FPDoc :
Free Pascal code documenter: Reference manual

Reference manual for FPDoc
Document version 0.9
December 21, 2005

Michaël Van Canneyt

Contents

Chapter 1

Introduction

1.1 About this document

This is the reference manual for FPDOC, a free documentation tool for Pascal units. It describes the usage of FPDOC and how to write documentation with it.

It attempts to be complete, but the tool is under continuous development, and so there may be some slight differences between the documentation and the actual program. In case of discrepancy, the sources of the tool are the final reference. A README or CHANGES file may be provided, and can also give some hints as to things which have changed. In case of doubt, these files or the sources are authoritative.

1.2 About FPDOC

FPDOC is a tool that combines a Pascal unit file and a description file in XML format and produces reference documentation for the unit. The reference documentation contains documentation for all of the identifiers found in the unit's interface section. The documentation is fully cross-referenced, making it easy to navigate. It is also possible to refer to other documentation sets written with FPDOC, making it possible to maintain larger documentation sets for large projects.

Contrary to some other documentation techniques, FPDOC does not require the presence of formatted comments in the source code. It takes a source file and a documentation file (in XML format) and merges these two together to a full documentation of the source. This means that the code doesn't get obfuscated with large pieces of comment, making it hard to read and understand.

FPDOC is package-oriented, which means that it considers units as part of a package. Documentation for all units in a package can be generated in one run.

At the moment of writing, the documentation can be generated in the following formats:

HTML Plain HTML. Javascript is used to be able to show a small window with class properties or class methods, but the generated HTML will work without JavaScript as well. Style sheets are used to do the markup, so the output can be customised.

XHTML As HTML, but using a more strict syntax.

LaTeX LaTeX files, which can be used with the fpc.sty file which comes with the Free Pascal documentation. From this output, PDF documents can be generated, and with the use of latex2rtf, RTF or Winhelp files. Text files can also be generated.

Text plain ascii text files. No cross-referencing exists. Other than that it resembles the LaTeX output in it's structure.

Man Unix man pages. Each function/procedure/method identifier is a man page. Constants are on a separate page, as are types, variables and resourcestrings.

Plans exist to create direct RTF output as well.

1.3 Getting more information.

If the documentation doesn't give an answer to your questions, you can obtain more information on the Internet, on the Free Pascal Website:

http://www.freepascal.org/

It contains links to download all FPDOC related material.

Finally, if you think something should be added to this manual (entirely possible), please do not hesitate and contact me at michael@freepascal.org.

Chapter 2

Compiling and Installing FPDOC

2.1 Compiling

In order to compile FPDOC, the following things are needed:

1. The fpdoc sources. These can be downloaded from the FPDOC website.
2. The Free Pascal compiler sources. FPDOC uses the scanner from the Free Pascal comiler to scan the source file.
3. The FCL units (or their sources) should be installed.
4. fpcmake is needed to create the makefile for fpdoc. It comes with Free Pascal, so if Free Pascal is installed, there should be no problem.
5. To make new internationalisation support files, rstconv must be installed, and the GNU gettext package.

Links to download all these programs can be found on the FPDOC website.

When the fpdoc sources have been unzipped, the Makefile must be generated. Before generating the makefile, the location of the compiler source directory should be indicated. In the Makefile.fpc file, which has a windows ini file format, locate the `fpcdir` entry in the `defaults` section:

```
fpcdir=../..
```

and change it so it points to the top-level Free Pascal source directory.

After that, running fpcmake will produce the Makefile, and running make should produce 2 executables: fpdoc and makeskel.

2.2 Installation

When installing from sources, a simple

```
make install
cd intl
make install
```

should completely install the documentation tool.

When installing from a archive with the binaries, it should be sufficient to copy the binaries to a directory in the PATH.

To have fpdoc available in several languages, the language files should be installed in the following directory on Unix systems:

```
/usr/local/share/locale/XX/LC_MESSAGES/
```

or

```
/usr/share/locale/XX/LC_MESSAGES/
```

Depending on the setup. Here XX should be replaced by the locale identifier.

Chapter 3

FPDOC usage

3.1 fpdoc

Using FPDOC is quite simple. It takes some command-line options, and based on these options, creates documentation. The command-line options can be given as long or short options, as is common for most GNU programs.

In principle, only 2 command-line options are needed:

package This specifies the name of the package for which documentation must be created. Exactly one package option can be specified.

input The name of a unit file for which documentation should be generated. This can be a simple filename, but can also contain some syntax options as they could be given to the Free Pascal scanner. More than one `input` option can be given, and documentation will be generated for all specified input files.

Some examples:

```
fpdoc --package=fcl --input=crt.pp
```

This will scan the crt.pp file and generate documentation for it in a directory called fcl.

```
fpdoc --package=fcl --input='-I../inc -S2 -DDebug classes.pp'
```

This will scan the file classes.pp, with the `DEBUG` symbol defined, the scanner will look for include files in the ../inc directory, and `OBJFPC`-mode syntax will be accepted.

(for more information about these options, see the Free Pascal compiler user's guide)

With the above commands, a set of documentation files will be generated in HTML format (this is the standard). There will be no description of any of the identifiers found in the unit's interface section, but all identifiers declarations will be present in the documentation.

The actual documentation (i.e. the description of each of the identifiers) resides in a description file, which can be specified with the `descr` option:

```
fpdoc --package=fcl --descr=crt.xml --input=crt.pp
```

This will scan the crt.pp file and generate documentation for it, using the descriptions found in the filecrt.xml file. The documentation will be written in a directory called fcl.

```
fpdoc --package=fcl --descr=classes.xml \
      --input='-I../inc -S2 -DDebug classes.pp'
```

All options should be given on one line. This will scan the file classes.pp, with the `DEBUG` symbol defined, the scanner will look for include files in the ../inc directory, and `OBJFPC`-mode syntax will be accepted.

More than one input file or description file can be given:

```
fpdoc --package=fcl --descr=classes.xml --descr=process.xml \
      --input='-I../inc -S2 -DDebug classes.pp' \
      --input='-I../inc -S2 -DDebug process.pp'
```

Here, documentation will be generated for 2 units: classes and `process`

The format of the description file is discussed in the next chapter.

Other formats can be generated, such as latex:

```
fpdoc --format=latex --package=fcl \
      --descr=classes.xml --descr=process.xml\
      --input='-I../inc -S2 -DDebug classes.pp' \
      --input='-I../inc -S2 -DDebug process.pp'
```

This will generate a LaTeX file called fcl.tex, which contains the documentation of the units classes and `process`. The latex file contains no document preamble, it starts with a chapter command. It is meant to be included (using the LaTeX include command) in a latex document with a preamble.

The output of FPDOC can be further customised by several command-line options, which will be explained in the next section.

3.2 FPDOC command-line options reference

In this section all FPDOC command-line options are explained.

3.2.1 content

This option tells FPDOC to generate a content file. A content file contains a list of all the possible anchors (labels) in the generated documentation file, and can be used to create cross-links in documentation for units in other packages, using the counterpart of the content option, the `import` option (section 3.2.7, page 11).

3.2.2 descr

This option specifies the name of a description file that contains the actual documentation for the unit. This option can be given several times, for several description files. The file will be searched relative to the current directory. No extension is added to the file, it should be a complete filename.

If the filename starts with an 'at' sign @, then it is interpreted as a text file which contains a list of filenames, one per line. Each of these files will be added to the list of description files.

The nodes in the description files will be merged into one big tree. This means that the documentation can be divided over multiple files. When merging the description files, nodes that occur twice will end up only once in the big node tree: the last node will always be the node that ends up in the parse tree. This means that the order of the various input commands or the ordering of the files in the file list is important.

Examples:

```
--descr=crt.xml
```

will tell FPDOC to read documentation from crt.xml, while

```
--descr=@fcl.lst
```

will tell FPDOC to read filenames from fcl.lst; each of the filenames found in it will be added to the list of files to be scanned for descriptions.

3.2.3 format

Specifies the output format in which the documentation will be generated. Currently, the following formats are known:

htm Plain HTML with 8.3 conforming filenames.

html HTML with long filenames.

xhtml XHTML with long filenames.

latex LaTex, which uses the fpc.sty style used by the Free Pascal documentation.

xml-struct Structured XML.

3.2.4 help

Gives a short copyright notice.

3.2.5 hide-protected

By default, the documentation will include descriptions and listings of protected fields and methods in classes or objects. This option changes this behaviour; if it is specified, no documentation will be generated for these methods. Note that public methods or properties that refer to these protected methods will then have a dangling (i.e. unavailable) link.

3.2.6 html-search

This option can be used when generating HTML documentation, to specify an url that can be used to search in the generated documentation. The URL will be included in the header of each generated page with a `Search` caption. The option is ignored for non-html output formats.

FPDOC does not generate a search page, this should be made by some external tool. Only the url to such a page can be specified.

Example:

```
--html-search=../search.html
```

3.2.7 import

Import a table of contents file, generated by FPDOC for another package with the `content` option (section 3.2.1, page 9). This option can be used to refer to documentation nodes in documentation sets for other packages. The argument consists of two parts: a filename, and a link prefix.

The filename is the name of the file that will be imported. The link prefix is a prefix that will be made to each HTML link; this needs to be done to be able to place the files in different directories.

Example:

```
--import=../fcl.cnt,../fcl
```

This will read the file fcl.cnt in the parent directory. For HTML documentation, all links to items in the fcl.cnt file, the link will be prepended with ../fcl.

This allows a setup where all packages have their own subdirectory of a common documentation directory, and all content files reside in the main documentation directory, as e.g. in the following directory tree:

```
/docs/fcl
     /fpdoc
     /fpgui
     /fpgfx
     /fpimg
```

The file fcl.cnt would reside in the docs directory. Similarly, for each package a contents file xxx.cnt could be places in that directory. Inside the subdirectory, commands as the above could be used to provide links to other documentation packages.

Note that for Latex documentation, this option is ignored.

3.2.8 input

This option tells FPDOC what input file should be used. The argument can be just a filename, but can also be a complete compiler command-line with options that concern the scanning of the Pascal source: defines, include files, syntax options, as they would be specified to the Free Pascal compiler when compiling the file. If a complete command is used, then it should be enclosed in single or double quotes, so the shell will not break them in parts.

It is possible to specify multiple input commands; they will be treated one by one, and documentation will be generated for each of the processed files.

3.2.9 lang

Select the language for the generated documentation. This will change all header names to the equivalent in the specified language. The documentation itself will not be translated, only the captions and headers used in the text.

Currently, valid choices are

de German.

fr French.

nl Dutch.

Example:

```
--lang=de
```

Will select German language for headers.

The language files should be installed correctly for this option to work. See the section on installing to check whether the languages are installed correctly.

3.2.10 latex-highlight

Switches on an internal latex syntax highlighter. This is not yet implemented. By default, syntax highlighting is provided by the syntax package that comes with Free Pascal.

3.2.11 output

This option tells FPDOC where the output file should be generated. How this option is interpreted depends on the format that is used. For latex, this is interpreted as the filename for the tex file. For all other formats, this is interpreted as the directory where all documentation files will be written. The directory will be created if it does not yet exist.

The filename or directory name is interpreted as relative to the current directory.

Example:

```
--format=html --output=docs/classes
```

will generate HTML documentation in the directory docs/classes.

```
--format=latex --output=docs/classes.tex
```

will generate latex documentation in the file docs/classes.

3.2.12 package

This option specifies the name of the package to be used. The package name will also be used as a default for the `output` option (section 3.2.11, page 12).

3.2.13 show-private

By default, no documentation is generated for private methods or fields of classes or objects. This option causes FPDOC to generate documentation for these methods and fields as well.

3.2.14 warn-no-node

If this option is given, then fpdoc will emit a warning if it cannot find a documentation node for some identifier. This can be used to see whether the description files are up-to-date, or whether they must be updated.

3.3 makeskel

3.3.1 introduction

The makeskel tool can be used to generate an empty description file for a unit. The description file will contain an element node for each identifier in the interface section of the Pascal unit.

It's usage is quite straightforward: the name of an input file (one or more) must be specified (as for FPDOC), an output file, and the name of a package:

```
makeskel --package=rtl --input=crt.pp --output=crt.xml
```

This will read the file crt.pp and will create a file crt.xml which contains empty nodes for all identifiers found in crt.pp, all in a package named `rtl`.

The `input` option can be given more than once, as for the fpdoc command:

```
makeskel --input='-Sn system.pp' --input=crt.pp --output=rtl.xml
```

As can be seen, the `input` option can contain some compiler options, as is the case for FPDOC. The above command will process the files system.pp and `crt.pp`, and will create **element** tags for the identifiers in both units in the file `rtl.xml`.

The output of makeskel is a valid, empty description file. It will contain a **module** tag for each unit specified, and each **module** will have **element** tags for each identifier in the unit.

Each **element** tag will by default contain **short**, **descr**, **errors** and **seealso** tags, but this can be customised.

3.4 Makeskel option reference

The output of makeskel can be customised using several options, which are discussed below.

3.4.1 descr

3.4.2 suse:descr

When in update mode (section 3.4.15, page 15), this opion can be used to add an existing documentation file, as for fpdoc. Nodes that are already in one of the existing documentation files will not be written to the output file.

3.4.3 disable-arguments

By default, for each function or procedure argument, a **element** tag will be generated. This option disables this behaviour: no **element** tags will be generated for procedure and function arguments.

3.4.4 disable-errors

If this option is specified, no **errors** tag will be generated in the element nodes. By default all element tags contain a **errors** node.

The **errors** tag is ignored when it is not needed; Normally, an **errors** tag is only needed for procedure and function elements.

3.4.5 disable-function-results

If this option is specified, then no **element** tag will be generated for function results. By default, makeskel will generate a result node for each function in the interface section. The result node is used in the documentation to document the return value of the function under a separate heading in the documentation page. Specifying this option suppresses the generation of the **element** tag for the function result.

3.4.6 disable-private

If this option is specified, then no **element** tags will be generated for private methods or fields of classes or objects. The default behaviour is to generate nodes for private methods or fields. It can be used to generate a skeleton for end-user and developer documentation.

3.4.7 disable-protected

If this option is specified, then no **element** tags will be generated for protected and private methods or fields of classes or objects. The default is to generate nodes for protected methods or fields. If this option is given, the option `-disable-private` is implied. It can be used to generate end-user-only documentation for classes.

3.4.8 disable-seealso

If this option is specified, no **seealso** tag will be generated in the element nodes. By default all **element** tags contain a **seealso** tag.

3.4.9 emitclassseparator

When this option is specified, at the beginning of the elements for the documentation of a class, a comment tag is emitted which contains a separator text. This can be useful to separate documentation of different classes and make the description file more understandable.

3.4.10 help

Makeskel emits a short copyright notice and exits when this option is specified.

3.4.11 input

This option is identical in meaning and functionality as the `input` option for FPDOC. (section 3.2.8, page 11) It specifies the Pascal unit source file that will be scanned and for which a skeleton description file will be generated. Multiple `input` options can be given, and **element** tags will be written for all the files, in one big output file.

3.4.12 lang

This option is used to specify the language for messages emitted by makeskel. The supported languages are identical to the ones for FPDOC:

de German.

fr French.

nl Dutch.

3.4.13 output

This option specifies the name of the output file. A full filename must be given, no extension will be added. If this option is omitted, the output will be sent to standard output.

When using update mode, the output file name should not appear in the list of existing documentation files. The makeskel program will do some elementary checks on this.

3.4.14 package

This option specifies the package name that will be used when generating the skeleton. It is a mandatory option.

3.4.15 update

This option tells makeskel to create an update file: it will read description files (section 3.2.2, page 9) and will only create documentation nodes for identifiers which do not yet have a documentation node in the read documentation files. The output file in this case can be merged with one (or more) of the documentation files: it's name should not appear in the list of existing documentation files. The makeskel program will do some elementary checks on this.

Chapter 4

The description file

4.1 Introduction

The description file is a XML document, which means that it is a kind of HTML or SGML like format, however it is more structured than HTML, making it easier to parse - and makes it easier to connect or merge it with a Pascal source file. Since the allowed syntax uses a lot of HTML tags, this makes it easy to write code for those that are familiar with writing HTML.

More information about the XML format, SGML and HTML can be found on the website of the W3 (World Wide Web) consortium: http://www.w3.org/

The remaining of this chapter assumes a basic knowledge of tags, their attributes and markup language, so these terms will not be explained here.

The minimal documentation file would look something like this:

```
<?xml version="1.0" encoding="ISO8859-1"?>
<fpdoc-descriptions>
<package name="fpc">
<module name="Classes">
</module>
</fpdoc-description>
</package>
```

The header **xml** tag is mandatory, and indicates that the file contains a documentation XML document.

Inside the document, one or more top-level **fpdoc-descriptions** tags may appear. Each of these tags can contain one or more **package** tags, which must have a *name* attribute. The name attribute will be used by fpdoc to select the documentation nodes.

Inside a **package** tag, one or more **module** tags may appear. there should be one **module** tag per unit that should be documented. The value of the *name* attribute of the `module` should be the name of the unit for which the **module** tag contains the documentation. The value of the `name` attribute is case insensitive, i.e.

```
<module name="CRT">
```

can be used for the documentation of the `crt` unit.

As it is above, the documentation description does not do much. To write real documentation, the **module** tag must be filled with the documentation for each identifier that appears in the unit interface header.

For each identifier in the unit interface header, the **module** should contain a tag that documents the identifier: this is the **element** tag. The name attribute of the element tag links the documentation to the identifier: the *name* attribute should have as value the fully qualified name of the identifier in the unit.

For example, to document the type

```
Type
  MyEnum = (meOne,meTwo,meThree);
```

an **element** tag called `myenum` should exist:

```
<element name="myenum">
</element>
```

But also for each of the three enumerated values an **element** tag should exist:

```
<element name="myenum.meOne">
</element>
<element name="myenum.meTwo">
</element>
<element name="myenum.meThree">
</element>
```

As it can be seen, the names of the identifiers follow a hierarchical structure. More about this in the next section.

Now the tags for the types are present, all that should be done is to fill it with the actual description. For this, several tags can be placed inside a **element** tag. The most important tag is the **descr** tag. The contents of the **descr** tag will be used to describe a type, function, constant or variable:

```
<element name="myenum">
<descr>
The MyEnum type is a simple enumeration type which is not
really useful, except for demonstration purposes.
</descr>
</element>
```

A second important tag is the **short** tag. It should contain a short description of the identifier, preferably a description that fits on one line. The **short** tag will be used in various overviews, at the top of a page in the HTML documentation (a synopsis) or will be used instead of the **descr** tag if that one is not available. It can also be used in different other cases: For instance the different values of an enumeration type will be laid out in a table, using the **short** description:

```
<element name="myenum.meOne">
<short>The first enumeration value</short>
</element>
<element name="myenum.meTwo">
<short>The second enumeration value</short>
</element>
<element name="myenum.meThree">
<short>The third enumeration value</short>
</element>
```

This will be converted to a table looking more or less like this:

meOne	The first enumeration value
meTwo	The second enumeration value
meThree	The third enumeration value

For functions and procedures, a list of possible error conditions can be documented inside a **errors** tag. This tag is equivalent to the **descr** tag, but is placed under a different heading in the generated documentation.

Finally, to cross-reference between related functions, types or classes, a **seealso** tag is also introduced. This will be used to lay out a series of links to related information. This tag can only have sub-tags which are `link` tags. For more about the **link** tag, see **link** (24).

To add a section or page of documentation that is not directly related to a single identifier in a unit, a **topic** tag can be used. This tag can appear inside a `package` or `module` tag, and can contain a **short** or **descr** tag. The contents of the **short** tag will be used for the title of the section or page. In on-line formats, a short list of related topics will appear in the package or module page, with links to the pages that contain the text of the topics. In a linear format, the topic sections will be inserted before the description of all identifiers.

If the topic appears in a **package** tag, then it can be nested: 2 levels of topics may be used. This is not so for topics inside a module: they can not be nested (the level of nesting in a linear documentation format is limited).

The following is an example of a valid topic tag:

```
<module name="CRT">
<topic name="UsingKeyboard">
<short>Using the keyboard functions</short>
<descr>
To use the keyboard functions of the CRT unit, one...
</descr>
</topic>
```

Topic nodes can be useful to add 'how to' sections to a unit, or to provide general IDE help.

4.2 Element names and cross-referencing

4.2.1 Element name conventions

As mentioned in the previous section, the `element` tag's *name* attribute is hierarchical. All levels in the hierarchy are denoted by a dot (.) in the name attribute.

As shown in the previous example, for an enumerated type, the various enumeration constants can be documented by specifying their name as `enumname.constname`. For example, given the type

```
Type
  MyEnum = (meOne,meTwo,meThree);
```

The various enumeration values can be documented using the element names `MyEnum.meOne`, `MyEnum.meTwo` and `MyEnum.meThree`:

```
<element name="myenum.meOne">
</element>
<element name="myenum.meTwo">
</element>
<element name="myenum.meThree">
</element>
```

Note that the casing of the name attribute need not be the same as the casing of the declaration.

This hierarchical structure can be used for all non-simple types:

- Enumeration type values.
- Fields in records, objects, classes. For nested record definitions, multiple levels are possible in the name.
- Methods of classes and objects.
- Properties of classes.
- Function and procedure arguments.
- Function results. The name is always the function name followed by `Result`.

4.2.2 Cross referencing: the `link` tag

To refer to another point in the documentation (a related function, class or whatever), a **link** tag exists. The `link` tag takes as a sole attribute a target *id* attribute. The link tag usually encloses a piece of text. In the HTML version of the documentation, this piece of text will function as a hyperlink. In the latex version, a page number reference will be printed.

The *id* attribute contains the name of an element to which the link refers. The name is not case sensitive, but it must be a fully qualified name.

An example of the link type would be:

```
The <link id="MyEnum">MyEnum</link> type is a simple type.
```

The link attribute also has a short form:

```
The <link id="MyEnum"/> type is a simple type.
```

In the short form, the value of the *id* attribute will be used as the text which will be hyperlinked. This is especially useful in the **seealso** tag.

To refer to a type in another unit, the unit name must be prepended to the *id* attribute:

```
<link id="myunit.myenum"/>
```

will link to the `myenum` type in a unit named myunit.

To refer to a node in the documentation of another package, the package name should be prepended to the *id* attribute, and it should be prepended with the hash symbol (#):

```
<link id="#fcl.classes.sofrombeginning"/>
```

will link to the constant `sofrombeginning` in the classes unit of the FCL reference documentation. Note that for this to work correctly, the contents file which was created when generating the documentation of the type must be imported correctly (see the `import` option).

4.3 Tag reference

4.3.1 Overview

The tags can roughly be divided in 2 groups:

1. Documentation structure tags. These are needed for fpdoc to do it's work. They determine what elements are documented. See table (4.1)

2. Text structure and formartting tags. These tags indicate blocks of text, such as paragraphs, tables, lists and remarks, but also specify formatting: apply formatting (make-up) to the text, or to provide links to other parts of the text. These mostly occur in text structure tags. See table (4.2)

Table 4.1: Documentation structure tags

Tag	Description	Page
descr	Element description	22
element	Identifier documentation	23
errors	Error section	23
fpdoc-description	Global tag	24
module	Unit tag	25
package	Package global tab	26
seealso	Cross-reference section	27
short	Short description	28
topic	Topic page	29

Table 4.2: Text formatting tags

Tag	Description	Page
b	Format bold	21
caption	Specify table caption	21
code	Syntax highlight code	21
dd	definition data	22
dl	definition list	22
dt	Definition term	23
i	format italics	24
li	list element	24
link	Cross-reference	24
ol	numbered list	25
p	paragraph	26
pre	Preformatted text	27
remark	remark paragraph	27
table	Table	28
td	Table cell	29
th	Table header	29
tr	Table row	30
u	format underlined	30
ul	bulleted list	30
var	format as variable	31

The nodes for formatting a text resemble closely the basic HTML formatting tags with the following exceptions:

- Each opening tag must have a corresponding closing tag.
- Tags are case sensitive.
- Tables and paragraphs are at the same level, i.e. a table cannot occur inside a paragraph. The same is true for all 'structural' tags such as lists,

Also, if special formatting tags such as a table or lists are inserted, then the remaining text must be inside a paragraph tag. This means that the following is wrong:

```
<descr>
Some beginning text
<ol>
<li>A list item</li>
</ol>
some ending text
</descr>
```

Instead, the correct XML should be

```
<descr>
<p>Some beginning text</p>
<ol>
<li>A list item</li>
</ol>
<p>some ending text</p>
</descr>
```

4.3.2 b : format bold

This tag will cause the text inside it to be formatted using a bold font.

Example:

```
Normal text <b>Bold text</b> normal text.
```

will be formatted as:
Normal text **Bold text** normal text.

See also: **i** (24), **u** (30).

4.3.3 caption : Specify table caption

This tag can occur inside a **table** tag and serves to set the table caption.

Example

```
<table>
<caption>This caption will end up above the table</caption>
<th><td>Column 1</td><td>Column 2</td></th>
</table>
```

See also: **table** (28)

4.3.4 code : format as pascal code

The **code** tag serves to insert Pascal code into the text. When possible this code will be highlighted in the output. It can be used to put highlighted Pascal code in the documentation of some identifier. It can occur inside **descr** or **errors** tags.

Note that any text surrounding the **code** tag should be placed in paragraph tags **p**.

Example:

```
<code>
With Strings do
  For i:=Count-1 downto 0 do
    Delete(i);
</code>
```

Seealso: **pre** (27), **p** (26)

4.3.5 descr : Descriptions

This is the actual documentation tag. The contents of this tag will be written as the documentation of the element. It can contain any mixture of text and markup tags. The **descr** tag can only occur inside a **element** or **module**.

Example:

```
<element name="MyEnym">
<descr>Myenum is a simple enumeration type</descr>
</element>
```

See also: **element** (23), **short** (28), **errors** (23), **seealso** (27)

4.3.6 dd : definition data.

The **dd** tag is used to denote the definition of a term in a definition list. It can occur only inside a definition list tag (**dl**), after a definition term (**dt**) tag. It's usage is identical to the one in HTML.

Example:

```
<dl>
<dt>FPC</dt><dd>stands for Free Pascal Compiler.</dd>
</dl>
```

Will be typeset as

FPC stands for Free Pascal Compiler.

See also: **dl** (22), **dt** (23), **ol** (25), **ul** (30)

4.3.7 dl : definition list.

Definition lists are meant to type a set of terms together with their explanation. It's usage is identical to the one in HTML, with the exception that it cannot occur inside ordinary text: surrounding text should always be enclosed in paragraph tags.

Example:

```
<dl>
<dt>meOne</dt><dd>First element of the enumeration type.</dd>
<dt>meTwo</dt><dd>Second element of the enumeration type.</dd>
<dt>meThree</dt><dd>Thir element of the enumeration type.</dd>
</dl>
```

Will be typeset as

meOne First element of the enumeration type.

meTwo Second element of the enumeration type.

meThree Third element of the enumeration type.

See also: **dt** (23), **dd** (22), **ol** (25), **ul** (30)

4.3.8 dt : definition term.

The **dt** tag is used in definition lists to enclose the term for which a definition is presented. It's usage is identical to the usage in HTML.

Example:

```
<dl>
<dt>FPC</dt><dd>stands for Free Pascal Compiler.</dd>
</dl>
```

Will be typeset as

FPC stands for Free Pascal Compiler.

See also: **dl** (22), **dd** (22), **ol** (25), **ul** (30)

4.3.9 element : Identifier documentation

The **element** contains the documentation for an identifier in a unit. It should occur inside a `module` tag. It can contain 4 tags:

short For a one-line description of the identifier. Is used as a header or is used in overviews of constants, types, variables or classes.

descr Contains the actual description of the identifier.

errors For functions an procedures this can be used to describe error conditions. It will be put in a separate section below the description section.

seealso Used to refer to other nodes. will be typeset in a separate section.

The **element** tag should have at least the *name* attribute, it is used to link the element node to the actual identifier in the Pascal unit. Other attributes may be added in future.

Example:

```
<element name="MyEnym">
<descr>Myenum is a simple enumeration type</descr>
</element>
```

See also: **descr** (22), **short** (28), **errors** (23), **seealso** (27)

4.3.10 errors : Error section.

The **errors** tag is used to document any errors that can occur when calling a function or procedure. it is placed in a different section in the generated documentation. It occurs inside a **element** tag, at the same level as a **descr** or **short** tag. It's contents can be any text or formatting tag.

Example:

```
<element name="MyDangerousFunction">
<descr>MyDangerousFunction is a dangerous function</descr>
<errors>When MyDangerousFunction fails, all is lost</errors>
</element>
```

See also: **descr** (22), **short** (28), **element** (23), **seealso** (27)

4.3.11 fpdoc-description : Global tag

The **fpdoc-description** tag is the topmost tag in a description file. It contains a series of **package** tags, one for each package that is described in the file.

See also: **package** (26)

4.3.12 i : Format italics

The **i** tag will cause its contents to be typeset in italics. It can occur mixed with any text.

Example:

```
Normal text <i>italic text</i> normal text.
```

will be formatted as:
Normal text *italic text* normal text.

See also: **b** (21), **u** (30)

4.3.13 li : list element

The tag **li** denotes an element in a **ol** or **ul** list. The usage is the same as for it's HTML counterpart: It can occur only inside one of the **ol** or **ul** list tags. It's contents may be arbitrary text and formatting tags, contrary to HTML tags, the **li** tag always must have a closing tag. Note that it cannot be used in a definition list (**dl** (22)).

Example:

```
<ul>
<li>First item in the list</li>
<li>Second item in the list</li>
</ul>
```

Will be typeset as

- First item in the list.
- Second item in the list.

See also: **ol** (25), **ul** (30).

4.3.14 link : Cross-reference

The **link** tag is used to insert a reference to an element inside some piece of text or inside the **seealso** section. It is similar in functionality to the

```
<A HREF="SomeAnchor">some linked text</A>
```

construct in HTML.

The mandatory *id* attribute of the **link** tag should have the name of an element tag in it. This name is not case sensitive. FPDOC will issue a warning if it cannot find a matching name. It will look for matching names in the current file, and in all content files that have been specified with the `import` command-line option.

The link tag can exist in 2 forms: one with separate closing tag, surrounding a piece of text, one without separate closing tag. If a piece of text is surrounded by the link tag, then the text will be converted to a hyperlink in the HTML documentation. If there is no surrounded text, then the value of the *id* attribute will be used as the text. This means that

```
<link id="TStream">TStream</link>
```

and

```
<link id="TStream"/>
```

are completely equivalent.

Example:

```
The <link id="TStringlist">stringlist</link> class is a descendent of the
<link id="TStrings"/> class.
```

See also: **element** (23), the `import` option (section 3.2.7, page 11).

4.3.15 module : Unit reference

The **module** tag encloses all **element** tags for a unit. It can contain only **element** tags for all identifiers in the unit and a **descr** tag describing the unit itself. The **module** tag should occur inside a **package** tag.

The *name* attribute should have as a value the name of the unit which is described by the module. This name is not case sensitive.

Example:

```
<module name="classes">
<descr>
The classes unit contains basic class definitions for the FCL.
</descr>
</module>
```

See also: **package** (26), **descr** (22), **element** (23)

4.3.16 ol : Numbered list.

The **ol** tag starts a numbered list. It can contain only **li** (24) tags, which denote the various elements in the list. Each item will be preceded by a number. The **ol** tag can occur inside a text, but surrounding text should always be enclosed in a **p** (26) paragraph tag, i.e. an **ol** tag should occur always at the same level as a **p** tag.

Example:

```
<p> some text before</p>
<ol>
```

```
<li>First item in the list</li>
<li>Second item in the list</li>
</ol>
```

Will be typeset as:

some text before

1. First item in the list.
2. Second item in the list.

See also: **li** (24), **ul** (30).

4.3.17 p : Paragraph

The **p** tag is the paragraph tag. Every description text should be enclosed in a **p** tag. Only when there is only one paragraph (and no lists or tables or remarks) in a description node, then the **p** tag may be skipped.

Note that if a description node contains a **table**, **pre**, **code** or any list tag, then the text surrounding these tags *must* be put inside a **p** paragraph tag. This is different from the behaviour in HTML.

The paragraph tag must always have an opening tag and a closing tag, unlike html where only the opening tag may be present.

Example:

```
<descr>
This is a paragraph which need not be surrounded by paragraph tags.
</descr>

<descr>
<p>
This is the first paragraph.
</p>
<p>
This is the second paragraph.
</p>
</descr>
```

See also: **table** (28), **dl** (22), **remark** (27),**code** (21), **ol** (25),**ul** (30),**ol** (25)

4.3.18 package : Package reference

The **package** tag indicates the package for which the description file contains documentation. A package is a collection of units which are logically grouped together (for a library, program, component suites). The *name* attribute of the **package** tag will be used to select the package node in the description file: Only the **package** node with name as specified by the `package` command-line option will be used when generating documentation. All other package nodes will be ignored.

The **package** node must always reside in a **fpdoc-description** node. It can contain a **descr** node, and various **module** nodes, one node per unit in the package.

See also: **fpdocdescription** (24), **module** (25), **descr** (22)

4.3.19 pre : Insert text as-is

The **pre** tag can be used to insert arbitrary text in the documentation. The text will not be formatted in any way, and will be displayed as it is encountered in the description node. It is functionally equivalent to the **pre** tag in HTML.

Note that if there is text surrounding the **pre** tag, it should be placed inside a **p** paragraph tag.

Example:

```
<pre>
This is some text.
  This is some more text

    And yet more text...
</pre>
```

This will be typeset as:

```
This is some text.
  This is some more text

    And yet more text...
```

See also: **code** (21), **p** (26)

4.3.20 remark : format as remark

A **remark** tag can be used to make a paragraph stand out. The **remark** is equivalent to the **p** tag, but it's contents is formatted in a way that makes it stand out from the rest of the text.

Note that any text before or after the **remark** tag must be enclosed in paragraph (**p**) tags.

Example:

```
<p>Normal text.</p>
<remark>
This text will stand out.
<example>
<p>Again normal text.</p>
```

Will be formatted as

Normal text.

Remark: This text will stand out.

Again normal text.

See also: **p** (26), **code** (21), **pre** (27)

4.3.21 seealso : Cross-reference section

The **seealso** section can occur inside any **element** tag, and will be used to create a list of cross-references. The contents of the **seealso** tag is a list of **link** tags. No other text is allowed inside this tag. Note that both the long and short form if the link tag may be used.

Example:

```
<seealso>
<link id="TStrings"/>
<link id="TStringList.Create">Create</link>
</seealso>
```

See also: **link** (24), **element** (23)

4.3.22 short : Short description

The `short` description is used to give a short description of an identifier. If possible, the description should fit on a single line of text. The contents of this tag will be used for the following purposes:

- Used as the synopsis on a page that describes an identifier.
- Used in overviews of constants, types, variables, record fields, functions and procedures, classes, and for method and property listings of classes.
- Replaces the **descr** tag in an **element** if no **descr** tag is present.
- In the description of an enumerated type, the enumeration values are typeset in a table, each row containing the name of the value and the short description.
- In the description of a function or procedure that accepts arguments, the arguments are followed by their short description.
- In the declaration of a class or record, each method, field or property is followed by the short description.

Example:

```
<element name="MyEnum.meOne">
<short>First element of MyEnum</short>
</element>
```

See also: **element** (23), **descr** (22)

4.3.23 table : Table start

The `table` tag starts a table, as in HTML. A table can contain **tr** (table row), **th** (table header row) or **caption** tags. Any text surrounding the table must be enclosed in paragraph tags (**p**).

Example:

```
<table>
<caption>
<var>TALignment</var> values and their meanings.
</caption>
<th><td>Value</td><td>Meaning</td></th>
<tr>
  <td><var>taLeftJustify</var></td>
  <td>Text is displayed aligned to the left.</td>
</tr>
<tr>
  <td><var>taRightJustify</var></td>
  <td>Text is displayed aligned to the right</td>
```

```
</tr>
<tr>
  <td><var>taCenter</var></td>
  <td>Text is displayed centred.</td>
</tr>
</table>
```

Will be formatted approximately as follows:

Value	Meaning
`taLeftJustify`	Text is displayed aligned to the left.
`taRightJustify`	Text is displayed aligned to the right
`taCenter`	Text is displayed centred.

See also: **th** (29), **caption** (21), **tr** (30), **p** (26)

4.3.24 td : Table cell

The **td** tag is used to denote one cell in a table. It occurs inside a **tr** or **th** tag, and can contain any text and formatting tags.

Example:

```
<table>
<tr><td>Cell (1,1)</td><td>Cell (2,1)</td></tr>
<tr><td>Cell (1,2)</td><td>Cell (2,2)</td></tr>
</table>
```

Will be formatted approximately as

Cell (1,1)	Cell (2,1)
Cell (1,2)	Cell (2,2)

See also: **table** (28), **th** (29), **tr** (30)

4.3.25 th : Table header

The `th` table header tag is used to denote the first row(s) of a table: It (they) will be made up differently from the other rows in the table. Exactly how it is made up depends on the format. In printed documentation (latex) a line will be drawn under the row. In HTML, the font and background may be formatted differently.

The **th** tag can only occur inside a **table** tag, and can contain only **td** tags.

Example:

```
<table>
<th><td>Cell (1,1)</td><td>Cell (2,1)</td></th>
<tr><td>Cell (1,2)</td><td>Cell (2,2)</td></tr>
</table>
```

Will be formatted approximately as

Cell (1,1)	Cell (2,1)
Cell (1,2)	Cell (2,2)

See also: **tr** (30), **table** (28)

The topic tag starts a topic page or section. A topic page or section is not linked to an identifier in some unit: it exists by itself. It must be inside a **package** or **module** tag. It must have a *name*

attribute (for cross-referencing). If it appears inside a **package**, it can be nested: a **topic** may be inside another **topic** tag.

```
<module name="CRT">
<topic name="UsingKeyboard">
<short>Using the keyboard functions</short>
<descr>
To use the keyboard functions of the CRT unit, one...
</descr>
</topic>
```

4.3.26 tr : table row

The **tr** tag denotes a row in a table. It works the same as in HTML. It can occur only in a **table** tag, and should contain only **td** table data tags.

Example:

```
<table>
<tr><td>Cell (1,1)</td><td>Cell (2,1)</td></tr>
<tr><td>Cell (1,2)</td><td>Cell (2,2)</td></tr>
</table>
```

Will be formatted approximately as

Cell (1,1)	Cell (2,1)
Cell (1,2)	Cell (2,2)

See also: **table** (28), **th** (29), **td** (29)

4.3.27 u : Format underlined

Example:

```
Normal text <u>underlined text</u> normal text.
```

will be formatted as:
Normal text underlined text normal text.

See also: **i** (24), **b** (21).

4.3.28 ul : bulleted list

The `ul` tag starts a bulleted list. This works as under HTML, with the exception that any text surrounding the list must be enclosed in paragraph tags (**p**). The list elements should be enclosed in **li** tags.

Example:

```
<p> some text before</p>
<ol>
<li>First item in the list</li>
<li>Second item in the list</li>
</ol>
```

Will be typeset as:

some text before

- First item in the list.
- Second item in the list.

See also: **li** (24), **ol** (25).

4.3.29 var : variable

The **var** tag is used to mark a piece of text as a variable (or, more general, as an identifier). It will be typeset differently from the surrounding text. Exactly how this is done depends on the output format.

Example:

```
The <var>Items</var> property gives indexed access to...
```

Will be typeset as

The `Items` property gives indexed access to...

See also: **b** (21), **u** (30), **i** (24), **code** (21)

Chapter 5

Generated output files.

5.1 HTML output

The HTML output consists of the following files, per unit:

1. A general unit description with the contents of the module's **descr** tag. The `uses` clause is documented here as well. All units in the `uses` clause together with their **short** description tags are typeset in a table.
2. A listing of all constants in the unit.
3. A listing of all types in the unit (except classes).
4. A listing of all variables in the unit.
5. A listing of all functions/procedures in the unit.
6. A listing of all classes in the unit.

All these overviews are hyperlinked to pages which contain the documentation of each identifier. Each page starts with the name of the identifier, plus a synopsis (made from the **short** tag's contents). After that follows the declaration, and the description. The description is filled with the **descr** node of the identifiers **element** tag.

If an `errors` tag was present, an 'Errors' section follows the description. Similarly, if there is a **seealso** tag, a 'See also' section with cross-reference links is made.

For classes, the declaration contains hyperlinks to separate pages which document all the members of the class. Each member in the declaration is followed by the **short** tag of the member's **element** tag, if one exists. As an extra, the class hierarchy is given, plus links to pop-up pages (if JavaScript is available, otherwise they are normal links) which contain alphabetical or hierarchical listings of the methods, fields or properties of the class.

For functions and procedures, the declaration will be typeset in such a way that all function arguments (if they are present) are in tabular format, followed by the short description of the argument. If it concerns a function, and a result element exists, the result description will be provided in a separate section, before the actual description.

The declaration of an enumerated type will be laid out in a table, with the enumeration value at the left, and the short description node of the value's element.

5.2 Latex output

The latex output is in one big file with the name of the package as specified on the command line. in this file, one chapter is made per unit.

Per unit the following sections are made:

1. A section with all constants.
2. A section with all types.
3. A section with all variables.
4. A section with all functions and procedures.
5. A section per declared class.

For the constants, types and variables, the declaration is given, followed by the **descr** node of the element corresponding to the identifier. All other nodes are ignored.

For functions and procedures, a subsection is made per procedure or function. This subsection consists of a list with the following entries:

Synopsis filled with the contents of the **short** tag.

Declaration Filled with the declaration of the function.

Arguments A tabular description of all arguments, if **short** tags are found for them.

Description Description of the function. Filled with the contents of the **descr** tag.

Errors Description of any error conditions. Filled with the contents of the **errors** tag.

See Also Cross-references to other functions. Filled with the contents of the **seealso** tag.

For classes, a subsection is made with an overview of implemented methods. Then a subsection is presented with available properties.

Then follows a subsection per method. These are formatted as a function, with an additional **Visibility** list element, giving the visibility of the function.

After the methods, a list of properties is given , formatted as a method, with an additional `Access` list element, specifying whether the property is read/write or not.