# Manual for pasdoc 0.6.20

Marco Schmidt

April 20, 2000

## Contents

# 1 Introduction

Disclaimer: This manual is still under development, so it will contain unfinished sections and sentences that won't make sense. I'll try to work on it more intensively later - promised!

Pasdoc creates documentation for Pascal unit files. Descriptions for variables, constants, types (called 'items' from now on) are taken from comments stored in the interface sections of unit source code files, each comment must be placed directly before the item's declaration. This way, you as a programmer can easily generate reference manuals of your libraries without having to deal with the details of document formats like HTML or LaTeX. Moreover, you can edit the source code and its descriptions in one place, no need to add or modify explanations in other files. The rest is done automatically, you should write a script / batch file that does the actual call to pasdoc... Download the latest version from

`http://pasdoc.sourceforge.net.`

For an example of source code that can be used with pasdoc, try the pasdoc sources themselves - type `pasdoc[.exe] -h *.pas` to generate HTML documentation. This way, you can get an impression of the looks of HTML or Tex documents and find out whether it matches your taste.

You can compile pasdoc with Turbo Pascal, Borland Pascal or Free Pascal (FPC). Delphi 5 support will be added soon – the current version crashes for some reason. Free Pascal is a GPL'd Pascal compiler, it is mostly compatible with Turbo Pascal and Delphi, available for Linux x86, Win32, Dos and OS/2.

# 2 Directives

As you may know, Pascal allows for *directives* in the source code. These are comments that contain commands for the compiler introduced by the dollar sign.

To distinguish different compilers, libraries or development stages, *conditional directives* make it possible to make the compiler ignore part of the file. An example:

```
unit SampleUnit;

{$ifdef WIN32}
uses Windows, WinProcs;
procedure WindowMove(H: TWindowHandle; DX, DY: Integer);
procedure WindowPrintText(H: TWindowHandle; X, Y: Integer; S: String);
{$else}
procedure ClearConsole;
procedure PrintText(S: String);
{$endif}

{$define DEBUG}
{$undef OPTIMIZE}
```

The `ifdef` part checks if a conditional directive called `WIN32` is currently defined (that would be the case for Delphi or FPC/Win32). If this is true, all code until `else` or `endif` are compiled, everything between `else` and `endif` is ignored. The contrary would apply if

the directive is not defined, e.g. under FPC/DOS or Borland Pascal. These statements can also be nested. Using `define` and `undef`, a programmer can add and delete directives, in the above example DEBUG and OPTIMIZE.

As pasdoc loads Pascal files in a similar way a compiler does, it must be able to regard conditional directives. All define and undef parts are evaluated by pasdoc, modifying an internal list of directives as source code is parsed.

Different from a real compiler, pasdoc starts with an empty list of conditional directives. To get back to the above example, if you want to write documentation for the `WIN32` code part, you must explicitly tell pasdoc that you want this directive defined. You can do so using the *Specify directive* or *Add directives from file* switch (see sections 5.4.7 and 5.4.9).

Next to those directives already presented, pasdoc only supports include files:

```
type TInteger = Integer;

{$I numbers.inc}

const MAX_FILES = 12;
```

In the above code, pasdoc would parse `TInteger`, get the include directive and start parsing the include file `numbers.inc`. This file could contain other directives, types or whatever. It is treated as it would be treated by any Pascal compiler.

Pascal compilers also know *switch directives*. These are boolean options, either on or off. They can be checked similar to conditional directives with the `$ifopt` directive. Pasdoc does not yet fully support these, but at least does not give up when it encounters one.

# 3 Adding descriptions

You can provide documentation for

- types,
- variables,
- constants,
- objects, classes, interfaces and
- units.

For each class, interface or object you can describe its fields, methods and properties. Write a comment describing the item (from now on I'll call this a description) before the name of the item itself is declared. The only exception are units - write the comment before the declaration of the `unit` keyword. Example:

```
type
  { This record type stores all information on a customer, including
    name, age and gender. }
  TCustomer = record
    Name: String;
```

```
    Age: Byte;
    Gender: Boolean;
  end;

{ Initializes a TCustomer record with the given parameters. }
procedure InitCustomer(Name: String; Age: Byte;
  Gender: Boolean; var Customer: TCustomer);
```

An interesting feature of pasdoc is its ability to link items from within comments. If you are currently writing about an array of integers `TIntArray` you've declared as a type, you might mention that the number of values it can store is specified in the constant `MAX_INTS`. You've probably already documented this constant when you declared it earlier in the same or another unit. Now, if you write `@link(MAX_INTS)` instead of simply `MAX_INTS`, pasdoc knows that you are referring to another item it has found or will find in the list of units you gave to it. After all input files have been parsed, pasdoc will start substituting all occurrences of `@link(something)` with "real" links. Depending on the type of output, these links could be hyperlinks (in HTML) or page references (in LaTeX). If the current output format is HTML, the description of `TIntArray` would contain a link to `MAX_INTS`. Viewing `TIntArray`'s description in your favourite web browser you'd now be able to click on `MAX_INTS` and the browser would jump to the definition of `MAX_INTS`, where you'd find more information on it.

If pasdoc cannot resolve a link (for whatever reasons), it will issue a warning to standard output and will write the content of `@link()` to the documentation file, not as a link, but in bold font.

## 4 Formatting your comments

In the description part for units, classes, interfaces and objects, you can include additional information on the author, date created, date last modified and an abstract. These information tags consist of the at character `@`, followed by an identifier like `abstract` and then text included in parentheses. As an example, lets take the well-known `DOS` unit. Its top part could look like this:

```
{
@abstract(provides access to file and directory operations)
@author(John Doe <doe@john-doe.com>)
@created(July 12, 1997)
@lastmod(June 20, 1999)
The DOS unit provides functionality to get information on files and
directories and to modify some of this information.
This includes disk space (e.g. @link(DiskFree)), access rights, file
and directory lists, changing the current directory, deleting files
and directories and creating directories.
Some of the functions are not available on all operating systems.
}
unit DOS;
```

Pasdoc would read this comment and store it with the unit information. After all Pascal source code files you gave to pasdoc have been read, pasdoc will try to process all tags, i.e., all

words introduced by a `@` character. If you want to use the character `@` itself, you must write it twice so that pasdoc knows you don't want to specify a tag. Note in the example above that the character does not need this special treatment within the parentheses, as shown in the author tag at the example of the email address. Following a list of all tags that you may use in pasdoc. A tag like `@link` must always be followed by an opening and then a closing parenthesis, even if you add nothing between them.

### 4.1 abstract

For some item types like classes or units you may write very large descriptions to give an adequate introduction. However, these large texts are not appropriate in an overview list. Use the abstract tag to give a short explanation of what an item is about. One row of text is a good rule of thumb. Of course, there should only be one abstract tag per description. Abstract tags in every item but classes, interfaces, objects or units will be ignored.

### 4.2 author

For each author who participated in writing an item, one author tag should be added. However, author tags will only result in documentation output for classes, interfaces, objects and units.

### 4.3 created

This tag should contain the date the item was created. At most one created tag should be in a description. Created tags will only result in documentation output for classes, interfaces, objects and units. There is no special format that must be followed.

### 4.4 exclude

If an exclude tag is found in a description, the item will not appear in the documentation. As a logical consequence, no information except the exclude tag itself should be written to the description. Whenever high-level items like units or classes are excluded from the documentation process, all items contained in them will not appear as well, e.g. constants or functions in an excluded unit or methods and fields in an excluded class.

### 4.5 lastmod

This tag should contain the date the item was last modified. At most one created tag should be in a description. Lastmod tags will only result in documentation output for classes, interfaces, objects and units. There is no special format that must be followed.

### 4.6 link

Use this tag to make your documentation more convenient to the reader. Whenever you mention another item in the description of an item, enclose the name of the mentioned item in a link tag, e.g.
`@link(GetName)`.
This will result in a hyperlink in HTML and a page reference in LaTeX.

# 5 Switches

This is a list of all switches (program parameters) supported by `pasdoc`. Enter `pasdoc --help` at the command line to get this list. Make sure you keep the exact spelling of the switches, pasdoc is case-sensitive. Most switches exist in two variations, a short one with a single dash and a longer one with two dashes. You can use either switch to get the same effect.

## 5.1 Documentation file format

After loading all Pascal source code files, pasdoc will write one or more output files, depending on the output file format. Choose the output format according to your needs – you might want to create several versions for

### 5.1.1 HTML

`-h`
`--html`

This switch makes pasdoc write HTML (Hypertext Markup Language) output. HTML files are usually displayed in a web browser, available on all modern computer systems. It is the default output file format. Several files will be created for this output type, one for each unit, class, interface and object, additionally some overview files with lists of all constants, types etc.

### 5.1.2 LaTeX

`-l`
`--latex`

This switch makes pasdoc write output that can be interpreted using TeX(or its derivate LaTeX). A single output file `docs.tex` will be created. With `latex docs.tex` you will create a file called `docs.dvi` which can then be converted to PostScript: `dvips docs.dvi`. If you prefer Adobes PDF file format, you might create it from the PostScript output using `ps2pdf docs.ps` or by using `pdflatex docs.tex` (the latter gives a nicer result). Instead of hyperlinks like in HTML, links in the comments will result in numbers that refer to the page where the linked item is explained.

## 5.2 Format-specific switches

The following switches can only be used with one output file format and are useless for the others.

### 5.2.1 No headers and footers in LaTeX

`-j`
`--notexheaders`

This switch will keep pasdoc from writing out standard declarations like `begin{document}` when writing output for LaTeX. This way, you will be able to include pasdoc's output by simply stating

```
\include(docs.tex)
```

in your LaTeXdocument.

#### 5.2.2 No homepage link

```
-n
--nohomepage
```

By default, pasdoc includes some information on itself and the document creation time at the bottom of each generated HTML file. This switch keeps pasdoc from adding that information.

### 5.3 Output language switches

You can specify the language that will be used for words in the output like *Methods* or *Classes, interfaces and objects*. Your choice will not influence the status messages printed by pasdoc to standard output – they will always be in English. Note that you can choose at most one language switch – if you specify none, the default language *English* will be used.

#### 5.3.1 Brasilian Portuguese

```
-b
--brasilian
```

Brasilian Portuguese will be used as language for headings and other text in the generated documentation.

#### 5.3.2 Catalan

```
-a
--catalan
```

Catalan will be used as language for headings and other text in the generated documentation.

#### 5.3.3 Dutch

```
-m
--dutch
```

Dutch will be used as language for headings and other text in the generated documentation.

#### 5.3.4 English

```
-e
--english
```

English will be used as language for headings and other text in the generated documentation. This is the default output language.

#### 5.3.5 French

```
-r
--french
```

French will be used as language for headings and other text in the generated documentation.

#### 5.3.6 German

```
-g
--german
```

German will be used as language for headings and other text in the generated documentation.

#### 5.3.7 Spanish

```
-i
--spanish
```

Spanish will be used as language for headings and other text in the generated documentation.

### 5.4 Other switches

#### 5.4.1 Include private fields, methods and properties

```
-p
--includeprivate
```

By default, private fields, methods and properties are not included in the documentation. However, with this switch you can force pasdoc to include these items.

#### 5.4.2 Output directory

```
-o DIRECTORY
--outputdirectory DIRECTORY
```

By default, pasdoc writes the output file(s) to the current directory. This switch defines a new output directory – this makes sense especially when you have many units and classes, they should get a subdirectory of their own, e.g. `htmloutput`.

### 5.4.3 Read file names from file

```
-s FILE
--sourcefilenames FILE
```

If you want pasdoc to write documentation for a large project involving many unit source code files, you can use this switch to load the file names from a text file. Pasdoc expects this file to have one file name in each row, no additional cleaning is done, so be careful not to include spaces or other whitespace like tabs.

### 5.4.4 Change verbosity level

```
-v LEVEL
--verbosity LEVEL
```

Using this switch in combination with an integer number $\geq 0$ lets you define the amount of information pasdoc writes to standard output. The default level is 2, this switch is optional. A level of 0 will result in no output at all.

### 5.4.5 Show help

```
-H
--help
```

This switch makes pasdoc print usage hints and supported switches to standard output (usually the console) and terminates.

### 5.4.6 Show version

```
-V
--version
```

A typical GNU switch, this makes pasdoc print program name and version in one row (e.g. `pasdoc 0.6.2`) to standard output. The program then terminates.

### 5.4.7 Specify a directive

```
-d DIRECTIVE
--directive DIRECTIVE
```

Adds `DIRECTIVE` to the list of conditional directives that are present whenever parsing a unit is started. The list of directives will be adjusted whenever a directive like `WIN32` or `FPC` is defined or undefined in the source code.

### 5.4.8 Specify an include file path

```
-u DIR
--includefilepath DIR
```

Adds `DIR` to the list of directories where pasdoc will search for an include file. Whenever an include file directive is encountered in the source code, pasdoc first tries to open that include file by the name found in that directive. This will work in all cases where the current directory contains that include file or when the file name contains a valid absolute or relative path.

For some projects, include files are kept in a special directory which is given to the compiler. To tell pasdoc where that directory is, use this switch.

#### 5.4.9 Specify directive file

```
-f FILE
--directivefile FILE
```

Adds all directives that can be found in text file `FILE`. Pasdoc expects one directive per text line. Same as adding each directive in that file using the `-d` switch.

## 6 Known problems and planned features

### 6.1 Documentation of program files

As was said before, only units are regarded by pasdoc. In an OOP environment for which pasdoc was written, an application is usually a class overriding an abstract application class, so all code that is ever needed in the program file looks like this:

```
program main;

uses myapp;

var App: TMyApplication;

begin
  App := TMyApplication.Create;
  App.Run;
  App.Destroy;
end.
```

So there isn't much to do for documentation. If you're not using OOP, you could at least try to move as much code as possible out of the main program to make things work with pasdoc.

### 6.2 Records

Pasdoc cannot create separate documentation for members of a record. In object-oriented programs, records will not appear most of the time because all encapsulated data will be part of a class or object. However, you can give a single explanation on a record type which could contain a description of all members.

### 6.3 Enumeration types

It is a good idea to list up all values an enumeration type can have in its description. Automatic listing of all values may become an option of pasdoc in the future.

### 6.4 Non-unique identifiers

In some larger projects, identifiers may be used in different contexts, e.g. as the name for a parameter and as a function name. Pasdoc will not be able to tell these contexts apart and as a result, will create in the above-mentioned example links (at least in HTML) from the argument name of a function to the type of the same name.

## 7 Adding support for another output format

If you want to write a different type of document than those supported, you can create another unit with a new object type that overrides `TDocGenerator` from unit `gendoc`. You'll have to override several of its methods to implement a new output format. As examples, you can always look at how the HTML and LaTeXgenerators work. First of all, you must decide whether your new output format will store the documentation in one (like LaTeX) or multiple files (like HTML).

## 8 Adding support for another language

Right now, seven languages are supported (Brasilian Portuguese, Catalan, Dutch, English, French, German and Spanish). If you want to add support for another language, all you have to do is

- get the latest version of pasdoc as source code,
- create a copy of the method `TDocGenerator.GetEnglishString` in the unit `gendoc` and rename it to fit your language,
- translate each English string to the corresponding string in your language and
- send it to me!

That's it... Some minor modifications will have to be added to `main.pas` and `chars.pas`, but all you have to do is translate roughly 45 strings.

## 9 Credits

Thanks to Michael van Canneyt, Marco van de Voort, Dan Damian, Philippe Jean Dit Bailleul, Jeff Wormsley, Johann Glaser, Gudrun Plato, Erwin Scheuch-Hellig, Iván Montes Velencoso, Mike Ravkin, Jean-Pierre Vial, Jon Korty, Martin Krumpolec, André Jager, Samuel Liddicott, Michael Hess, Ivan Tarapcik, Marc Weustink, Pascal Berger, Rolf Offermanns and Rodrigo Urubatan Ferreira Jardim for contributing ideas, bug reports and fixes, help and code!