# Contributor's Guidelines

***Release 2.1.0***

August 17, 2015

CHAPTER

ONE

# DPDK CODING STYLE

## 1.1 Description

This document specifies the preferred style for source files in the DPDK source tree. It is based on the Linux Kernel coding guidelines and the FreeBSD 7.2 Kernel Developer's Manual (see man style(9)), but was heavily modified for the needs of the DPDK.

## 1.2 General Guidelines

The rules and guidelines given in this document cannot cover every situation, so the following general guidelines should be used as a fallback:

- The code style should be consistent within each individual file.
- In the case of creating new files, the style should be consistent within each file in a given directory or module.
- The primary reason for coding standards is to increase code readability and comprehensibility, therefore always use whatever option will make the code easiest to read.

Line length is recommended to be not more than 80 characters, including comments. [Tab stop size should be assumed to be 8-characters wide].

---

**Note:** The above is recommendation, and not a hard limit. However, it is expected that the recommendations should be followed in all but the rarest situations.

---

## 1.3 C Comment Style

### 1.3.1 Usual Comments

These comments should be used in normal cases. To document a public API, a doxygen-like format must be used: refer to *Doxygen Guidelines*.

```
/*
 * VERY important single-line comments look like this.
 */

/* Most single-line comments look like this. */

/*
```

```
 * Multi-line comments look like this.  Make them real sentences. Fill
 * them so they look like real paragraphs.
 */
```

### 1.3.2 License Header

Each file should begin with a special comment containing the appropriate copyright and license for the file. Generally this is the BSD License, except for code for Linux Kernel modules. After any copyright header, a blank line should be left before any other contents, e.g. include statements in a C file.

## 1.4 C Preprocessor Directives

### 1.4.1 Header Includes

In DPDK sources, the include files should be ordered as following:

1. libc includes (system includes first)
2. DPDK EAL includes
3. DPDK misc libraries includes
4. application-specific includes

Include files from the local application directory are included using quotes, while includes from other paths are included using angle brackets: "<>".

Example:

```
#include <stdio.h>
#include <stdlib.h>

#include <rte_eal.h>

#include <rte_ring.h>
#include <rte_mempool.h>

#include "application.h"
```

### 1.4.2 Header File Guards

Headers should be protected against multiple inclusion with the usual:

```
#ifndef _FILE_H_
#define _FILE_H_

/* Code */

#endif /* _FILE_H_ */
```

### 1.4.3 Macros

Do not `#define` or declare names except with the standard DPDK prefix: RTE_. This is to ensure there are no collisions with definitions in the application itself.

The names of "unsafe" macros (ones that have side effects), and the names of macros for manifest constants, are all in uppercase.

The expansions of expression-like macros are either a single token or have outer parentheses. If a macro is an inline expansion of a function, the function name is all in lowercase and the macro has the same name all in uppercase. If the macro encapsulates a compound statement, enclose it in a do-while loop, so that it can be used safely in if statements. Any final statement-terminating semicolon should be supplied by the macro invocation rather than the macro, to make parsing easier for pretty-printers and editors.

For example:

```
#define MACRO(x, y) do {                                        \
        variable = (x) + (y);                                   \
        (y) += 2;                                               \
} while(0)
```

---

**Note:** Wherever possible, enums and inline functions should be preferred to macros, since they provide additional degrees of type-safety and can allow compilers to emit extra warnings about unsafe code.

---

### 1.4.4 Conditional Compilation

- When code is conditionally compiled using `#ifdef` or `#if`, a comment may be added following the matching `#endif` or `#else` to permit the reader to easily discern where conditionally compiled code regions end.
- This comment should be used only for (subjectively) long regions, regions greater than 20 lines, or where a series of nested `#ifdef`'s may be confusing to the reader. Exceptions may be made for cases where code is conditionally not compiled for the purposes of lint(1), or other tools, even though the uncompiled region may be small.
- The comment should be separated from the `#endif` or `#else` by a single space.
- For short conditionally compiled regions, a closing comment should not be used.
- The comment for `#endif` should match the expression used in the corresponding `#if` or `#ifdef`.
- The comment for `#else` and `#elif` should match the inverse of the expression(s) used in the preceding `#if` and/or `#elif` statements.
- In the comments, the subexpression `defined(FOO)` is abbreviated as "FOO". For the purposes of comments, `#ifndef FOO` is treated as `#if !defined(FOO)`.

```
#ifdef KTRACE
#include <sys/ktrace.h>
#endif

#ifdef COMPAT_43
/* A large region here, or other conditional code. */
#else /* !COMPAT_43 */
/* Or here. */
#endif /* COMPAT_43 */

#ifndef COMPAT_43
/* Yet another large region here, or other conditional code. */
#else /* COMPAT_43 */
```

```
/* Or here. */
#endif /* !COMPAT_43 */
```

**Note:** Conditional compilation should be used only when absolutely necessary, as it increases the number of target binaries that need to be built and tested.

## 1.5 C Types

### 1.5.1 Integers

For fixed/minimum-size integer values, the project uses the form uintXX_t (from stdint.h) instead of older BSD-style integer identifiers of the form u_intXX_t.

### 1.5.2 Enumerations

- Enumeration values are all uppercase.

```
enum enumtype { ONE, TWO } et;
```

- Enum types should be used in preference to macros #defining a set of (sequential) values.
- Enum types should be prefixed with `rte_` and the elements by a suitable prefix [generally starting RTE_<enum>_ - where <enum> is a shortname for the enum type] to avoid namespace collisions.

### 1.5.3 Bitfields

The developer should group bitfields that are included in the same integer, as follows:

```
struct grehdr {
  uint16_t rec:3,
       srr:1,
       seq:1,
       key:1,
       routing:1,
       csum:1,
       version:3,
       reserved:4,
       ack:1;
/* ... */
}
```

### 1.5.4 Variable Declarations

In declarations, do not put any whitespace between asterisks and adjacent tokens, except for tokens that are identifiers related to types. (These identifiers are the names of basic types, type qualifiers, and typedef-names other than the one being declared.) Separate these identifiers from asterisks using a single space.

For example:

```
int *x;         /* no space after asterisk */
int * const x;  /* space after asterisk when using a type qualifier */
```

- All externally-visible variables should have an `rte_` prefix in the name to avoid namespace collisions.
- Do not use uppercase letters - either in the form of ALL_UPPERCASE, or CamelCase - in variable names. Lower-case letters and underscores only.

### 1.5.5 Structure Declarations

- In general, when declaring variables in new structures, declare them sorted by use, then by size (largest to smallest), and then in alphabetical order. Sorting by use means that commonly used variables are used together and that the structure layout makes logical sense. Ordering by size then ensures that as little padding is added to the structure as possible.
- For existing structures, additions to structures should be added to the end so for backward compatibility reasons.
- Each structure element gets its own line.
- Try to make the structure readable by aligning the member names using spaces as shown below.
- Names following extremely long types, which therefore cannot be easily aligned with the rest, should be separated by a single space.

```
struct foo {
        struct foo      *next;          /* List of active foo. */
        struct mumble   amumble;        /* Comment for mumble. */
        int             bar;            /* Try to align the comments. */
        struct verylongtypename *baz;   /* Won't fit with other members */
};
```

- Major structures should be declared at the top of the file in which they are used, or in separate header files if they are used in multiple source files.
- Use of the structures should be by separate variable declarations and those declarations must be extern if they are declared in a header file.
- Externally visible structure definitions should have the structure name prefixed by `rte_` to avoid namespace collisions.

### 1.5.6 Queues

Use queue(3) macros rather than rolling your own lists, whenever possible. Thus, the previous example would be better written:

```
#include <sys/queue.h>

struct foo {
        LIST_ENTRY(foo) link;      /* Use queue macros for foo lists. */
        struct mumble   amumble;   /* Comment for mumble. */
        int             bar;       /* Try to align the comments. */
        struct verylongtypename *baz;   /* Won't fit with other members */
};
LIST_HEAD(, foo) foohead;          /* Head of global foo list. */
```

DPDK also provides an optimized way to store elements in lockless rings. This should be used in all data-path code, when there are several consumer and/or producers to avoid locking for concurrent access.

### 1.5.7 Typedefs

Avoid using typedefs for structure types.

For example, use:

```
struct my_struct_type {
/* ... */
};

struct my_struct_type my_var;
```

rather than:

```
typedef struct my_struct_type {
/* ... */
} my_struct_type;

my_struct_type my_var
```

Typedefs are problematic because they do not properly hide their underlying type; for example, you need to know if the typedef is the structure itself, as shown above, or a pointer to the structure. In addition, they must be declared exactly once, whereas an incomplete structure type can be mentioned as many times as necessary. Typedefs are difficult to use in stand-alone header files. The header that defines the typedef must be included before the header that uses it, or by the header that uses it (which causes namespace pollution), or there must be a back-door mechanism for obtaining the typedef.

Note that #defines used instead of typedefs also are problematic (since they do not propagate the pointer type correctly due to direct text replacement). For example, `#define pint int *` does not work as expected, while `typedef int *pint` does work. As stated when discussing macros, typedefs should be preferred to macros in cases like this.

When convention requires a typedef; make its name match the struct tag. Avoid typedefs ending in `_t`, except as specified in Standard C or by POSIX.

---

**Note:** It is recommended to use typedefs to define function pointer types, for reasons of code readability. This is especially true when the function type is used as a parameter to another function.

---

For example:

```
/**
 * Definition of a remote launch function.
 */
typedef int (lcore_function_t)(void *);

/* launch a function of lcore_function_t type */
int rte_eal_remote_launch(lcore_function_t *f, void *arg, unsigned slave_id);
```

## 1.6 C Indentation

### 1.6.1 General

- Indentation is a hard tab, that is, a tab character, not a sequence of spaces,

---

**Note:** Global whitespace rule in DPDK, use tabs for indentation, spaces for alignment.

---

- Do not put any spaces before a tab for indentation.
- If you have to wrap a long statement, put the operator at the end of the line, and indent again.
- For control statements (if, while, etc.), continuation it is recommended that the next line be indented by two tabs, rather than one, to prevent confusion as to whether the second line of the control statement forms part of the statement body or not. Alternatively, the line continuation may use additional spaces to line up to an appropriately point on the preceding line, for example, to align to an opening brace.

---

**Note:** As with all style guidelines, code should match style already in use in an existing file.

---

```
while (really_long_variable_name_1 == really_long_variable_name_2 &&
    var3 == var4){  /* confusing to read as */
    x = y + z;      /* control stmt body lines up with second line of */
    a = b + c;      /* control statement itself if single indent used */
}

if (really_long_variable_name_1 == really_long_variable_name_2 &&
        var3 == var4){  /* two tabs used */
    x = y + z;          /* statement body no longer lines up */
    a = b + c;
}

z = a + really + long + statement + that + needs +
        two + lines + gets + indented + on + the +
        second + and + subsequent + lines;
```

- Do not add whitespace at the end of a line.
- Do not add whitespace or a blank line at the end of a file.

### 1.6.2 Control Statements and Loops

- Include a space after keywords (if, while, for, return, switch).
- Do not use braces ({ and }) for control statements with zero or just a single statement, unless that statement is more than a single line in which case the braces are permitted.

```
for (p = buf; *p != '\0'; ++p)
        ;       /* nothing */
for (;;)
        stmt;
for (;;) {
        z = a + really + long + statement + that + needs +
                two + lines + gets + indented + on + the +
                second + and + subsequent + lines;
}
for (;;) {
```

```
        if (cond)
                stmt;
}
if (val != NULL)
        val = realloc(val, newsize);
```

- Parts of a for loop may be left empty.

```
for (; cnt < 15; cnt++) {
        stmt1;
        stmt2;
}
```

- Closing and opening braces go on the same line as the else keyword.
- Braces that are not necessary should be left out.

```
if (test)
        stmt;
else if (bar) {
        stmt;
        stmt;
} else
        stmt;
```

### 1.6.3 Function Calls

- Do not use spaces after function names.
- Commas should have a space after them.
- No spaces after ( or [ or preceding the ] or ) characters.

```
error = function(a1, a2);
if (error != 0)
        exit(error);
```

### 1.6.4 Operators

- Unary operators do not require spaces, binary operators do.
- Do not use parentheses unless they are required for precedence or unless the statement is confusing without them. However, remember that other people may be more easily confused than you.

### 1.6.5 Exit

Exits should be 0 on success, or 1 on failure.

```
        exit(0);        /*
                         * Avoid obvious comments such as
                         * "Exit 0 on success."
                         */
}
```

### 1.6.6 Local Variables

- Variables should be declared at the start of a block of code rather than in the middle. The exception to this is when the variable is `const` in which case the declaration must be at the point of first use/assignment.
- When declaring variables in functions, multiple variables per line are OK. However, if multiple declarations would cause the line to exceed a reasonable line length, begin a new set of declarations on the next line rather than using a line continuation.
- Be careful to not obfuscate the code by initializing variables in the declarations, only the last variable on a line should be initialized. If multiple variables are to be initialised when defined, put one per line.
- Do not use function calls in initializers, except for `const` variables.

```
int i = 0, j = 0, k = 0;  /* bad, too many initializer */

char a = 0;         /* OK, one variable per line with initializer */
char b = 0;

float x, y = 0.0;  /* OK, only last variable has initializer */
```

### 1.6.7 Casts and sizeof

- Casts and sizeof statements are not followed by a space.
- Always write sizeof statements with parenthesis. The redundant parenthesis rules do not apply to sizeof(var) instances.

## 1.7 C Function Definition, Declaration and Use

### 1.7.1 Prototypes

- It is recommended (and generally required by the compiler) that all non-static functions are prototyped somewhere.
- Functions local to one source module should be declared static, and should not be prototyped unless absolutely necessary.
- Functions used from other parts of code (external API) must be prototyped in the relevant include file.
- Function prototypes should be listed in a logical order, preferably alphabetical unless there is a compelling reason to use a different ordering.
- Functions that are used locally in more than one module go into a separate header file, for example, "extern.h".
- Do not use the __P macro.
- Functions that are part of an external API should be documented using Doxygen-like comments above declarations. See *Doxygen Guidelines* for details.
- Functions that are part of the external API must have an `rte_` prefix on the function name.

- Do not use uppercase letters - either in the form of ALL_UPPERCASE, or CamelCase - in function names. Lower-case letters and underscores only.
- When prototyping functions, associate names with parameter types, for example:

```
void function1(int fd); /* good */
void function2(int);    /* bad */
```

- Short function prototypes should be contained on a single line. Longer prototypes, e.g. those with many parameters, can be split across multiple lines. The second and subsequent lines should be further indented as for line statement continuations as described in the previous section.

```
static char *function1(int _arg, const char *_arg2,
       struct foo *_arg3,
       struct bar *_arg4,
       struct baz *_arg5);
static void usage(void);
```

---

**Note:** Unlike function definitions, the function prototypes do not need to place the function return type on a separate line.

---

### 1.7.2 Definitions

- The function type should be on a line by itself preceding the function.
- The opening brace of the function body should be on a line by itself.

```
static char *
function(int a1, int a2, float fl, int a4)
{
```

- Do not declare functions inside other functions. ANSI C states that such declarations have file scope regardless of the nesting of the declaration. Hiding file declarations in what appears to be a local scope is undesirable and will elicit complaints from a good compiler.
- Old-style (K&R) function declaration should not be used, use ANSI function declarations instead as shown below.
- Long argument lists should be wrapped as described above in the function prototypes section.

```
/*
 * All major routines should have a comment briefly describing what
 * they do. The comment before the "main" routine should describe
 * what the program does.
 */
int
main(int argc, char *argv[])
{
        char *ep;
        long num;
        int ch;
```

## 1.8 C Statement Style and Conventions

### 1.8.1 NULL Pointers

- NULL is the preferred null pointer constant. Use NULL instead of `(type *)0` or `(type *)NULL`, except where the compiler does not know the destination type e.g. for variadic args to a function.
- Test pointers against NULL, for example, use:

```
if (p == NULL) /* Good, compare pointer to NULL */
```

```
if (!p) /* Bad, using ! on pointer */
```

- Do not use ! for tests unless it is a boolean, for example, use:

```
if (*p == '\0') /* check character against (char)0 */
```

### 1.8.2 Return Value

- Functions which create objects, or allocate memory, should return pointer types, and NULL on error. The error type should be indicated may setting the variable `rte_errno` appropriately.
- Functions which work on bursts of packets, such as RX-like or TX-like functions, should return the number of packets handled.
- Other functions returning int should generally behave like system calls: returning 0 on success and -1 on error, setting `rte_errno` to indicate the specific type of error.
- Where already standard in a given library, the alternative error approach may be used where the negative value is not -1 but is instead `-errno` if relevant, for example, `-EINVAL`. Note, however, to allow consistency across functions returning integer or pointer types, the previous approach is preferred for any new libraries.
- For functions where no error is possible, the function type should be `void` not `int`.
- Routines returning `void *` should not have their return values cast to any pointer type. (Typecasting can prevent the compiler from warning about missing prototypes as any implicit definition of a function returns int, which, unlike `void *`, needs a typecast to assign to a pointer variable.)

---

**Note:** The above rule about not typecasting `void *` applies to malloc, as well as to DPDK functions.

---

- Values in return statements should not be enclosed in parentheses.

### 1.8.3 Logging and Errors

In the DPDK environment, use the logging interface provided:

```
#define RTE_LOGTYPE_TESTAPP1 RTE_LOGTYPE_USER1
#define RTE_LOGTYPE_TESTAPP2 RTE_LOGTYPE_USER2

/* enable these logs type */
rte_set_log_type(RTE_LOGTYPE_TESTAPP1, 1);
```

```
rte_set_log_type(RTE_LOGTYPE_TESTAPP2, 1);

/* log in debug level */
rte_set_log_level(RTE_LOG_DEBUG);
RTE_LOG(DEBUG, TESTAPP1, "this is is a debug level message\n");
RTE_LOG(INFO, TESTAPP1, "this is is a info level message\n");
RTE_LOG(WARNING, TESTAPP1, "this is is a warning level message\n");

/* log in info level */
rte_set_log_level(RTE_LOG_INFO);
RTE_LOG(DEBUG, TESTAPP2, "debug level message (not displayed)\n");
```

### 1.8.4 Branch Prediction

- When a test is done in a critical zone (called often or in a data path) the code can use the `likely()` and `unlikely()` macros to indicate the expected, or preferred fast path. They are expanded as a compiler builtin and allow the developer to indicate if the branch is likely to be taken or not. Example:

```
#include <rte_branch_prediction.h>
if (likely(x > 1))
  do_stuff();
```

---

**Note:** The use of `likely()` and `unlikely()` should only be done in performance critical paths, and only when there is a clearly preferred path, or a measured performance increase gained from doing so. These macros should be avoided in non-performance-critical code.

---

### 1.8.5 Static Variables and Functions

- All functions and variables that are local to a file must be declared as `static` because it can often help the compiler to do some optimizations (such as, inlining the code).
- Functions that should be inlined should to be declared as `static inline` and can be defined in a .c or a .h file.

---

**Note:** Static functions defined in a header file must be declared as `static inline` in order to prevent compiler warnings about the function being unused.

---

### 1.8.6 Const Attribute

The `const` attribute should be used as often as possible when a variable is read-only.

### 1.8.7 Inline ASM in C code

The `asm` and `volatile` keywords do not have underscores. The AT&T syntax should be used. Input and output operands should be named to avoid confusion, as shown in the following example:

```
asm volatile("outb %[val], %[port]"
        : :
        [port] "dN" (port),
        [val] "a" (val));
```

### 1.8.8 Control Statements

- Forever loops are done with for statements, not while statements.
- Elements in a switch statement that cascade should have a FALLTHROUGH comment. For example:

```
switch (ch) {         /* Indent the switch. */
case 'a':             /* Don't indent the case. */
        aflag = 1;    /* Indent case body one tab. */
        /* FALLTHROUGH */
case 'b':
        bflag = 1;
        break;
case '?':
default:
        usage();
        /* NOTREACHED */
}
```

CHAPTER

TWO

# DESIGN

## 2.1 Environment or Architecture-specific Sources

In DPDK and DPDK applications, some code is specific to an architecture (i686, x86_64) or to an executive environment (bsdapp or linuxapp) and so on. As far as is possible, all such instances of architecture or env-specific code should be provided via standard APIs in the EAL.

By convention, a file is common if it is not located in a directory indicating that it is specific. For instance, a file located in a subdir of "x86_64" directory is specific to this architecture. A file located in a subdir of "linuxapp" is specific to this execution environment.

---

**Note:** Code in DPDK libraries and applications should be generic. The correct location for architecture or executive environment specific code is in the EAL.

---

When absolutely necessary, there are several ways to handle specific code:

- Use a `#ifdef` with the CONFIG option in the C code. This can be done when the differences are small and they can be embedded in the same C file:
- Use the CONFIG option in the Makefile. This is done when the differences are more significant. In this case, the code is split into two separate files that are architecture or environment specific. This should only apply inside the EAL library.

### 2.1.1 Per Architecture Sources

The following config options can be used:

- CONFIG_RTE_ARCH is a string that contains the name of the architecture.
- CONFIG_RTE_ARCH_I686, CONFIG_RTE_ARCH_X86_64, CONFIG_RTE_ARCH_X86_64_32 or CONFIG_RTE_ARCH_PPC_64 are defined only if we are building for those architectures.

### 2.1.2 Per Execution Environment Sources

The following config options can be used:

- CONFIG_RTE_EXEC_ENV is a string that contains the name of the executive environment.
- CONFIG_RTE_EXEC_ENV_BSDAPP or CONFIG_RTE_EXEC_ENV_LINUXAPP are defined only if we are building for this execution environment.

## 2.2 Library Statistics

### 2.2.1 Description

This document describes the guidelines for DPDK library-level statistics counter support. This includes guidelines for turning library statistics on and off and requirements for preventing ABI changes when implementing statistics.

### 2.2.2 Mechanism to allow the application to turn library statistics on and off

Each library that maintains statistics counters should provide a single build time flag that decides whether the statistics counter collection is enabled or not. This flag should be exposed as a variable within the DPDK configuration file. When this flag is set, all the counters supported by current library are collected for all the instances of every object type provided by the library. When this flag is cleared, none of the counters supported by the current library are collected for any instance of any object type provided by the library:

```
# DPDK file config/common_linuxapp, config/common_bsdapp, etc.
CONFIG_RTE_<LIBRARY_NAME>_STATS_COLLECT=y/n
```

The default value for this DPDK configuration file variable (either "yes" or "no") is decided by each library.

### 2.2.3 Prevention of ABI changes due to library statistics support

The layout of data structures and prototype of functions that are part of the library API should not be affected by whether the collection of statistics counters is turned on or off for the current library. In practical terms, this means that space should always be allocated in the API data structures for statistics counters and the statistics related API functions are always built into the code, regardless of whether the statistics counter collection is turned on or off for the current library.

When the collection of statistics counters for the current library is turned off, the counters retrieved through the statistics related API functions should have a default value of zero.

### 2.2.4 Motivation to allow the application to turn library statistics on and off

It is highly recommended that each library provides statistics counters to allow an application to monitor the library-level run-time events. Typical counters are: number of packets received/dropped/transmitted, number of buffers allocated/freed, number of occurrences for specific events, etc.

However, the resources consumed for library-level statistics counter collection have to be spent out of the application budget and the counters collected by some libraries might not be relevant to the current application. In order to avoid any unwanted waste of resources and/or performance impacts, the application should decide at build time whether the collection of library-level statistics counters should be turned on or off for each library individually.

Library-level statistics counters can be relevant or not for specific applications:

- For Application A, counters maintained by Library X are always relevant and the application needs to use them to implement certain features, such as traffic accounting, logging,

application-level statistics, etc. In this case, the application requires that collection of statistics counters for Library X is always turned on.

- For Application B, counters maintained by Library X are only useful during the application debug stage and are not relevant once debug phase is over. In this case, the application may decide to turn on the collection of Library X statistics counters during the debug phase and at a later stage turn them off.
- For Application C, counters maintained by Library X are not relevant at all. It might be that the application maintains its own set of statistics counters that monitor a different set of run-time events (e.g. number of connection requests, number of active users, etc). It might also be that the application uses multiple libraries (Library X, Library Y, etc) and it is interested in the statistics counters of Library Y, but not in those of Library X. In this case, the application may decide to turn the collection of statistics counters off for Library X and on for Library Y.

The statistics collection consumes a certain amount of CPU resources (cycles, cache bandwidth, memory bandwidth, etc) that depends on:

- Number of libraries used by the current application that have statistics counters collection turned on.
- Number of statistics counters maintained by each library per object type instance (e.g. per port, table, pipeline, thread, etc).
- Number of instances created for each object type supported by each library.
- Complexity of the statistics logic collection for each counter: when only some occurrences of a specific event are valid, additional logic is typically needed to decide whether the current occurrence of the event should be counted or not. For example, in the event of packet reception, when only TCP packets with destination port within a certain range should be recorded, conditional branches are usually required. When processing a burst of packets that have been validated for header integrity, counting the number of bits set in a bitmask might be needed.

CHAPTER

THREE

# MANAGING ABI UPDATES

## 3.1 Description

This document details some methods for handling ABI management in the DPDK. Note this document is not exhaustive, in that C library versioning is flexible allowing multiple methods to achieve various goals, but it will provide the user with some introductory methods

## 3.2 General Guidelines

1. Whenever possible, ABI should be preserved
2. The addition of symbols is generally not problematic
3. The modification of symbols can generally be managed with versioning
4. The removal of symbols generally is an ABI break and requires bumping of the LIBABIVER macro

## 3.3 What is an ABI

An ABI (Application Binary Interface) is the set of runtime interfaces exposed by a library. It is similar to an API (Application Programming Interface) but is the result of compilation. It is also effectively cloned when applications link to dynamic libraries. That is to say when an application is compiled to link against dynamic libraries, it is assumed that the ABI remains constant between the time the application is compiled/linked, and the time that it runs. Therefore, in the case of dynamic linking, it is critical that an ABI is preserved, or (when modified), done in such a way that the application is unable to behave improperly or in an unexpected fashion.

## 3.4 The DPDK ABI policy

ABI versions are set at the time of major release labeling, and the ABI may change multiple times, without warning, between the last release label and the HEAD label of the git tree.

ABI versions, once released, are available until such time as their deprecation has been noted in the Release Notes for at least one major release cycle. For example consider the case where the ABI for DPDK 2.0 has been shipped and then a decision is made to modify it during the development of DPDK 2.1. The decision will be recorded in the Release Notes for the DPDK 2.1 release and the modification will be made available in the DPDK 2.2 release.

ABI versions may be deprecated in whole or in part as needed by a given update.

Some ABI changes may be too significant to reasonably maintain multiple versions. In those cases ABI's may be updated without backward compatibility being provided. The requirements for doing so are:

1. At least 3 acknowledgments of the need to do so must be made on the dpdk.org mailing list.
2. The changes (including an alternative map file) must be gated with the `RTE_NEXT_ABI` option, and provided with a deprecation notice at the same time. It will become the default ABI in the next release.
3. A full deprecation cycle, as explained above, must be made to offer downstream consumers sufficient warning of the change.
4. At the beginning of the next release cycle, every `RTE_NEXT_ABI` conditions will be removed, the `LIBABIVER` variable in the makefile(s) where the ABI is changed will be incremented, and the map files will be updated.

Note that the above process for ABI deprecation should not be undertaken lightly. ABI stability is extremely important for downstream consumers of the DPDK, especially when distributed in shared object form. Every effort should be made to preserve the ABI whenever possible. The ABI should only be changed for significant reasons, such as performance enhancements. ABI breakage due to changes such as reorganizing public structure fields for aesthetic or readability purposes should be avoided.

## 3.5 Examples of Deprecation Notices

The following are some examples of ABI deprecation notices which would be added to the Release Notes:

- The Macro #RTE_F00 is deprecated and will be removed with version 2.0, to be replaced with the inline function `rte_foo()`.
- The function `rte_mbuf_grok()` has been updated to include a new parameter in version 2.0. Backwards compatibility will be maintained for this function until the release of version 2.1
- The members of `struct rte_foo` have been reorganized in release 2.0 for performance reasons. Existing binary applications will have backwards compatibility in release 2.0, while newly built binaries will need to reference the new structure variant `struct rte_foo2`. Compatibility will be removed in release 2.2, and all applications will require updating and rebuilding to the new structure at that time, which will be renamed to the original `struct rte_foo`.
- Significant ABI changes are planned for the `librte_dostuff` library. The upcoming release 2.0 will not contain these changes, but release 2.1 will, and no backwards compatibility is planned due to the extensive nature of these changes. Binaries using this library built prior to version 2.1 will require updating and recompilation.

## 3.6 Versioning Macros

When a symbol is exported from a library to provide an API, it also provides a calling convention (ABI) that is embodied in its name, return type and arguments. Occasionally that function may need to change to accommodate new functionality or behavior. When that occurs, it is desirable to allow for backward compatibility for a time with older binaries that are dynamically linked to the DPDK.

To support backward compatibility the `lib/librte_compat/rte_compat.h` header file provides macros to use when updating exported functions. These macros are used in conjunction with the `rte_<library>_version.map` file for a given library to allow multiple versions of a symbol to exist in a shared library so that older binaries need not be immediately recompiled.

The macros exported are:

- `VERSION_SYMBOL(b, e, n)`: Creates a symbol version table entry binding versioned symbol b@DPDK_n to the internal function `b_e`.
- `BIND_DEFAULT_SYMBOL(b, e, n)`: Creates a symbol version entry instructing the linker to bind references to symbol b to the internal symbol `b_e`.
- `MAP_STATIC_SYMBOL(f, p)`: Declare the prototype `f`, and map it to the fully qualified function `p`, so that if a symbol becomes versioned, it can still be mapped back to the public symbol name.

## 3.7 Examples of ABI Macro use

### 3.7.1 Updating a public API

Assume we have a function as follows

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param)
{
        ...
}
```

Assume that struct rte_acl_ctx is a private structure, and that a developer wishes to enhance the acl api so that a debugging flag can be enabled on a per-context basis. This requires an addition to the structure (which, being private, is safe), but it also requires modifying the code as follows

```
/*
 * Create an acl context object for apps to
 * manipulate
 */
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param, int debug)
{
        ...
}
```

Note also that, being a public function, the header file prototype must also be changed, as must all the call sites, to reflect the new ABI footprint. We will maintain previous ABI versions that are accessible only to previously compiled binaries

The addition of a parameter to the function is ABI breaking as the function is public, and existing application may use it in its current form. However, the compatibility macros in DPDK allow a developer to use symbol versioning so that multiple functions can be mapped to the same public symbol based on when an application was linked to it. To see how this is done, we start with the requisite libraries version map file. Initially the version map file for the acl library looks like this

```
DPDK_2.0 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_create;
     rte_acl_dump;
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
     rte_acl_reset_rules;
     rte_acl_set_ctx_classify;

     local: *;
};
```

This file needs to be modified as follows

```
DPDK_2.0 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_create;
     rte_acl_dump;
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
     rte_acl_reset_rules;
     rte_acl_set_ctx_classify;

     local: *;
};

DPDK_2.1 {
     global:
     rte_acl_create;

} DPDK_2.0;
```

The addition of the new block tells the linker that a new version node is available (DPDK_2.1), which contains the symbol rte_acl_create, and inherits the symbols from the DPDK_2.0 node. This list is directly translated into a list of exported symbols when DPDK is compiled as a shared library

Next, we need to specify in the code which function map to the rte_acl_create symbol at which versions. First, at the site of the initial symbol definition, we need to update the function so that it is uniquely named, and not in conflict with the public symbol name

```
 struct rte_acl_ctx *
-rte_acl_create(const struct rte_acl_param *param)
+rte_acl_create_v20(const struct rte_acl_param *param)
{
        size_t sz;
        struct rte_acl_ctx *ctx;
        ...
```

Note that the base name of the symbol was kept intact, as this is condusive to the macros used for versioning symbols. That is our next step, mapping this new symbol name to the initial symbol name at version node 2.0. Immediately after the function, we add this line of code

```
VERSION_SYMBOL(rte_acl_create, _v20, 2.0);
```

Remembering to also add the rte_compat.h header to the requisite c file where these changes are being made. The above macro instructs the linker to create a new symbol `rte_acl_create@DPDK_2.0`, which matches the symbol created in older builds, but now points to the above newly named function. We have now mapped the original rte_acl_create symbol to the original function (but with a new name)

Next, we need to create the 2.1 version of the symbol. We create a new function name, with a different suffix, and implement it appropriately

```
struct rte_acl_ctx *
rte_acl_create_v21(const struct rte_acl_param *param, int debug);
{
     struct rte_acl_ctx *ctx = rte_acl_create_v20(param);

     ctx->debug = debug;

     return ctx;
}
```

This code serves as our new API call. Its the same as our old call, but adds the new parameter in place. Next we need to map this function to the symbol `rte_acl_create@DPDK_2.1`. To do this, we modify the public prototype of the call in the header file, adding the macro there to inform all including applications, that on re-link, the default rte_acl_create symbol should point to this function. Note that we could do this by simply naming the function above rte_acl_create, and the linker would chose the most recent version tag to apply in the version script, but we can also do this in the header file

```
struct rte_acl_ctx *
-rte_acl_create(const struct rte_acl_param *param);
+rte_acl_create(const struct rte_acl_param *param, int debug);
+BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 2.1);
```

The BIND_DEFAULT_SYMBOL macro explicitly tells applications that include this header, to link to the rte_acl_create_v21 function and apply the DPDK_2.1 version node to it. This method is more explicit and flexible than just re-implementing the exact symbol name, and allows for other features (such as linking to the old symbol version by default, when the new ABI is to be opt-in for a period.

One last thing we need to do. Note that we've taken what was a public symbol, and duplicated it into two uniquely and differently named symbols. We've then mapped each of those back to the public symbol `rte_acl_create` with different version tags. This only applies to dynamic linking, as static linking has no notion of versioning. That leaves this code in a position of no longer having a symbol simply named `rte_acl_create` and a static build will fail on that missing symbol.

To correct this, we can simply map a function of our choosing back to the public symbol in the static build with the `MAP_STATIC_SYMBOL` macro. Generally the assumption is that the most recent version of the symbol is the one you want to map. So, back in the C file where, immediately after `rte_acl_create_v21` is defined, we add this

```
struct rte_acl_create_v21(const struct rte_acl_param *param, int debug)
{
     ...
}
MAP_STATIC_SYMBOL(struct rte_acl_create(const struct rte_acl_param *param, int debug), rte_acl_
```

That tells the compiler that, when building a static library, any calls to the symbol `rte_acl_create` should be linked to `rte_acl_create_v21`

That's it, on the next shared library rebuild, there will be two versions of rte_acl_create, an old DPDK_2.0 version, used by previously built applications, and a new DPDK_2.1 version, used by future built applications.

### 3.7.2 Deprecating part of a public API

Lets assume that you've done the above update, and after a few releases have passed you decide you would like to retire the old version of the function. After having gone through the ABI deprecation announcement process, removal is easy. Start by removing the symbol from the requisite version map file:

```
    DPDK_2.0 {
         global:

         rte_acl_add_rules;
         rte_acl_build;
         rte_acl_classify;
         rte_acl_classify_alg;
         rte_acl_classify_scalar;
         rte_acl_dump;
-        rte_acl_create
         rte_acl_find_existing;
         rte_acl_free;
         rte_acl_ipv4vlan_add_rules;
         rte_acl_ipv4vlan_build;
         rte_acl_list_dump;
         rte_acl_reset;
         rte_acl_reset_rules;
         rte_acl_set_ctx_classify;

         local: *;
    };

    DPDK_2.1 {
         global:
         rte_acl_create;
    } DPDK_2.0;
```

Next remove the corresponding versioned export .. code-block:: c

-VERSION_SYMBOL(rte_acl_create, _v20, 2.0);

Note that the internal function definition could also be removed, but its used in our example by the newer version _v21, so we leave it in place. This is a coding style choice.

Lastly, we need to bump the LIBABIVER number for this library in the Makefile to indicate to applications doing dynamic linking that this is a later, and possibly incompatible library version:

```
-LIBABIVER := 1
+LIBABIVER := 2
```

### 3.7.3 Deprecating an entire ABI version

While removing a symbol from and ABI may be useful, it is often more practical to remove an entire version node at once. If a version node completely specifies an API, then removing part of it, typically makes it incomplete. In those cases it is better to remove the entire node

To do this, start by modifying the version map file, such that all symbols from the node to be removed are merged into the next node in the map

In the case of our map above, it would transform to look as follows

```
DPDK_2.1 {
     global:

     rte_acl_add_rules;
     rte_acl_build;
     rte_acl_classify;
     rte_acl_classify_alg;
     rte_acl_classify_scalar;
     rte_acl_dump;
     rte_acl_create
     rte_acl_find_existing;
     rte_acl_free;
     rte_acl_ipv4vlan_add_rules;
     rte_acl_ipv4vlan_build;
     rte_acl_list_dump;
     rte_acl_reset;
     rte_acl_reset_rules;
     rte_acl_set_ctx_classify;

     local: *;
};
```

Then any uses of BIND_DEFAULT_SYMBOL that pointed to the old node should be updated to point to the new version node in any header files for all affected symbols.

```
-BIND_DEFAULT_SYMBOL(rte_acl_create, _v20, 2.0);
+BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 2.1);
```

Lastly, any VERSION_SYMBOL macros that point to the old version node should be removed, taking care to keep, where need old code in place to support newer versions of the symbol.

## 3.8 Running the ABI Validator

The `scripts` directory in the DPDK source tree contains a utility program, `validate-abi.sh`, for validating the DPDK ABI based on the Linux ABI Compliance Checker.

This has a dependency on the `abi-compliance-checker` and `and abi-dumper` utilities which can be installed via a package manager. For example:

```
sudo yum install abi-compliance-checker
sudo yum install abi-dumper
```

The syntax of the `validate-abi.sh` utility is:

```
./scripts/validate-abi.sh <TAG1> <TAG2> <TARGET>
```

Where `TAG1` and `TAG2` are valid git tags on the local repo and target is the usual DPDK compilation target.

For example to test the current committed HEAD against a previous release tag we could add a temporary tag and run the utility as follows:

```
git tag MY_TEMP_TAG
./scripts/validate-abi.sh v2.0.0 MY_TEMP_TAG x86_64-native-linuxapp-gcc
```

After the validation script completes (it can take a while since it need to compile both tags) it will create compatibility reports in the `./compat_report` directory. Listed incompatibilities can be found as follows:

```
grep -lr Incompatible compat_reports/
```

CHAPTER

FOUR

# DPDK DOCUMENTATION GUIDELINES

This document outlines the guidelines for writing the DPDK Guides and API documentation in RST and Doxygen format.

It also explains the structure of the DPDK documentation and shows how to build the Html and PDF versions of the documents.

## 4.1 Structure of the Documentation

The DPDK source code repository contains input files to build the API documentation and User Guides.

The main directories that contain files related to documentation are shown below:

```
lib
|-- librte_acl
|-- librte_cfgfile
|-- librte_cmdline
|-- librte_compat
|-- librte_eal
|   |-- ...
...
doc
|-- api
+-- guides
    |-- freebsd_gsg
    |-- linux_gsg
    |-- prog_guide
    |-- sample_app_ug
    |-- guidelines
    |-- testpmd_app_ug
    |-- rel_notes
    |-- nics
    |-- xen
    |-- ...
```

The API documentation is built from Doxygen comments in the header files. These files are mainly in the `lib/librte_*` directories although some of the Poll Mode Drivers in `drivers/net` are also documented with Doxygen.

The configuration files that are used to control the Doxygen output are in the `doc/api` directory.

The user guides such as *The Programmers Guide* and the *FreeBSD* and *Linux Getting Started* Guides are generated from RST markup text files using the Sphinx Documentation Generator.

These files are included in the `doc/guides/` directory. The output is controlled by the `doc/guides/conf.py` file.

## 4.2 Role of the Documentation

The following items outline the roles of the different parts of the documentation and when they need to be updated or added to by the developer.

- **Release Notes**

  The Release Notes document which features have been added in the current and previous releases of DPDK and highlight any known issues. The Releases Notes also contain notifications of features that will change ABI compatibility in the next major release.

  Developers should update the Release Notes to add a short description of new or updated features. Developers should also update the Release Notes to add ABI announcements if necessary, (see `/contributing/versioning` for details).

- **API documentation**

  The API documentation explains how to use the public DPDK functions. The API index page shows the generated API documentation with related groups of functions.

  The API documentation should be updated via Doxygen comments when new functions are added.

- **Getting Started Guides**

  The Getting Started Guides show how to install and configure DPDK and how to run DPDK based applications on different OSes.

  A Getting Started Guide should be added when DPDK is ported to a new OS.

- **The Programmers Guide**

  The Programmers Guide explains how the API components of DPDK such as the EAL, Memzone, Rings and the Hash Library work. It also explains how some higher level functionality such as Packet Distributor, Packet Framework and KNI work. It also shows the build system and explains how to add applications.

  The Programmers Guide should be expanded when new functionality is added to DPDK.

- **App Guides**

  The app guides document the DPDK applications in the `app` directory such as `testpmd`.

  The app guides should be updated if functionality is changed or added.

- **Sample App Guides**

  The sample app guides document the DPDK example applications in the examples directory. Generally they demonstrate a major feature such as L2 or L3 Forwarding, Multi Process or Power Management. They explain the purpose of the sample application, how to run it and step through some of the code to explain the major functionality.

  A new sample application should be accompanied by a new sample app guide. The guide for the Skeleton Forwarding app is a good starting reference.

- **Network Interface Controller Drivers**

  The NIC Drivers document explains the features of the individual Poll Mode Drivers, such as software requirements, configuration and initialization.

  New documentation should be added for new Poll Mode Drivers.

- **Guidelines**

  The guideline documents record community process, expectations and design directions.

  They can be extended, amended or discussed by submitting a patch and getting community approval.

## 4.3 Building the Documentation

### 4.3.1 Dependencies

The following dependencies must be installed to build the documentation:

- Doxygen.
- Sphinx (also called python-sphinx).
- TexLive (at least TexLive-core, extra Latex support and extra fonts).
- Inkscape.

Doxygen generates documentation from commented source code. It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install doxygen

# Red Hat/Fedora.
sudo yum     -y install doxygen
```

Sphinx is a Python documentation tool for converting RST files to Html or to PDF (via LaTeX). It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install python-sphinx

# Red Hat/Fedora.
sudo yum     -y install python-sphinx

# Or, on any system with Python installed.
sudo easy_install -U sphinx
```

For further information on getting started with Sphinx see the Sphinx Tutorial.

---

**Note:** To get full support for Figure and Table numbering it is best to install Sphinx 1.3.1 or later.

---

Inkscape is a vector based graphics program which is used to create SVG images and also to convert SVG images to PDF images. It can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install inkscape
```

```
# Red Hat/Fedora.
sudo yum     -y install inkscape
```

TexLive is an installation package for Tex/LaTeX. It is used to generate the PDF versions of the documentation. The main required packages can be installed as follows:

```
# Ubuntu/Debian.
sudo apt-get -y install texlive-latex-extra texlive-fonts-extra \
                        texlive-fonts-recommended


# Red Hat/Fedora, selective install.
sudo yum     -y install texlive-collection-latexextra \
                        texlive-collection-fontsextra
```

### 4.3.2 Build commands

The documentation is built using the standard DPDK build system. Some examples are shown below:

- Generate all the documentation targets:

  ```
  make doc
  ```

- Generate the Doxygen API documentation in Html:

  ```
  make doc-api-html
  ```

- Generate the guides documentation in Html:

  ```
  make doc-guides-html
  ```

- Generate the guides documentation in Pdf:

  ```
  make doc-guides-pdf
  ```

The output of these commands is generated in the `build` directory:

```
build/doc
      |-- html
      |   |-- api
      |   +-- guides
      |
      +-- pdf
          +-- guides
```

---

**Note:** Make sure to fix any Sphinx or Doxygen warnings when adding or updating documentation.

---

The documentation output files can be removed as follows:

```
make doc-clean
```

## 4.4 Document Guidelines

Here are some guidelines in relation to the style of the documentation:

- Document the obvious as well as the obscure since it won't always be obvious to the reader. For example an instruction like "Set up 64 2MB Hugepages" is better when followed by a sample commandline or a link to the appropriate section of the documentation.

- Use American English spellings throughout. This can be checked using the `aspell` utility:

```
aspell --lang=en_US --check doc/guides/sample_app_ug/mydoc.rst
```

## 4.5 RST Guidelines

The RST (reStructuredText) format is a plain text markup format that can be converted to Html, PDF or other formats. It is most closely associated with Python but it can be used to document any language. It is used in DPDK to document everything apart from the API.

The Sphinx documentation contains a very useful RST Primer which is a good place to learn the minimal set of syntax required to format a document.

The official reStructuredText website contains the specification for the RST format and also examples of how to use it. However, for most developers the RST Primer is a better resource.

The most common guidelines for writing RST text are detailed in the Documenting Python guidelines. The additional guidelines below reiterate or expand upon those guidelines.

### 4.5.1 Line Length

- The recommended style for the DPDK documentation is to put sentences on separate lines. This allows for easier reviewing of patches. Multiple sentences which are not separated by a blank line are joined automatically into paragraphs, for example:

```
Here is an example sentence.
Long sentences over the limit shown below can be wrapped onto
a new line.
These three sentences will be joined into the same paragraph.

This is a new paragraph, since it is separated from the
previous paragraph by a blank line.
```

  This would be rendered as follows:

  > *Here is an example sentence. Long sentences over the limit shown below can be wrapped onto a new line. These three sentences will be joined into the same paragraph.*
  >
  > *This is a new paragraph, since it is separated from the previous paragraph by a blank line.*

- Long sentences should be wrapped at 120 characters +/- 10 characters. They should be wrapped at words.
- Lines in literal blocks must by less than 80 characters since they aren't wrapped by the document formatters and can exceed the page width in PDF documents.

### 4.5.2 Whitespace

- Standard RST indentation is 3 spaces. Code can be indented 4 spaces, especially if it is copied from source files.
- No tabs. Convert tabs in embedded code to 4 or 8 spaces.

- No trailing whitespace.
- Add 2 blank lines before each section header.
- Add 1 blank line after each section header.
- Add 1 blank line between each line of a list.

### 4.5.3 Section Headers

- Section headers should use the use the following underline formats:

```
Level 1 Heading
===============


Level 2 Heading
---------------


Level 3 Heading
~~~~~~~~~~~~~~~


Level 4 Heading
^^^^^^^^^^^^^^^
```

- Level 4 headings should be used sparingly.
- The underlines should match the length of the text.
- In general, the heading should be less than 80 characters, for conciseness.
- As noted above:
  - Add 2 blank lines before each section header.
  - Add 1 blank line after each section header.

### 4.5.4 Lists

- Bullet lists should be formatted with a leading * as follows:

```
* Item one.

* Item two is a long line that is wrapped and then indented to match
  the start of the previous line.

* One space character between the bullet and the text is preferred.
```

- Numbered lists can be formatted with a leading number but the preference is to use `#.` which will give automatic numbering. This is more convenient when adding or removing items:

```
#. Item one.

#. Item two is a long line that is wrapped and then indented
   to match the start of the e first line.

#. Item two is a long line that is wrapped and then indented to match
   the start of the previous line.
```

- Definition lists can be written with or without a bullet:

```
* Item one.

  Some text about item one.

* Item two.

  Some text about item two.
```

- All lists, and sub-lists, must be separated from the preceding text by a blank line. This is a syntax requirement.
- All list items should be separated by a blank line for readability.

### 4.5.5 Code and Literal block sections

- Inline text that is required to be rendered with a fixed width font should be enclosed in backquotes like this: ``text`, so that it appears like this: `text`.
- Fixed width, literal blocks of texts should be indented at least 3 spaces and prefixed with `::` like this:

```
Here is some fixed width text::

   0x0001 0x0001 0x00FF 0x00FF
```

- It is also possible to specify an encoding for a literal block using the `.. code-block::` directive so that syntax highlighting can be applied. Examples of supported highlighting are:

```
.. code-block:: console
.. code-block:: c
.. code-block:: python
.. code-block:: diff
.. code-block:: none
```

That can be applied as follows:

```
.. code-block:: c

   #include<stdio.h>

   int main() {

      printf("Hello World\n");

      return 0;
   }
```

Which would be rendered as:

```c
#include<stdio.h>

int main() {

   printf("Hello World\n");

   return 0;
}
```

- The default encoding for a literal block using the simplified `::` directive is `none`.

- Lines in literal blocks must be less than 80 characters since they can exceed the page width when converted to PDF documentation. For long literal lines that exceed that limit try to wrap the text at sensible locations. For example a long command line could be documented like this and still work if copied directly from the docs:

```
build/app/testpmd -c7 -n3 --vdev=eth_pcap0,iface=eth0     \
                          --vdev=eth_pcap1,iface=eth1     \
                          -- -i --nb-cores=2 --nb-ports=2 \
                             --total-num-mbufs=2048
```

- Long lines that cannot be wrapped, such as application output, should be truncated to be less than 80 characters.

### 4.5.6 Images

- All images should be in SVG scalar graphics format. They should be true SVG XML files and should not include binary formats embedded in a SVG wrapper.
- The DPDK documentation contains some legacy images in PNG format. These will be converted to SVG in time.
- Inkscape is the recommended graphics editor for creating the images. Use some of the older images in `doc/guides/prog_guide/img/` as a template, for example `mbuf1.svg` or `ring-enqueue.svg`.
- The SVG images should include a copyright notice, as an XML comment.
- Images in the documentation should be formatted as follows:
  - The image should be preceded by a label in the format `.. _figure_XXXX:` with a leading underscore and where XXXX is a unique descriptive name.
  - Images should be included using the `.. figure::` directive and the file type should be set to * (not `.svg`). This allows the format of the image to be changed if required, without updating the documentation.
  - Images must have a caption as part of the `.. figure::` directive.
- Here is an example of the previous three guidelines:

```
.. _figure_mempool:

.. figure:: img/mempool.*

   A mempool in memory with its associated ring.
```

- Images can then be linked to using the `:numref:` directive:

```
The mempool layout is shown in :numref:figure_mempool.
```

  This would be rendered as: *The mempool layout is shown in Fig 6.3*.

  **Note**: The `:numref:` directive requires Sphinx 1.3.1 or later. With earlier versions it will still be rendered as a link but won't have an automatically generated number.

- The caption of the image can be generated, with a link, using the `:ref:` directive:

```
:ref:figure_mempool
```

  This would be rendered as: *A mempool in memory with its associated ring.*

### 4.5.7 Tables

- RST tables should be used sparingly. They are hard to format and to edit, they are often rendered incorrectly in PDF format, and the same information can usually be shown just as clearly with a definition or bullet list.
- Tables in the documentation should be formatted as follows:
  - The table should be preceded by a label in the format `.. _table_XXXX:` with a leading underscore and where XXXX is a unique descriptive name.
  - Tables should be included using the `.. table::` directive and must have a caption.
- Here is an example of the previous two guidelines:

```
.. _table_qos_pipes:

.. table:: Sample configuration for QOS pipes.

   +----------+----------+----------+
   | Header 1 | Header 2 | Header 3 |
   |          |          |          |
   +==========+==========+==========+
   | Text     | Text     | Text     |
   +----------+----------+----------+
   | ...      | ...      | ...      |
   +----------+----------+----------+
```

- Tables can be linked to using the `:numref:` and `:ref:` directives, as shown in the previous section for images. For example:

```
The QOS configuration is shown in :numref:table_qos_pipes.
```

- Tables should not include merged cells since they are not supported by the PDF renderer.

### 4.5.8 Hyperlinks

- Links to external websites can be plain URLs. The following is rendered as http://dpdk.org:

```
http://dpdk.org
```

- They can contain alternative text. The following is rendered as Check out DPDK:

```
Check out DPDK <http://dpdk.org>_
```

- An internal link can be generated by placing labels in the document with the format `.. _label_name`.
- The following links to the top of this section: *Hyperlinks*:

```
.. _links:

Hyperlinks
~~~~~~~~~~

* The following links to the top of this section: :ref:links:
```

**Note:** The label must have a leading underscore but the reference to it must omit it. This is a frequent cause of errors and warnings.

- The use of a label is preferred since it works across files and will still work if the header text changes.

## 4.6 Doxygen Guidelines

The DPDK API is documented using Doxygen comment annotations in the header files. Doxygen is a very powerful tool, it is extremely configurable and with a little effort can be used to create expressive documents. See the Doxygen website for full details on how to use it.

The following are some guidelines for use of Doxygen in the DPDK API documentation:

- New libraries that are documented with Doxygen should be added to the Doxygen configuration file: `doc/api/doxy-api.conf`. It is only required to add the directory that contains the files. It isn't necessary to explicitly name each file since the configuration matches all `rte_*.h` files in the directory.
- Use proper capitalization and punctuation in the Doxygen comments since they will become sentences in the documentation. This in particular applies to single line comments, which is the case the is most often forgotten.
- Use @ style Doxygen commands instead of \ style commands.
- Add a general description of each library at the head of the main header files:

```
/**
 * @file
 * RTE Mempool.
 *
 * A memory pool is an allocator of fixed-size object. It is
 * identified by its name, and uses a ring to store free objects.
 * ...
 */
```

- Document the purpose of a function, the parameters used and the return value:

```
/**
 * Attach a new Ethernet device specified by arguments.
 *
 * @param devargs
 *  A pointer to a strings array describing the new device
 *  to be attached. The strings should be a pci address like
 *  0000:01:00.0 or **virtual** device name like eth_pcap0.
 * @param port_id
 *  A pointer to a port identifier actually attached.
 *
 * @return
 *  0 on success and port_id is filled, negative on error.
 */
int rte_eth_dev_attach(const char *devargs, uint8_t *port_id);
```

- Doxygen supports Markdown style syntax such as bold, italics, fixed width text and lists. For example the second line in the `devargs` parameter in the previous example will be rendered as:

  > The strings should be a pci address like `0000:01:00.0` or **virtual** device name like `eth_pcap0`.

- Use - instead of * for lists within the Doxygen comment since the latter can get confused with the comment delimiter.

- Add an empty line between the function description, the `@params` and `@return` for readability.
- Place the `@params` description on separate line and indent it by 2 spaces. (It would be better to use no indentation since this is more common and also because checkpatch complains about leading whitespace in comments. However this is the convention used in the existing DPDK code.)
- Documented functions can be linked to simply by adding `()` to the function name:

```
/**
 * The functions exported by the application Ethernet API to setup
 * a device designated by its port identifier must be invoked in
 * the following order:
 *     - rte_eth_dev_configure()
 *     - rte_eth_tx_queue_setup()
 *     - rte_eth_rx_queue_setup()
 *     - rte_eth_dev_start()
 */
```

  In the API documentation the functions will be rendered as links, see the online section of the rte_ethdev.h docs that contains the above text.

- The `@see` keyword can be used to create a *see also* link to another file or library. This directive should be placed on one line at the bottom of the documentation section.

```
/**
 * ...
 *
 * Some text that references mempools.
 *
 * @see eal_memzone.c
 */
```

- Doxygen supports two types of comments for documenting variables, constants and members: prefix and postfix:

```
/** This is a prefix comment. */
#define RTE_FOO_ERROR  0x023.

#define RTE_BAR_ERROR  0x024. /**< This is a postfix comment. */
```

- Postfix comments are preferred for struct members and constants if they can be documented in the same way:

```
struct rte_eth_stats {
    uint64_t ipackets; /**< Total number of received packets. */
    uint64_t opackets; /**< Total number of transmitted packets.*/
    uint64_t ibytes;   /**< Total number of received bytes. */
    uint64_t obytes;   /**< Total number of transmitted bytes. */
    uint64_t imissed;  /**< Total of RX missed packets. */
    uint64_t ibadcrc;  /**< Total of RX packets with CRC error. */
    uint64_t ibadlen;  /**< Total of RX packets with bad length. */
}
```

  Note: postfix comments should be aligned with spaces not tabs in accordance with the *DPDK Coding Style*.

- If a single comment type can't be used, due to line length limitations then prefix comments should be preferred. For example this section of the code contains prefix comments, postfix comments on the same line and postfix comments on a separate line:

```
/** Number of elements in the elt_pa array. */
uint32_t    pg_num __rte_cache_aligned;
```

```
uint32_t    pg_shift;     /**< LOG2 of the physical pages. */
uintptr_t   pg_mask;      /**< Physical page mask value. */
uintptr_t   elt_va_start;
/**< Virtual address of the first mempool object. */
uintptr_t   elt_va_end;
/**< Virtual address of the <size + 1> mempool object. */
phys_addr_t elt_pa[MEMPOOL_PG_NUM_DEFAULT];
/**< Array of physical page addresses for the mempool buffer. */
```

This doesn't have an effect on the rendered documentation but it is confusing for the developer reading the code. It this case it would be clearer to use prefix comments throughout:

```
/** Number of elements in the elt_pa array. */
uint32_t    pg_num __rte_cache_aligned;
/** LOG2 of the physical pages. */
uint32_t    pg_shift;
/** Physical page mask value. */
uintptr_t   pg_mask;
/** Virtual address of the first mempool object. */
uintptr_t   elt_va_start;
/** Virtual address of the <size + 1> mempool object. */
uintptr_t   elt_va_end;
/** Array of physical page addresses for the mempool buffer. */
phys_addr_t elt_pa[MEMPOOL_PG_NUM_DEFAULT];
```

- Check for Doxygen warnings in new code by checking the API documentation build:

  ```
  make doc-api-html >/dev/null
  ```

- Read the rendered section of the documentation that you have added for correctness, clarity and consistency with the surrounding text.