

# Manual for pasdoc 0.8

Marco Schmidt & Carl Eric Codère & Johannes Berg

April 19, 2004

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Directives</b>	<b>4</b>
<b>3</b>	<b>Adding descriptions</b>	<b>6</b>
<b>4</b>	<b>Formatting your comments</b>	<b>7</b>
4.1	@ . . . . .	9
4.2	abstract . . . . .	9
4.3	author . . . . .	9
4.4	classname, inherited, name . . . . .	9
4.5	code . . . . .	10
4.6	created . . . . .	10
4.7	cvs . . . . .	10
4.8	exclude . . . . .	10
4.9	false . . . . .	11
4.10	html . . . . .	11
4.11	lastmod . . . . .	11
4.12	link . . . . .	12
4.13	longcode . . . . .	12
4.14	nil . . . . .	12
4.15	param . . . . .	13
4.16	raises . . . . .	13
4.17	return, returns . . . . .	13
4.18	true . . . . .	13
<b>5</b>	<b>Switches</b>	<b>13</b>
5.1	Documentation file format . . . . .	14
5.1.1	HTML . . . . .	14
5.1.2	htmlhelp . . . . .	14
5.1.3	L <sup>A</sup> T <sub>E</sub> X . . . . .	14
5.1.4	LaTeX2rtf . . . . .	15
5.2	Format-specific switches . . . . .	15
5.2.1	No generator information . . . . .	15
5.2.2	Specify name of document . . . . .	15
5.2.3	Specify footers and headers to use . . . . .	16
5.3	Comment Marker switches . . . . .	16
5.4	Output language switches . . . . .	16

5.5	Other switches . . . . .	17
5.5.1	Include / Exclude class Members by visibility . . . . .	17
5.5.2	Output directory . . . . .	18
5.5.3	Read file names from file . . . . .	18
5.5.4	Change verbosity level . . . . .	18
5.5.5	Show help . . . . .	19
5.5.6	Specify a directive . . . . .	19
5.5.7	Specify an include file path . . . . .	19
5.5.8	Specify directive file . . . . .	19
5.5.9	Set title of document . . . . .	20
5.5.10	Include uses list . . . . .	20
5.5.11	Full link output . . . . .	20
5.5.12	Non documented switches . . . . .	21
<b>6</b>	<b>Known problems</b>	<b>21</b>
6.1	Documentation of program files . . . . .	21
6.2	Records . . . . .	22
6.3	Non-unique identifiers . . . . .	22
<b>7</b>	<b>Adding support for another output format</b>	<b>22</b>
<b>8</b>	<b>Credits</b>	<b>22</b>

## 1 Introduction

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 L<sup>A</sup>T<sub>E</sub>X.

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] --format html *.pas` to generate HTML documentation.

You can compile pasdoc with Free Pascal (version 1.0 or higher), as well as with Delphi and Kylix.

## 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 5.5.6 and 5.5.8).

Next to those directives already presented, pasdoc also 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 (including enumerations),
- variables,
- constants,
- classes, interfaces, objects,
- procedures, functions and
- units.

Providing a description for the different items is fairly easy. You simply need to provide a comment containing the description before the name of the item itself.

For units, the comment declaration must be done before 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);
```

It is possible to specify that items will only be documented when certain comment markers are present at the start of the comment. Please refer to 5.3 for further information. Furthermore, undocumented items may or may not appear in the final document, depending on the output format.

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  $\text{\LaTeX}$ ). 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 a special font.

## 4 Formatting your comments

All comments recognize special directives that are used for different purposes. Each of these special directives starts with the at character `@`, followed by an identifier and optionally followed by text between parentheses.

As an example, let's 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 is 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.

The following tags are supported:

**@@** represents the @ character

**@abstract** This is an abstract of the description (nowadays called "management summary")

**@author** Treat the argument as an author's name and email address

**@classname** PasDoc inserts the class name here.

**@code** Treat argument as code example

**@created** Creation date of item

**@cvs** The argument is related to source versioning with e.g. cvs

**@exclude** The current item is to be excluded from documentation

**@false** PasDoc inserts the specially formatted text 'false' here.

**@html** Inserts html code in the output

**@inherited** PasDoc inserts the name of the ancestor class here.

**@lastmod** last modified date of item

**@link** The argument is the name of another entity, PasDoc will add a link to it here.

**@longcode** Format the text and output it in fixed width font, with correct formatting.

**@name** PasDoc inserts the name of the item (class, object, function, variable...) here

**@nil** PasDoc inserts the specially formatted text 'nil' here.

**@param** Treat first argument as parameter name and all following arguments as the description

**@raises** Treat first argument as exception name and all following arguments as the description

**@return, @returns** Description of a function's return value

**@true** PasDoc inserts the specially formatted text 'true' here.

#### 4.1 @

Represents the @ character, for example if you want to use one of the tags literally

#### 4.2 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.

The abstract text will appear in the overview section of the documentation (if the document format supports this overview section), and will also appear as the first paragraph of the item full documentation.

#### 4.3 author

For each author who participated in writing an item, one author tag should be added. The format of the author tag should conform to the following specification : **@author**(Name <URI>)

Author tags will only result in documentation output for classes, interfaces, objects and units.

Example:

```
@author(Johannes Berg <email@address.here>)
```

#### 4.4 classname, inherited, name

PasDoc uses the tags **@inherited**, **@classname** and **@name** as placeholders for the names of the ancestor class, current class and name of the current item respectively.

Example:

```
{ @name is a method of @classname which overrides the method of
  @inherited to do something completely different...}
```

#### 4.5 code

PasDoc uses the tag `@code` to mark example code which is preformatted and should not be changed in the output. It will usually appear in a teletype font in the final documentation.

Example:

```
{: how to declare a variable.
Example:
@code(
var
  SomeVariable: SomeType;)
}
```

#### 4.6 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.

#### 4.7 cvs

This tag is used to extract the last modification date and authors of the item. The parameter of this tag should conform to the *Author : ccodere* or *Date : 2004/04/2002 : 01 : 52* string of cvs or rcs.

```
@cvs($Date: 2004/04/20 02:01:52 $)
```

#### 4.8 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.

The following example will produce no documentation, as the entire unit will be excluded from the documentation process.

Example:

```
{@exclude }
unit myunit;

interface

procedure hello;

implementation

procedure hello;
begin
  WriteLn('Hello');
end;

end.
```

#### 4.9 false

PasDoc inserts the specially formatted text 'false' here at the location of the tag. This tag does not have any parameters.

#### 4.10 html

Pasdoc directly outputs the text that is between parentheses, without any conversion for the html output format. For other formats, the text is converted to standard text.

There is no syntax checking on the validity of the HTML syntax.

If there are no parentheses, @HTML is directly written to the output documentation.

#### 4.11 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.

## 4.12 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 L<sup>A</sup>T<sub>E</sub>X.

## 4.13 longcode

Use this tag to output code, and pre-formatted text. The output text will closely resemble the text typed, and will be represented in a fixed width font. In the case of pascal code typed within this tag, it will be pretty-printed first.

To be able to put special characters in this tag, the tag should be followed by a # and finished with a # before the closing parentheses.

Example:

```
    @longCode(#
procedure TForm1.FormCreate(Sender: TObject);
var
  i: integer;
begin
  // Note that your comments are formatted.
  {$H+} // You can even include compiler directives.
  // reserved words are formatted in bold.
  for i := 1 to 10 do
  begin
    It is OK to include pseudo-code like this line.
    // It will be formatted as if it were meaningful pascal code.
  end;
end;
    #)
```

## 4.14 nil

PasDoc inserts the specially formatted text 'nil' here. This tag does not have any parameters.

#### 4.15 param

Treats first argument as parameter name and all following text as the description of this parameter.

Example:

```
{ A small description
  @param(Filepath The file to open)
}
constructor Init(FilePath : String);
```

#### 4.16 raises

Treats the first argument as exception name and all following text as the description for this exception.

Example:

```
{ A small description
  @raises(EMyException Raises this exception)
}
constructor Init;
```

#### 4.17 return, returns

Treat the text in the argument as the description of the returns value of this function or method.

#### 4.18 true

PasDoc inserts the specially formatted text 'true' here at the location of the tag. This tag does not have any parameters.

## 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

```
-O html  
--format html
```

This switch makes pasdoc write HTML (Hypertext Markup Language) output. HTML files are usually displayed in a web browser, which is 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.

This is the preferred output for online viewing.

It is to note that even undocumented items will appear in the final output file format.

### 5.1.2 htmlhelp

```
-O htmlhelp  
--format htmlhelp
```

This switch makes pasdoc write HTML (Hypertext Markup Language) output. It also writes additional files that can be used to create Microsoft htmlhelp files. Please consult the htmlhelp Microsoft SDK for more information.

### 5.1.3 L<sup>A</sup>T<sub>E</sub>X

```
-O latex  
--format latex
```

This switch makes pasdoc write output that can be interpreted using L<sup>A</sup>T<sub>E</sub>X. This is the preferred output format for printing the documentation.

A single output file, either having the name specified with the `-N` switch, or the default name `docs.tex` will be created.

With `latex` you will be able to create a dvi file that can then be converted to a Postscript file using `dvips`. Or you can also directly generate a huge HTML file by using `htlatex`, or a PDF file by using `pdflatex`.

It is to note that the output generated by `pasdoc` has been optimized for `pdflatex` and `htlatex`.

It is to note that only documented items will appear in the final output file format.

#### 5.1.4 LaTeX2rtf

```
-O latex2rtf
--format latex2rtf
```

This switch makes `pasdoc` write output that can be interpreted using `latex2rtf`. This is the preferred output format for adding the documentation to word processor documentation.

A single output file, either having the name specified with the `-N` switch, or the default name `docs.tex` will be created. This file can then be converted to rtf by using `latex2rtf`.

This output will only work with the `latex2rtf` tool. Using other tools might not produce the expected results.

It is to note that only documented items will appear in the final output file format.

## 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 generator information

```
-X
--exclude-generator
```

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.

### 5.2.2 Specify name of document

```
-N NAME --name NAME
```

When the output format of the documentation is not HTML (such as latex, or CHM), this specifies the name of the final name of the documentation. If this is not specified, it uses the defaultdocs filename.

### 5.2.3 Specify footers and headers to use

`-F FILENAME --footer FILENAME -H FILENAME --header FILENAME`

You can specify texts files which PasDoc should use as header or footer for all generated html pages. This option is only available for the html output format.

Example:

```
pasdoc --header myheader.txt --footer myfooter.txt
```

## 5.3 Comment Marker switches

It is possible for pasdoc to ignore comments that do not start with the correct start markers. By default, all comments are treated as item descriptions. This can be changed using the following switches:

`--staronly` Parse only `{**}`, `(***` and `//**` style comments

`--marker` Parse only `{<marker>}`, `(*i;markeri` and `//i;markeri` comments.  
Overrides the staronly option, which is a shortcut for `'-marker=**'`

`--marker-optional` Do not require the markers given in `-marker` but remove them from the comment if they exist.

## 5.4 Output language switches

`-L lg`

`--language lg`

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.

The `lg` parameter can take one of the following values:

**ba** Bosnian (Codepage 1250)

**br** Portugese / Brazilian

**ct** Catalan  
**dk** Danish  
**en** English  
**fr** French  
**de** German  
**id** Indonesian  
**it** Italian  
**ja** Japanese  
**pl** Polish  
**ru.1251** Russian (Codepage 1251)  
**ru.866** Russian (Codepage 866)  
**ru.KOI8** Russian (KOI-8)  
**sk** Slovak  
**es** Spanish  
**se** Swedish

## **5.5 Other switches**

### **5.5.1 Include / Exclude class Members by visibility**

**-M**  
**--visible-members**

By default all non-private fields, methods properties are included in the documentation. This switch permits to change which items of the specified visibility will be documented.

The possible arguments, separated by a comma are:

**private**  
**protected**  
**public**

published

automated

In the following example only the private and protected members will be documented, all others will be ignored.

```
pasdoc --visible-members private,protected
```

### 5.5.2 Output directory

`-E DIRECTORY`

`--output 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`.

### 5.5.3 Read file names from file

`-S FILE`

`--source 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.

### 5.5.4 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.

### 5.5.5 Show help

-?  
--help

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

### 5.5.6 Specify a directive

-D DIRECTIVE  
--define 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. Each define should be separated by the others by a comma, as shown in the following example:

```
pasdoc --define debug,hello,world
```

### 5.5.7 Specify an include file path

-I DIR  
--include DIR

Adds `DIR` to the list of directories where pasdoc will search for include files. 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.

It is possible to use this switch more than once on the command line.

```
pasdoc --include c:\mysources\include --include c:\3rdparty\somelib\include
```

### 5.5.8 Specify directive file

-d DIRECTORY  
--conditionals DIRECTORY

Adds the defines specified in a file `DIRECTORY` 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. There should be one defibe per line in the conditional file.

```
pasdoc --conditionals /home/me/pascal/myconditionals
```

### 5.5.9 Set title of document

```
-T "STRING"  
--title "STRING"
```

This option sets the title of the output document. The characters in the title should be enclosed in double quotes.

By default, depending on the documentation format, the document contains either no title, or the name of the unit being documented.

Example:

```
pasdoc -T "This is my document title"
```

### 5.5.10 Include uses list

```
--write-uses-list
```

PasDoc can optionally include the list of units in a unit's uses clause to that unit's description.

Example:

```
pasdoc --write-uses-list
```

If a unit in the uses list is part of the documentation, it will be clickable in the output.

By default this option is disabled.

### 5.5.11 Full link output

```
--full-link
```

This option controls the behaviour of "`@link(unit.procedure)`" type links. If it is set, the output generated will look like this:

`unit.procedure` with the "unit" part linking to the unit and the "procedure" part linking to the procedure inside the unit. If it is unset, then the output will only be `procedure`.

### 5.5.12 Non documented switches

This lists the other unusual switches that are recognized by pasdoc:

- R, --description read description from this file
- C, --content Read Contents for HtmlHelp from file
- numericfilenames Causes the html generator to create numeric file-names
- graphviz-uses write a GVUses.gviz file that can be used for the dot program from GraphViz to generate a unit dependency graph.
- graphviz-classes write a GVClasses.gviz file that can be used for the dot program from GraphViz to generate a class hierarchy graph.
- abbreviations abbreviation file, format is "[name] value", value is trimmed, lines that do not start with '[' (or whitespace before that) are ignored
- aspell enable aspell, giving language as parameter, currently only done in HTML output
- ignore-words When spell-checking, ignore the words in that file. The file should contain one word on every line, no comments allowed
- cache-dir Cache directory for parsed files (default not set)

## 6 Known problems

### 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 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 `PasDoc.Gen.pas`. 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  $\text{\LaTeX}$  generators work. First of all, you must decide whether your new output format will store the documentation in one (like  $\text{\LaTeX}$ ) or multiple files (like HTML).

## 8 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 Kory, 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!