Txt2FPDoc : Reference Manual

for Txt2FPDoc version 0.8.6
July 2015

Yann Mérignac

This manual is for `Txt2FPDoc` version 0.8.6 (last updated 30 July 2015).

Table of Contents

1 Introduction

1.1 About this document

This document is the reference manual for Txt2FPDoc version 0.8.6.

Chapter 2 explains how to compile and install Txt2FPDoc. (Chapter 2 [Compiling and Installing Txt2FPDoc], page 4)

Chapters 3 , 4 and 5 explain how to write documentation for your code with Txt2FPDoc. (Chapter 3 [Documenting Your Code], page 5, Chapter 4 [Descriptions], page 18, Chapter 5 [The Preprocessor], page 22)

Chapter 6 explains how to run Txt2FPDoc. (Chapter 6 [Running Txt2FPDoc], page 24)

Appendices A, B and C describe in detail the Txt2FPDoc syntax. They are not to be read but to serve as a reference. (Appendix A [Headers], page 25, Appendix B [Structure of Main Components], page 28, Appendix C [Documentation File Grammar], page 30)

Appendix D provides complete examples, i.e., source code, documentation file and commands to build html documentation. (Appendix D [Complete Examples], page 32)

1.2 Conventions

Commands that are entered by the user are shown preceded by '~$'.

```
~$ fpc myprog.pas
~$ ./myprog
```

Examples of Txt2FPDoc input are set out from the normal text, and shown in a fixed width font, like this

```
1 Function: Multiply
2
3 Multiplies two integers.
```

The line numbers at the beginning of the lines must not be typed. They are present to serve as references in the explanations.

1.3 About Txt2FPDoc

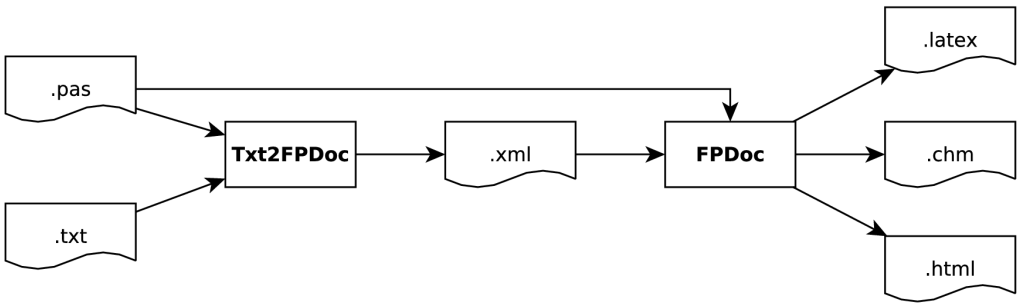

Figure: Process of creating documentation with Txt2FPDoc

Txt2FPDoc is a documentation tool for Free Pascal and Lazarus code. It's a command line utility that converts a documentation file written in human readable language to an XML file suitable for FPDoc.

Txt2FPDoc tries to make the documentation file easy to read (almost as readable as plain English) and easy to write (no cryptic tags, only light markups like '**bold**' and '//italic//').

The documentation can be stored in one or more text files or directly included in the source code.

Here is a quick example of how to document a function for `Txt2FPDoc`:

Example: Documentation of `Multiply` for `Txt2FPDoc`

```
1 {** Function: Multiply -- Multiplies two integers.
2
3     Parameters:
4       - A -- first integer
5       - B -- second integer
6
7     Result:
8       The two integers multiplied together. }
9 function Multiply(A, B: Integer) : Integer;
```

For comparison, here's the XML code that you should write to get the same result with `FPDoc`:

Example: Documentation of `Multiply` for `FPDoc`

```
 1  <element name="Multiply">
 2    <short>Multiplies two integers.</short>
 3  </element>
 4
 5  <element name="Multiply.Result">
 6    <short>The two integers multiplied together.</short>
 7  </element>
 8
 9  <element name="Multiply.B">
10    <short>second integer</short>
11  </element>
12
13  <element name="Multiply.A">
14    <short>first integer</short>
15  </element>
```

Last, the following screenshot shows how it could be rendered in a web browser once converted to HTML by `FPDoc`.

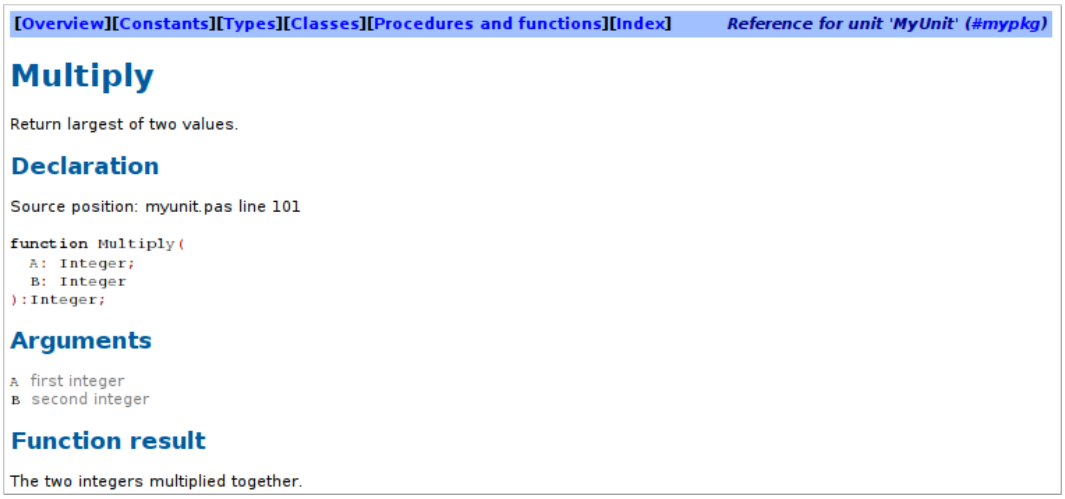

Image: Documentation of `Multiply` in a web brower

2 Compiling and Installing Txt2FPDoc

2.1 Compiling

In order to compile Txt2FPDoc you need :

- the Free Pascal compiler : http://www.freepascal.org/
- the Txt2FPDoc sources : http://yann.merignac.free.fr/txt2fpdoc.html

When the Txt2FPDoc sources have been uncompressed enter the Txt2FPDoc source directory and run the following commands:

```
~$ fpc pasconf
~$ ./pasconf
~$ make
```

For extra configuration options try running `./pasconf -h`.

2.2 Installing

To install Txt2FPDoc on a Unix like system run `make install`.

On Windows just copy `txt2fpdoc.exe` to a directory listed in your `PATH` environment variable.

3 Documenting Your Code

This section will teach you how to use `Txt2FPDoc` to document your code. To achieve this goal we will write a documentation file for the following unit:

Example: `myunit.pas`

```
 1  unit MyUnit;
 2
 3  interface
 4
 5  function Max(A, B: Integer) : Integer;
 6  function Min(A, B: Integer) : Integer;
 7  function Multiply(A, B: Integer) : Integer;
 8
 9  implementation
10  end.
```

3.1 Basic Rules

1. `Txt2FPDoc` **knows nothing about Pascal.**

   It just offers an alternative to XML for writing documentation. It does not check that code and documentation are consistent.

2. **Indentation as no meaning.**

   The meaning of a line depends only on its first non-blank characters.

3. **Special chars can be escaped by '\'.**

   Some characters have a special meaning. For examples '`-`' on the beginning of a line means that the line is an unordered list item. Starting the line by '`\-`' makes `Txt2FPDoc` consider this line as simple text. A true '\' can be obtained by '`\\`'.

3.2 Minimal Documentation File

The minimal documentation file would look something like this:

Example: Minimal documentation file

```
1  Package: MyPkg
2
3  Module: MyUnit
4
5  End Module: MyUnit
6
7  End Package: MyPkg
```

Lines starting by a header name immediately followed by a colon (like lines 1, 3, 5, 7) are called header lines. You'll use a header line for each element you want to document. There are many header names, more than 60, but only a few (see Section A.1 [Core Headers], page 25) are necessary to write documentation all others are just synonyms. Even better, you can start writing real documentation with only 10 headers and, you already know four of them: *Package*, *End Package*, *Module*, and *End Module*.

Important: A header is recognized only if there is no space between it and the colon. If a space is present the line is considered as a simple text.

The minimal documentation file consists of a package block that contains a module block. The package block starts by a *Package* header (line 1) and ends by a *End Package* header (line 7). The module block starts by a *Module* header (line 3) and ends by a *End Module* header (line 5).

For now this file documents nothing but it is syntactically correct. We could even make it shorter by removing the last two lines since the *End Package* and *End Module* headers are optional. However, some error messages are sometimes easier to understand if they are present.

For a complete list of all header names see Appendix A [Headers], page 25.

Note: The idea of using headers comes from Natural Docs[1] and to a lesser extent from ROBODoc[2].

3.3 Comments and Separators

Example: Comments ans separators

```
 1  **************
 2  Package: MyPkg
 3  **************
 4
 5  # This is a comment
 6
 7  ~~~~~~~~~~~~~~~
 8  Module: MyUnit
 9  ~~~~~~~~~~~~~~~
10
11  End Module: MyUnit
12
13  End Package: MyPkg
```

Lines starting by '`#`' are comments and are ignored (line 5).

Lines consisting of a single character from ['`-`', '`=`', '`_`', '`*`', '`%`', '`~`'] repeated 5 times or more are separators and, like comments, are ignored (lines 1, 3, 7, 9). Using separators increases the readability of documentation.

3.4 Adding Descriptions

Almost all the elements that we can document can have a short description and a long description. So let's add descriptions to the module *MyUnit*.

Example: First descriptions

```
 1  Package: MyPkg
 2
 3  Module: MyUnit
 4
 5    MyUnit contains some essential routines.
 6
 7    This routines are the result of many years of
 8    hard work.
 9
10    I hope they will help to save mankind.
11
12  End Module: MyUnit
13
14  End Package: MyPkg
```

The short description is the first paragraph following the header line (line 5). It ends on the blank line (line 6). Then, all following paragraphs (lines 7-10) are part of the long description.

[1] http://www.naturaldocs.org/

[2] http://rfsber.home.xs4all.nl/Robo/

Short descriptions can also start on the header line, after a double hyphen ('--').

Example: Short description on header line

```
1  Module: MyUnit -- MyUnit contains some essential
2    routines.
```

Short and long descriptions can be beautified by the use of some markups (see Section 4.4 [Beautifiers], page 19).

Example: Descriptions with beautifiers

```
1  Package: MyPkg
2
3  Module: MyUnit
4
5    ''MyUnit'' contains some essential routines.
6
7    This routines are the result of many years of
8    **hard work**.
9
10   I hope they will help to save mankind.
11
12 End Module: MyUnit
13
14 End Package: MyPkg
```

- *'MyUnit'* (line 5) will be emphasized.
- *'hard work'* (line 8) will be rendered bold.

See Chapter 4 [Descriptions], page 18 for more information on descriptions.

3.5 Documenting Elements

We now have to add an *Element* block for each identifier we want to document.

Here are the two possible simplifed formats for an *Element* block (see [Structure of an Element], page 28 for the full format).

The first possible format is:

```
Element: Name [-- ShortDescription]

[LongDescription]
```

The second possible format is:

```
Element: Name

[ShortDescription]

[LongDescription]
```

Below is an example of documentation for the `Max` function and its arguments `A` and `B`.

Example: Documenting `Max`

```
1  Package: MyPkg
2
3  Module: MyUnit
4
5    MyUnit contains some essential routines.
6
7    This routines are the result of many years of
```

```
 8    hard work.
 9
10    I hope they will help to save mankind.
11
12  Element: Max -- Return largest of two values.
13
14    Max returns the maximum of A and B.
15
16  Element: Max.A -- first value
17
18  Element: Max.B -- second value
19
20  End Module: MyUnit
21
22  End Package: MyPkg
```

- Documentation for function `Max` starts in line 12 with a short description (line 12) and a long description (line 14).
- Documentation for parameter `A` of `Max` is in line 16.
- Documentation for parameter `B` of `Max` is in line 18.

3.6 Sub Elements

Documenting elements one by one is not really satisfactory. It's easier to write than XML, but not as readable as one might hope. Fortunately we can do better thanks to *Sub Elements*.

Example: Sub Elements

```
 1  Package: MyPkg
 2
 3  Module: MyUnit
 4
 5    MyUnit contains some essential routines.
 6
 7    This routines are the result of many years of
 8    hard work.
 9
10    I hope they will help to save mankind.
11
12  Element: Max -- Return largest of two values.
13
14    Max returns the maximum of A and B.
15
16    Sub Elements:
17      - A -- first value
18      - B -- second value
19
20  End Module: MyUnit
21
22  End Package: MyPkg
```

A *Sub Elements* section starts by a *Sub Elements* header on is own line (line 16). Then comes a list off sub elements (line 17-18), in the format '`- Identifier -- ShortDescription`'. The short description of a sub element may extend over multiple lines. Sub elements can only have a short description if you want to add a more detailed description use a normal *Element* header.

3.7 Synonyms

Rather than using generic header names as a *Module*, *Element* and *Sub Elements* we can use more specific synonyms that will make documentation more readable.

Some examples:

- Instead of *Module* you can use: *Unit*, *Program*, *Library*
- Instead of *Element* you can use: *Function*, *Procedure*, *Constant*, *Var*, *Type*, ...
- Instead of *Sub Elements* you can use: *Parameters*, *Enumerators*, *Properties*, ...

See Appendix A [Headers], page 25 for a complete list of existing headers.

Example: A better readability thanks to synonyms

```
1  Package: MyPkg
2
3  Unit: MyUnit
4
5    MyUnit contains some essential routines.
6
7    This routines are the result of many years of
8    hard work.
9
10   I hope they will help to save mankind.
11
12 Function: Max -- Return largest of two values.
13
14   Max returns the maximum of A and B.
15
16   Parameters:
17     - A -- first value
18     - B -- second value
19
20 End Unit: MyUnit
21
22 End Package: MyPkg
```

3.8 Function Result

Document the result of the `Multiply` function is actually documenting `Multiply.Result`.

We can do as we would for a normal element.

Example: As a normal element

```
1  Element: Multiply.Result -- The two integers multiplied together.
```

We can also consider that `Multiply.Result` is a sub element of `Multiply`.

Example: As a sub element

```
1  Function: Multiply -- Multiplies 2 integers.
2
3    Sub Elements:
4      - A      -- first integer
5      - B      -- second integer
6      - Result -- The two integers multiplied together.
```

But the best way is to use a *Result* header:

Example: With *Result* header

```
1  Function: Multiply -- Multiplies 2 integers.
2
3    Parameters:
4      - A -- first integer
5      - B -- second integer
6
7    Result:
8
9      The two integers multiplied together.
```

A *Result* header can only appear once per element.

3.9 References

When writing the documentation of an element you often need to refer to another element. For example, in the `Max` function, we could add a link to the `Min` function.

The format of a link is '`<<Target>>`' or '`<<Text Target>>`' (see Section 4.5 [Links], page 19).

Example: Links in descriptions

```
1  Function: Max -- Return largest of two values.
2
3    Max returns the maximum of A and B.
4
5    See <<Min>>.
6
7    Parameters:
8      - A -- first value
9      - B -- second value
10
11 Function: Min -- Return smallest of two values.
12
13   Min returns the minimum of A and B.
14
15   <<Click Here Max>>.
16
17   Parameters:
18     - A -- first value
19     - B -- second value
```

But often it is better that these links are grouped in a special section. For this we can use the *See Also* header.

Example: Using a *See Also* header

```
1  Function: Max -- Return largest of two values.
2
3    Max returns the maximum of A and B.
4
5    Parameters:
6      - A -- first value
7      - B -- second value
8
9    See Also: <<Min>>
10
11 Function: Min -- Return smallest of two values.
```

```
12
13    Min returns the minimum of A and B.
14
15    Parameters:
16      - A -- first value
17      - B -- second value
18
19    See Also: <<Max>>
```

A *See Also* header can only appear once per element.

3.10 Mixing Code and Documentation

Thanks to the `Txt2FPDoc` preprocessor, we can mix source code and documentation. This is achieved with the include directive.

Example: `documentation.txt`

```
1  Package: MyPkg
2
3  $include 'myunit.pas'
4
5  End Package: MyPkg
```

The `$include` directive (line 3) imports `myunit.pas`. As it is a pascal file, only documentation comments are imported (see Section 5.1 [$include], page 22).

And here is the file `myunit.pas` with documentation located in comments.

Example: `myunit.pas`

```
1  {**
2    Unit: MyUnit
3
4    MyUnit contains some essential routines.
5
6    This unit is the result of many years of
7    hard work.
8
9    I  hope they will help to save mankind. }
10 unit MyUnit;
11
12 interface
13
14 {**
15 Function: Max -- Return largest of two values.
16
17   Max returns the maximum of A and B.
18
19   Parameters:
20     - A -- first value
21     - B -- second value }
22 function Max(A, B: Integer) : Integer;
23
24 {**
25   Function: Min -- Return smallest of two values.
26
27   Min returns the minimum of A and B.
```

```
28
29    Parameters:
30      - A -- first value
31      - B -- second value }
32  function Min(A, B: Integer) : Integer;
33
34  implementation
35
36  end. {** End Unit: MyUnit}
```

3.11 Abbreviated Syntax

When you have many similar elements to document the traditional syntax can be somewhat cumbersome.

Example: Constants documented one by one

```
 1  const
 2    {** Constant: Black -- Black color }
 3    Black = 0;
 4
 5    {** Constant: Blue -- Blue color }
 6    Blue = 1;
 7
 8    {** Constant: Green -- Green color }
 9    Green = 2;
10
11    {** Constant: Yellow -- Yellow color }
12    Yellow = 3;
13
14    {** Constant: Red -- Red color }
15    Red = 4;
```

Using *Elements* header or one of its synonyms can save you some typing. A *Elements* header is used as a *Sub Elements* header except that the elements of the list that follows are not considered as of sub elements but as elements in the current level.

Example: Constants documented using *Elements*

```
 1  const
 2    {** Elements:
 3          - Black  -- Black color
 4          - Blue   -- Blue color
 5          - Green  -- Green color
 6          - Yellow -- Yellow color
 6          - Red    -- Red color   }
 7    Black = 0;
 8    Blue = 1;
 9    Green = 2;
10    Yellow = 3;
11    Red = 4;
```

3.12 Composite Types

Example: An example class

```
type
  TRectangle = class
  private
    fHeight: Real;
    fWidth: Real;
  public
    constructor Create(W, H : Real);

    function Area : Real;
    procedure SetSize(W, H : Real);

    property Height : Real read fHeight write fHeight;
    property Width : Real read fWidth write fWidth;
  end;
```

Suppose you want to document the `TRectangle` class above. You can do it like we have learn to do it until now.

Example: `TRectangle` documentation

```
Type: TRectangle -- A class that implements a rectangle.

Constructor: TRectangle.Create -- Creates a new rectangle.

  Parameters:
    - W -- width
    - H -- height

Function: TRectangle.Area -- Returns the rectangle area.

Procedure: TRectangle.SetSize -- Sets the rectangle size.

  Parameters:
    - W -- width
    - H -- height

Property: TRectangle.Height -- rectangle height

Property: TRectangle.Width -- rectangle width
```

Alternatively you can choose to use *Struct*/*End Struct* headers and there synonyms.

Example: `TRectangle` documented with *Struct*

```
Struct: TRectangle -- A class that implements a rectangle.

  Constructor: Create -- Creates a new rectangle.

    Parameters:
      - W -- width
      - H -- height

  Function: Area -- Returns the rectangle area.

```

```
11    Procedure: SetSize -- Sets the rectangle size.
12
13      Parameters:
14        - W -- width
15        - H -- height
16
17    Property: Height -- rectangle height
18    Property: Width -- rectangle width
19
20  End Struct: TRectangle
```

All elements documented between *Struct* (line 1) and *End Struct* (line 20) are considered as sub elements of `TRectangle`. The closing *End Struct* is mandatory.

Of course you can also use synonyms of *Struct*/*End Struct*, like *Class*/*End Class*, and mix documentation and code.

Example: `TRectangle` documented in source code with *Class*

```
 1  type
 2    {** Class: TRectangle -- A class that implements a rectangle. }
 3    TRectangle = class
 4    private
 5      fHeight: Real;
 6      fWidth: Real;
 7    public
 8      {** Constructor: Create -- Creates a new rectangle.
 9
10          Parameters:
11            - W -- width
12            - H -- height }
13      constructor Create(W, H : Real);
14
15      {** Function: Area -- Returns the rectangle area. }
16      function Area : Real;
17
18      {** Procedure: SetSize -- Sets the rectangle size.
19
20          Parameters:
21            - W -- width
22            - H -- height }
23      procedure SetSize(W, H : Real);
24
25      {** Property: Height -- rectangle height }
26      property Height : Real read fHeight write fHeight;
27
28      {** Property: Width -- rectangle width }
29      property Width : Real read fWidth write fWidth;
30    end; {** End Class: TRectangle}
```

3.13 Annotations

With *Note* and *Notes* headers you can annotate the documentation. These notes are intended to documentation writer. They appear in the final documentation only if option `--emit-notes` of `FPDoc` is used.

Example: *Note* and *Notes* example

```
1  Function: Min -- Return smallest of two values.
2
3    Min returns the minimum of A and B.
4
5    Parameters:
6      - A -- first value
7      - B -- second value
8
9    See Also: <Max>
10
11   Note:
12
13     ``Min`` needs more documenting.
14
15   Notes:
16     - Add an example
17     - Find better names for ``A`` and ``B``
```

3.14 Topics

If you want to include with your documentation pages that are not related to a Pascal identifier, you must use a *Topic* header. You can insert a *Topic* in a *Package* or in a *Module*. A *Topic* has a name, a short description and a long description. If a *Topic* appears in a *Package* it can also have *Sub Topics* after his long description.

Example: *Topic* example

```
1  Topic: UsingKeyboard -- Using the keyboard functions
2
3  To use the keyboard functions of the CRT unit, one...
4
5  Sub Topic: UsingShiftKey -- Using the shift key
6
7  ...
```

3.15 Errors

The *Errors* header is used to document errors that can happen during the execution of a procedure or a function.

Example: *Errors* example

```
1  Function: Min -- Return smallest of two values.
2
3    Min returns the minimum of A and B.
4
5    Parameters:
6      - A -- first value
7      - B -- second value
8
9    See Also: <<Max>>
10
```

```
11    Errors:
12
13      When ``Min`` fails, dangerous things can
14      happen.
```

A *Errors* header can only appear once per element after all *Sub Elements*, *Result*, *See Also*, *Note* and *Notes* headers.

3.16 Examples

The *Example* header enables to include source code files into the documentation of an element. It can appear multiple times. `Txt2FPDoc` does not verify that the file exists.

Example: Importing an example file

```
 1  Function: Min -- Return smallest of two values.
 2
 3    Min returns the minimum of A and B.
 4
 5    Parameters:
 6      - A -- first value
 7      - B -- second value
 8
 9    See Also: <<Max>>
10
11    Example: 'demo_min.pas'
```

A *Example* header can only appear after all *Sub Elements*, *Result*, *See Also*, *Note* and *Notes* headers.

3.17 Version

The *Version* header enables to add version information into the documentation of an element.

Example: Adding version informations

```
 1  Function: Min -- Return smallest of two values.
 2
 3    Min returns the minimum of A and B.
 4
 5    Parameters:
 6      - A -- first value
 7      - B -- second value
 8
 9    See Also: <<Max>>
10
11    Version:
12
13      Deprecated since version 3.2.
```

A *Version* header can only appear once per element after all *Sub Elements*, *Result*, *See Also*, *Note* and *Notes* headers.

3.18 Inserting a Short Description

Occasionally, you may need to insert the short description of an element in the current text. To do this, you can copy yourself the short description, or you can insert the element name enclosed in '`@`', as in the following example[3].

[3] This example uses a definition list (see Section 4.6 [Lists], page 20).

Example: Inserting a short description

```
1  Summary:
2
3  ::‘‘Min‘‘      -- @Min@
4  ::‘‘Max‘‘      -- @Max@
5  ::‘‘Multiply‘‘ -- @Multiply@
```

4 Descriptions

Some headers can be followed by descriptions. Some only accept short descriptions, others accept long descriptions and others accept descriptions of the two kinds.

4.1 Short Descriptions

Example: Short descriptions after '--'

```
1  Element: Item_1 -- This is the short description of Item_1.
3
4    And this is the first paragraph
5    of its long descripion.
6
7  Element: Item_2 -- This is the short description
8    of Item_2.
9
10   And this is the first paragraph
11   of its long descripion.
```

A short description can start on the header line, after a double hyphen '--' (line 1 and 7), and extends until the next non-text line (line 3 and 9).

Example: Short descriptions as paragraphs

```
1  Element: Item_1
2    This is the short description
3    of Item_1.
4
5    And this is the first paragraph
6    of its long descripion.
7
8  Element: Item_2
9
10   This is the short description
11   of Item_2.
12
13   And this is the first paragraph
14   of its long descripion.
```

If it does not start in the same line, the short description will be the first paragraph found after the header. In the example above, the short description of `Item_1` extends from line 2 to 3 and the short description of `B` from lines 10 to 11.

You can also create an empty description, as in the example below.

Example: Empty short description

```
1  Element: Item_1 --
2
3    This is the first paragraph of
4    the long descripion of Item_1.
```

4.2 Long Descriptions

The long description begins right after the short description. It consists of a sequence of:

- paragraphs (see Section 4.3 [Paragraphs], page 19)
- lists (see Section 4.6 [Lists], page 20)
- preformatted blocks (see Section 4.7 [Preformatted Blocks], page 20)

- code blocks (see Section 4.8 [Code Blocks], page 20)
- images (see Section 4.9 [Images], page 21)
- tables (see Section 4.10 [Tables], page 21)
- remarks (see Section 4.11 [Remarks], page 21)

4.3 Paragraphs

A paragraph is a block of text terminated by a blank line, a header, a list item, a preformatted line or the end of file.

Example: Paragraphs

```
1  A paragraph is made by one or
2  more lines.
3
4  A blank line starts a new
5  paragraph.
```

4.4 Beautifiers

Some markups can be used to beautify texts in paragraphs, short descriptions, list items and table cells. If needed, they can be escaped with '\'.

Bold

Text enclosed in '**' is rendered in a bold font.

Italic

Text enclosed in '//' is printed in italic.

Emphasized

Text enclosed in '‘‘' is marked as a variable or as an identifier. It will be printed in a different font (which depends on the output format).

Underlined

Text enclosed in '__' is printed underlined.

Warning: Due to some FPDoc bug, support for underlined style is disabled at the time of this writing.

Example: Beautifiers

```
1  For beautifiers we have **bold**,
2  //italic//, ‘‘emphasized‘‘ and
3  __underlined__.
```

4.5 Links

Paragraphs may also contain links. There are two types of links: internal links and external links.

Internal Links

Internal links are links to documented elements. They exist in 2 forms:

- '<<target>>'
- '<<text target>>'

Example: Internal links

```
1  The <<stringlist TStringlist>> class is a descendent
2  of the  <<TStrings>> class.
```

External Links

External links are links to URLs. They exist in 2 forms:

- '`[[target]]`'
- '`[[text target]]`'

Example: External links

```
1  [[Here http://www.freepascal.org]]
2  is the website of Free Pascal.
3
4  Check out the following site: [[http://www.freepascal.org]].
```

4.6 Lists

Unordered Lists

An unordered list is a sequence of items in the format '`- Text`'.

Example: Unordered list

```
1  Some text before
2    - First item in the list.
3    - Second item in the
4      list.
```

Ordered Lists

An ordered list is a sequence of items in the format '`+ Text`'.

Example: Ordered list

```
1  Some text before
2    + First item in the list.
3    + Second item in the list.
```

Definitions Lists

An definition list is a sequence of items in the format '`:: Term -- Definition`'.

Example: Definition list

```
1  ::FPC -- Free Pascal compiler
2  ::Lazarus -- An IDE for Free
3      Pascal
```

4.7 Preformatted Blocks

A preformatted block is a sequence of lines beginning by '`>`'. Lines of a preformatted blocks are not formatted and are printed in a fixed-font.

Example: Preformatted block

```
1  This is normal text.
2  > This is
3  >   a **preformatted**
4  >     text.
```

4.8 Code Blocks

The *Code* header serves to insert pascal code into documentation. All preformatted lines following a *Code* header are considered part of the code block.

Example: Code block

```
1  Code:
2  > if Ok then
3  >   WriteLn('Hello!');
```

4.9 Images

The *Image* header serves to insert an image given in an external file. *Image* header is in the format '`Image: FileName -- Caption`'. `FileName` must be quoted.

Example: Image insertion

```
1  Pascal, named in honor of the French mathematician
2  and philosopher Blaise Pascal, was developed by
3  Niklaus Wirth.
4
5  Here is Blaise Pascal, see how digital photography
6  has made enormous progress since his time.
7
8  Image: 'images/blaise-pascal.png' -- Blaise Pascal
```

4.10 Tables

A table consists of one or more row lines. A row line starts with a '`|`'. Table columns are separated by '`|`'.

Example: Simple table

```
1  | taLeftJustify  | Text is displayed aligned to the left
2  | taRightJustify | Text is displayed aligned to the right
3  | taCenter       | Text is displayed centred
```

If the first row line starts with '`||`' it is considered as a header line.

Example: Table with headers

```
1  ||Value           | Meaning
2  | taLeftJustify   | Text is displayed aligned to the left
3  | taRightJustify  | Text is displayed aligned to the right
4  | taCenter        | Text is displayed centred
```

You can add a table caption by adding a *Table* header.

Example: Table with caption

```
1  Table: Text Alignment
2  ||Value           | Meaning
3  | taLeftJustify   | Text is displayed aligned to the left
4  | taRightJustify  | Text is displayed aligned to the right
5  | taCenter        | Text is displayed centred
```

4.11 Remarks

Example: Remarks

```
1  Remark: This is remark 1.
2
3  Remark:
4
5  This is remark 2.
```

5 The Preprocessor

`Txt2FPDoc` has a preprocessor. All lines starting by '`$`' are preprocessor's directives. The preprocessor accepts the following directives:

- `$include` (see Section 5.1 [$include], page 22)
- `$include_many` (see Section 5.2 [$include_many], page 22)
- `$include_preformatted` (see Section 5.3 [$include_preformatted], page 22)
- `$include_numbered` (see Section 5.4 [$include_numbered], page 22)

`$include` and `$include_many` directives are the solution if you want to mix code and documentation or split your documentation into several files.

Note:

Unrecognized directives are treated as comment lines. This behavior may change in future releases.

5.1 $include

The `$include` directive lets you import a file. If the included file is a pascal file then only documentation comments, those starting by '`{**`' or '`(**`' or '`//**`', are imported. Pascal files are recognized by their extension, which must be one of : '`.pas`', '`.pp`', '`.p`' or '`.inc`'.

Example: Importing files

```
1  $include 'src/myunit.pas'
2  $include 'topics.txt'
```

5.2 $include_many

The `$include_many` directive lets you import content of several files matching a pattern as if they were imported one by one with `$include`. File inclusion order is not guaranteed.

Example: Importing several pascal files

```
1  $include_many 'src/*.pas'
```

5.3 $include_preformatted

The `$include_preformatted` directive lets you import a file as a preformatted text (a '`>`' is inserted at the beginning of each line). This is useful to import license text or code examples.

Example: Importing a license file

```
1  Topic: License
2
3  $include_preformatted 'COPYING'
```

Example: Importing a code example

```
1  Code:
2
3  $include_preformatted 'hello.pas'
```

5.4 $include_numbered

The `$include_numbered` directive is the same as the `$include_preformatted` directive except that it automatically numbers the lines of the imported file. Line numbers generated in this way can be used as a reference later in the documentation.

Example: Importing a code example with line numbers

```
1  Code:
2  $include_numbered 'hello.pas'
3
4  **Line 6** shows how to write a simple message in pascal
5  with ``WriteLn``.
```

Warning: Syntax highlighting of code examples included in this way may be inaccurate if multiline comments are present. Solutions:

- Use only one line comments.
- Do not use syntax highlighting, ie, import the file as preformatted text (by removing the *Code* header in example above).

6 Running Txt2FPDoc

Generating the XML file for FPDoc is very simple. Assuming your documentation file is documention.txt, you can convert it to documentation.xml with the following command:

```
~$ txt2fpdoc -o documentation.xml documentation.txt
```

Sometimes something goes wrong and it can be helpful to see the documentation analyzed by Txt2FPDoc after passing through the preprocessor. This can be done with:

```
~$ txt2fpdoc --dump documentation.txt
```

Txt2FPDoc options are:

`-h, --help`
: Print a usage message, then exit.

`-v, --version`
: Print the version number, then exit.

`-o, --output=OUTFILE`
: Specify output file name.

`-d, --dump`
: Dump preprocessor output, then exit.

Appendix A Headers

A.1 Core Headers

Core headers are the minimum set of headers required to use all the capabilities of `Txt2FPDoc`. Some headers of the core headers set have synonyms that can make documentation easier to read and easier to write.

Core Headers	**Synonyms**
Element	*Callback*, *Constant*, *Constructor*, *Const*, *Destructor*, *Enumeration*, *Enumerator*, *Enum*, *Field*, *Function*, *Func*, *Member*, *Method*, *Parameter*, *Param*, *Procedure*, *Proc*, *Property*, *Type*, *Variable*, *Var*
Elements	*Consts*, *Constants*, *Types*, *Vars*, *Variables*
End Module	*End Library*, *End Program*, *End Unit*
End Package	none
End Struct	*End Class*, *End Interface*, *End Object*, *End Record*
Errors	none
Example	none
Image	none
Module	*Library*, *Program*, *Unit*
Note	none
Notes	none
Package	none
Remark	none
Result	*Returns*
See Also	none
Struct	*Class*, *Interface*, *Object*, *Record*
Sub Elements	*Arguments*, *Enumerators*, *Fields*, *Members*, *Parameters*, *Params*, *Properties*
Sub Topic	none
Table	none

Topic	none
Version	none

A.2 Alphabetical List of Headers

Arguments (synonym of *Sub Elements*)
Callback (synonym of *Element*)
Class (synonym of *Struct*)
Const (synonym of *Element*)
Constant (synonym of *Element*)
Constants (synonym of *Elements*)
Constructor (synonym of *Element*)
Consts (synonym of *Elements*)
Destructor (synonym of *Element*)
Element
Elements
End Class (synonym of *End Struct*)
End Interface (synonym of *End Struct*)
End Library (synonym of *End Module*)
End Module
End Object (synonym of *End Struct*)
End Package
End Program (synonym of *End Module*)
End Record (synonym of *End Struct*)
End Struct
End Unit (synonym of *End Module*)
Enum (synonym of *Element*)
Enumeration (synonym of *Element*)
Enumerator (synonym of *Element*)
Enumerators (synonym of *Sub Elements*)
Errors
Example
Field (synonym of *Element*)
Fields (synonym of *Sub Elements*)
Func (synonym of *Element*)
Function (synonym of *Element*)
Interface (synonym of *Struct*)
Image
Library (synonym of *Module*)
Member (synonym of *Element*)
Members (synonym of *Sub Elements*)
Method (synonym of *Element*)

Module
Note
Notes
Object (synonym of *Struct*)
Package
Param (synonym of *Element*)
Parameter (synonym of *Element*)
Parameters (synonym of *Sub Elements*)
Params (synonym of *Sub Elements*)
Proc (synonym of *Element*)
Procedure (synonym of *Element*)
Program (synonym of *Module*)
Properties (synonym of *Sub Elements*)
Property (synonym of *Element*)
Record (synonym of *Struct*)
Remark
Result
Returns (synonym of *Result*)
See Also
Struct
Sub Elements
Sub Topic
Table
Topic
Type (synonym of *Element*)
Types (synonym of *Elements*)
Unit (synonym of *Module*)
Var (synonym of *Element*)
Variable (synonym of *Element*)
Variables (synonym of *Elements*)
Vars (synonym of *Elements*)
Version

Appendix B Structure of Main Components

Structure of a Package

Short Format

```
Package: PkgName [-- ShortDescription]

[LongDescription]

{Module | Topic}

[End Package: PkgName]
```

Long Format

```
Package: PkgName

ShortDescription

[LongDescription]

{Module | Topic}

[End Package: PkgName]
```

Structure of a Module

Short Format

```
Module: ModuleName [-- ShortDescription]

[LongDescription]

{Element | Topic}

[End Module: ModuleName]
```

Long Format

```
Module: ModuleName

ShortDescription

[LongDescription]

{Element | Topic}

[End Module: ModuleName]
```

Structure of an Element

Short Format

```
Element: Name [-- ShortDescription]

[LongDescription]

{SubElements | Result | SeeAlso | Note | Notes}

{Errors | Example | Version}
```

Long Format

```
Element: Name

ShortDescription

[LongDescription]

{SubElements | Result | SeeAlso | Note | Notes}

{Errors | Example | Version}
```

Result, *See Also*, *Errors* and *Version* headers can appear only once inside an element.

Structure of a Topic

Short Format

```
Topic: Name [-- ShortDescription]

[LongDescription]

{SubTopic}
```

Long Format

```
Topic: Name

ShortDescription

[LongDescription]

{SubTopic}
```

Structure of a Sub Topic

A *Sub Topic* can only occur if its parent is directly inside a package.

Short Format

```
Sub Topic: Name [-- ShortDescription]

[LongDescription]
```

Long Format

```
Sub Topic: Name

ShortDescription

[LongDescription]
```

Appendix C Documentation File Grammar

Conventions

module
: denotes a non terminal

‘--’
: denotes a literal

IDENTIFIER
: denotes a terminal

[x]
: denotes zero or one occurrences of x.

{ x }
: denotes zero or more occurrences of x.

(x | y)
: means one of either x or y.

‘^’
: denotes start of line

‘$’
: denotes end of line

Grammar

documentation → *package*
package → *package-line description* {*module* | *topic*} [*end-package-line*]
package-line → ^ ‘Package:’ *IDENTIFIER* [‘--’ *SHORT-DESCRIPTION*] $
end-package-line → ^ ‘End Package:’ *IDENTIFIER* $
description → { *description-item* }
description-item → *paragraph* | *code* | *remark* | *list* | *preformatted* | *image* | *table*
module → *module-line description* { *note* | *notes* } *module-element-list* [*end-module-line*]
module-element-list → { *element* | *elements* | *struct* | *topic* }
module-line → ^ ‘Module:’ *IDENTIFIER* [‘--’ *SHORT-DESCRIPTION*] $
end-module-line → ^ ‘End Module:’ *IDENTIFIER* $
topic → *topic-line description* { *note* | *notes* } { *sub-topic* }
sub-topic → *sub-topic-line description* { *note* | *notes* }
topic-line → ^ ‘Topic:’ *IDENTIFIER* [‘--’ *SHORT-DESCRIPTION*] $
sub-topic-line → ^ ‘Sub Topic:’ *IDENTIFIER* [‘--’ *SHORT-DESCRIPTION*] $
paragraph → { *TEXT-LINE* }
code → *code-line preformatted*
code-line → ^ ‘Code:’ $
remark → ^ ‘Remark:’ *SHORT-DESCRIPTION* $
list → *ordered-list* | *unordered-list* | *definition-list*
definition-list → { *definition-list-line* }

definition-list-line → ^ '::' *TERM* '--' *DEFINITION* $
ordered-list → { *ordered-list-line* }
ordered-list-line → ^ '+' *TEXT* $
unordered-list → { *unordered-list-line* }
unordered-list-line → ^ '-' *TEXT* $
preformatted → { *preformatted-line* }
preformatted-line → ^ '>' *TEXT* $
image → ^ 'Image:' *FILE-NAME* '--' *CAPTION* $
table → [*table-line*] [*table-header-line*] { *table-row-line* }
table-line → ^ 'Table:' *CAPTION* $
table-header-line → ^ '||' { *CELL* '|' } $
table-row-line → ^ '|' { *CELL* '|' } $
note → ^ 'Note:' *SHORT-DESCRIPTION* $
notes → *notes-line* { *unordered-list* }
notes-line → ^ 'Notes:' $
element → *element-line description element-extra* [{*errors* | *example* | *version*}]
element-extra → { *sub-elements* | *result* | *see-also* | *note* | *notes* }
element-line → ^ 'Element:' *IDENTIFIER* ['--' *SHORT-DESCRIPTION*] $
elements → *elements-line* { *elements-item-line* }
elements-line → ^ 'Elements:' $
elements-item-line → ^ '-' *IDENTIFIER* '--' [*SHORT-DESCRIPTION*] $
struct → *struct-line description struct-element-list* [*end-struct-line*]
struct-element-list → { *element* | *elements* | *struct* | *topic* }
struct-line → ^ 'Struct:' *IDENTIFIER* ['--' *SHORT-DESCRIPTION*] $
end-struct-line → ^ 'End Struct:' *IDENTIFIER* $
errors → *errors-line description*
errors-line → ^ 'Errors:' $
example → ^ 'Example:' *FILE-NAME* $
version → *version-line description*
version-line → ^ 'Version:' $
sub-elements → *sub-elements-line* { *sub-elements-item* }
sub-elements-line → ^ 'Sub Elements:' $
sub-elements-item → '-' *IDENTIFIER* '--' *SHORT-DESCRIPTION*
result → ^ 'Result:' *SHORT-DESCRIPTION* $
see-also → ^ 'See Also:' *LINKS* $

Appendix D Complete Examples

D.1 Documentation in Source Code

Example: documentation.txt

```
Package: MyPkg
##############

$include 'myunit.pas'

End Package: MyPkg
##################
```

Example: myunit.pas

```
{**
  Unit: MyUnit
  ------------

  ``MyUnit`` contains some essential routines.

  This routines are the result of many years of **hard work**.

  I hope they will help to save mankind.
}
unit MyUnit;

interface

{**
  Function: Max -- Return largest of two values.

  ``Max`` returns the maximum of ``A`` and ``B``.

  Parameters:
    - A -- first value
    - B -- second value

  See Also: <<Min>>
}
function Max(A, B: Integer) : Integer;

{**
  Function: Min -- Return smallest of two values.

  ``Min`` returns the minimum of ``A`` and ``B``.

  Parameters:
    - A -- first value
    - B -- second value

  See Also: <<Max>>
```

```
38  }
39  function Min(A, B: Integer) : Integer;
40
41  {**
42    Function: Multiply -- Multiplies 2 integers.
43
44    Parameters:
45      - A -- first integer
46      - B -- second integer
47
48    Result:
49
50      The two integers multiplied together.
51  }
52  function Multiply(A, B: Integer) : Integer;
53
54  implementation
55  end. {** End Unit: MyUnit }
```

Example: Building documentation

```
~$ txt2fpdoc -o documentation.xml documentation.txt
~$ fpdoc --package=mypkg --descr=documentation.xml --dont-trim \
     --format=html --output=out --input=myunit.pas
```

D.2 Documentation and Source Code Apart

Example: myunit.pas

```
 1  unit MyUnit;
 2
 3  interface
 4
 5  function Max(A, B: Integer) : Integer;
 6  function Min(A, B: Integer) : Integer;
 7  function Multiply(A, B: Integer) : Integer;
 8
 9  implementation
10  end.
```

Example: documentation.txt

```
 1  Package: MyPkg
 2  ###############
 3
 4  Unit: MyUnit
 5  ============
 6
 7    ‘‘MyUnit‘‘ contains some essential routines.
 8
 9    This routines are the result of many years of **hard work**.
10
11    I hope they will help to save mankind.
12
13  Function: Max -- Return largest of two values.
```

```
14  ---------------------------------------------
15
16    ''Max'' returns the maximum of ''A'' and ''B''.
17
18    Parameters:
19      - A -- first value
20      - B -- second value
21
22   See Also: <<Min>>
23
24  Function: Min -- Return smallest of two values.
25  ---------------------------------------------
26
27    ''Min'' returns the minimum of ''A'' and ''B''.
28
29   Parameters:
30      - A -- first value
31      - B -- second value
32
33   See Also: <<Max>>
34
35  Function: Multiply -- Multiplies 2 integers.
36  ------------------------------------------
37
38    Parameters:
39      - A -- first integer
40      - B -- second integer
41
42    Result:
43
44      The two integers multiplied together.
45
46  End Unit: MyUnit
47  ================
48
49  End Package: MyPkg
50  ###################
```

Example: Building documentation

```
~$ txt2fpdoc -o documentation.xml documentation.txt
~$ fpdoc --package=mypkg --descr=documentation.xml --dont-trim \
     --format=html --output=out --input=myunit.pas
```

Index

H

I

L

M

N

O

P

R

S

T

U

V

X