Linux Kernel Documentation Guide

Release

The kernel development community

August 18, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Linux kernel uses Sphinx to generate pretty documentation from reStructuredText files under `Documentation`. To build the documentation in HTML or PDF formats, use `make htmldocs` or `make pdfdocs`. The generated documentation is placed in `Documentation/output`.

The reStructuredText files may contain directives to include structured documentation comments, or kernel-doc comments, from source files. Usually these are used to describe the functions and types and design of the code. The kernel-doc comments have some special structure and formatting, but beyond that they are also treated as reStructuredText.

Finally, there are thousands of plain text documentation files scattered around `Documentation`. Some of these will likely be converted to reStructuredText over time, but the bulk of them will remain in plain text.

CHAPTER

TWO

SPHINX INSTALL

The ReST markups currently used by the Documentation/ files are meant to be built with `Sphinx` version 1.3 or upper. If you're desiring to build PDF outputs, it is recommended to use version 1.4.6 or upper.

There's a script that checks for the Spinx requirements. Please see *Checking for Sphinx dependencies* for further details.

Most distributions are shipped with Sphinx, but its toolchain is fragile, and it is not uncommon that upgrading it or some other Python packages on your machine would cause the documentation build to break.

A way to get rid of that is to use a different version than the one shipped on your distributions. In order to do that, it is recommended to install Sphinx inside a virtual environment, using `virtualenv-3` or `virtualenv`, depending on how your distribution packaged Python 3.

Note:

1. *Sphinx versions below 1.5 don't work properly with Python's docutils version 0.13.1 or upper. So, if you're willing to use those versions, you should run `pip install 'docutils==0.12'`.*
2. *It is recommended to use the RTD theme for html output. Depending on the Sphinx version, it should be installed in separate, with `pip install sphinx_rtd_theme`.*
3. *Some ReST pages contain math expressions. Due to the way Sphinx work, those expressions are written using LaTeX notation. It needs texlive installed with amdfonts and amsmath in order to evaluate them.*

In summary, if you want to install Sphinx version 1.4.9, you should do:

```
$ virtualenv sphinx_1.4
$ . sphinx_1.4/bin/activate
(sphinx_1.4) $ pip install -r Documentation/sphinx/requirements.txt
```

After running `. sphinx_1.4/bin/activate`, the prompt will change, in order to indicate that you're using the new environment. If you open a new shell, you need to rerun this command to enter again at the virtual environment before building the documentation.

Image output

The kernel documentation build system contains an extension that handles images on both GraphViz and SVG formats (see *Figures & Images*).

For it to work, you need to install both GraphViz and ImageMagick packages. If those packages are not installed, the build system will still build the documentation, but won't include any images at the output.

PDF and LaTeX builds

Such builds are currently supported only with Sphinx versions 1.4 and upper.

For PDF and LaTeX output, you'll also need XeLaTeX version 3.14159265.

Depending on the distribution, you may also need to install a series of `texlive` packages that provide the minimal set of functionalities required for XeLaTeX to work.

Checking for Sphinx dependencies

There's a script that automatically check for Sphinx dependencies. If it can recognize your distribution, it will also give a hint about the install command line options for your distro:

```
$ ./scripts/sphinx-pre-install
Checking if the needed tools for Fedora release 26 (Twenty Six) are available
Warning: better to also install "texlive-luatex85".
You should run:

        sudo dnf install -y texlive-luatex85
        /usr/bin/virtualenv sphinx_1.4
        . sphinx_1.4/bin/activate
        pip install -r Documentation/sphinx/requirements.txt

Can't build as 1 mandatory dependency is missing at ./scripts/sphinx-pre-install line 468.
```

By default, it checks all the requirements for both html and PDF, including the requirements for images, math expressions and LaTeX build, and assumes that a virtual Python environment will be used. The ones needed for html builds are assumed to be mandatory; the others to be optional.

It supports two optional parameters:

`--no-pdf` Disable checks for PDF;

`--no-virtualenv` Use OS packaging for Sphinx instead of Python virtual environment.

CHAPTER

THREE

SPHINX BUILD

The usual way to generate the documentation is to run `make htmldocs` or `make pdfdocs`. There are also other formats available, see the documentation section of `make help`. The generated documentation is placed in format-specific subdirectories under `Documentation/output`.

To generate documentation, Sphinx (`sphinx-build`) must obviously be installed. For prettier HTML output, the Read the Docs Sphinx theme (`sphinx_rtd_theme`) is used if available. For PDF output you'll also need XeLaTeX and `convert(1)` from ImageMagick (https://www.imagemagick.org). All of these are widely available and packaged in distributions.

To pass extra options to Sphinx, you can use the `SPHINXOPTS` make variable. For example, use `make SPHINXOPTS=-v htmldocs` to get more verbose output.

To remove the generated documentation, run `make cleandocs`.

CHAPTER

FOUR

WRITING DOCUMENTATION

Adding new documentation can be as simple as:

1. Add a new `.rst` file somewhere under `Documentation`.
2. Refer to it from the Sphinx main TOC tree in `Documentation/index.rst`.

This is usually good enough for simple documentation (like the one you're reading right now), but for larger documents it may be advisable to create a subdirectory (or use an existing one). For example, the graphics subsystem documentation is under `Documentation/gpu`, split to several `.rst` files, and has a separate `index.rst` (with a `toctree` of its own) referenced from the main index.

See the documentation for Sphinx and reStructuredText on what you can do with them. In particular, the Sphinx reStructuredText Primer is a good place to get started with reStructuredText. There are also some Sphinx specific markup constructs.

Specific guidelines for the kernel documentation

Here are some specific guidelines for the kernel documentation:

- Please don't go overboard with reStructuredText markup. Keep it simple. For the most part the documentation should be plain text with just enough consistency in formatting that it can be converted to other formats.
- Please keep the formatting changes minimal when converting existing documentation to reStructuredText.
- Also update the content, not just the formatting, when converting documentation.
- Please stick to this order of heading adornments:
  1. = with overline for document title:

```
==============
Document title
==============
```

  2. = for chapters:

```
Chapters
========
```

  3. - for sections:

```
Section
-------
```

  4. ~ for subsections:

```
Subsection
~~~~~~~~~~
```

Although RST doesn't mandate a specific order ("Rather than imposing a fixed number and order of section title adornment styles, the order enforced will be the order as encountered."), having the higher levels the same overall makes it easier to follow the documents.

- For inserting fixed width text blocks (for code examples, use case examples, etc.), use `::` for anything that doesn't really benefit from syntax highlighting, especially short snippets. Use `.. code-block:: <language>` for longer code blocks that benefit from highlighting.

the C domain

The **Sphinx C Domain** (name c) is suited for documentation of C API. E.g. a function prototype:

```
.. c:function:: int ioctl( int fd, int request )
```

The C domain of the kernel-doc has some additional features. E.g. you can *rename* the reference name of a function with a common name like open or `ioctl`:

```
.. c:function:: int ioctl( int fd, int request )
   :name: VIDIOC_LOG_STATUS
```

The func-name (e.g. ioctl) remains in the output but the ref-name changed from `ioctl` to `VIDIOC_LOG_STATUS`. The index entry for this function is also changed to `VIDIOC_LOG_STATUS` and the function can now referenced by:

```
:c:func:`VIDIOC_LOG_STATUS`
```

list tables

We recommend the use of *list table* formats. The *list table* formats are double-stage lists. Compared to the ASCII-art they might not be as comfortable for readers of the text files. Their advantage is that they are easy to create or modify and that the diff of a modification is much more meaningful, because it is limited to the modified content.

The `flat-table` is a double-stage list similar to the `list-table` with some additional features:

- column-span: with the role `cspan` a cell can be extended through additional columns
- row-span: with the role `rspan` a cell can be extended through additional rows
- auto span rightmost cell of a table row over the missing cells on the right side of that table-row. With Option `:fill-cells:` this behavior can changed from *auto span* to *auto fill*, which automatically inserts (empty) cells instead of spanning the last cell.

options:

- `:header-rows:` [int] count of header rows
- `:stub-columns:` [int] count of stub columns
- `:widths:` [[int] [int] ...] widths of columns
- `:fill-cells:` instead of auto-spanning missing cells, insert missing cells

roles:

- `:cspan:` [int] additional columns (*morecols*)
- `:rspan:` [int] additional rows (*morerows*)

The example below shows how to use this markup. The first level of the staged list is the *table-row*. In the *table-row* there is only one markup allowed, the list of the cells in this *table-row*. Exceptions are *comments* (`..`) and *targets* (e.g. a ref to `:ref:`last row <last row>`` / *last row*).

```
.. flat-table:: table title
   :widths: 2 1 1 3

   * - head col 1
     - head col 2
     - head col 3
     - head col 4

   * - column 1
     - field 1.1
     - field 1.2 with autospan

   * - column 2
     - field 2.1
     - :rspan:`1` :cspan:`1` field 2.2 - 3.3

   * .. _`last row`:

     - column 3
```

Rendered as:

Table 4.1: table title

head col 1	head col 2	head col 3	head col 4
column 1	field 1.1	field 1.2 with autospan	
column 2	field 2.1	field 2.2 - 3.3	
column 3			

CHAPTER

FIVE

FIGURES & IMAGES

If you want to add an image, you should use the `kernel-figure` and `kernel-image` directives. E.g. to insert a figure with a scalable image format use SVG (*SVG image example*):

```
.. kernel-figure::  svg_image.svg
   :alt:    simple SVG image

   SVG image example
```

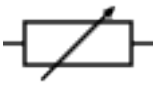

Fig. 5.1: SVG image example

The kernel figure (and image) directive support **DOT** formated files, see

- DOT: http://graphviz.org/pdf/dotguide.pdf
- Graphviz: http://www.graphviz.org/content/dot-language

A simple example (*DOT's hello world example*):

```
.. kernel-figure::  hello.dot
   :alt:    hello world

   DOT's hello world example
```

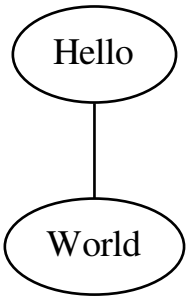

Fig. 5.2: DOT's hello world example

Embed *render* markups (or languages) like Graphviz's **DOT** is provided by the `kernel-render` directives.:

```
.. kernel-render:: DOT
   :alt: foobar digraph
   :caption: Embedded **DOT** (Graphviz) code

   digraph foo {
    "bar" -> "baz";
   }
```

How this will be rendered depends on the installed tools. If Graphviz is installed, you will see an vector image. If not the raw markup is inserted as *literal-block* (*Embedded DOT (Graphviz) code*).

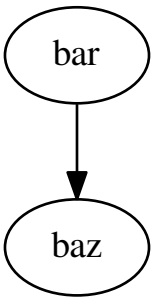

Fig. 5.3: Embedded **DOT** (Graphviz) code

The *render* directive has all the options known from the *figure* directive, plus option `caption`. If `caption` has a value, a *figure* node is inserted. If not, a *image* node is inserted. A `caption` is also needed, if you want to refer it (*Embedded SVG markup*).

Embedded **SVG**:

```
.. kernel-render:: SVG
   :caption: Embedded **SVG** markup
   :alt: so-nw-arrow

   <?xml version="1.0" encoding="UTF-8"?>
   <svg xmlns="http://www.w3.org/2000/svg" version="1.1" ...>
      ...
   </svg>
```

Fig. 5.4: Embedded **SVG** markup

CHAPTER

SIX

WRITING KERNEL-DOC COMMENTS

The Linux kernel source files may contain structured documentation comments in the kernel-doc format to describe the functions, types and design of the code. It is easier to keep documentation up-to-date when it is embedded in source files.

Note:

The kernel-doc format is deceptively similar to javadoc, gtk-doc or Doxygen, yet distinctively different, for historical reasons. The kernel source contains tens of thousands of kernel-doc comments. Please stick to the style described here.

The kernel-doc structure is extracted from the comments, and proper Sphinx C Domain function and type descriptions with anchors are generated from them. The descriptions are filtered for special kernel-doc highlights and cross-references. See below for details.

Every function that is exported to loadable modules using `EXPORT_SYMBOL` or `EXPORT_SYMBOL_GPL` should have a kernel-doc comment. Functions and data structures in header files which are intended to be used by modules should also have kernel-doc comments.

It is good practice to also provide kernel-doc formatted documentation for functions externally visible to other kernel files (not marked `static`). We also recommend providing kernel-doc formatted documentation for private (file `static`) routines, for consistency of kernel source code layout. This is lower priority and at the discretion of the maintainer of that kernel source file.

How to format kernel-doc comments

The opening comment mark `/**` is used for kernel-doc comments. The `kernel-doc` tool will extract comments marked this way. The rest of the comment is formatted like a normal multi-line comment with a column of asterisks on the left side, closing with `*/` on a line by itself.

The function and type kernel-doc comments should be placed just before the function or type being described in order to maximise the chance that somebody changing the code will also change the documentation. The overview kernel-doc comments may be placed anywhere at the top indentation level.

Running the `kernel-doc` tool with increased verbosity and without actual output generation may be used to verify proper formatting of the documentation comments. For example:

```
scripts/kernel-doc -v -none drivers/foo/bar.c
```

The documentation format is verified by the kernel build when it is requested to perform extra gcc checks:

```
make W=n
```

Function documentation

The general format of a function and function-like macro kernel-doc comment is:

```
/**
 * function_name() - Brief description of function.
 * @arg1: Describe the first argument.
 * @arg2: Describe the second argument.
 *        One can provide multiple line descriptions
 *        for arguments.
 *
 * A longer description, with more discussion of the function function_name()
 * that might be useful to those using or modifying it. Begins with an
 * empty comment line, and may include additional embedded empty
 * comment lines.
 *
 * The longer description may have multiple paragraphs.
 *
 * Context: Describes whether the function can sleep, what locks it takes,
 *          releases, or expects to be held. It can extend over multiple
 *          lines.
 * Return: Describe the return value of foobar.
 *
 * The return value description can also have multiple paragraphs, and should
 * be placed at the end of the comment block.
 */
```

The brief description following the function name may span multiple lines, and ends with an argument description, a blank comment line, or the end of the comment block.

Function parameters

Each function argument should be described in order, immediately following the short function description. Do not leave a blank line between the function description and the arguments, nor between the arguments.

Each `@argument:` description may span multiple lines.

Note:

If the `@argument` description has multiple lines, the continuation of the description should start at the same column as the previous line:

```
* @argument: some long description
*            that continues on next lines
```

or:

```
* @argument:
*         some long description
*         that continues on next lines
```

If a function has a variable number of arguments, its description should be written in kernel-doc notation as:

```
* @...: description
```

Function context

The context in which a function can be called should be described in a section named `Context`. This should include whether the function sleeps or can be called from interrupt context, as well as what locks it takes, releases and expects to be held by its caller.

Examples:

```
* Context: Any context.
* Context: Any context. Takes and releases the RCU lock.
* Context: Any context. Expects <lock> to be held by caller.
* Context: Process context. May sleep if @gfp flags permit.
* Context: Process context. Takes and releases <mutex>.
* Context: Softirq or process context. Takes and releases <lock>, BH-safe.
* Context: Interrupt context.
```

Return values

The return value, if any, should be described in a dedicated section named `Return`.

Note:

1. *The multi-line descriptive text you provide does not recognize line breaks, so if you try to format some text nicely, as in:*

```
* Return:
* 0 - OK
* -EINVAL - invalid argument
* -ENOMEM - out of memory
```

this will all run together and produce:

```
Return: 0 - OK -EINVAL - invalid argument -ENOMEM - out of memory
```

So, in order to produce the desired line breaks, you need to use a ReST list, e. g.:

```
* Return:
* * 0             - OK to runtime suspend the device
* * -EBUSY        - Device should not be runtime suspended
```

2. *If the descriptive text you provide has lines that begin with some phrase followed by a colon, each of those phrases will be taken as a new section heading, which probably won't produce the desired effect.*

Structure, union, and enumeration documentation

The general format of a struct, union, and enum kernel-doc comment is:

```
/**
 * struct struct_name - Brief description.
 * @member1: Description of member1.
 * @member2: Description of member2.
 *           One can provide multiple line descriptions
 *           for members.
 *
 * Description of the structure.
 */
```

You can replace the `struct` in the above example with `union` or `enum` to describe unions or enums. `member` is used to mean struct and union member names as well as enumerations in an enum.

The brief description following the structure name may span multiple lines, and ends with a member description, a blank comment line, or the end of the comment block.

Members

Members of structs, unions and enums should be documented the same way as function parameters; they immediately succeed the short description and may be multi-line.

Inside a struct or union description, you can use the `private:` and `public:` comment tags. Structure fields that are inside a `private:` area are not listed in the generated output documentation.

The `private:` and `public:` tags must begin immediately following a `/*` comment marker. They may optionally include comments between the `:` and the ending `*/` marker.

Example:

```
/**
 * struct my_struct - short description
 * @a: first member
 * @b: second member
 * @d: fourth member
 *
 * Longer description
 */
struct my_struct {
    int a;
    int b;
/* private: internal use only */
    int c;
/* public: the next one is public */
    int d;
};
```

Nested structs/unions

It is possible to document nested structs and unions, like:

```
/**
 * struct nested_foobar - a struct with nested unions and structs
 * @memb1: first member of anonymous union/anonymous struct
 * @memb2: second member of anonymous union/anonymous struct
 * @memb3: third member of anonymous union/anonymous struct
 * @memb4: fourth member of anonymous union/anonymous struct
 * @bar: non-anonymous union
 * @bar.st1: struct st1 inside @bar
 * @bar.st2: struct st2 inside @bar
 * @bar.st1.memb1: first member of struct st1 on union bar
 * @bar.st1.memb2: second member of struct st1 on union bar
 * @bar.st2.memb1: first member of struct st2 on union bar
 * @bar.st2.memb2: second member of struct st2 on union bar
 */
struct nested_foobar {
  /* Anonymous union/struct*/
  union {
    struct {
      int memb1;
      int memb2;
  }
    struct {
```

```
      void *memb3;
      int memb4;
    }
  }
  union {
    struct {
      int memb1;
      int memb2;
    } st1;
    struct {
      void *memb1;
      int memb2;
    } st2;
  } bar;
};
```

Note:

1. *When documenting nested structs or unions, if the struct/union `foo` is named, the member `bar` inside it should be documented as `@foo.bar:`*
2. *When the nested struct/union is anonymous, the member `bar` in it should be documented as `@bar:`*

In-line member documentation comments

The structure members may also be documented in-line within the definition. There are two styles, single-line comments where both the opening `/**` and closing `*/` are on the same line, and multi-line comments where they are each on a line of their own, like all other kernel-doc comments:

```
/**
 * struct foo - Brief description.
 * @foo: The Foo member.
 */
struct foo {
      int foo;
      /**
       * @bar: The Bar member.
       */
      int bar;
      /**
       * @baz: The Baz member.
       *
       * Here, the member description may contain several paragraphs.
       */
      int baz;
      union {
              /** @foobar: Single line description. */
              int foobar;
      };
      /** @bar2: Description for struct @bar2 inside @foo */
      struct {
              /**
               * @bar2.barbar: Description for @barbar inside @foo.bar2
               */
              int barbar;
      } bar2;
};
```

Typedef documentation

The general format of a typedef kernel-doc comment is:

```
/**
 * typedef type_name - Brief description.
 *
 * Description of the type.
 */
```

Typedefs with function prototypes can also be documented:

```
/**
 * typedef type_name - Brief description.
 * @arg1: description of arg1
 * @arg2: description of arg2
 *
 * Description of the type.
 *
 * Context: Locking context.
 * Return: Meaning of the return value.
 */
typedef void (*type_name)(struct v4l2_ctrl *arg1, void *arg2);
```

Highlights and cross-references

The following special patterns are recognized in the kernel-doc comment descriptive text and converted to proper reStructuredText markup and Sphinx C Domain references.

Attention: *The below are **only** recognized within kernel-doc comments, **not** within normal reStructure documents.*

funcname() Function reference.

@parameter Name of a function parameter. (No cross-referencing, just formatting.)

%CONST Name of a constant. (No cross-referencing, just formatting.)

``literal`` A literal block that should be handled as-is. The output will use a `monospaced font`.

Useful if you need to use special characters that would otherwise have some meaning either by kernel-doc script of by reStructuredText.

This is particularly useful if you need to use things like %ph inside a function description.

$ENVVAR Name of an environment variable. (No cross-referencing, just formatting.)

&struct name Structure reference.

&enum name Enum reference.

&typedef name Typedef reference.

&struct_name->member or &struct_name.member Structure or union member reference. The cross-reference will be to the struct or union definition, not the member directly.

&name A generic type reference. Prefer using the full reference described above instead. This is mostly for legacy comments.

Cross-referencing from reStructuredText

To cross-reference the functions and types defined in the kernel-doc comments from reStructuredText documents, please use the Sphinx C Domain references. For example:

```
See function :c:func:`foo` and struct/union/enum/typedef :c:type:`bar`.
```

While the type reference works with just the type name, without the struct/union/enum/typedef part in front, you may want to use:

```
See :c:type:`struct foo <foo>`.
See :c:type:`union bar <bar>`.
See :c:type:`enum baz <baz>`.
See :c:type:`typedef meh <meh>`.
```

This will produce prettier links, and is in line with how kernel-doc does the cross-references.

For further details, please refer to the Sphinx C Domain documentation.

Overview documentation comments

To facilitate having source code and comments close together, you can include kernel-doc documentation blocks that are free-form comments instead of being kernel-doc for functions, structures, unions, enums, or typedefs. This could be used for something like a theory of operation for a driver or library code, for example.

This is done by using a `DOC:` section keyword with a section title.

The general format of an overview or high-level documentation comment is:

```
/**
 * DOC: Theory of Operation
 *
 * The whizbang foobar is a dilly of a gizmo. It can do whatever you
 * want it to do, at any time. It reads your mind. Here's how it works.
 *
 * foo bar splat
 *
 * The only drawback to this gizmo is that is can sometimes damage
 * hardware, software, or its subject(s).
 */
```

The title following `DOC:` acts as a heading within the source file, but also as an identifier for extracting the documentation comment. Thus, the title must be unique within the file.

CHAPTER

SEVEN

INCLUDING KERNEL-DOC COMMENTS

The documentation comments may be included in any of the reStructuredText documents using a dedicated kernel-doc Sphinx directive extension.

The kernel-doc directive is of the format:

```
.. kernel-doc:: source
   :option:
```

The *source* is the path to a source file, relative to the kernel source tree. The following directive options are supported:

export: *[source-pattern ...]* Include documentation for all functions in *source* that have been exported using EXPORT_SYMBOL or EXPORT_SYMBOL_GPL either in *source* or in any of the files specified by *source-pattern*.

The *source-pattern* is useful when the kernel-doc comments have been placed in header files, while EXPORT_SYMBOL and EXPORT_SYMBOL_GPL are next to the function definitions.

Examples:

```
.. kernel-doc:: lib/bitmap.c
   :export:

.. kernel-doc:: include/net/mac80211.h
   :export: net/mac80211/*.c
```

internal: *[source-pattern ...]* Include documentation for all functions and types in *source* that have **not** been exported using EXPORT_SYMBOL or EXPORT_SYMBOL_GPL either in *source* or in any of the files specified by *source-pattern*.

Example:

```
.. kernel-doc:: drivers/gpu/drm/i915/intel_audio.c
   :internal:
```

doc: *title* Include documentation for the DOC: paragraph identified by *title* in *source*. Spaces are allowed in *title*; do not quote the *title*. The *title* is only used as an identifier for the paragraph, and is not included in the output. Please make sure to have an appropriate heading in the enclosing reStructuredText document.

Example:

```
.. kernel-doc:: drivers/gpu/drm/i915/intel_audio.c
   :doc: High Definition Audio over HDMI and Display Port
```

functions: *function [...]* Include documentation for each *function* in *source*.

Example:

```
.. kernel-doc:: lib/bitmap.c
   :functions: bitmap_parselist bitmap_parselist_user
```

Without options, the kernel-doc directive includes all documentation comments from the source file.

The kernel-doc extension is included in the kernel source tree, at `Documentation/sphinx/kerneldoc.py`. Internally, it uses the `scripts/kernel-doc` script to extract the documentation comments from the source.

How to use kernel-doc to generate man pages

If you just want to use kernel-doc to generate man pages you can do this from the kernel git tree:

```
$ scripts/kernel-doc -man $(git grep -l '/\*\*' -- :^Documentation :^tools) | scripts/split-man.pl /tmp/
```

CHAPTER

EIGHT

INCLUDING UAPI HEADER FILES

Sometimes, it is useful to include header files and C example codes in order to describe the userspace API and to generate cross-references between the code and the documentation. Adding cross-references for userspace API files has an additional vantage: Sphinx will generate warnings if a symbol is not found at the documentation. That helps to keep the uAPI documentation in sync with the Kernel changes. The *parse_headers.pl* provide a way to generate such cross-references. It has to be called via Makefile, while building the documentation. Please see `Documentation/media/Makefile` for an example about how to use it inside the Kernel tree.

parse_headers.pl

NAME

parse_headers.pl - parse a C file, in order to identify functions, structs, enums and defines and create cross-references to a Sphinx book.

SYNOPSIS

parse_headers.pl [<options>] <C_FILE> <OUT_FILE> [<EXCEPTIONS_FILE>]

Where <options> can be: –debug, –help or –man.

OPTIONS

–debug

Put the script in verbose mode, useful for debugging.

–usage

Prints a brief help message and exits.

–help

Prints a more detailed help message and exits.

DESCRIPTION

Convert a C header or source file (C_FILE), into a ReStructured Text included via ..parsed-literal block with cross-references for the documentation files that describe the API. It accepts an optional EXCEPTIONS_FILE with describes what elements will be either ignored or be pointed to a non-default reference.

The output is written at the (OUT_FILE).

It is capable of identifying defines, functions, structs, typedefs, enums and enum symbols and create cross-references for all of them. It is also capable of distinguish #define used for specifying a Linux ioctl.

The EXCEPTIONS_FILE contain two types of statements: **ignore** or **replace**.

The syntax for the ignore tag is:

ignore **type name**

The **ignore** means that it won't generate cross references for a **name** symbol of type **type**.

The syntax for the replace tag is:

replace **type name new_value**

The **replace** means that it will generate cross references for a **name** symbol of type **type**, but, instead of using the default replacement rule, it will use **new_value**.

For both statements, **type** can be either one of the following:

ioctl

The ignore or replace statement will apply to ioctl definitions like:

#define VIDIOC_DBG_S_REGISTER _IOW('V', 79, struct v4l2_dbg_register)

define

The ignore or replace statement will apply to any other #define found at C_FILE.

typedef

The ignore or replace statement will apply to typedef statements at C_FILE.

struct

The ignore or replace statement will apply to the name of struct statements at C_FILE.

enum

The ignore or replace statement will apply to the name of enum statements at C_FILE.

symbol

The ignore or replace statement will apply to the name of enum statements at C_FILE.

For replace statements, **new_value** will automatically use :c:type: references for **typedef**, **enum** and **struct** types. It will use :ref: for **ioctl**, **define** and **symbol** types. The type of reference can also be explicitly defined at the replace statement.

EXAMPLES

ignore define _VIDEODEV2_H

Ignore a #define _VIDEODEV2_H at the C_FILE.

ignore symbol PRIVATE

On a struct like:

enum foo { BAR1, BAR2, PRIVATE };

It won't generate cross-references for **PRIVATE**.

replace symbol BAR1 :c:type:'foo' replace symbol BAR2 :c:type:'foo'

On a struct like:

enum foo { BAR1, BAR2, PRIVATE };

It will make the BAR1 and BAR2 enum symbols to cross reference the foo symbol at the C domain.

BUGS

Report bugs to Mauro Carvalho Chehab <mchehab@kernel.org>

COPYRIGHT