kernel-doc tests

Release 20230630

John Doe

Jun 30, 2023

C SOURCES

Wtihin this section you will find some LinuxDoc HowTo tests and examples for common use cases. The kernel-doc comments are taken from the source files *source all-in-a-tumble.c* and *source all-in-a-tumble.h*.

CHAPTER ONE

SOURCE OF ALL-IN-A-TUMBLE.[CH]

Below you find the *source code* from the example files

- *source all-in-a-tumble.h* and
- *source all-in-a-tumble.c*.

Within these source files *here* you see some:

```
/* parse-SNIP: ... */
```

aka Snippets, which we will use in section: *kernel-doc Test*.

1.1 source all-in-a-tumble.h

```
/* parse-markup: reST */

/**
 * DOC: About Examples
 *
 * The files :ref:`all-in-a-tumble.c-src` and :ref:`all-in-a-tumble.h-src` are
 * including all examples of the :ref:`linuxdoc-howto` documentation.  These
 * files are also used as a test of the kernel-doc parser, to see how kernel-doc
 * content will be rendered and where the parser might fail.
 *
 * And ... The content itself is nonsense / don't look to close ;-)
 */

// testing:
//
// .. kernel-doc::  ./all-in-a-tumble.c
//     :export:  ./all-in-a-tumble.h

/* parse-SNIP:  EXPORT_SYMBOL */
EXPORT_SYMBOL_GPL_FUTURE(user_function)

int user_function(int a, ...)
/* parse-SNAP: */

```

(continues on next page)

(continued from previous page)

```
/* parse-SNIP:  user_sum-h */
int user_sum(int a, int b);
/* parse-SNAP: */


/**
 * block_touch_buffer - mark a buffer accessed
 * @bh: buffer_head being touched
 *
 * Called from touch_buffer().
 */
DEFINE_EVENT(block_buffer, block_touch_buffer,

        TP_PROTO(struct buffer_head *bh),

        TP_ARGS(bh)
);

/**
 * block_dirty_buffer - mark a buffer dirty
 * @bh: buffer_head being dirtied
 *
 * Called from mark_buffer_dirty().
 */
DEFINE_EVENT(block_buffer, block_dirty_buffer,

        TP_PROTO(struct buffer_head *bh),

        TP_ARGS(bh)
);

// The parse-SNIP/SNAP comments are used to include the C sorce code as snippets
// into a reST document. These are the examples of the kernel-doc-HOWTO book.

/* parse-SNIP: theory-of-operation */
/**
 * 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 it can sometimes damage hardware,
 * software, or its subject(s).
 *
 * DOC: multiple DOC sections
 *
 * It's not recommended to place more than one "DOC:" section in the same
 * comment block. To insert a new "DOC:" section, create a new comment block and
 * to create a sub-section use the reST markup for headings, see documentation
```

(continues on next page)

(continued from previous page)

```
 * of function rst_mode()
 */
/* parse-SNAP: */

/* parse-SNIP: lorem */
/**
 * DOC: lorem ipsum
 *
 * Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor
 * incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
 * nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi
 * consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore
 * eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident,
 * sunt in culpa qui officia deserunt mollit anim id est laborum.
 */
/* parse-SNAP: */


/* parse-SNIP: my_long_struct */
/**
 * struct my_long_struct - short description with &my_struct->a and &my_struct->b
 * @foo: The Foo member.
 *
 * Longer description
 */
struct my_long_struct {
        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 @my_long_struct */
        struct {
                /**
                 * @bar2.barbar: Description for @barbar inside @my_long_struct.bar2
                 */
                int barbar;
        } bar2;
};
/* parse-SNAP: */


```

(continues on next page)

(continued from previous page)

```
/* parse-SNIP: my_union */
/**
 * union my_union - short description
 * @a: first member
 * @b: second member
 *
 * Longer description
 */
union my_union {
    int a;
    int b;
};
/* parse-SNAP: */


/* parse-SNIP: my_enum */
/**
 * enum my_enum - log level
 * @QUIET: logs nothing
 * @INFO: logs info messages
 * @WARN: logs warn and info messages
 * @DEBUG: logs debug, warn and info messages
 */

enum my_enum {
  QUIET,
  INFO,
  WARN,
  DEBUG
};
/* parse-SNAP: */


/* parse-SNIP: my_typedef */
/**
 * typedef my_typedef - useless typdef of int
 *
 */
typedef int my_typedef;
/* parse-SNAP: */


/* parse-SNIP: rst_mode */
/**
 * rst_mode - dummy to demonstrate reST & kernel-doc markup in comments
 * @a: first argument
 * @b: second argument
 * Context: :c:func:`in_gizmo_mode`.
 *
 * Long description. This function has two integer arguments. The first is
 * ``parameter_a`` and the second is ``parameter_b``.
 *
```

(continues on next page)

(continued from previous page)

```
 * As long as the reST / sphinx-doc toolchain uses `intersphinx
 * <http://www.sphinx-doc.org/en/stable/ext/intersphinx.html>`__ you can refer
 * definitions *outside* like :c:type:`struct media_device <media_device>`.  If
 * the description of ``media_device`` struct is found in any of the intersphinx
 * locations, a hyperref to this target is generated a build time.
 *
 * Example:
 *   int main() {
 *     printf("Hello World\n");
 *     return 0;
 *   }
 *
 * Return: Sum of ``parameter_a`` and the second is ``parameter_b``.
 *
 * highlighting:
 * The highlight pattern, are non regular reST markups. They are only available
 * within kernel-doc comments, helping C developers to write short and compact
 * documentation.
 *
 * - user_function() : function
 * - @a : name of a parameter
 * - &struct my_struct : name of a structure (including the word struct)
 * - &union my_union : name of a union
 * - &my_struct->a or &my_struct.b -  member of a struct or union.
 * - &enum my_enum : name of a enum
 * - &typedef my_typedef : name of a typedef
 * - %CONST : name of a constant.
 * - $ENVVAR : environmental variable
 *
 * The kernel-doc parser translates the pattern above to the corresponding reST
 * markups. You don't have to use the *highlight* pattern, if you prefer *pure*
 * reST, use the reST markup.
 *
 * - :c:func:`user_function` : function
 * - ``a`` : name of a parameter
 * - :c:type:`struct my_struct <my_struct>` : name of a structure (including the word␣
↪struct)
 * - :c:type:`union my_union <my_union>` : name of a union
 * - :c:type:`my_struct->a <my_struct>` or :c:type:`my_struct.b <my_struct>` -  member of␣
↪a struct or union.
 * - :c:type:`enum my_enum <my_enum>` : name of a enum
 * - :c:type:`typedef my_typedef <my_typedef>` : name of a typedef
 * - ``CONST`` : name of a constant.
 * - ``\$ENVVAR`` : environmental variable
 *
 * Since the prefixes ``$...``, ``&...`` and ``@...`` are used to markup the
 * highlight pattern, you have to escape them in other uses: \$lorem, \&lorem,
 * \%lorem and \@lorem. To esacpe from function highlighting, use lorem\().
 *
 * Parser Mode:
 * This is an example with activated reST additions, in this section you will
 * find some common inline markups.
```

(continues on next page)

(continued from previous page)

```
 *
 * Within the *reST mode* the kernel-doc parser pass through all markups to the
 * reST toolchain, except the *vintage highlighting* but including any
 * whitespace. With this, the full reST markup is available in the comments.
 *
 * This is a link to the `Linux kernel source tree
 * <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/>`_.
 *
 * This description is only to show some reST inline markups like *emphasise*
 * and **emphasis strong**. The following is a demo of a reST list markup:
 *
 * Definition list:
 * :def1: lorem
 * :def2: ipsum
 *
 * Ordered List:
 * - item one
 * - item two
 * - item three with
 *   a linebreak
 *
 * Literal blocks:
 * The next example shows a literal block::
 *
 *     +------+          +------+
 *     |\     |\        /|     /|
 *     | +----+-+      +-+----+ |
 *     | |    | |      | |    | |
 *     +-+----+ |      | +----+-+
 *      \|     \|      |/     |/
 *       +------+      +------+
 *         foo()         bar()
 *
 * Highlighted code blocks:
 * The next example shows a code block, with highlighting C syntax in the
 * output.
 *
 * .. code-block:: c
 *
 *     // Hello World program
 *     #include<stdio.h>
 *     int main()
 *     {
 *        printf("Hello World");
 *     }
 *
 *
 * reST sectioning:
 *
 * colon markup: sectioning by colon markup in reST mode is less ugly. ;-)
 *
 * A kernel-doc section like *this* section is translated into a reST
```

(continues on next page)

(continued from previous page)

```
 * *subsection*. This means, you can only use the following *sub-levels* within a
 * kernel-doc section.
 *
 * a subsubsection
 * ^^^^^^^^^^^^^^^
 *
 * lorem ipsum
 *
 * a paragraph
 * """""""""""
 *
 * lorem ipsum
 *
 */
int rst_mode(int a, char *b)
{
  return a + b;
}
/* parse-SNAP: */


/* parse-markup: kernel-doc */

/**
 * vintage - short description of this function
 * @parameter_a: first argument
 * @parameter_b: second argument
 * Context: in_gizmo_mode().
 *
 * This is a test of a typical markup from *vintage* kernel-doc.  Don't look to
 * close here, it is only for testing some kernel-doc parser stuff.
 *
 * Long description. This function has two integer arguments. The first is
 * @parameter_a and the second is @parameter_b.
 *
 * Example: user_function(22);
 *
 * Return: Sum of @parameter_a and @parameter_b.
 *
 * highlighting:
 *
 * - vintage()    : function
 * - @parameter_a : name of a parameter
 * - $ENVVAR      : environmental variable
 * - &my_struct   : name of a structure (up to two words including ``struct``)
 * - %CONST       : name of a constant.
 *
 * Parser Mode: *vintage* kernel-doc mode
 *
 * Within the *vintage kernel-doc mode* ignores any whitespace or inline
 * markup.
 *
```

(continues on next page)

(continued from previous page)

```
 * - Inline markup like *emphasis* or **emphasis strong**
 * - Literals and/or block indent:
 *
 *      a + b
 *
 * In kernel-doc *vintage* mode, there are no special block or inline markups
 * available. Markups like the one above result in ambiguous reST markup which
 * could produce error messages in the subsequently sphinx-build
 * process. Unexpected outputs are mostly the result.
 *
 * This is a link https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/
 * to the Linux kernel source tree
 *
 * colon markup: sectioning by colon markup in vintage mode is partial ugly. ;-)
 *
 */
int vintage(int parameter_a, char parameter_b)
{
  return a + b;
}

/* some C&P for extended tests
 */

/**
* struct nfp_flower_priv - Flower APP per-vNIC priv data
* @nn:                 Pointer to vNIC
* @mask_id_seed:       Seed used for mask hash table
* @flower_version:     HW version of flower
* @mask_ids:           List of free mask ids
* @mask_table:         Hash table used to store masks
* @flow_table:         Hash table used to store flower rules
*/
struct nfp_flower_priv {
      struct nfp_net *nn;
      u32 mask_id_seed;
      u64 flower_version;
      struct nfp_fl_mask_id mask_ids;
      DECLARE_HASHTABLE(mask_table, NFP_FLOWER_MASK_HASH_BITS);
      DECLARE_HASHTABLE(flow_table, NFP_FLOWER_HASH_BITS);
};

/**
 * enum foo - foo
 * @F1: f1
 * @F2: f2
 */
enum foo {
        F1,

        F2,
};
```

(continues on next page)

(continued from previous page)

```


/**
* struct something - Lorem ipsum dolor sit amet.
* @foofoo: lorem
* @barbar: ipsum
*/

struct something {
        struct foo

        foofoo;

        struct bar

        barbar;
};

/**
 * struct lineevent_state - contains the state of a userspace event
 * @gdev: the GPIO device the event pertains to
 * @label: consumer label used to tag descriptors
 * @desc: the GPIO descriptor held by this event
 * @eflags: the event flags this line was requested with
 * @irq: the interrupt that trigger in response to events on this GPIO
 * @wait: wait queue that handles blocking reads of events
 * @events: KFIFO for the GPIO events (testing DECLARE_KFIFO)
 * @foobar: testing DECLARE_KFIFO_PTR
 * @read_lock: mutex lock to protect reads from colliding with adding
 * new events to the FIFO
 */
struct lineevent_state {
        struct gpio_device *gdev;
        const char *label;
        struct gpio_desc *desc;
        u32 eflags;
        int irq;
        wait_queue_head_t wait;
        DECLARE_KFIFO(events, struct gpioevent_data, 16);
        DECLARE_KFIFO_PTR(foobar, struct lirc_scancode);
        struct mutex read_lock;
};
```

1.2 source `all-in-a-tumble.c`

```c
// this test some kernel-doc stuff

/* parse-SNIP: hello-world */
#include<stdio.h>
int main() {
  printf("Hello World\n");
  return 0;
}
/* parse-SNAP: */

/* parse-SNIP: user_function */
/**
 * user_function() - function that can only be called in user context
 * @a: some argument
 * @...: ellipsis operator
 *
 * This function makes no sense, it's only a kernel-doc demonstration.
 *
 * Example:
 * x = user_function(22);
 *
 * Return:
 * Returns first argument
 */
int
user_function(int a, ...)
{
        return a;
}
/* parse-SNAP: */


/* parse-SNIP: user_sum-c */
/**
 * user_sum() - another function that can only be called in user context
 * @a: first argument
 * @b: second argument
 *
 * This function makes no sense, it's only a kernel-doc demonstration.
 *
 * Example:
 * x = user_sum(1, 2);
 *
 * Return:
 * Returns the sum of the @a and @b
 */
API_EXPORTED
int user_sum(int a, int b)
{
        return a + b;
```

(continues on next page)

(continued from previous page)

```
}
/* parse-SNAP: */

/* parse-SNIP: internal_function */
/**
 * internal_function - the answer
 *
 * Context: !sanity()
 *
 * Return:
 * The answer to the ultimate question of life, the universe and everything.
 */
int internal_function()
{
        return 42;
}
/* parse-SNAP: */

/* parse-SNIP: test_SYSCALL */
/**
 * sys_tgkill - send signal to one specific thread
 * @tgid: the thread group ID of the thread
 * @pid: the PID of the thread
 * @sig: signal to be sent
 *
 * Return:
 *
 * This syscall also checks the @tgid and returns -ESRCH even if the PID
 * exists but it's not belonging to the target process anymore. This
 * method solves the problem of threads exiting and PIDs getting reused.
 */
SYSCALL_DEFINE3(tgkill, pid_t, tgid, pid_t, pid, int, sig)
{
        ...
}

/* parse-SNAP: */

/* parse-SNIP: rarely_code_styles*/
/**
* enum rarely_enum - enum to test parsing rarely code styles
* @F1: f1
* @F2: f2
*/
enum rarely_enum {
        F1,

        F2,
};


/**
```

(continues on next page)

(continued from previous page)

```
* struct rarely_struct - struct to test parsing rarely code styles
* @foofoo: lorem
* @barbar: ipsum
*/

struct rarely_struct {
        struct foo

        foofoo;

        struct bar

        barbar;
};


```

CHAPTER

TWO

RENDERED `ALL-IN-A-TUMBLE.[CH]`

Below you find the rendered reST markup, generated from kernel-doc comments of the example files *all-in-a-tumble.h* and *all-in-a-tumble.c*. This content will be produced by the kernel-doc parser and inserted in the document by using the following directives:

```
.. kernel-doc::  /src/all-in-a-tumble.c
   :module: example

.. kernel-doc::  /src/all-in-a-tumble.h
   :module: example
```

The option `:module:` is optional, to find out why we use this option *here*, see kernel-doc options.

2.1 all-in-a-tumble.h

2.1.1 About Examples

The files *source all-in-a-tumble.c* and *source all-in-a-tumble.h* are including all examples of the LinuxDoc HowTo documentation. These files are also used as a test of the kernel-doc parser, to see how kernel-doc content will be rendered and where the parser might fail.

And … The content itself is nonsense / don't look to close ;-)

2.1.2 trace_block_touch_buffer

void **`trace_block_touch_buffer`**(struct buffer_head *bh)

mark a buffer accessed

Parameters

- **bh** (`struct buffer_head*`) – buffer_head being touched

Description

Called from `touch_buffer()`.

2.1.3 trace_block_dirty_buffer

void **`trace_block_dirty_buffer`**(struct buffer_head *bh)

mark a buffer dirty

Parameters

- **bh** (`struct buffer_head*`) – buffer_head being dirtied

Description

Called from `mark_buffer_dirty()`.

2.1.4 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 it can sometimes damage hardware, software, or its subject(s).

2.1.5 multiple DOC sections

It's not recommended to place more than one "DOC:" section in the same comment block. To insert a new "DOC:" section, create a new comment block and to create a sub-section use the reST markup for headings, see documentation of function *`rst_mode()`*

2.1.6 lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

2.1.7 struct my_long_struct

struct **`my_long_struct`**

short description with `my_struct->a` and `my_struct->b`

Definition

```
struct my_long_struct {
    int foo;
    int bar;
    int baz;
    union {
        int foobar;
    } ;
    struct {
        int barbar;
    } bar2;
}
```

Members

foo
  The Foo member.

bar
  The Bar member.

baz
  The Baz member.

  Here, the member description may contain several paragraphs.

{unnamed_union}
  anonymous

foobar
  Single line description.

bar2
  Description for struct `bar2` inside `my_long_struct`

bar2.barbar
  Description for `barbar` inside `my_long_struct.bar2`

Description

Longer description

2.1.8 union my_union

union **`my_union`**
  short description

Definition

```
union my_union {
    int a;
    int b;
}
```

Members

a
  first member

b
  second member

Description

Longer description

2.1.9 enum my_enum

enum **my_enum**

log level

Definition

```
enum my_enum {
    QUIET,
    INFO,
    WARN,
    DEBUG
};
```

Constants

QUIET
logs nothing

INFO
logs info messages

WARN
logs warn and info messages

DEBUG
logs debug, warn and info messages

2.1.10 typedef my_typedef

type **my_typedef**

useless typdef of int

2.1.11 rst_mode

int **rst_mode**(int a, char *b)

dummy to demonstrate reST & kernel-doc markup in comments

Parameters

- **a** (int) – first argument
- **b** (char*) – second argument Context: `in_gizmo_mode()`.

Description

Long description. This function has two integer arguments. The first is `parameter_a` and the second is `parameter_b`.

As long as the reST / sphinx-doc toolchain uses intersphinx you can refer definitions *outside* like `struct media_device`. If the description of `media_device` struct is found in any of the intersphinx locations, a hyperref to this target is generated a build time.

Example

```
int main() {
  printf("Hello World\n");
  return 0;
}
```

Return

Sum of `parameter_a` and the second is `parameter_b`.

highlighting

The highlight pattern, are non regular reST markups. They are only available within kernel-doc comments, helping C developers to write short and compact documentation.

- *`user_function()`* : function
- `a` : name of a parameter
- `struct my_struct` : name of a structure (including the word struct)
- *`union my_union`* : name of a union
- `my_struct->a` or `my_struct.b` - member of a struct or union.
- *`enum my_enum`* : name of a enum
- *`typedef my_typedef`* : name of a typedef
- `CONST` : name of a constant.
- `$ENVVAR` : environmental variable

The kernel-doc parser translates the pattern above to the corresponding reST markups. You don't have to use the *highlight* pattern, if you prefer *pure* reST, use the reST markup.

- *`user_function()`* : function
- `a` : name of a parameter
- `struct my_struct` : name of a structure (including the word struct)
- *`union my_union`* : name of a union
- `my_struct->a` or `my_struct.b` - member of a struct or union.
- *`enum my_enum`* : name of a enum
- *`typedef my_typedef`* : name of a typedef
- `CONST` : name of a constant.

- `$ENVVAR` : environmental variable

Since the prefixes `$...`, `&...` and `@...` are used to markup the highlight pattern, you have to escape them in other uses: $lorem, &lorem, %lorem and @lorem. To esacpe from function highlighting, use lorem().

Parser Mode

This is an example with activated reST additions, in this section you will find some common inline markups.

Within the *reST mode* the kernel-doc parser pass through all markups to the reST toolchain, except the *vintage highlighting* but including any whitespace. With this, the full reST markup is available in the comments.

This is a link to the Linux kernel source tree.

This description is only to show some reST inline markups like *emphasise* and **emphasis strong**. The following is a demo of a reST list markup:

Definition list

def1
    lorem

def2
    ipsum

Ordered List

- item one
- item two
- item three with a linebreak

Literal blocks

The next example shows a literal block:

```
+------+          +------+
|\     |\        /|     /|
| +----+-+      +-+----+ |
| |    | |      | |    | |
+-+----+ |      | +----+-+
 \|     \|      |/     |/
  +------+      +------+
   foo()         bar()
```

Highlighted code blocks

The next example shows a code block, with highlighting C syntax in the output.

```c
// Hello World program
#include<stdio.h>
int main()
{
   printf("Hello World");
}
```

reST sectioning

colon markup: sectioning by colon markup in reST mode is less ugly. ;-)

A kernel-doc section like *this* section is translated into a reST *subsection*. This means, you can only use the following *sub-levels* within a kernel-doc section.

a subsubsection

lorem ipsum

a paragraph

lorem ipsum

2.1.12 vintage

int **vintage**(int parameter_a, char parameter_b)

short description of this function

Parameters

- **parameter_a** (int) – first argument
- **parameter_b** (char) – second argument

Context

in_gizmo_mode().

Description

This is a test of a typical markup from *vintage* kernel-doc. Don't look to close here, it is only for testing some kernel-doc parser stuff.

Long description. This function has two integer arguments. The first is `parameter_a` and the second is `parameter_b`.

Example

```
user_function(22);
```

Return

Sum of `parameter_a` and `parameter_b`.

highlighting

- *`vintage()`* : function
- `parameter_a` : name of a parameter
- `$ENVVAR` : environmental variable
- `struct my_struct` : name of a structure (up to two words including ``struct``)
- `CONST` : name of a constant.

Parser Mode

vintage kernel-doc mode

Within the *vintage kernel-doc mode* ignores any whitespace or inline markup.

- Inline markup like *emphasis* or **emphasis strong**
- Literals and/or block indent:

a + b

In kernel-doc *vintage* mode, there are no special block or inline markups available. Markups like the one above result in ambiguous reST markup which could produce error messages in the subsequently sphinx-build process. Unexpected outputs are mostly the result.

This is a link https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/ to the Linux kernel source tree

colon markup

sectioning by colon markup in vintage mode is partial ugly. ;-)

2.1.13 struct nfp_flower_priv

struct **nfp_flower_priv**

Flower APP per-vNIC priv data

Definition

```
struct nfp_flower_priv {
    struct nfp_net *nn;
    u32 mask_id_seed;
    u64 flower_version;
    struct nfp_fl_mask_id mask_ids;
    DECLARE_HASHTABLE(mask_table, NFP_FLOWER_MASK_HASH_BITS);
    DECLARE_HASHTABLE(flow_table, NFP_FLOWER_HASH_BITS);
}
```

Members

nn
Pointer to vNIC

mask_id_seed
Seed used for mask hash table

flower_version
HW version of flower

mask_ids
List of free mask ids

mask_table
Hash table used to store masks

flow_table
Hash table used to store flower rules

2.1.14 enum foo

enum **foo**

foo

Definition

```
enum foo {
    F1,
    F2
};
```

Constants

F1
f1

F2
f2

2.1.15 struct something

struct **something**

Lorem ipsum dolor sit amet.

Definition

```
struct something {
    struct foo foofoo;
    struct bar barbar;
}
```

Members

foofoo
lorem

barbar
ipsum

2.1.16 struct lineevent_state

struct **lineevent_state**

contains the state of a userspace event

Definition

```
struct lineevent_state {
    struct gpio_device *gdev;
    const char *label;
    struct gpio_desc *desc;
    u32 eflags;
    int irq;
    wait_queue_head_t wait;
    DECLARE_KFIFO(events, struct gpioevent_data, 16);
    DECLARE_KFIFO_PTR(foobar, struct lirc_scancode);
    struct mutex read_lock;
}
```

Members

gdev
the GPIO device the event pertains to

label
consumer label used to tag descriptors

desc
the GPIO descriptor held by this event

eflags
the event flags this line was requested with

irq
the interrupt that trigger in response to events on this GPIO

wait
wait queue that handles blocking reads of events

events
KFIFO for the GPIO events (testing DECLARE_KFIFO)

foobar
testing DECLARE_KFIFO_PTR

read_lock
mutex lock to protect reads from colliding with adding new events to the FIFO

2.2 all-in-a-tumble.c

2.2.1 user_function

int **`user_function`**(int a, ...)
function that can only be called in user context

Parameters

- **`a`** (`int`) – some argument
- **`ellipsis`** (`ellipsis`) – ellipsis operator

Description

This function makes no sense, it's only a kernel-doc demonstration.

Example

```
x = user_function(22);
```

Return

Returns first argument

2.2.2 user_sum

int **user_sum**(int a, int b)

another function that can only be called in user context

Parameters

- **a** (int) – first argument
- **b** (int) – second argument

Description

This function makes no sense, it's only a kernel-doc demonstration.

Example

```
x = user_sum(1, 2);
```

Return

Returns the sum of the `a` and `b`

2.2.3 internal_function

int **internal_function**(void)

the answer

Parameters

- **void** – no arguments

Context

!sanity()

Return

The answer to the ultimate question of life, the universe and everything.

2.2.4 sys_tgkill

long **sys_tgkill**(pid_t tgid, pid_t pid, int sig)

send signal to one specific thread

Parameters

- **tgid** (pid_t) – the thread group ID of the thread
- **pid** (pid_t) – the PID of the thread
- **sig** (int) – signal to be sent

Return

This syscall also checks the `tgid` and returns -ESRCH even if the PID exists but it's not belonging to the target process anymore. This method solves the problem of threads exiting and PIDs getting reused.

2.2.5 enum rarely_enum

enum **rarely_enum**

enum to test parsing rarely code styles

Definition

```
enum rarely_enum {
    F1,
    F2
};
```

Constants

F1
f1

F2
f2

2.2.6 struct rarely_struct

struct **rarely_struct**

struct to test parsing rarely code styles

Definition

```
struct rarely_struct {
    struct foo foofoo;
    struct bar barbar;
}
```

Members

foofoo
lorem

barbar
ipsum

CHAPTER

THREE

DOC SECTIONS

For a very simple example we use this DOC section from *source all-in-a-tumble.h*:

```
/**
 * DOC: lorem ipsum
 *
 * Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor
 * incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
 * nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi
 * consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore
 * eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident,
 * sunt in culpa qui officia deserunt mollit anim id est laborum.
 */
```

To insert content with heading use:

```
.. kernel-doc::  /src/all-in-a-tumble.h
   :doc: lorem ipsum
   :module: test
```

With the module name `test` the title can be linked with:

```
Here is a link to DOC: :ref:`test.lorem-ipsum`
```

Here is a link to DOC *lorem ipsum* …

DOC section with header

3.1 lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.2 option `:no-header:`

To insert just the content, without the header use option :no-header::

```
.. kernel-doc::  /src/all-in-a-tumble.h
   :doc: lorem ipsum
   :no-header:
```

DOC section without header

Lorem ipsum dolor sit amet, consectetur adipisici elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.3 multiple DOC sections

Its always recommended to separate different DOC sections in different comments. Nevertheless, a few tests are to be carried out here with it. The DOC section tests are based on this comment:

```
/**
 * 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 it can sometimes damage hardware,
 * software, or its subject(s).
 *
 * DOC: multiple DOC sections
 *
 * It's not recommended to place more than one "DOC:" section in the same
 * comment block. To insert a new "DOC:" section, create a new comment block and
 * to create a sub-section use the reST markup for headings, see documentation
 * of function rst_mode()
 */
```

```
.. kernel-doc::  /src/all-in-a-tumble.h
   :doc: Theory of Operation
   :no-header:
```

DOC section

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.

3.3.1 foo bar splat

The only drawback to this gizmo is that it can sometimes damage hardware, software, or its subject(s).

```
.. kernel-doc::  /src/all-in-a-tumble.h
   :doc: multiple DOC sections
```

DOC section

3.3.2 multiple DOC sections

It's not recommended to place more than one "DOC:" section in the same comment block. To insert a new "DOC:" section, create a new comment block and to create a sub-section use the reST markup for headings, see documentation of function `rst_mode()`

CHAPTER FOUR

OPTION :MAN-SECT:

In the *option :export:* example, we can add a `:man-sect:  2` option, to generate man pages with the kernel-doc-man builder for all exported symbols. The usage is:

```
.. kernel-doc::  /src/all-in-a-tumble.c
   :export:  all-in-a-tumble.h
   :module: test
   :man-sect: 2
```

In the conf.py file we set man_pages and kernel_doc_mansect:

```
kernel_doc_mansect = None
man_pages = [ ]
```

To place and gzip the manuals in `dist/docs/man` Folder see kernel-doc-man Builder.

You can include the man-page as a download item in your HTML like this (relative build path is needed):

```
:download:`user_function.2.gz   <../../dist/docs/man/user_function.2.gz>`
```

Or just set a link to the man page file (relative HTML URL is needed)

```
hyperlink to: `user_function.2.gz <../man/user_function.2.gz>`_
```

To view a (downloaded) man-page use:

```
$ man ~/Downloads/user_function.2.gz
```

CHAPTER

FIVE

EXPORTED SYMBOLS

5.1 option `:export:`

In the *source all-in-a-tumble.h* header file we export:

```
EXPORT_SYMBOL_GPL_FUTURE(user_function)

int user_function(int a, ...)
```

The documentation of the exported symbols is in *source all-in-a-tumble.c*. To gather exports from *source all-in-a-tumble.h* and *source all-in-a-tumble.c* and parses comments from *source all-in-a-tumble.c* use kernel-doc options:

```
.. kernel-doc::  /src/all-in-a-tumble.c
   :export:  /src/all-in-a-tumble.h
   :module: test
```

---

exported symbols

5.1.1 user_function

int **user_function**(int a, ...)

function that can only be called in user context

Parameters

- **a** (int) – some argument
- **ellipsis** (ellipsis) – ellipsis operator

Description

This function makes no sense, it’s only a kernel-doc demonstration.

Example

```
x = user_function(22);
```

Return

Returns first argument

5.2 options `:export:`, `:exp-method:`, `:exp-ids:`

This test gathers function from *source all-in-a-tumble.c* whose function attributes mark them as exported:

```
/**
 * user_sum() - another function that can only be called in user context
 * @a: first argument
 * @b: second argument
 *
 * This function makes no sense, it's only a kernel-doc demonstration.
 *
 * Example:
 * x = user_sum(1, 2);
 *
 * Return:
 * Returns the sum of the @a and @b
 */
API_EXPORTED
int user_sum(int a, int b)
{
        return a + b;
}
```

and that are present in *source all-in-a-tumble.h*:

```
int user_sum(int a, int b);
```

To insert the documentation use:

```
.. kernel-doc::  /src/all-in-a-tumble.c
   :export:  /src/all-in-a-tumble.h
   :exp-method: attribute
   :exp-ids: API_EXPORTED
   :module: test_fnattrs
```

The `exp-method` and `exp-ids` could be respectively omitted if `kernel_doc_exp_method` and `kernel_doc_exp_ids` are set in the sphinx configuration.

exported symbols

5.2.1 user_sum

int **user_sum**(int a, int b)

another function that can only be called in user context

Parameters

- **a** (int) – first argument
- **b** (int) – second argument

Description

This function makes no sense, it's only a kernel-doc demonstration.

Example

```
x = user_sum(1, 2);
```

Return

Returns the sum of the a and b

5.3 option :internal:

Include documentation for all documented definitions, **not** exported. This test gathers exports from *source all-in-a-tumble.h* and *source all-in-a-tumble.c* and parses comments from *source all-in-a-tumble.c*, from where only the *not exported* definitions are used in the reST output:

```
.. kernel-doc::  /src/all-in-a-tumble.c
   :internal:  all-in-a-tumble.h
   :module: test_internal
```

The example also shows, that mixing different values for

- `:exp-method: -> [macro|attribute]` and
- `:exp-ids: -> [EXPORT_SYMBOL|API_EXPORTED]`

in one source file is not well supported:

internal symbols

5.3.1 user_sum

int **user_sum**(int a, int b)

another function that can only be called in user context

Parameters

- **a** (int) – first argument
- **b** (int) – second argument

Description

This function makes no sense, it's only a kernel-doc demonstration.

Example

```
x = user_sum(1, 2);
```

Return

Returns the sum of the a and b

5.3.2 internal_function

int **internal_function**(void)

the answer

Parameters

- **void** – no arguments

Context

!sanity()

Return

The answer to the ultimate question of life, the universe and everything.

5.3.3 sys_tgkill

long **sys_tgkill**(pid_t tgid, pid_t pid, int sig)

send signal to one specific thread

Parameters

- **tgid** (pid_t) – the thread group ID of the thread
- **pid** (pid_t) – the PID of the thread
- **sig** (int) – signal to be sent

Return

This syscall also checks the `tgid` and returns -ESRCH even if the PID exists but it's not belonging to the target process anymore. This method solves the problem of threads exiting and PIDs getting reused.

5.3.4 enum rarely_enum

enum **rarely_enum**

enum to test parsing rarely code styles

Definition

```
enum rarely_enum {
    F1,
    F2
};
```

Constants

F1
f1

F2
f2

5.3.5 struct rarely_struct

struct **rarely_struct**

struct to test parsing rarely code styles

Definition

```
struct rarely_struct {
    struct foo foofoo;
    struct bar barbar;
}
```

Members

foofoo
lorem

barbar
ipsum

CHAPTER

SIX

SYSCALL MACRO

In the Kernel's source is a macro: SYSCALL_DEFINEn(). By example:

```
/**
 * sys_tgkill - send signal to one specific thread
 * @tgid: the thread group ID of the thread
 * @pid: the PID of the thread
 * @sig: signal to be sent
 *
 * Return:
 *
 * This syscall also checks the @tgid and returns -ESRCH even if the PID
 * exists but it's not belonging to the target process anymore. This
 * method solves the problem of threads exiting and PIDs getting reused.
 */
SYSCALL_DEFINE3(tgkill, pid_t, tgid, pid_t, pid, int, sig)
{
        ...
}
```

```
.. kernel-doc::  /src/all-in-a-tumble.c
   :symbols:  sys_tgkill
```

---

missing exports

6.1 sys_tgkill

long **sys_tgkill**(pid_t tgid, pid_t pid, int sig)

send signal to one specific thread

Parameters

- **tgid** (pid_t) – the thread group ID of the thread
- **pid** (pid_t) – the PID of the thread
- **sig** (int) – signal to be sent

6.1.1 Return

This syscall also checks the `tgid` and returns -ESRCH even if the PID exists but it's not belonging to the target process anymore. This method solves the problem of threads exiting and PIDs getting reused.

INDEX