requirement is described in [2.7.3 Supplement argument types of intern operation links \(anchors in html\)](#) page 19

The ZmL translator searches the HTML file as given in the link. It should be (also for the proper link) the relative path starting from the odt. From the HTML file all anchors are checked and write in an index container. If the anchors refer to an operation, the key of the index is the operation name. If the operation is not unique, means the same name is found twice or more, then the first occurrence is preserved, but a “?” is appended on end.

For ZmL to odt translation:

- If the operation name is found in the list, without the “?” on end, then the stored

full anchor is replaced for the link, its complete for this unique operation.

- If the operation name is not found in the list or it is not unique, then the link is stored in odt as given in VmL. If it is complete with all arguments, it is proper. If the link is faulty (non exists operation), it is able to see in odt and should be corrected either in the VmL with new translation to odt or in odt.

For odt to ZmL translation

- It may be expected that the links in odt are proper, or not to correct them in VmL.

- The operation name is searched in the index from HTML. If is is found and ends with “?” then the link from odt is not changed. The operation is ok. But whether the argument types are ok, this is not checked by odt to VmL, should be checked viewing the document and tracking links.

- If the operation is found with its complete label, and it is unique, then `...html#operation(...)` is written as link in the VmL file. Then not the complicated argument string is disturbing, it is simple

readable and unique. Back translation to odt will get the link in its valid form again.

- If the software is changed later, a new translation VmL to odt will get the new now valid anchor of the operation, without additional effort.

- If the software is changed in a kind, that a previous unique operation link will get non unique, then the VmL to odt produces the first occurrence with following “?”, and back translation preserves it. On editing VmL this should be obviously. It may be simple to correct the link comparing in the software ore generated Javadoc.

The link for unique operations may look like:

```
<:@link:../../docuSrcJava_vishiaLibreOffc/
org/vishia/odt/readOdt/
WriteOdt.html#main(...)::
WriteOdt.main(cmdLineArguments).>
(<:@link:https://vishia.org/LibreOffc/...::
www.>
```

The text of the link is not changed and not used. It may be nice to do not write there the formally arguments, instead mnemonic argument descriptions..

3.12 Transliteration of specific characters

There are some non or bad visible characters which needs transliteration. This is:

Non breaking space

ZmL	adoc	html	UTF-16	UTF-8
\	{nbsp}	 	\u00a0	c2 a0

LibreOffice: “*Insert – Formatting mark – Insert Non breaking space Sh-Ctrl-space*”.

Appearance inside a text only visible in LibreOffice itself: etc. pp.

The non breaking space is a normal space, in editors usual shown as a simple space, but a line break on this position is prevented. The transliteration in ZmL is simple, only a backslash before the space is written, to see it.

Using for example for *etc. pp*, which should not break the line between *etc.* and the following *pp*.

Zero width space (optional break)

ZmL	adoc	html	UTF-16	UTF-8
\	{zws}		\u200b	e2 80 8b

LibreOffice: “*Insert – Formatting mark – No width Optional Break Ctrl-*”.

Appearance inside a text only visible in LibreOffice itself: A verylongword

A white space with an appearance of non space. But a line wrap can be inserted on this position if necessary.

word joiner

ZmL	adoc	html	UTF-16	UTF-8
\+	{wj}		\u2060	e2 81 a0

LibreOffice: “*Insert – Formatting mark – Word Joiner*”.

Appearance inside a text only visible in LibreOffice itself: word joiner

A non visible character which prevents a line break on this position, though a breaking possible character follows, with the next following words (also after some spaces or tabs).

... **3.2 Syntax overview by examples** ²⁵see page

3.13 Using Character styles, semantic text span

General character styles are written in form:

```
paragraph ... <:ChStyle:text.> ...
further text
```

For software documentation, often code snippets should be included in the current text. For that the character style should be written in ZmL as simple as possible. A line in C is written as `<:cC:operation(float x);.>` and appears in LibreOffice as

Usual used styles are:

Code block appearance

```
Simple code block
with some lines.
```

```
Cmd line
or file tree presentation
```

```
REM A windows batch file
or a shell script
```

```
##Some configuration data
a = "test"
```

```
void javaOperation(float arg) {
    return;
}
```

```
void cppOperation(float arg) {
    return;
}
```

```
##This is a otx script:
<:otx: VarV_UFB: evSrc, fb, evin, din>
```

```
A Zml code snippet <:c:code characters.>
```

Copy this part in your document to copy the styles, and to see how the styles appear.

`operation(float x);`. To compare, in Javadoc it is written as `[Cpp]`operation(float x)`` whereby the `cpp` should be declared as `class="language-Cpp"` in the CSS for HTML (Cascade Style Sheets). In LibreOffice `cc` should exist as character style.

Especially for source code Some paragraph styles and character styles should be existing with the same font, size and color:

P-style, C-Style and appearance:

- `Code`, `ccode`: And here is simple code
- `CodeCmd`, `cCmd`: this is a `cmd` call arguments example
- `CodeScript`, `cS`: A part of a script
- `CodeCfg`: `cCfg`: `config` data some configuration data
- `CodeJava`, `cJ`: `javaOperation` with arguments
- `CodeCpp`, `cc`: also C or C++ language `cppOperation()` given
- `CodeOtx`, `cotx`: A specific code style in line written as `<:otx: VarV_UFB:`
- `CodeZmL`, `cZmL`: Specific code style for this topic `<:c:code characters.>`
- `cm`: A `Marker` should be used also inside code blocks and in the explanation. Should look demonstrative

The simple writable direct formatting styles for italic (usual ctrl-i), bold (ctrl-b) and the combination of both is translated to the character style `Quotation`, `Strong Emphasis` and `Emphasis`. Their appearance can be

controlled by the style, it is usual *Quotation*, **Strong Emphasis** and *Emphasis*. It is written in ZmL as `<:Q:Quotation.>`, `<:S:Strong Emphasis.>` and `<:E:Emphasis.>`

4 Handling Writing and Converting

Table of Contents

4 Handling Writing and Converting.....	42
4.1 Directory tree structure in the working area.....	42
4.2 Daily work on the documents.....	43
4.3 Convert from given vml to a new odt.....	44
4.4 Convert from LibreOffice odt to a vml.....	45

– Above the theory and some practical approaches. Now the real practice.

This chapter is related to the download zip file `TemplateZmLodt-2024-09-28.zip` and also following.

4.1 Directory tree structure in the working area

The template zip file contains one document as template for technical documentation as explained in the chapters above, especially chapter 2 **Some decisions how to write a technical documentation** page 10

The general thinking is: do the work on any location in the file system, maybe on a local hard disk, maybe also in network. You have to consider one location where you do the work, and some other locations for global things, the LibreOffice tool itself, for Java etc. The global things are organized by the operation system in its specific known manner. The working location is yours.

The next maybe important thinking is: use symbolic linked directories if necessary. For example images can be saved anywhere else, not in your working area for the documentation. Because the images are used also for other things, for html view, for creating and editing the images itself, etc. The same is with linked files. Here also for the example some automatic generated Javadoc files are referenced for technical documentation. You may similar stuff do with Doxygen as tool or from any other tools. Automatically generated documentations, often in html format, completes the technical documentation.

A symbolic linked directory in Unix/Linux should be familiar, use the `ln -s path/to`

`link` as command. For Windows it is also possible, since ~ Vista, but not so familiar known. It is the `mklink /J link path\to`, see Windows-help.

One other important idea here is: src and build are side by side, src does not contain any temporaries, and src has a second level for diverse components, and a third level for the parts of component. This tree structure in the working area of the hard disk is explained in <https://www.vishia.org/SwEng/?html/srcFileTree.html> and shortly presented here as:

working area on your hard disk:

```

+-build
| +-Component
|   +-odt
|     +-dbg
|     +-backwork
+-tools
+-src
  +-load_tools
  +-Component
    +-odt null...here is the docu
    | +-makeDocu
    +-asciidoc
    +-img null...as symbolic link
    +-html null...symbolic link
    +-...more for this component,
      for example C/pp sources

```

The directory tree in your working location regards a separation between the real sources and generate files, and some temporary files which are interesting to view, for backup, etc, but not to save it persistently. It is a bad concision to merge

these files between the sources. The sources should only contain persistent files, and that files are always in kByte and low MByte range. Images and generated docu (html) may have more MByte, but these are links and can be excluded if the sources are copied or zipped.

There is also a smaller directory tree thinking for the ready to use documentation:

```
Any location for documentation
+-pdf
+-img
+-html/generatedDocu
```

This is similar or equivalent the inner tree of the sources above. Instead `odt` here is `pdf`. The links from the generated pdf document

4.2 Daily work on the documents

General both files should be in focus of the work. The question is: What is the master file, the `docu.odt` or the `source.vml`?

Answer: Use both, after changing in one of them, convert to the other, before continue. Convert at least to `source.vml`? before finish work.

The `source.vml` file is the candidate for a version management (often `git` is used), because this is textual content possible to simple compare for version tracking. The `docu.odt` file should be always possible to generate from the `source.vml`. But nevertheless it may be recommended to save an `docu.odt` versions, which is used to publish (as pdf), independent from a git for the `source.vml` sources. It is also possible to generate a `source.vml` from this stored `docu.odt` newly for comparison.

How to generate the `docu.odt` – see following **4.3 Convert from given vml to a new odt**. If you have done the generation, then the script should automatically open the LibreOffice editor to work with. The first action should be: `tools - update all`. Then you have a proper view to your `source.vml` document. It can be seen only as view (as for example after rendering to PDF in

are **the same relative links** as also from the `odt`. That's an important regulation to work proper with relative links. **Do not use absolute links** in your file organization because the absolute links are only valid for your work in the moment. Other environments may have other considerations. Also if all PCs in the department of a company are the same, they are not the same guaranteed in a few years, they are not the same in an other department.

The proposal and here used directory tree structure regards

This is arranged in a su

AsciiDoc or LaTeX), you can parallel look for other content there, and work furthermore on your `source.vml`.

But with the new generated `docu.odt` you should press only one time the icon in LibreOffice for => **ZmL**, see **4.4 Convert from LibreOffice odt to a vml 45**. The you get a new `source.vml`, which may has changed some line breaks, with the same content. You can/should immediately compare this new generated `source.vml` with your last work. This is automatically saved as copy in `build/CMPN/odt/source.vml` as action on conversion. If you have always open a text diff tool, for example using <https://winmerge.org>, then this work is immediately started by only press “refresh” in this tool. The differences should be all able to accept (exclusively sophisticated constructs which are not clean programmed in any of the involved tools). Then you can work furthermore after “refresh” in your ZmL editor.

If this is too costly, you can also skip this step and trust on the tool chain.

It is also possible to work furthermore in the `docu.odt` with LibreOffice. But especially then, you may generate a new `source.vml` by

pressing => **ZmL** and comparing with your diff tool, to be sure that all is ok. Also a checkin or commit in your source version system or store manually versions may be recommended.

If you want to adjust the page dispositions, or only search for small writing errors, or think about details and amount of content, editing in LibreOffice may be a proper decision.

If you detect that some links are faulty, should be adjusted because the linked files are moved in the file system, or a generally error for link locations is given, or images are moved, or one writing mistake is given a lot of times, then it is better to edit this in the `source.vml`, because it is better to adjust these things with search and replace in the `source.vml`, especially in primary hidden information in LibreOffice. In nullthe `source.vml` they are part of the plain text.

4.3 Convert from given vml to a new odt

Generation the `docu.odt` or just conversion from `source.vml` is done by start of a command line action, starting with a batch or a shell script. In the given download

```
echo off
echo called: %0
cd %~d0%~p0
echo currdir=%CD%
makeDocu\L0ffc-ZmL2odt.bat TechnicalDocTwoColumns
```

The marked argument is the file name of the `TechnicalDocTwoColumns.odt` as destination file to create and also the `TechnicalDocTwoColumns.vml` as input file. The rest is done by the called script `makeDocu\L0ffc-ZmL2odt.bat` which is universal for all documents in this folder.

Note: the first line `cd %~d0%~p0` assures that the current directory is the directory of this

If you want to change the chapter arrangement, or the decision about column arrangement, this may be better done in the `source.vml`.

Which editor should be used for `source.vml`? You can use all plain text editing tools. You can better use an editor which supports AsciiDoc especially if an outline is shown. This outline gives an overview over the maybe long source text. I use editing in Eclipse with the specific *“Wiki text editor”* set to the *“Markup language = AsciiDoc”* or just the *“AsciiDoc editor”* available in the market place of Eclipse. This is proper because the chapter syntax is the same and the outline works. Only supporting the specific syntax for for example with short keys is not supported, unfortunately there is not a specific ZmL editor.

example for this documentation it is the file `TechnicalDocTwoColumns.vml2odt.bat` in the directory `../src/example/odt`. This file contains only:

called file, which path is contained in the first (`%0`) argument. This allows calling the batch from another current directory with a path, maybe also from inside any tool.

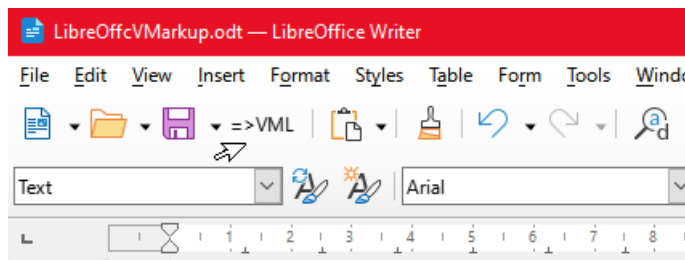
The called script `-L0ffc-ZmL2odt.bat` is explained in chapter **5.1.2 callZmL2odt.bat and callZmL2odt.sh** page 48

4.4 Convert from LibreOffice odt to a vml

Figure 9: ZmL-icon.png

This can be done anytime in the opened LibreOffice editor, if a proper Macro in LibreOffice was installed. The macro is presented by an icon. It is stored with the name `ZmLwr` in the central LibreOffice Macros under "Standard". It calls a batch file `L0ffc_odt2ZmL.bat` (or shell script for Linux). This is described in the chapter [5.1.3 Macro =>ZmL and script -callOdt2ZmL.bat and .callOdt2ZmL.sh](#) page. The scripts may be need to adapt for different environments, for example using another source tree structure ([4.1 Directory tree structure in the working area](#)).

After pressing the button `=>ZmL` the following command window is shown:



```

C:\WINDOWS\system32\cmd.exe
called: L0ffc_odt2VML.bat Version 2024-08-25
arg1: U:\vishia\LibreOffc\source.wrk\src\srcJava_vishiaLibreOffc\odt\
arg2: LibreOffcVMarkup.odt
Environment variables:
NAME_SRC=LibreOffcVMarkup
DIR_SRC=U:\vishia\LibreOffc\source.wrk\src\srcJava_vishiaLibreOffc\odt\
DIRCMPN=U:\vishia\LibreOffc\source.wrk\src\srcJava_vishiaLibreOffc
NAMECMP=srcJava_vishiaLibreOffc
BUILDCMPN=U:\vishia\LibreOffc\source.wrk\build\srcJava_vishiaLibreOffc\odt
copied: U:\vishia\LibreOffc\source.wrk\build\srcJava_vishiaLibreOffc\odt\backWork\LibreOffcVMarkup-C.vml
==== generate LibreOffcVMarkup.vml.adoc ==== ?
U:\vishia\LibreOffc\source.wrk\src\srcJava_vishiaLibreOffc\odt

U:\vishia\LibreOffc\source.wrk\src\srcJava_vishiaLibreOffc\odt>java -version
java version "1.8.0_341"
Java(TM) SE Runtime Environment (build 1.8.0_341-b10)
Java HotSpot(TM) 64-Bit Server VM (build 25.341-b10, mixed mode)

U:\vishia\LibreOffc\source.wrk\src\srcJava_vishiaLibreOffc\odt>java -cp "..\..\..\tools\vishiaBase.jar;..\..\..\tools\vishiaDocTools.jar" org.vishia.odt.readOdt.ReadOdt --@C:\Programs\batch\L0ffc_odt2VML.bat:args

successfull back: U:\vishia\LibreOffc\source.wrk\build\srcJava_vishiaLibreOffc\odt\LibreOffcVMarkup.vml
*** finished LibreOffcVMarkup.vml @U:\vishia\LibreOffc\source.wrk\src\srcJava_vishiaLibreOffc\odt\. EXIT-code=0
Drücken Sie eine beliebige Taste . . .
  
```

Figure 10: cmdWIndowOdt2ZmL.png

You see what is happen, and look and compare the `myDocu.vml` output.

Any call of the `ReadOdt` via Java saves the currently found `source.vml` file either beside the given file with `source.back.vml` or with the same name in the `build/component/odt/...` directory to compare with last version. This comparison may be important to see the progress, and decide which version should be commit to the version repository (git). It is also possible to create one backup file on each converting action with a countered

name. This produces trash (recommended on the RAM disk), some files, only kByte to less MByte range. But you can compare your work. This temporary trash files are all deleted on call of the script `makeDocu/+createCleanBuild.bat`.

The current (last) `vml` file is always stored beside the odt file determined by the calling arguments of java, ready to commit in a source repository and to work with it mutual with the `docu.odt` file in LibreOffice.

5 Implementation

Table of Contents

5 Implementation.....	46
5.1 Explanation of the scripts to call the converter.....	46
5.1.1 Script to clean and create a build (temporary output) location.....	46
5.1.2 callZmL2odt.bat and callZmL2odt.sh.....	48
5.1.3 Macro =>ZmL and script -callOdt2ZmL.bat and .callOdt2ZmL.sh.....	50
5.2 Get the tools.....	52
5.3 Test ref.....	52

5.1 Explanation of the scripts to call the converter

The scripts can or also should be adapted if another file tree structure is desired (see **4.1 Directory tree structure in the working area** page 42 or just only for some different outputs, for example having one persistent file in the temporary location on ZmL writing or only the last one. If you have lesser experience with such scripts, you can also try to change only one line, look

what's happen, and use it. Step by step you may understand the script concepts.

In the moment (2014-09) there are only Windows batch scripts. Also shell scripts are possible in the same way, and they work also in Windows, if for example “mingw” is installed coming with a “git” installation. But yet not ready.

5.1.1 Script to clean and create a build (temporary output) location

Why using a **build** or just temporary location? See **4.1 Directory tree structure in the working area** page 42. The idea is, that the temporary location should be a part of the working tree. This is possible using a symbolic linked directory. The working tree may be on a network location, and the link

destination for **build** should be local, or as I prefer, on a (local) RAM disk. That is done by the script in `src/example/odt/makeScript/+createCleanBuild.bat`.

This **build** directory may be unnecessary by setting some arguments only using the source directory, see next chapter.

```

echo off
echo called: %0
set BUILD_TMP="%TMP%\TemplateZmLodt"           <A>
set CMPN="example"                             <C1>
cd /D "%~d0%~p0"                               <B>
cd ..
set CURRDIR="%CD%"
echo INFO: directory of docu is %CURRDIR%
REM get the the directory name before \odt, it is the 'component'
cd ..
set DIRCMPN=%CD%
echo Test DIRCMPN=+++DIRCMPN%+                <C>
REM extract CMPN as the directory name before \odt, it is the 'component'
REM be carefully, about faulty spaces on end of CMPN (bug of windows) if writing ...nxA )
For %%A in (%DIRCMPN%) do ( echo Test: FOR results in %%A)
For %%A in (%DIRCMPN%) do ( Set CMPN=%%~nxA)    <C>
echo Test CMPN=+++CMPN%+
::pause
REM clean and create tmp location:              <D>
if exist %BUILD_TMP%\build\%CMPN% rmdir /S/Q %BUILD_TMP%\build\%CMPN%
::dir %BUILD_TMP%\build

```

```

if not exist %BUILD_TMP% mkdir %BUILD_TMP% <E>
if not exist %BUILD_TMP%\build mkdir %BUILD_TMP%\build
mkdir %BUILD_TMP%\build\%CMPN%
mkdir %BUILD_TMP%\build\%CMPN%\odt
mkdir %BUILD_TMP%\build\%CMPN%\odt\Backwork
mkdir %BUILD_TMP%\build\%CMPN%\odt\dbg <E>
if not exist ..\..\..\build ( <F>
  cd ..\..\..\
  mklink /J build %BUILD_TMP%\build
  cd %CURRDIR%
)
dir ..\..\..\build <G>
dir ..\..\..\build\%CMPN%
dir ..\..\..\build\%CMPN%\odt
if not "%1"=="NOPAUSE" pause <H>

```

This is the script in the `TemplateZmLodt-2024-09-28.zip` in `src/example/odt/makeScript/createCleanBuild.bat`. Because showing in column width only a few lines are broken marked with `...` on end and begin of the continued line. If you copy this content from the documentation, remove this line break with the `...`, then it is complete.

<A> In this line the destination directory in the `TMP` is defined. You should be carefully that the directory is not clashed with any other workspace temporary. It is for the whole workspace' `.../build` folder. If the `.../build` is created in another file also (especially in the root of the workspace in `+clean_mklink_build.bat`, use the same location.

 This `cd` sets the current directory to the given directory of this file, also if it called from another directory. `%0` is the path of the batch file. After the following `cd ..` then current is the `src/example/odt` directory.

<C> The statements between gets the component's name. This is the directory name between `src/COMPONENT/odt`. Then name is known, it is `src/example/odt` because of the position of the batch file. The 4th line marked with <C1> is the more simple variant instead. But these statements between <C>

makes the batch file compatible also for other components. The statements are a little bit some sophisticated Windows commands. But it works. The `For` statement should only be necessary because the `.%~nxA` cannot work with a simple environment variable, only with a `For` variable or with an argument as for example `%%~nx1`. Ask Windows for further explanation.

<D> This `rmdir` removes the whole content in the linked location if it is linked, or also in the non-linked location. It cleans. But it cleans only the space for this component `%CMPN%`, not for others in the working tree. The batch file on the root in the working tree `+clean.bat` cleans all in the `build` directory and the `build` directory itself.

<E> This statements create the necessary folders in the temporary location for this component `%CMPN%`. It creates also the base directories for the working space if not existing.

<F> This is only for show the result.

<G> The command window remains open to see what was done, if it is start by double click. If it is start via another script, the `pause` can be prevented by the argument `NOPAUSE`.

5.1.2 callZmL2odt.bat and callZmL2odt.sh

This script is called from a specific batch file, see 4.3 Convert from given vml to a new odt page 30. It is called with only %1, the name of the vml and odt file:

```

echo off
set NAME_SRC=%1
set CURRDIR=%CD%
set BUILDCOMPONENT=..\..\..\build\srcJava_vishiaLibreOffc\odt
::set JCP="..\..\..\tools\vishiaBase.jar;..\..\..\tools\vishiaDocTools.jar"
set JCP=D:/vishia/Java/Eclipse_Pj/vishiaJavaAll_22-03/bin
:checkBuildLoop
if not exist %BUILDCOMPONENT% (
    echo the ../build does not exist, please start createCleanBuild.bat first.
    pause
    goto :checkBuildLoop
)
:loop
echo off
echo =====
cd
echo ==== generate %NAME_SRC%.odt ==== ?
pause
::cls
if exist .~lock.%NAME_SRC%.odt# (
    echo %NAME_SRC%.odt is open, close it!
    pause
    goto loop
)
    echo Detect %NAME_SRC%.odt is closed,
    REM copy the vml.adoc version used last for generate odt, to compare.
    REM use a text diff tool to see what is changed after new generation.
    REM the argument --@C:path means that this file is used to read arguments.
(args)
    echo on
    copy %NAME_SRC%.odt %BUILDCOMPONENT%\backWork\%NAME_SRC%.odt
    java -cp %JCP% org.vishia.odt.readOdt.WriteOdt --@makeDocu/-Loffc-ZmL2odt.bat:args
::args ##
::-i:$NAME_SRC.vml
::-ilabel:*.Labels.txt
::-flink:../html/*.html
::-odt:$NAME_SRC.odt
::-cfg:makeDocu/Asciidoc2LibreOdt.cfg
::-www:https://vishia.org/
::-r:$BUILDCOMPONENT/$NAME_SRC.ReadZmL.report.txt
    echo off
    if errorlevel 1 (
        echo ERROR
        pause
        goto :loop
    )
    echo start LibreOffice...:
    echo first: Menu: "tools - update all" in LibreOffice,
    echo then you can edit inside %NAME_SRC%.odt and save it,
    echo ...
    echo start LibreOffice ...or press ctrl-C or close cmd-window
    pause
    call LibreOffice.bat %NAME_SRC%.odt
    echo ---
    echo press any key to re-generate or ctrl-C or close the window with mouse.
:: goto :loop

```

Let's explain from inner to outer:

(JAVA): The batch script calls the conversion program via `java`. This command is normally a very long line because of some arguments. It is shortened here by using environment variables. The 'class path' is contained in the variable `JCP`, it is set on top of the file. Here either two jar files are given, or for development phase also the Eclipse output directory may be immediately used from the specified location. This line may be necessary to adapt, if the tools, the jar files are available of an global position in file system instead. Here it is the folder `../../tools`. See also 5.2 Get the tools page 38

```
\**The second argument is the class which
does          the          work:
org.vishia.odt.readOdt.writeOdt.
```

(args) Then, after the `--@` exactly this same file `makeDocu/-Loffc-ZmL2odt.bat` is named as containing arguments. The following `:args` identifies this as label inside this file, where the arguments starts. It is searched in the file on the first few positions on beginning of lines. The string before, here `::` is recognized as indentation or marker string, for the batch file it is the comment designation. And last not least also the comment sequence for arguments is

declared in this line here with `##`. All immediately following lines with the given indentation, here `::`, are lines for arguments. Environment variables are also dissolved if they are written as `$(ENV)` or also only `$ENV` if it is unique as identifier.

Now the meaning of the arguments:

(input): After `-i:` the input file is named. Because of the environment variable `NAME_SRC` it is the name given as argument `%1` of this file. The current directory is the `odt` directory, hence it is all.

(iLabel), (fLink): This is an important possibility if the input file contains bookmark references as used for internal references in LibreOffice, but this internal references are not given because the document is split in parts. All files found on the given directory with the given extension, but not the file with the own `NAME_SRC` are read. They contain bookmarks or labels from their conversion. Instead the destination of the reference will be found in another document. See **3.9 Possibility of include and dispersion** Then with the next `-fLink:` the directory and file extension is named where the label should be referred to. The file name is contained in the given `-iLabel:*` file.

5.1.3 Macro =>ZmL and script -callOdt2ZmL.bat and .callOdt2ZmL.sh

```

REM ***** BASIC *****

Sub Main

REM https://ask.libreoffice.org/t/how-to-check-folders-name-by-macro/44665

dim sCurFileURL As String
dim sCurFileSys As String
dim sFolderSys As String
dim aPaths as Variant
dim odtName as String

sCurFileURL = ThisComponent.getURL()
sCurFileSys = ConvertFromURL(sCurFileURL)
aPaths=Split(sCurFileSys,"\")

odtDir = ""
For i = Lbound(aPaths) to Ubound(aPaths)-1
    odtDir = odtDir & aPaths(i) & "\"
Next i
odtName = aPaths(Ubound(aPaths))

REM MsgBox "Current Path : name: " & odtDir & " : " & odtName

REM call the proper script in the directory beside
REM 1.
argument: A batch file to call
REM 2.
argument: 1= Focus os window in standard size
REM 3.
argument only one string argument for the batch as %1
REM 4.
argument true then libre office waits for finish shell (sync)
REM see https://help.libreoffice.org/6.4/en-US/text/sbasic/shared/03130500.html
REM String(1,34) is the " (quotation char) ASCII = 34 = 0x22, used for surround "odtDir"
REM more as one argument separated with space, arguments maybe surround with "arg1", as "arg1"
arg2
Shell("C:\Programs\BATCH\LOffc_odt2ZmL.bat", 1, String(1,34) & odtDir & String(1,34) & " " &
odtName)
REM Shell(odtDir & "LOffc_odt2ZmL.bat", 1, String(1,34) & odtDir & String(1,34) & " " &
odtName)

End Sub

```

You should adapt the line for the Shell start, the directory of the batch file.

A batch file `../odt/makeDocu/LOffc_odt2ZmL.bat` for Windows-PC (possible of course also a shell script) is stored in the shown local directory. This script can be the same for all conversions, but can be adapted for tests and should be adapted in some special arguments, see description below.

todo describe

The batch file contains:

This should be explained.

The core action is (JAVA) call java with the class `org.vishia.odt.readOdt.ReadOdt`. But because this batch file is globally valid the arguments should be prepared with environment variables. They are shown on (ENV). The a little bit sophisticated one is the `BUILDCMPN`. This is the directory where some temporary files and backup files are stored. I use for that a RAM-Disk which has the advantage, it is faster, and does not consume the lifetime of a hard disk or SSD. The amount of stored data is in kByte till MByte range for a `vm1` file. For this solution a `build` directory beside `src` in the working tree is presumed but prepared. The `build` is a symbolic link (Junction in MS-Windows) to the `TMP` directory, and the last one is on RAM-Disk. There is a small batch script `+clean_mklink_build.bat` to clean and prepare this `build`:

```
echo off
set BUILD_TMP=%TMP%\LOffcZmL
cd %~d0%~p0
REM clean build, should be a symbolic link
if exist build rmdir /S/Q build
REM clean tmp location:
if exist %BUILD_TMP% rmdir /S/Q %BUILD_TMP%
mkdir %BUILD_TMP%\build
mklink /J build %BUILD_TMP%\build
echo test.txt > build/test.txt
```

This is a universal batch, only the location on the used directory inside `%TMP%`

5.2 Get the tools

load_tools

5.3 Test ref

Xxx ref to [5.3 Test ref page 52 html / Internals LibreOffcZMarkup.pdf](#): [1.4 WriteOdt](#)refext:

Docu file: Internals_LibreOffcZMarkup

- 1 *Internals page 2 (#internal)*
- 1.1 *Read content.xml from the odt file to internal data page 2 (#Impl-ReadOdg-XMLread)*
- 1.2 *ReadOdt page 3 (#\$Label_1)*
- 1.3 *Write content.xml to the odt file from internal data page 3 (#\$Label_2)*
- 1.4 *WriteOdt page 3 (#internal-WriteOdt)*

should be determined. It creates a symbolic link in MS-Windows and before, it cleans all.

If you start this batch you have a cleaned situations, but you have also lost all your temporary files inside the build.

The Java arguments (Java Args) should be explained: Usual arguments are part of the command line. But this very long command line will get non obviously. Hence inside Java there is the possibility to read arguments from any file. This file is given with `--@path/to/file` and this is exactly this same file. With additional the label `--@path/to/file.args` The identifier `args` is searched in this file on the first few positions on beginning of lines. The string before, here `::` is recognized as indentation or marker string, for the batch file it is the comment designation. And last not least also the comment sequence for arguments is declared in this line her with `##`. All immediately following lines with the indentation, here `::` are lines for arguments. Environment variables are also dissolved if they are written as `$(ENV)` or also only `$ENV` if it is unique as identifier.

But all in all, this files runs, produces a simple output on pressing the icon button: