1. Documentation generation
    1. Get the MkDoc tool
    2. Building the doc
    3. Adding files and modules
    4. Configuration tokens
2. Writing documentation in headers
    1. The rules
    2. Comments format
    3. Useful tags
3. What goes in the generated doc
4. Generated output
5. Linking from Trac to the API Reference Manual

This document explains how to write correct documentation in header source files for the MutekH API reference manual.

The manual is generate from both plain documentation file located in the `doc/` directory and some chosen header source files. These files are processed using the ?MkDoc tool. This tool **comes with a manual**, but it's still worth detailing some MutekH specific issues here.

Documentation generation

Get the MkDoc tool

Installation is quite easy, you just have to get the source and add the `mkdoc/src` directory in your `PATH` environment variable:

```
svn co http://svn.savannah.gnu.org/svn/mkdoc/trunk mkdoc
export PATH=$PATH:$PWD/mkdoc/src
```

You can install the package in a permanent way too with `make install`.

Building the doc

The MutekH build system handle all the tool invocation stuff:

```
$ make doc
```

Adding files and modules

If you want to add new header files, just insert a line in your **local Makefile with the relative header name**:

```
doc_headers = stdio.h sys/types.h
```

Every documented header file must contains a documentation for the file itself. This description comment must contain a `@file` tag not to be attached to the next declaration (see example below).

Files, macros and symbols can be grouped in a documentation module. This is a good idea to **create a documentation module for each MutekH library**. This can be done by declaring the new module **in the `doc/top.mkdoc` file**:

```
@moduledef{My great library}
 @short This is a simple module example
 This module ... blabla ... long description.
@end group
```

The MutekH module **must be added in the global `doc/header_list.mk` file** too.

Module ownership can be declared in a per declaration basis using the `@module` tag, however it's inherited by default. This means all symbols declared in a header file associated with a module are owned by the same module.

Configuration tokens

Build system configuration tokens are documented in each `.config` file using the mandatory `%desc` line. These descriptions and all token relationships information is used to generate a **dedicated section** in the documentation. Module **ownership must be specified** in `.config` files for tokens being associated with their module:

```
%module My great library

%config CONFIG_MY_OPTION
desc Please try to write a really useful description here from now !!!
%config end
```

Writing documentation in headers

The rules

It's really important to follow these basic rules when writing the documentation. Not following these rules carefully would make your documentation useless, as reading the source code directly will be more attractive !!!

Here are the rules:

- Please always **make complete sentences** starting with **C**apitals. (It would be great to apply this to configuration tokens description too).
- Do not make sentences for short description of files and modules introduced by a `@short` tag.
- Do not forget to **insert reference tags** for each identifier and macro name, so that MkDoc can generate an hyperlink. Not having a direct link is really frustrating for the reader.
- Remember that you can not count on other declarations proximity to make things obvious, contiguous things in the header **may not be contiguous in the document**.
- Try to **mark all internal declarations as such**, this will help sorting lists with relevant items first, using the right style, and make things clear.
- Consider the **warning messages** from MkDoc and **read carefully the output document** to double check previous rules and see if everything is easily understandable and make sens.

Comments format

Code documentation is inserted in source file **using special comment formats** beginning with `/**` (exactly 2 stars) or `//<`. The former must be placed before the associated declaration. The latter must be placed after the declaration but **doesn't work with macros** if on the same line (it's a feature, read the manual :)).

```
/** This is a special error code. */
#define EPOUFCTOUT 42

int useless; //< @this may not be used.
```

The `@this` tag can be used as a shortcut which is replaced by something like "*This static function*", "*This variable*" ...

```
/**
  @file
  @short Useless header file
  @module {My great library}

  Some optional long description and documentation...
*/

/** get a useless value (BAD) */
int badly_commented();

/** This function gets a useless value. */
int well_commented();

/** @this gets a useless value. */
int smartly_commented();
```

Useful tags

Many other tags are useful:

- `@internal`: This tag must be used to mark internals functions. In the MutekH project the policy is to mark all symbols which should not be used directly by the user as internal. **Internal are still displayed** in the documentation but marked as such.
- `@hidden`: This tag must be used to hide something you really don't want to appear in the documentation. Note that **preprocessor macros are hidden by default** if not commented.
- `@multiple`: This tag can be used to apply the same documentation more than once, up to the next documentation comment.
- `@param` and `@return`: These tags can be used to document parameters and return values. Please **don't use it for all functions** if the information are obvious. The `@return` tag starts a sentence like "*The return value is*". The `@param` tag adds a "*Parameters list:*" if at the beginning of a paragraph. Do not forget to insert an empty line when beginning a normal text paragraph after these tags.
- `@showcontent`: This tag can be used to make the content of a preprocessor macro and numerical values of enums visible in the documentation.
- `@ref` and `@see`: These tags may be used when refering to an other symbol. The `@ref` tag just insert a link to the specified symbol where it appears. `@see` tags automatically generate a nice sentence at symbol documentation end pointing to all related symbols. Do not forget to **prepend a sharp (#) sign to macro names** as they are in a different name space to avoid collisions. Headers and modules can be refered too with @ and + sign prefix.
- `@url` can be used to add an internet link. You may use this to link against this wiki.

Some examples:

```
/** @hidden */
#define while if

/**
  @multiple @internal
  @this is an internal error code.
  @see #EPOUFCTOUT @see #ESTUPIDUSER
 */
#define EFOO 1
#define EBAR 2

/**
  @this is really complex, you have better using the @ref works_better function instead.

  @param a Action to perform
  @param b Repeat count
  @param c Special parameter
  @return the negative error code.
 */
int la_fonction_qui_fait_tout(char *a, int b, const short c);

/**
  @this expands to something hard to explain.
  @showcontent
*/
#define X(a,b) do { int c = a(b); } while (0)

/**
  @this defines the prototype for the generic function.
  @see my_prototype_t
*/
#define MYPROTOTYPE(n) int (n)(int a, int b)

/**
  This function type does a lot of cool things which are described here.
  @param a The first parameter
  @param b The second parameter
  @return the special AlmSum operator result.
  @see #MYPROTOTYPE
*/
typedef MYPROTOTYPE(my_prototype_t);
```

Many other tags can be used to format text, insert examples, write structured plain documentation... Please refer to the ?MkDoc manual **for detailled syntax and usage** of discussed tags and list of other cool tags.

What goes in the generated doc

- Its embedded preprocessor enables MkDoc to **expose declarations which take place in macro expansion** with information about involved header files, macros and exact line positions. Please **do not comment things twice** when repeated but enclosed in a `#if #else #endif` as this is useless unless you want to show different macro contents for instance.
- The preprocessor **keeps track of nested conditional context** and the resulting conditional expression required for a declaration to actually take place is exposed in the documentation. This **includes configuration tokens** which are described in the documentation. Note that macros which are not documented are ignored here.
- Multiple declarations of the **same macro or symbol name are taken into account separately** but generate links to homonym declarations. This is useful for repeated declaration with different preprocessor conditions and for prototypes repeated in different files.
- Links against online MutekH source code repository with file and line number are generated correctly for each declaration **provided that the svn revision number is coherent** with the one in `doc/mkdoc.conf`

.

Generated output

The default is to generate html documentation in `doc/html`, but tweaks and other formats are available as described in MkDoc manual. You may for instance generate a latex source file:

```
make doc MKDOCFLAGS='--doc-format latex'
```

Linking from Trac to the API Reference Manual

Using the InterMapTxt feature of Trac, you can provide links from anywhere in the Trac site to the API reference manual using links looking like:

`api:`*type*`:`*name* where

- ◊ *type* can be `macro`, `function`, `enum`, `struct`, or `field`
- ◊ *name* is the name (without the `#` for macros, and like `struct_name.field_name` for *field*s)

Example:

```
 * [api:function:pthread_create link to pthread_create]
 * [api:macro:CONFIG_CPU_MAXCOUNT fancy text]
 * [api:struct:context_s CPU context structure]
 * [api:struct:fdt_walker_s.on_node_entry field on_node_entry of fdt_walker_s]
```

Yields

- ◊ link to pthread_create
- ◊ fancy text
- ◊ CPU context structure
- ◊ field on_node_entry of fdt_walker_s