
kernel-doc tests

Release 20230630

John Doe

Jun 30, 2023

C SOURCES

1	source of all-in-a-tumble. [ch]	3
1.1	source all-in-a-tumble.h	3
1.2	source all-in-a-tumble.c	12
2	rendered all-in-a-tumble. [ch]	15
2.1	all-in-a-tumble.h	16
2.2	all-in-a-tumble.c	26
3	DOC sections	31
3.1	lorem ipsum	31
3.2	option :no-header:	32
3.3	multiple DOC sections	32
4	option :man-sect:	35
5	exported symbols	37
5.1	option :export:	37
5.2	options :export:, :exp-method:, :exp-ids:	38
5.3	option :internal:	39
6	SYSCALL macro	43
6.1	sys_tgkill	43
	Index	45

Within 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*.

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

```
1 /* parse-markup: reST */
2
3 /**
4  * DOC: About Examples
5  *
6  * The files :ref:`all-in-a-tumble.c-src` and :ref:`all-in-a-tumble.h-src` are
7  * including all examples of the :ref:`linuxdoc-howto` documentation. These
8  * files are also used as a test of the kernel-doc parser, to see how kernel-doc
9  * content will be rendered and where the parser might fail.
10 *
11 * And ... The content itself is nonsense / don't look to close ;-)
12 */
13
14 // testing:
15 //
16 // .. kernel-doc:: ./all-in-a-tumble.c
17 //      :export: ./all-in-a-tumble.h
18
19 /* parse-SNIP: EXPORT_SYMBOL */
20 EXPORT_SYMBOL_GPL_FUTURE(user_function)
21
22 int user_function(int a, ...)
23 /* parse-SNAP: */
24
```

(continues on next page)

(continued from previous page)

```

25 /* parse-SNIP: user_sum-h */
26 int user_sum(int a, int b);
27 /* parse-SNAP: */
28
29
30 /**
31  * block_touch_buffer - mark a buffer accessed
32  * @bh: buffer_head being touched
33  *
34  * Called from touch_buffer().
35  */
36 DEFINE_EVENT(block_buffer, block_touch_buffer,
37
38             TP_PROTO(struct buffer_head *bh),
39
40             TP_ARGS(bh)
41 );
42
43 /**
44  * block_dirty_buffer - mark a buffer dirty
45  * @bh: buffer_head being dirtied
46  *
47  * Called from mark_buffer_dirty().
48  */
49 DEFINE_EVENT(block_buffer, block_dirty_buffer,
50
51             TP_PROTO(struct buffer_head *bh),
52
53             TP_ARGS(bh)
54 );
55
56 // The parse-SNIP/SNAP comments are used to include the C source code as snippets
57 // into a reST document. These are the examples of the kernel-doc-HOWTO book.
58
59 /* parse-SNIP: theory-of-operation */
60 /**
61  * DOC: Theory of Operation
62  *
63  * The whizbang foobar is a dilly of a gizmo. It can do whatever you
64  * want it to do, at any time. It reads your mind. Here's how it works.
65  *
66  * foo bar splat
67  * -----
68  *
69  * The only drawback to this gizmo is that it can sometimes damage hardware,
70  * software, or its subject(s).
71  *
72  * DOC: multiple DOC sections
73  *
74  * It's not recommended to place more than one "DOC:" section in the same
75  * comment block. To insert a new "DOC:" section, create a new comment block and
76  * to create a sub-section use the reST markup for headings, see documentation

```

(continues on next page)

(continued from previous page)

```

77  * of function rst_mode()
78  */
79  /* parse-SNAP: */
80
81  /* parse-SNIP: lorem */
82  /**
83   * DOC: lorem ipsum
84   *
85   * Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed eiusmod tempor
86   * incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
87   * nostrud exercitation ullamco laboris nisi ut aliquid ex ea commodi
88   * consequat. Quis aute iure reprehenderit in voluptate velit esse cillum dolore
89   * eu fugiat nulla pariatur. Excepteur sint obcaecat cupiditat non proident,
90   * sunt in culpa qui officia deserunt mollit anim id est laborum.
91   */
92  /* parse-SNAP: */
93
94
95  /* parse-SNIP: my_long_struct */
96  /**
97   * struct my_long_struct - short description with &my_struct->a and &my_struct->b
98   * @foo: The Foo member.
99   *
100  * Longer description
101  */
102  struct my_long_struct {
103      int foo;
104      /**
105       * @bar: The Bar member.
106       */
107      int bar;
108      /**
109       * @baz: The Baz member.
110       *
111       * Here, the member description may contain several paragraphs.
112       */
113      int baz;
114      union {
115          /** @foobar: Single line description. */
116          int foobar;
117      };
118      /** @bar2: Description for struct @bar2 inside @my_long_struct */
119      struct {
120          /**
121           * @bar2.barbar: Description for @barbar inside @my_long_struct.bar2
122           */
123          int barbar;
124      } bar2;
125  };
126  /* parse-SNAP: */
127
128

```

(continues on next page)

(continued from previous page)

```
129 /* parse-SNIP: my_union */
130 /**
131  * union my_union - short description
132  * @a: first member
133  * @b: second member
134  *
135  * Longer description
136  */
137 union my_union {
138     int a;
139     int b;
140 };
141 /* parse-SNAP: */
142
143
144 /* parse-SNIP: my_enum */
145 /**
146  * enum my_enum - log level
147  * @QUIET: logs nothing
148  * @INFO: logs info messages
149  * @WARN: logs warn and info messages
150  * @DEBUG: logs debug, warn and info messages
151  */
152
153 enum my_enum {
154     QUIET,
155     INFO,
156     WARN,
157     DEBUG
158 };
159 /* parse-SNAP: */
160
161
162 /* parse-SNIP: my_typedef */
163 /**
164  * typedef my_typedef - useless typedef of int
165  *
166  */
167 typedef int my_typedef;
168 /* parse-SNAP: */
169
170
171 /* parse-SNIP: rst_mode */
172 /**
173  * rst_mode - dummy to demonstrate reST & kernel-doc markup in comments
174  * @a: first argument
175  * @b: second argument
176  * Context: :c:func:`in_gizmo_mode`.
177  *
178  * Long description. This function has two integer arguments. The first is
179  * ``parameter_a`` and the second is ``parameter_b``.
180  *
```

(continues on next page)

(continued from previous page)

```

181 * As long as the reST / sphinx-doc toolchain uses `intersphinx
182 * <http://www.sphinx-doc.org/en/stable/ext/intersphinx.html>`__ you can refer
183 * definitions *outside* like :c:type:`struct media_device <media_device>`. If
184 * the description of ``media_device`` struct is found in any of the intersphinx
185 * locations, a hyperref to this target is generated a build time.
186 *
187 * Example:
188 *     int main() {
189 *         printf("Hello World\n");
190 *         return 0;
191 *     }
192 *
193 * Return: Sum of ``parameter_a`` and the second is ``parameter_b``.
194 *
195 * highlighting:
196 * The highlight pattern, are non regular reST markups. They are only available
197 * within kernel-doc comments, helping C developers to write short and compact
198 * documentation.
199 *
200 * - user_function() : function
201 * - @a : name of a parameter
202 * - &struct my_struct : name of a structure (including the word struct)
203 * - &union my_union : name of a union
204 * - &my_struct->a or &my_struct.b - member of a struct or union.
205 * - &enum my_enum : name of a enum
206 * - &typedef my_typedef : name of a typedef
207 * - %CONST : name of a constant.
208 * - $ENVVAR : environmental variable
209 *
210 * The kernel-doc parser translates the pattern above to the corresponding reST
211 * markups. You don't have to use the *highlight* pattern, if you prefer *pure*
212 * reST, use the reST markup.
213 *
214 * - :c:func:`user_function` : function
215 * - ``a`` : name of a parameter
216 * - :c:type:`struct my_struct <my_struct>` : name of a structure (including the word_
↳ struct)
217 * - :c:type:`union my_union <my_union>` : name of a union
218 * - :c:type:`my_struct->a <my_struct>` or :c:type:`my_struct.b <my_struct>` - member of_
↳ a struct or union.
219 * - :c:type:`enum my_enum <my_enum>` : name of a enum
220 * - :c:type:`typedef my_typedef <my_typedef>` : name of a typedef
221 * - ``CONST`` : name of a constant.
222 * - ``$ENVVAR`` : environmental variable
223 *
224 * Since the prefixes ``$...``, ``&...`` and ``@...`` are used to markup the
225 * highlight pattern, you have to escape them in other uses: \$lorem, &lorem,
226 * %lorem and @lorem. To escape from function highlighting, use lorem\().
227 *
228 * Parser Mode:
229 * This is an example with activated reST additions, in this section you will
230 * find some common inline markups.

```

(continues on next page)

(continued from previous page)

```

231 *
232 * Within the reST mode the kernel-doc parser pass through all markups to the
233 * reST toolchain, except the vintage highlighting but including any
234 * whitespace. With this, the full reST markup is available in the comments.
235 *
236 * This is a link to the \Linux kernel source tree
237 * <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/>`_.
238 *
239 * This description is only to show some reST inline markups like emphasise
240 * and emphasis strong. The following is a demo of a reST list markup:
241 *
242 * Definition list:
243 * :def1: lorem
244 * :def2: ipsum
245 *
246 * Ordered List:
247 * - item one
248 * - item two
249 * - item three with
250 *   a linebreak
251 *
252 * Literal blocks:
253 * The next example shows a literal block::
254 *
255 *      +-----+      +-----+
256 *      |\      |\      /\      /\
257 *      | +-----+  +-----+ |
258 *      | |      | |      | |      | |
259 *      +-----+ |      | +-----+
260 *      \|      \|      /\      /\
261 *      +-----+      +-----+
262 *      foo()          bar()
263 *
264 * Highlighted code blocks:
265 * The next example shows a code block, with highlighting C syntax in the
266 * output.
267 *
268 * .. code-block:: c
269 *
270 *     // Hello World program
271 *     #include<stdio.h>
272 *     int main()
273 *     {
274 *         printf("Hello World");
275 *     }
276 *
277 *
278 * reST sectioning:
279 *
280 * colon markup: sectioning by colon markup in reST mode is less ugly. ;- )
281 *
282 * A kernel-doc section like this section is translated into a reST

```

(continues on next page)

(continued from previous page)

```

283  * *subsection*. This means, you can only use the following *sub-levels* within a
284  * kernel-doc section.
285  *
286  * a subsection
287  * ^^^^^^^^^^^^^^^^^^^^
288  *
289  * lorem ipsum
290  *
291  * a paragraph
292  * ^^^^^^^^^^^^
293  *
294  * lorem ipsum
295  *
296  */
297  int rst_mode(int a, char *b)
298  {
299      return a + b;
300  }
301  /* parse-SNAP: */
302
303
304  /* parse-markup: kernel-doc */
305
306  /**
307   * vintage - short description of this function
308   * @parameter_a: first argument
309   * @parameter_b: second argument
310   * Context: in_gizmo_mode().
311   *
312   * This is a test of a typical markup from *vintage* kernel-doc. Don't look to
313   * close here, it is only for testing some kernel-doc parser stuff.
314   *
315   * Long description. This function has two integer arguments. The first is
316   * @parameter_a and the second is @parameter_b.
317   *
318   * Example: user_function(22);
319   *
320   * Return: Sum of @parameter_a and @parameter_b.
321   *
322   * highlighting:
323   *
324   * - vintage() : function
325   * - @parameter_a : name of a parameter
326   * - $ENVVAR : environmental variable
327   * - &my_struct : name of a structure (up to two words including ``struct``)
328   * - %CONST : name of a constant.
329   *
330   * Parser Mode: *vintage* kernel-doc mode
331   *
332   * Within the *vintage kernel-doc mode* ignores any whitespace or inline
333   * markup.
334   *

```

(continues on next page)

(continued from previous page)

```

335 * - Inline markup like *emphasis* or **emphasis strong**
336 * - Literals and/or block indent:
337 *
338 *     a + b
339 *
340 * In kernel-doc *vintage* mode, there are no special block or inline markups
341 * available. Markups like the one above result in ambiguous reST markup which
342 * could produce error messages in the subsequently sphinx-build
343 * process. Unexpected outputs are mostly the result.
344 *
345 * This is a link https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/
346 * to the Linux kernel source tree
347 *
348 * colon markup: sectioning by colon markup in vintage mode is partial ugly. ;-)
349 *
350 */
351 int vintage(int parameter_a, char parameter_b)
352 {
353     return a + b;
354 }
355
356 /* some C&P for extended tests
357 */
358
359 /**
360 * struct nfp_flower_priv - Flower APP per-vNIC priv data
361 * @nn:                Pointer to vNIC
362 * @mask_id_seed:      Seed used for mask hash table
363 * @flower_version:    HW version of flower
364 * @mask_ids:          List of free mask ids
365 * @mask_table:        Hash table used to store masks
366 * @flow_table:        Hash table used to store flower rules
367 */
368 struct nfp_flower_priv {
369     struct nfp_net *nn;
370     u32 mask_id_seed;
371     u64 flower_version;
372     struct nfp_fl_mask_id mask_ids;
373     DECLARE_HASHTABLE(mask_table, NFP_FLOWER_MASK_HASH_BITS);
374     DECLARE_HASHTABLE(flow_table, NFP_FLOWER_HASH_BITS);
375 };
376
377 /**
378 * enum foo - foo
379 * @F1: f1
380 * @F2: f2
381 */
382 enum foo {
383     F1,
384
385     F2,
386 };

```

(continues on next page)

(continued from previous page)

```
387
388
389 /**
390  * struct something - Lorem ipsum dolor sit amet.
391  * @foofoo: lorem
392  * @barbar: ipsum
393  */
394
395 struct something {
396     struct foo
397
398     foofoo;
399
400     struct bar
401
402     barbar;
403 };
404
405 /**
406  * struct lineevent_state - contains the state of a userspace event
407  * @gdev: the GPIO device the event pertains to
408  * @label: consumer label used to tag descriptors
409  * @desc: the GPIO descriptor held by this event
410  * @eflags: the event flags this line was requested with
411  * @irq: the interrupt that trigger in response to events on this GPIO
412  * @wait: wait queue that handles blocking reads of events
413  * @events: Kfifo for the GPIO events (testing DECLARE_KFIFO)
414  * @foobar: testing DECLARE_KFIFO_PTR
415  * @read_lock: mutex lock to protect reads from colliding with adding
416  * new events to the FIFO
417  */
418 struct lineevent_state {
419     struct gpio_device *gdev;
420     const char *label;
421     struct gpio_desc *desc;
422     u32 eflags;
423     int irq;
424     wait_queue_head_t wait;
425     DECLARE_KFIFO(events, struct gpioevent_data, 16);
426     DECLARE_KFIFO_PTR(foobar, struct lirc_scancode);
427     struct mutex read_lock;
428 };
```

1.2 source all-in-a-tumble.c

```
1 // this test some kernel-doc stuff
2
3 /* parse-SNIP: hello-world */
4 #include<stdio.h>
5 int main() {
6     printf("Hello World\n");
7     return 0;
8 }
9 /* parse-SNAP: */
10
11 /* parse-SNIP: user_function */
12 /**
13  * user_function() - function that can only be called in user context
14  * @a: some argument
15  * @...: ellipsis operator
16  *
17  * This function makes no sense, it's only a kernel-doc demonstration.
18  *
19  * Example:
20  * x = user_function(22);
21  *
22  * Return:
23  * Returns first argument
24  */
25 int
26 user_function(int a, ...)
27 {
28     return a;
29 }
30 /* parse-SNAP: */
31
32
33 /* parse-SNIP: user_sum-c */
34 /**
35  * user_sum() - another function that can only be called in user context
36  * @a: first argument
37  * @b: second argument
38  *
39  * This function makes no sense, it's only a kernel-doc demonstration.
40  *
41  * Example:
42  * x = user_sum(1, 2);
43  *
44  * Return:
45  * Returns the sum of the @a and @b
46  */
47 API_EXPORTED
48 int user_sum(int a, int b)
49 {
50     return a + b;
```

(continues on next page)

(continued from previous page)

```

51 }
52 /* parse-SNAP: */
53
54 /* parse-SNIP: internal_function */
55 /**
56  * internal_function - the answer
57  *
58  * Context: !sanity()
59  *
60  * Return:
61  * The answer to the ultimate question of life, the universe and everything.
62  */
63 int internal_function()
64 {
65     return 42;
66 }
67 /* parse-SNAP: */
68
69 /* parse-SNIP: test_SYSCALL */
70 /**
71  * sys_tgkill - send signal to one specific thread
72  * @tgid: the thread group ID of the thread
73  * @pid: the PID of the thread
74  * @sig: signal to be sent
75  *
76  * Return:
77  *
78  * This syscall also checks the @tgid and returns -ESRCH even if the PID
79  * exists but it's not belonging to the target process anymore. This
80  * method solves the problem of threads exiting and PIDs getting reused.
81  */
82 SYSCALL_DEFINE3(tgkill, pid_t, tgid, pid_t, pid, int, sig)
83 {
84     ...
85 }
86
87 /* parse-SNAP: */
88
89 /* parse-SNIP: rarely_code_styles*/
90 /**
91  * enum rarely_enum - enum to test parsing rarely code styles
92  * @F1: f1
93  * @F2: f2
94  */
95 enum rarely_enum {
96     F1,
97
98     F2,
99 };
100
101
102 /**

```

(continues on next page)

(continued from previous page)

```
103 * struct rarely_struct - struct to test parsing rarely code styles
104 * @foofoo: lorem
105 * @barbar: ipsum
106 */
107
108 struct rarely_struct {
109     struct foo
110
111     foofoo;
112
113     struct bar
114
115     barbar;
116 };
117
118
```

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`.

- *all-in-a-tumble.h*
 - *About Examples*
 - *trace_block_touch_buffer*
 - *trace_block_dirty_buffer*
 - *Theory of Operation*
 - *multiple DOC sections*
 - *lorem ipsum*
 - *struct my_long_struct*
 - *union my_union*
 - *enum my_enum*
 - *typedef my_typedef*
 - *rst_mode*
 - *vintage*
 - *struct nfp_flower_priv*
 - *enum foo*
 - *struct something*
 - *struct lineevent_state*
- *all-in-a-tumble.c*
 - *user_function*

```
- user_sum
- internal_function
- sys_tgkill
- enum rarely_enum
- struct rarely_struct
```

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 adipisicing elit, sed eiusmod tempor incididunt 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 typedef 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 escape 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.

```
// 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 subsection

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

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 adipisicing elit, sed eiusmod tempor
 * incididunt 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 adipisicing elit, sed eiusmod tempor incididunt 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 adipisicing elit, sed eiusmod tempor incididunt 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()`

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
```


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

SYSCALL MACRO

In the Kernel's source is a macro: `SYSCALL_DEFINE()`. 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.

E

example.foo (*C enum*), 24
example.internal_function (*C function*), 27
example.lineevent_state (*C struct*), 25
example.my_enum (*C enum*), 19
example.my_long_struct (*C struct*), 17
example.my_typedef (*C type*), 19
example.my_union (*C union*), 18
example.nfp_flower_priv (*C struct*), 24
example.rarely_enum (*C enum*), 28
example.rarely_struct (*C struct*), 29
example.rst_mode (*C function*), 19
example.something (*C struct*), 25
example.sys_tgkill (*C function*), 28
example.trace_block_dirty_buffer (*C function*),
16
example.trace_block_touch_buffer (*C function*),
16
example.user_function (*C function*), 26
example.user_sum (*C function*), 27
example.vintage (*C function*), 22

S

sys_tgkill (*C function*), 43

T

test.user_function (*C function*), 37
test_fnattrs.user_sum (*C function*), 39
test_internal.internal_function (*C function*), 40
test_internal.rarely_enum (*C enum*), 41
test_internal.rarely_struct (*C struct*), 41
test_internal.sys_tgkill (*C function*), 40
test_internal.user_sum (*C function*), 40