DPDK

DATA PLANE DEVELOPMENT KIT

Contributor's Guidelines

Release 20.05.0

May 26, 2020

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.

Note: Uses of `bool` in structures are not preferred as is wastes space and it's also not clear as to what type size the bool is. A preferred use of `bool` is mainly as a return type from functions that return true/false, and maybe local variable functions.

Ref: LKML

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 initialized 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:

```
/* register log types for this application */
int my_logtype1 = rte_log_register("myapp.log1");
int my_logtype2 = rte_log_register("myapp.log2");

/* set global log level to INFO */
rte_log_set_global_level(RTE_LOG_INFO);

/* only display messages higher than NOTICE for log2 (default
 * is DEBUG) */
rte_log_set_level(my_logtype2, RTE_LOG_NOTICE);

/* enable all PMD logs (whose identifier string starts with "pmd.") */
rte_log_set_level_pattern("pmd.*", RTE_LOG_DEBUG);

/* log in debug level */
rte_log_set_global_level(RTE_LOG_DEBUG);
RTE_LOG(DEBUG, my_logtype1, "this is a debug level message\n");
RTE_LOG(INFO, my_logtype1, "this is a info level message\n");
RTE_LOG(WARNING, my_logtype1, "this is a warning level message\n");
RTE_LOG(WARNING, my_logtype2, "this is a debug level message (not displayed)\n");

/* log in info level */
rte_log_set_global_level(RTE_LOG_INFO);
RTE_LOG(DEBUG, my_logtype1, "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 */
}
```

1.9 Dynamic Logging

DPDK provides infrastructure to perform logging during runtime. This is very useful for enabling debug output without recompilation. To enable or disable logging of a particular topic, the `--log-level` parameter can be provided to EAL, which will change the log level. DPDK code can register topics, which allows the user to adjust the log verbosity for that specific topic.

In general, the naming scheme is as follows: `type.section.name`

- Type is the type of component, where `lib`, `pmd`, `bus` and `user` are the common options.
- Section refers to a specific area, for example a poll-mode-driver for an ethernet device would use `pmd.net`, while an eventdev PMD uses `pmd.event`.
- The name identifies the individual item that the log applies to. The name section must align with the directory that the PMD code resides. See examples below for clarity.

Examples:

- The virtio network PMD in `drivers/net/virtio` uses `pmd.net.virtio`
- The eventdev software poll mode driver in `drivers/event/sw` uses `pmd.event.sw`
- The octeontx mempool driver in `drivers/mempool/octeontx` uses `pmd.mempool.octeontx`
- The DPDK hash library in `lib/librte_hash` uses `lib.hash`

1.9.1 Specializations

In addition to the above logging topic, any PMD or library can further split logging output by using "specializations". A specialization could be the difference between initialization code, and logs of events that occur at runtime.

An example could be the initialization log messages getting one specialization, while another specialization handles mailbox command logging. Each PMD, library or component can create as many specializations as required.

A specialization looks like this:

- Initialization output: `type.section.name.init`
- PF/VF mailbox output: `type.section.name.mbox`

A real world example is the i40e poll mode driver which exposes two specializations, one for initialization `pmd.net.i40e.init` and the other for the remaining driver logs `pmd.net.i40e.driver`.

Note that specializations have no formatting rules, but please follow a precedent if one exists. In order to see all current log topics and specializations, run the `app/test` binary, and use the `dump_log_types`

1.10 Python Code

All Python code should work with Python 2.7+ and 3.2+ and be compliant with PEP8 (Style Guide for Python Code).

The `pep8` tool can be used for testing compliance with the guidelines.

1.11 Integrating with the Build System

DPDK supports being built in two different ways:

- using `make` - or more specifically "GNU make", i.e. `gmake` on FreeBSD
- using the tools `meson` and `ninja`

Any new library or driver to be integrated into DPDK should support being built with both systems. While building using `make` is a legacy approach, and most build-system enhancements are being done using `meson` and `ninja` there are no plans at this time to deprecate the legacy `make` build system.

Therefore all new component additions should include both a `Makefile` and a `meson.build` file, and should be added to the component lists in both the `Makefile` and `meson.build` files in the relevant top-level directory: either `lib` directory or a `driver` subdirectory.

1.11.1 Makefile Contents

The `Makefile` for the component should be of the following format, where `<name>` corresponds to the name of the library in question, e.g. hash, lpm, etc. For drivers, the same format of Makefile is used.

```
# pull in basic DPDK definitions, including whether library is to be
# built or not
include $(RTE_SDK)/mk/rte.vars.mk
```

```
# library name
LIB = librte_<name>.a

# any library cflags needed. Generally add "-O3 $(WERROR_FLAGS)"
CFLAGS += -O3
CFLAGS += $(WERROR_FLAGS)

# the symbol version information for the library
EXPORT_MAP := rte_<name>_version.map

# all source filenames are stored in SRCS-y
SRCS-$(CONFIG_RTE_LIBRTE_<NAME>) += rte_<name>.c

# install includes
SYMLINK-$(CONFIG_RTE_LIBRTE_<NAME>)-include += rte_<name>.h

# pull in rules to build the library
include $(RTE_SDK)/mk/rte.lib.mk
```

1.11.2 Meson Build File Contents - Libraries

The `meson.build` file for a new DPDK library should be of the following basic format.

```
sources = files('file1.c', ...)
headers = files('file1.h', ...)
```

This will build based on a number of conventions and assumptions within the DPDK itself, for example, that the library name is the same as the directory name in which the files are stored.

For a library `meson.build` file, there are number of variables which can be set, some mandatory, others optional. The mandatory fields are:

sources Default Value = []. This variable should list out the files to be compiled up to create the library. Files must be specified using the meson `files()` function.

The optional fields are:

build Default Value = true Used to optionally compile a library, based on its dependencies or environment. When set to "false" the `reason` value, explained below, should also be set to explain to the user why the component is not being built. A simple example of use would be:

```
if not is_linux
        build = false
        reason = 'only supported on Linux'
endif
```

cflags Default Value = [<-march/-mcpu flags>]. Used to specify any additional cflags that need to be passed to compile the sources in the library.

deps Default Value = ['eal']. Used to list the internal library dependencies of the library. It should be assigned to using += rather than overwriting using =. The dependencies should be specified as strings, each one giving the name of a DPDK library, without the `librte_` prefix. Dependencies are handled recursively, so specifying e.g. `mempool`, will automatically also make the library depend upon the mempool library's dependencies too - `ring` and `eal`. For libraries that only depend upon EAL, this variable may be omitted from the `meson.build` file. For example:

```
deps += ['ethdev']
```

ext_deps Default Value = []. Used to specify external dependencies of this library. They should be returned as dependency objects, as returned from the meson `dependency()` or `find_library()` functions. Before returning these, they should be checked to ensure the dependencies have been found, and, if not, the `build` variable should be set to `false`. For example:

```
my_dep = dependency('libX', required: 'false')
if my_dep.found()
        ext_deps += my_dep
else
        build = false
endif
```

headers Default Value = []. Used to return the list of header files for the library that should be installed to $PREFIX/include when `ninja install` is run. As with source files, these should be specified using the meson `files()` function.

includes: Default Value = []. Used to indicate any additional header file paths which should be added to the header search path for other libs depending on this library. EAL uses this so that other libraries building against it can find the headers in subdirectories of the main EAL directory. The base directory of each library is always given in the include path, it does not need to be specified here.

name Default Value = library name derived from the directory name. If a library's .so or .a file differs from that given in the directory name, the name should be specified using this variable. In practice, since the convention is that for a library called `librte_xyz.so`, the sources are stored in a directory `lib/librte_xyz`, this value should never be needed for new libraries.

Note: The name value also provides the name used to find the function version map file, as part of the build process, so if the directory name and library names differ, the `version.map` file should be named consistently with the library, not the directory

objs Default Value = []. This variable can be used to pass to the library build some pre-built objects that were compiled up as part of another target given in the included library `meson.build` file.

reason Default Value = '<unknown reason>'. This variable should be used when a library is not to be built i.e. when `build` is set to "false", to specify the reason why a library will not be built. For missing dependencies this should be of the form `'missing dependency, "libname"'`.

use_function_versioning Default Value = false. Specifies if the library in question has ABI versioned functions. If it has, this value should be set to ensure that the C files are compiled twice with suitable parameters for each of shared or static library builds.

1.11.3 Meson Build File Contents - Drivers

For drivers, the values are largely the same as for libraries. The variables supported are:

build As above.

cflags As above.

deps As above.

ext_deps As above.

includes Default Value = <driver directory> Some drivers include a base directory for additional source files and headers, so we have this variable to allow the headers from that base directory to be found when compiling driver sources. Should be appended to using += rather than overwritten using =. The values appended should be meson include objects got using the `include_directories()` function. For example:

```
includes += include_directories('base')
```

name As above, though note that each driver class can define it's own naming scheme for the resulting `.so` files.

objs As above, generally used for the contents of the `base` directory.

pkgconfig_extra_libs Default Value = [] This variable is used to pass additional library link flags through to the DPDK pkgconfig file generated, for example, to track any additional libraries that may need to be linked into the build - especially when using static libraries. Anything added here will be appended to the end of the `pkgconfig --libs` output.

reason As above.

sources [mandatory] As above

version As above

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 (freebsd or linux) 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 “linux” 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:

  ```
  #ifdef RTE_ARCH_I686
  toto();
  #else
  titi();
  #endif
  ```

- 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.

Note: As in the linux kernel, the `CONFIG_` prefix is not used in C code. This is only needed in Makefiles or shell scripts.

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_FREEBSD` or `CONFIG_RTE_EXEC_ENV_LINUX` 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_linux, config/common_freebsd, 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.

2.3 PF and VF Considerations

The primary goal of DPDK is to provide a userspace dataplane. Managing VFs from a PF driver is a control plane feature and developers should generally rely on the Linux Kernel for that.

Developers should work with the Linux Kernel community to get the required functionality upstream. PF functionality should only be added to DPDK for testing and prototyping purposes while the kernel work is ongoing. It should also be marked with an "EXPERIMENTAL" tag. If the functionality isn't upstreamable then a case can be made to maintain the PF functionality in DPDK without the EXPERIMENTAL tag.

CHAPTER

THREE

ABI POLICY

3.1 Description

This document details the management policy that ensures the long-term stability of the DPDK ABI and API.

3.2 General Guidelines

1. Major ABI versions are declared no more frequently than yearly. Compatibility with the major ABI version is mandatory in subsequent releases until a new major ABI version is declared.
2. Major ABI versions are usually but not always declared aligned with a *LTS release*.
3. The ABI version is managed at a project level in DPDK, and is reflected in all non-experimental *library's soname*.
4. The ABI should be preserved and not changed lightly. ABI changes must follow the outlined *deprecation process*.
5. The addition of symbols is generally not problematic. The modification of symbols is managed with *ABI Versioning*.
6. The removal of symbols is considered an *ABI breakage*, once approved these will form part of the next ABI version.
7. Libraries or APIs marked as *experimental* may change without constraint, as they are not considered part of an ABI version. Experimental libraries have the major ABI version `0`.
8. Updates to the *minimum hardware requirements*, which drop support for hardware which was previously supported, should be treated as an ABI change.

Note: In 2019, the DPDK community stated its intention to move to ABI stable releases, over a number of release cycles. This change begins with maintaining ABI stability through one year of DPDK releases starting from DPDK 19.11. This policy will be reviewed in 2020, with intention of lengthening the stability period. Additional implementation detail can be found in the release notes.

3.2.1 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.

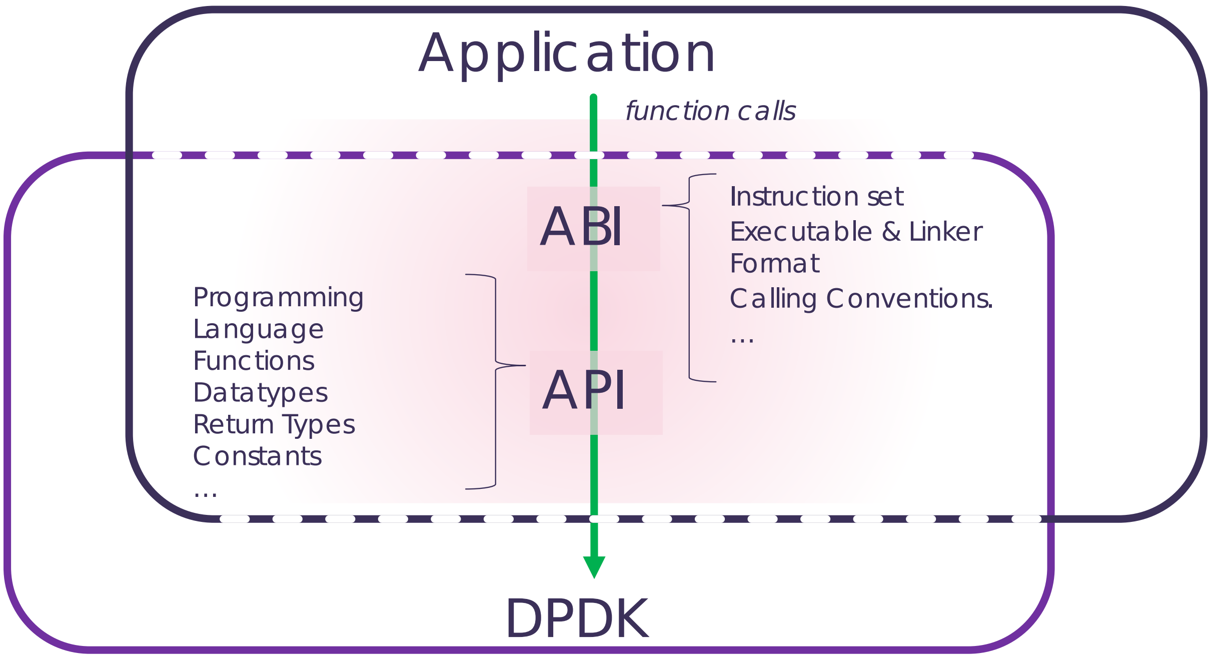


Fig. 3.1: Illustration of DPDK API and ABI.

3.2.2 What is an ABI version?

An ABI version is an instance of a library's ABI at a specific release. Certain releases are considered to be milestone releases, the yearly LTS release for example. The ABI of a milestone release may be declared as a 'major ABI version', where this ABI version is then supported for some number of subsequent releases and is annotated in the library's *soname*.

ABI version support in subsequent releases facilitates application upgrades, by enabling applications built against the milestone release to upgrade to subsequent releases of a library without a rebuild.

More details on major ABI version can be found in the *ABI versioning* guide.

3.3 The DPDK ABI policy

A new major ABI version is declared no more frequently than yearly, with declarations usually aligning with a LTS release, e.g. ABI 20 for DPDK 19.11. Compatibility with the major ABI version is then mandatory in subsequent releases until the next major ABI version is declared, e.g. ABI 21 for DPDK 20.11.

At the declaration of a major ABI version, major version numbers encoded in libraries' sonames are bumped to indicate the new version, with the minor version reset to `0`. An example would be `librte_eal.so.20.3` would become `librte_eal.so.21.0`.

The ABI may then change multiple times, without warning, between the last major ABI version increment and the HEAD label of the git tree, with the condition that ABI compatibility with the major ABI version is preserved and therefore sonames do not change.

Minor versions are incremented to indicate the release of a new ABI compatible DPDK release, typically the DPDK quarterly releases. An example of this, might be that `librte_eal.so.20.1` would indicate the first ABI compatible DPDK release, following the declaration of the new major ABI version `20`.

An ABI version is supported in all new releases until the next major ABI version is declared. When changing the major ABI version, the release notes will detail all ABI changes.

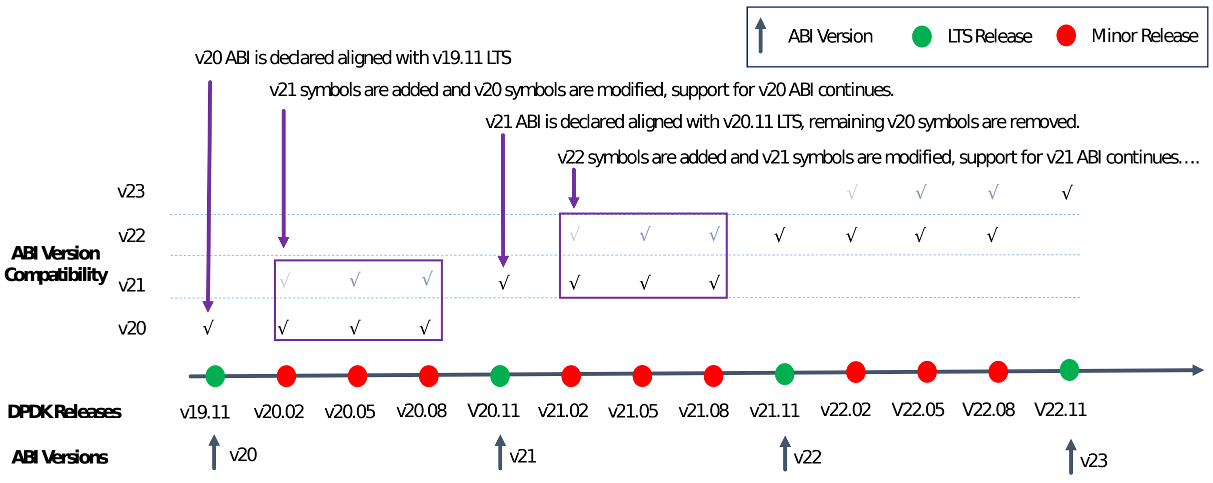

ABI Version
LTS Release
Minor Release
v20 ABI is declared aligned with v19.11 LTS
v21 symbols are added and v20 symbols are modified, support for v20 ABI continues.
v21 ABI is declared aligned with v20.11 LTS, remaining v20 symbols are removed.
v22 symbols are added and v21 symbols are modified, support for v21 ABI continues….
ABI Version Compatibility
v23
v22
v21
v20
DPDK Releases
v19.11
v20.02
v20.05
v20.08
V20.11
v21.02
v21.05
v21.08
v21.11
v22.02
V22.05
V22.08
V22.11
ABI Versions
v20
v21
v22
v23

Fig. 3.2: Mapping of new ABI versions and ABI version compatibility to DPDK releases.

3.3.1 ABI Changes

The ABI may still change after the declaration of a major ABI version, that is new APIs may be still added or existing APIs may be modified.

Warning: Note that, this policy details the method by which the ABI may be changed, with due regard to preserving compatibility and observing deprecation notices. This process however should not be undertaken lightly, as a general rule ABI stability is extremely important for downstream consumers of DPDK. The API should only be changed for significant reasons, such as performance enhancements. API breakages due to changes such as reorganizing public structure fields for aesthetic or readability purposes should be avoided.

The requirements for changing the ABI are:

1. At least 3 acknowledgments of the need to do so must be made on the dpdk.org mailing list.

   - The acknowledgment of the maintainer of the component is mandatory, or if no maintainer is available for the component, the tree/sub-tree maintainer for that component must acknowledge the ABI change instead.
   - The acknowledgment of three members of the technical board, as delegates of the technical board acknowledging the need for the ABI change, is also mandatory.

   - It is also recommended that acknowledgments from different "areas of interest" be sought for each deprecation, for example: from NIC vendors, CPU vendors, end-users, etc.

2. Backward compatibility with the major ABI version must be maintained through *ABI Versioning*, with *forward-only* compatibility offered for any ABI changes that are indicated to be part of the next ABI version.

   - In situations where backward compatibility is not possible, read the section on *ABI Breakages*.
   - No backward or forward compatibility is offered for API changes marked as `experimental`, as described in the section on *Experimental APIs and Libraries*.
   - In situations in which an `experimental` symbol has been stable for some time. When promoting the symbol to become part of the next ABI version, the maintainer may choose to provide an alias to the `experimental` tag, so as not to break consuming applications.

3. If a newly proposed API functionally replaces an existing one, when the new API becomes non-experimental, then the old one is marked with `__rte_deprecated`.

   - The depreciated API should follow the notification process to be removed, see *Examples of Deprecation Notices*.
   - At the declaration of the next major ABI version, those ABI changes then become a formal part of the new ABI and the requirement to preserve ABI compatibility with the last major ABI version is then dropped.
   - The responsibility for removing redundant ABI compatibility code rests with the original contributor of the ABI changes, failing that, then with the contributor's company and then finally with the maintainer.

Note: Note that forward-only compatibility is offered for those changes made between major ABI versions. As a library's soname can only describe compatibility with the last major ABI version, until the next major ABI version is declared, these changes therefore cannot be resolved as a runtime dependency through the soname. Therefore any application wishing to make use of these ABI changes can only ensure that its runtime dependencies are met through Operating System package versioning.

Note: Updates to the minimum hardware requirements, which drop support for hardware which was previously supported, should be treated as an ABI change, and follow the relevant deprecation policy procedures as above: 3 acks, technical board approval and announcement at least one release in advance.

3.3.2 ABI Breakages

For those ABI changes that are too significant to reasonably maintain multiple symbol versions, there is an amended process. In these cases, ABIs may be updated without the requirement of backward compatibility being provided. These changes must follow the same process *described above* as non-breaking changes, however with the following additional requirements:

1. ABI breaking changes (including an alternative map file) can be included with deprecation notice, in wrapped way by the `RTE_NEXT_ABI` option, to provide more details about oncoming changes. `RTE_NEXT_ABI` wrapper will be removed at the declaration of the next major ABI version.

2. Once approved, and after the deprecation notice has been observed these changes will form part of the next declared major ABI version.

3.3.3 Examples of ABI Changes

The following are examples of allowable ABI changes occurring between declarations of major ABI versions.

- DPDK 19.11 release defines the function `rte_foo()` ; `rte_foo()` is part of the major ABI version `20`.
- DPDK 20.02 release defines a new function `rte_foo(uint8_t bar)`. This is not a problem as long as the symbol `rte_foo@DPDK20` is preserved through *ABI Versioning*.
  - The new function may be marked with the `__rte_experimental` tag for a number of releases, as described in the section *Experimental*.
  - Once `rte_foo(uint8_t bar)` becomes non-experimental, `rte_foo()` is declared as `__rte_deprecated` and an deprecation notice is provided.
- DPDK 19.11 is not re-released to include `rte_foo(uint8_t bar)`, the new version of `rte_foo` only exists from DPDK 20.02 onwards as described in the *note on forward-only compatibility*.
- DPDK 20.02 release defines the experimental function `__rte_experimental rte_baz()`. This function may or may not exist in the DPDK 20.05 release.
- An application `dPacket` wishes to use `rte_foo(uint8_t bar)`, before the declaration of the DPDK `21` major ABI version. The application can only ensure its runtime dependencies are met by specifying `DPDK (>= 20.2)` as an explicit package dependency, as the soname can only indicate the supported major ABI version.
- At the release of DPDK 20.11, the function `rte_foo(uint8_t bar)` becomes formally part of then new major ABI version DPDK `21` and `rte_foo()` may be removed.

3.3.4 Examples of Deprecation Notices

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

- The Macro `#RTE_FOO` is deprecated and will be removed with ABI version 21, to be replaced with the inline function `rte_foo()`.
- The function `rte_mbuf_grok()` has been updated to include a new parameter in version 20.2. Backwards compatibility will be maintained for this function until the release of the new DPDK major ABI version 21, in DPDK version 20.11.
- The members of `struct rte_foo` have been reorganized in DPDK 20.02 for performance reasons. Existing binary applications will have backwards compatibility in release 20.02, while newly built binaries will need to reference the new structure variant `struct rte_foo2`. Compatibility will be removed in release 20.11, 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 20.02 will not contain these changes, but release 20.11 will, and no backwards compatibility is

planned due to the extensive nature of these changes. Binaries using this library built prior to ABI version 21 will require updating and recompilation.

3.4 Experimental

3.4.1 APIs

APIs marked as `experimental` are not considered part of an ABI version and may change without warning at any time. Since changes to APIs are most likely immediately after their introduction, as users begin to take advantage of those new APIs and start finding issues with them, new DPDK APIs will be automatically marked as `experimental` to allow for a period of stabilization before they become part of a tracked ABI version.

Note that marking an API as experimental is a multi step process. To mark an API as experimental, the symbols which are desired to be exported must be placed in an EXPERIMENTAL version block in the corresponding libraries' version map script. Secondly, the corresponding prototypes of those exported functions (in the development header files), must be marked with the `__rte_experimental` tag (see `rte_compat.h`). The DPDK build makefiles perform a check to ensure that the map file and the C code reflect the same list of symbols. This check can be circumvented by defining `ALLOW_EXPERIMENTAL_API` during compilation in the corresponding library Makefile.

In addition to tagging the code with `__rte_experimental`, the doxygen markup must also contain the EXPERIMENTAL string, and the MAINTAINERS file should note the EXPERIMENTAL libraries.

For removing the experimental tag associated with an API, deprecation notice is not required. Though, an API should remain in experimental state for at least one release. Thereafter, the normal process of posting patch for review to mailing list can be followed.

After the experimental tag has been formally removed, a tree/sub-tree maintainer may choose to offer an alias to the experimental tag so as not to break applications using the symbol. The alias is then dropped at the declaration of next major ABI version.

3.4.2 Libraries

Libraries marked as `experimental` are entirely not considered part of an ABI version, and may change without warning at any time. Experimental libraries always have a major ABI version of `0` to indicate they exist outside of *ABI Versioning* , with the minor version incremented with each ABI change to library.

CHAPTER

FOUR

ABI VERSIONING

This document details the mechanics of ABI version management in DPDK.

4.1 What is a library's soname?

System libraries usually adopt the familiar major and minor version naming convention, where major versions (e.g. `librte_eal 20.x,21.x`) are presumed to be ABI incompatible with each other and minor versions (e.g. `librte_eal 20.1,20.2`) are presumed to be ABI compatible. A library's soname. is typically used to provide backward compatibility information about a given library, describing the lowest common denominator ABI supported by the library. The soname or logical name for the library, is typically comprised of the library's name and major version e.g. `librte_eal.so.20`.

During an application's build process, a library's soname is noted as a runtime dependency of the application. This information is then used by the dynamic linker when resolving the applications dependencies at runtime, to load a library supporting the correct ABI version. The library loaded at runtime therefore, may be a minor revision supporting the same major ABI version (e.g. `librte_eal.20.2`), as the library used to link the application (e.g `librte_eal.20.0`).

4.2 Major ABI versions

An ABI version change to a given library, especially in core libraries such as `librte_mbuf`, may cause an implicit ripple effect on the ABI of it's consuming libraries, causing ABI breakages. There may however be no explicit reason to bump a dependent library's ABI version, as there may have been no obvious change to the dependent library's API, even though the library's ABI compatibility will have been broken.

This interdependence of DPDK libraries, means that ABI versioning of libraries is more manageable at a project level, with all project libraries sharing a **single ABI version**. In addition, the need to maintain a stable ABI for some number of releases as described in the section *ABI Policy*, means that ABI version increments need to carefully planned and managed at a project level.

Major ABI versions are therefore declared typically aligned with an LTS release and is then supported some number of subsequent releases, shared across all libraries. This means that a single project level ABI version, reflected in all individual library's soname, library filenames and associated version maps persists over multiple releases.

```
$ head ./lib/librte_acl/rte_acl_version.map
DPDK_20 {
        global:
...
```

```
$ head ./lib/librte_eal/rte_eal_version.map
DPDK_20 {
        global:
...
```

When an ABI change is made between major ABI versions to a given library, a new section is added to that library's version map describing the impending new ABI version, as described in the section *Examples of ABI Macro use*. The library's soname and filename however do not change, e.g. `libacl.so.20`, as ABI compatibility with the last major ABI version continues to be preserved for that library.

```
$ head ./lib/librte_acl/rte_acl_version.map
DPDK_20 {
        global:
...

DPDK_21 {
        global:

} DPDK_20;
...

$ head ./lib/librte_eal/rte_eal_version.map
DPDK_20 {
        global:
...
```

However when a new ABI version is declared, for example DPDK `21`, old depreciated functions may be safely removed at this point and the entire old major ABI version removed, see the section *Deprecating an entire ABI version* on how this may be done.

```
$ head ./lib/librte_acl/rte_acl_version.map
DPDK_21 {
        global:
...

$ head ./lib/librte_eal/rte_eal_version.map
DPDK_21 {
        global:
...
```

At the same time, the major ABI version is changed atomically across all libraries by incrementing the major version in the ABI_VERSION file. This is done globally for all libraries that declare a stable ABI. For libraries marked as EXPERIMENTAL, their major ABI version is always set to 0.

4.2.1 Minor ABI versions

Each non-LTS release will also increment minor ABI version, to permit multiple DPDK versions being installed alongside each other. Both stable and experimental ABI's are versioned using the global version file that is updated at the start of each release cycle, and are managed at the project level.

4.3 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 may be required to allow for backward compatibility for a time with older binaries that are dynamically linked to the DPDK.

To support backward compatibility the `rte_function_versioning.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 `be`.
- `BIND_DEFAULT_SYMBOL(b, e, n)`: Creates a symbol version entry instructing the linker to bind references to symbol `b` to the internal symbol `be`.
- `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.
- `__vsym`: Annotation to be used in a declaration of the internal symbol `be` to signal that it is being used as an implementation of a particular version of symbol `b`.
- `VERSION_SYMBOL_EXPERIMENTAL(b, e)`: Creates a symbol version table entry binding versioned symbol `b@EXPERIMENTAL` to the internal function `be`. The macro is used when a symbol matures to become part of the stable ABI, to provide an alias to experimental for some time.

4.3.1 Examples of ABI Macro use

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_20 {
     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_20 {
     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_21 {
     global:
     rte_acl_create;

} DPDK_20;
```

The addition of the new block tells the linker that a new version node `DPDK_21` is available, which contains the symbol rte_acl_create, and inherits the symbols from the DPDK_20 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 maps 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)
+struct rte_acl_ctx * __vsym
+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 conducive to the macros used for versioning symbols and we have annotated the function as `__vsym`, an implementation of a versioned symbol . That is our next step, mapping this new symbol name to the initial symbol name at version node 20. Immediately after the function, we add the VERSION_SYMBOL macro.

```
#include <rte_function_versioning.h>

...
VERSION_SYMBOL(rte_acl_create, _v20, 20);
```

Remembering to also add the rte_function_versioning.h header to the requisite c file where these changes are being made. The macro instructs the linker to create a new symbol `rte_acl_create@DPDK_20`, 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).

Please see the section *Enabling versioning macros* to enable this macro in the meson/ninja build. Next, we need to create the new `v21` version of the symbol. We create a new function name, with the `v21` suffix, and implement it appropriately.

```
struct rte_acl_ctx * __vsym
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 new default symbol `rte_acl_create@DPDK_21`. To do this, immediately after the function, we add the BIND_DEFAULT_SYMBOL macro.

```
#include <rte_function_versioning.h>

...
BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 21);
```

The macro instructs the linker to create the new default symbol `rte_acl_create@DPDK_21`, which points to the above newly named function.

We finally modify the prototype of the call in the public header file, such that it contains both versions of the symbol and the public API.

```
struct rte_acl_ctx *
rte_acl_create(const struct rte_acl_param *param);

struct rte_acl_ctx * __vsym
rte_acl_create_v20(const struct rte_acl_param *param);
```

```
struct rte_acl_ctx * __vsym
rte_acl_create_v21(const struct rte_acl_param *param, int debug);
```

And that's it, on the next shared library rebuild, there will be two versions of rte_acl_create, an old DPDK_20 version, used by previously built applications, and a new DPDK_21 version, used by future built applications.

Note: **Before you leave**, please take care reviewing the sections on *mapping static symbols*, *enabling versioning macros*, and *ABI deprecation*.

Mapping static symbols

Now 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_ctx * __vsym
rte_acl_create_v21(const struct rte_acl_param *param, int debug)
{
     ...
}
MAP_STATIC_SYMBOL(struct rte_acl_ctx *rte_acl_create(const struct rte_acl_param *param, int deb
```

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`

Enabling versioning macros

Finally, we need to indicate to the meson/ninja build system to enable versioning macros when building the library or driver. In the libraries or driver where we have added symbol versioning, in the `meson.build` file we add the following

```
use_function_versioning = true
```

at the start of the head of the file. This will indicate to the tool-chain to enable the function version macros when building. There is no corresponding directive required for the `make` build system.

Aliasing experimental symbols

In situations in which an `experimental` symbol has been stable for some time, and it becomes a candidate for promotion to the stable ABI. At this time, when promoting the symbol, maintainer may choose to provide an alias to the `experimental` symbol version, so as not to break consuming applications.

The process to provide an alias to `experimental` is similar to that, of *symbol versioning* described above. Assume we have an experimental function `rte_acl_create` as follows:

```
#include <rte_compat.h>

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

In the map file, experimental symbols are listed as part of the `EXPERIMENTAL` version node.

```
DPDK_20 {
     global:
     ...

     local: *;
};

EXPERIMENTAL {
     global:

     rte_acl_create;
};
```

When we promote the symbol to the stable ABI, we simply strip the `__rte_experimental` annotation from the function and move the symbol from the `EXPERIMENTAL` node, to the node of the next major ABI version as follow.

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

We then update the map file, adding the symbol `rte_acl_create` to the `DPDK_21` version node.

```
DPDK_20 {
     global:
     ...

     local: *;
};

DPDK_21 {
     global:

     rte_acl_create;
} DPDK_20;
```

Although there are strictly no guarantees or commitments associated with *experimental symbols*, a maintainer may wish to offer an alias to experimental. The process to add an alias to experimental, is similar to the symbol versioning process. Assuming we have an experimental symbol as before, we now add the symbol to both the `EXPERIMENTAL` and `DPDK_21` version nodes.

```
#include <rte_compat.h>;
#include <rte_function_versioning.h>
```

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

__rte_experimental
struct rte_acl_ctx *
rte_acl_create_e(const struct rte_acl_param *param)
{
   return rte_acl_create(param);
}
VERSION_SYMBOL_EXPERIMENTAL(rte_acl_create, _e);

struct rte_acl_ctx *
rte_acl_create_v21(const struct rte_acl_param *param)
{
   return rte_acl_create(param);
}
BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 21);
```

In the map file, we map the symbol to both the `EXPERIMENTAL` and `DPDK_21` version nodes.

```
DPDK_20 {
     global:
     ...

     local: *;
};

DPDK_21 {
     global:

     rte_acl_create;
} DPDK_20;

EXPERIMENTAL {
     global:

     rte_acl_create;
};
```

Note: Please note, similar to *symbol versioning*, when aliasing to experimental you will also need to take care of *mapping static symbols*.

Deprecating part of a public API

Lets assume that you've done the above updates, and in preparation for the next major ABI version 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_20 {
     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_21 {
       global:
       rte_acl_create;
  } DPDK_20;
```

Next remove the corresponding versioned export.

```
-VERSION_SYMBOL(rte_acl_create, _v20, 20);
```

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 and declare it as static. This is a coding style choice.

Deprecating an entire ABI version

While removing a symbol from an ABI may be useful, it is more practical to remove an entire version node at once, as is typically done at the declaration of a major ABI version. 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_21 {
     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, 20);
+BIND_DEFAULT_SYMBOL(rte_acl_create, _v21, 21);
```

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.

4.4 Running the ABI Validator

The `devtools` directory in the DPDK source tree contains a utility program, `check-abi.sh`, for validating the DPDK ABI based on the libabigail abidiff utility.

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

```
devtools/check-abi.sh <refdir> <newdir>
```

Where <refdir> specifies the directory housing the reference build of DPDK, and <newdir> specifies the DPDK build directory to check the ABI of.

The ABI compatibility is automatically verified when using a build script from `devtools`, if the variable `DPDK_ABI_REF_VERSION` is set with a tag, as described in *ABI check recommendations*.

CHAPTER

FIVE

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.

5.1 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_eal
|   |-- ...
...
doc
|-- api
+-- guides
    |-- freebsd_gsg
    |-- linux_gsg
    |-- prog_guide
    |-- sample_app_ug
    |-- guidelines
    |-- testpmd_app_ug
    |-- rel_notes
    |-- nics
    |-- ...
```

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.

5.2 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 release.

  Developers should include updates to the Release Notes with patch sets that relate to any of the following sections:

  - New Features
  - Resolved Issues (see below)
  - Known Issues
  - API Changes
  - ABI Changes
  - Shared Library Versions

  Resolved Issues should only include issues from previous releases that have been resolved in the current release. Issues that are introduced and then fixed within a release cycle do not have to be included here.

  Refer to the Release Notes from the previous DPDK release for the correct format of each section.

- **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.

5.3 Building the Documentation

5.3.1 Dependencies

The following dependencies must be installed to build the documentation:

- Doxygen.
- Sphinx (also called python-sphinx).
- TexLive (at least TexLive-core and the extra Latex support).
- 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 dnf     -y install doxygen
```

Sphinx is a Python documentation tool for converting RST files to Html or to PDF (via LaTeX). For full support with figure and table captioning the latest version of Sphinx can be installed as follows:

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

# Red Hat/Fedora.
sudo dnf     -y install python-pip
sudo pip install --upgrade sphinx
sudo pip install --upgrade sphinx_rtd_theme
```

For further information on getting started with Sphinx see the Sphinx Getting Started.

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 dnf     -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-lang-greek

# Red Hat/Fedora, selective install.
sudo dnf     -y install texlive-collection-latexextra texlive-greek-fontenc
```

Latexmk is a perl script for running LaTeX for resolving cross references, and it also runs auxiliary programs like bibtex, makeindex if necessary, and dvips. It has also a number of other useful capabilities (see man 1 latexmk).

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

# Red Hat/Fedora.
sudo dnf     -y install latexmk
```

5.3.2 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
```

5.4 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
  ```

5.5 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.

5.5.1 Line Length

- Lines in sentences should be less than 80 characters and wrapped at words. Multiple sentences which are not separated by a blank line are joined automatically into paragraphs.
- Lines in literal blocks **must** be less than 80 characters since they are not wrapped by the document formatters and can exceed the page width in PDF documents.

  Long literal command lines can be shown wrapped with backslashes. For example:

  ```
  testpmd -l 2-3 -n 4 \
          --vdev=virtio_user0,path=/dev/vhost-net,queues=2,queue_size=1024 \
          -- -i --tx-offloads=0x0000002c --enable-lro --txq=2 --rxq=2 \
          --txd=1024 --rxd=1024
  ```

5.5.2 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.

5.5.3 Section Headers

- Section headers should 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.

5.5.4 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 previous line.

  #. Item three.
  ```

- 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.

5.5.5 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 -l 0-2 -n3 --vdev=net_pcap0,iface=eth0     \
                            --vdev=net_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.

5.5.6 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-enqueue1.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.*

5.5.7 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.

5.5.8 Hyperlinks

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

```
https://dpdk.org
```

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

```
`Check out DPDK <https://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.

5.6 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:

  ```
  /**
   * Try to take the lock.
   *
   * @param sl
   *   A pointer to the spinlock.
   * @return
   *   1 if the lock is successfully taken; 0 otherwise.
   */
  int rte_spinlock_trylock(rte_spinlock_t *sl);
  ```

- 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 `net_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.

CHAPTER

SIX

CONTRIBUTING CODE TO DPDK

This document outlines the guidelines for submitting code to DPDK.

The DPDK development process is modeled (loosely) on the Linux Kernel development model so it is worth reading the Linux kernel guide on submitting patches: How to Get Your Change Into the Linux Kernel. The rationale for many of the DPDK guidelines is explained in greater detail in the kernel guidelines.

6.1 The DPDK Development Process

The DPDK development process has the following features:

- The code is hosted in a public git repository.
- There is a mailing list where developers submit patches.
- There are maintainers for hierarchical components.
- Patches are reviewed publicly on the mailing list.
- Successfully reviewed patches are merged to the repository.
- Patches should be sent to the target repository or sub-tree, see below.
- All sub-repositories are merged into main repository for `-rc1` and `-rc2` versions of the release.
- After the `-rc2` release all patches should target the main repository.

The mailing list for DPDK development is dev@dpdk.org. Contributors will need to register for the mailing list in order to submit patches. It is also worth registering for the DPDK Patchwork

If you are using the GitHub service, you can link your repository to the `travis-ci.org` build service. When you push patches to your GitHub repository, the travis service will automatically build your changes.

The development process requires some familiarity with the `git` version control system. Refer to the Pro Git Book for further information.

6.2 Source License

The DPDK uses the Open Source BSD-3-Clause license for the core libraries and drivers. The kernel components are GPL-2.0 licensed. DPDK uses single line reference to Unique License Identifiers in source files as defined by the Linux Foundation's SPDX project.

DPDK uses first line of the file to be SPDX tag. In case of *#!* scripts, SPDX tag can be placed in 2nd line of the file.

For example, to label a file as subject to the BSD-3-Clause license, the following text would be used:

`SPDX-License-Identifier: BSD-3-Clause`

To label a file as dual-licensed with BSD-3-Clause and GPL-2.0 (e.g., for code that is shared between the kernel and userspace), the following text would be used:

`SPDX-License-Identifier: (BSD-3-Clause OR GPL-2.0)`

Refer to `licenses/README` for more details.

6.3 Maintainers and Sub-trees

The DPDK maintenance hierarchy is divided into a main repository `dpdk` and sub-repositories `dpdk-next-*`.

There are maintainers for the trees and for components within the tree.

Trees and maintainers are listed in the `MAINTAINERS` file. For example:

```
Crypto Drivers
--------------
M: Some Name <some.name@email.com>
T: git://dpdk.org/next/dpdk-next-crypto

Intel AES-NI GCM PMD
M: Some One <some.one@email.com>
F: drivers/crypto/aesni_gcm/
F: doc/guides/cryptodevs/aesni_gcm.rst
```

Where:

- `M` is a tree or component maintainer.
- `T` is a repository tree.
- `F` is a maintained file or directory.

Additional details are given in the `MAINTAINERS` file.

The role of the component maintainers is to:

- Review patches for the component or delegate the review. The review should be done, ideally, within 1 week of submission to the mailing list.
- Add an `acked-by` to patches, or patchsets, that are ready for committing to a tree.
- Reply to questions asked about the component.

Component maintainers can be added or removed by submitting a patch to the `MAINTAINERS` file. Maintainers should have demonstrated a reasonable level of contributions or reviews to the component area. The maintainer should be confirmed by an `ack` from an established contributor. There can be more than one component maintainer if desired.

The role of the tree maintainers is to:

- Maintain the overall quality of their tree. This can entail additional review, compilation checks or other tests deemed necessary by the maintainer.

- Commit patches that have been reviewed by component maintainers and/or other contributors. The tree maintainer should determine if patches have been reviewed sufficiently.
- Ensure that patches are reviewed in a timely manner.
- Prepare the tree for integration.
- Ensure that there is a designated back-up maintainer and coordinate a handover for periods where the tree maintainer can't perform their role.

Tree maintainers can be added or removed by submitting a patch to the `MAINTAINERS` file. The proposer should justify the need for a new sub-tree and should have demonstrated a sufficient level of contributions in the area or to a similar area. The maintainer should be confirmed by an `ack` from an existing tree maintainer. Disagreements on trees or maintainers can be brought to the Technical Board.

The backup maintainer for the master tree should be selected from the existing sub-tree maintainers from the project. The backup maintainer for a sub-tree should be selected from among the component maintainers within that sub-tree.

6.4 Getting the Source Code

The source code can be cloned using either of the following:

main repository:

```
git clone git://dpdk.org/dpdk
git clone https://dpdk.org/git/dpdk
```

sub-repositories (list):

```
git clone git://dpdk.org/next/dpdk-next-*
git clone https://dpdk.org/git/next/dpdk-next-*
```

6.5 Make your Changes

Make your planned changes in the cloned `dpdk` repo. Here are some guidelines and requirements:

- Follow the *DPDK Coding Style* guidelines.
- If you add new files or directories you should add your name to the `MAINTAINERS` file.
- Initial submission of new PMDs should be prepared against a corresponding repo.
  - Thus, for example, initial submission of a new network PMD should be prepared against dpdk-next-net repo.
  - Likewise, initial submission of a new crypto or compression PMD should be prepared against dpdk-next-crypto repo.
  - For other PMDs and more info, refer to the `MAINTAINERS` file.
- New external functions should be added to the local `version.map` file. See the *ABI policy* and *ABI versioning* guides. New external functions should also be added in alphabetical order.
- Important changes will require an addition to the release notes in `doc/guides/rel_notes/`. See the *Release Notes section of the Documentation Guidelines* for details.

- Test the compilation works with different targets, compilers and options, see *Checking Compilation*.
- Don't break compilation between commits with forward dependencies in a patchset. Each commit should compile on its own to allow for `git bisect` and continuous integration testing.
- Add tests to the `app/test` unit test framework where possible.
- Add documentation, if relevant, in the form of Doxygen comments or a User Guide in RST format. See the *Documentation Guidelines*.

Once the changes have been made you should commit them to your local repo.

For small changes, that do not require specific explanations, it is better to keep things together in the same patch. Larger changes that require different explanations should be separated into logical patches in a patchset. A good way of thinking about whether a patch should be split is to consider whether the change could be applied without dependencies as a backport.

It is better to keep the related documentation changes in the same patch file as the code, rather than one big documentation patch at the end of a patchset. This makes it easier for future maintenance and development of the code.

As a guide to how patches should be structured run `git log` on similar files.

6.6 Commit Messages: Subject Line

The first, summary, line of the git commit message becomes the subject line of the patch email. Here are some guidelines for the summary line:

- The summary line must capture the area and the impact of the change.
- The summary line should be around 50 characters.
- The summary line should be lowercase apart from acronyms.
- It should be prefixed with the component name (use git log to check existing components). For example:

  ```
  ixgbe: fix offload config option name

  config: increase max queues per port
  ```
- Use the imperative of the verb (like instructions to the code base).
- Don't add a period/full stop to the subject line or you will end up two in the patch name: `dpdk_description..patch`.

The actual email subject line should be prefixed by `[PATCH]` and the version, if greater than v1, for example: `PATCH v2`. The is generally added by `git send-email` or `git format-patch`, see below.

If you are submitting an RFC draft of a feature you can use `[RFC]` instead of `[PATCH]`. An RFC patch doesn't have to be complete. It is intended as a way of getting early feedback.

6.7 Commit Messages: Body

Here are some guidelines for the body of a commit message:

- The body of the message should describe the issue being fixed or the feature being added. It is important to provide enough information to allow a reviewer to understand the purpose of the patch.
- When the change is obvious the body can be blank, apart from the signoff.
- The commit message must end with a `Signed-off-by:` line which is added using:

  ```
  git commit --signoff # or -s
  ```

  The purpose of the signoff is explained in the Developer's Certificate of Origin section of the Linux kernel guidelines.

  Note: All developers must ensure that they have read and understood the Developer's Certificate of Origin section of the documentation prior to applying the signoff and submitting a patch.

- The signoff must be a real name and not an alias or nickname. More than one signoff is allowed.
- The text of the commit message should be wrapped at 72 characters.
- When fixing a regression, it is required to reference the id of the commit which introduced the bug, and put the original author of that commit on CC. You can generate the required lines using the following git alias, which prints the commit SHA and the author of the original code:

  ```
  git config alias.fixline "log -1 --abbrev=12 --format='Fixes: %h (\"%s\")%nCc: %ae'"
  ```

  The output of `git fixline <SHA>` must then be added to the commit message:

  ```
  doc: fix some parameter description

  Update the docs, fixing description of some parameter.

  Fixes: abcdefgh1234 ("doc: add some parameter")
  Cc: author@example.com

  Signed-off-by: Alex Smith <alex.smith@example.com>
  ```

- When fixing an error or warning it is useful to add the error message and instructions on how to reproduce it.
- Use correct capitalization, punctuation and spelling.

In addition to the `Signed-off-by:` name the commit messages can also have tags for who reported, suggested, tested and reviewed the patch being posted. Please refer to the *Tested, Acked and Reviewed by* section.

6.7.1 Patch Fix Related Issues

Coverity is a tool for static code analysis. It is used as a cloud-based service used to scan the DPDK source code, and alert developers of any potential defects in the source code. When fixing an issue found by Coverity, the patch must contain a Coverity issue ID in the body of the commit message. For example:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Coverity issue: 12345
Fixes: abcdefgh1234 ("doc: add some parameter")
```

```
Cc: author@example.com

Signed-off-by: Alex Smith <alex.smith@example.com>
```

Bugzilla is a bug- or issue-tracking system. Bug-tracking systems allow individual or groups of developers effectively to keep track of outstanding problems with their product. When fixing an issue raised in Bugzilla, the patch must contain a Bugzilla issue ID in the body of the commit message. For example:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Bugzilla ID: 12345
Fixes: abcdefgh1234 ("doc: add some parameter")
Cc: author@example.com

Signed-off-by: Alex Smith <alex.smith@example.com>
```

6.7.2 Patch for Stable Releases

All fix patches to the master branch that are candidates for backporting should also be CCed to the stable@dpdk.org mailing list. In the commit message body the Cc: stable@dpdk.org should be inserted as follows:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Fixes: abcdefgh1234 ("doc: add some parameter")
Cc: stable@dpdk.org

Signed-off-by: Alex Smith <alex.smith@example.com>
```

For further information on stable contribution you can go to *Stable Contribution Guide*.

6.8 Creating Patches

It is possible to send patches directly from git but for new contributors it is recommended to generate the patches with `git format-patch` and then when everything looks okay, and the patches have been checked, to send them with `git send-email`.

Here are some examples of using `git format-patch` to generate patches:

```
# Generate a patch from the last commit.
git format-patch -1

# Generate a patch from the last 3 commits.
git format-patch -3

# Generate the patches in a directory.
git format-patch -3 -o ~/patch/

# Add a cover letter to explain a patchset.
git format-patch -3 -o ~/patch/ --cover-letter

# Add a prefix with a version number.
git format-patch -3 -o ~/patch/ -v 2
```

Cover letters are useful for explaining a patchset and help to generate a logical threading to the patches. Smaller notes can be put inline in the patch after the `---` separator, for example:

```
Subject: [PATCH] fm10k/base: add FM10420 device ids

Add the device ID for Boulder Rapids and Atwood Channel to enable
drivers to support those devices.

Signed-off-by: Alex Smith <alex.smith@example.com>
---

ADD NOTES HERE.

 drivers/net/fm10k/base/fm10k_api.c  | 6 ++++++
 drivers/net/fm10k/base/fm10k_type.h | 6 ++++++
 2 files changed, 12 insertions(+)
...
```

Version 2 and later of a patchset should also include a short log of the changes so the reviewer knows what has changed. This can be added to the cover letter or the annotations. For example:

```
---
v3:
* Fixed issued with version.map.

v2:
* Added i40e support.
* Renamed ethdev functions from rte_eth_ieee15888_*() to rte_eth_timesync_*()
  since 802.1AS can be supported through the same interfaces.
```

6.9 Checking the Patches

Patches should be checked for formatting and syntax issues using the `checkpatches.sh` script in the `devtools` directory of the DPDK repo. This uses the Linux kernel development tool `checkpatch.pl` which can be obtained by cloning, and periodically, updating the Linux kernel sources.

The path to the original Linux script must be set in the environment variable `DPDK_CHECKPATCH_PATH`.

Spell checking of commonly misspelled words can be enabled by downloading the codespell dictionary:

```
https://raw.githubusercontent.com/codespell-project/codespell/master/codespell_lib/data/dictior
```

The path to the downloaded `dictionary.txt` must be set in the environment variable `DPDK_CHECKPATCH_CODESPELL`.

Environment variables required by the development tools, are loaded from the following files, in order of preference:

```
.develconfig
~/.config/dpdk/devel.config
/etc/dpdk/devel.config.
```

Once the environment variable is set, the script can be run as follows:

```
devtools/checkpatches.sh ~/patch/
```

The script usage is:

```
checkpatches.sh [-h] [-q] [-v] [patch1 [patch2] ...]]"
```

Where:

- `-h`: help, usage.
- `-q`: quiet. Don't output anything for files without issues.
- `-v`: verbose.
- `patchX`: path to one or more patches.

Then the git logs should be checked using the `check-git-log.sh` script.

The script usage is:

```
check-git-log.sh [range]
```

Where the range is a `git log` option.

6.10 Checking Compilation

6.10.1 Makefile System

Compilation of patches and changes should be tested using the `test-build.sh` script in the `devtools` directory of the DPDK repo:

```
devtools/test-build.sh x86_64-native-linux-gcc+next+shared
```

The script usage is:

```
test-build.sh [-h] [-jX] [-s] [config1 [config2] ...]]
```

Where:

- `-h`: help, usage.
- `-jX`: use X parallel jobs in "make".
- `-s`: short test with only first config and without examples/doc.
- `config`: default config name plus config switches delimited with a + sign.

Examples of configs are:

```
x86_64-native-linux-gcc
x86_64-native-linux-gcc+next+shared
x86_64-native-linux-clang+shared
```

The builds can be modified via the following environmental variables:

- `DPDK_BUILD_TEST_CONFIGS` (target1+option1+option2 target2)
- `DPDK_BUILD_TEST_DIR`
- `DPDK_DEP_CFLAGS`
- `DPDK_DEP_LDFLAGS`
- `DPDK_DEP_PCAP` (y/[n])
- `DPDK_NOTIFY` (notify-send)

These can be set from the command line or in the config files shown above in the *Checking the Patches*.

The recommended configurations and options to test compilation prior to submitting patches are:

```
x86_64-native-linux-gcc+shared+next
x86_64-native-linux-clang+shared
i686-native-linux-gcc

export DPDK_DEP_ZLIB=y
export DPDK_DEP_PCAP=y
export DPDK_DEP_SSL=y
```

6.10.2 Meson System

Compilation of patches is to be tested with `devtools/test-meson-builds.sh` script.

The script internally checks for dependencies, then builds for several combinations of compilation configuration. By default, each build will be put in a subfolder of the current working directory. However, if it is preferred to place the builds in a different location, the environment variable `DPDK_BUILD_TEST_DIR` can be set to that desired location. For example, setting `DPDK_BUILD_TEST_DIR=__builds` will put all builds in a single subfolder called "__builds" created in the current directory. Setting `DPDK_BUILD_TEST_DIR` to an absolute directory path e.g. `/tmp` is also supported.

6.11 Checking ABI compatibility

By default, ABI compatibility checks are disabled.

To enable them, a reference version must be selected via the environment variable `DPDK_ABI_REF_VERSION`.

The `devtools/test-build.sh` and `devtools/test-meson-builds.sh` scripts then build this reference version in a temporary directory and store the results in a subfolder of the current working directory. The environment variable `DPDK_ABI_REF_DIR` can be set so that the results go to a different location.

6.12 Sending Patches

Patches should be sent to the mailing list using `git send-email`. You can configure an external SMTP with something like the following:

```
[sendemail]
    smtpuser = name@domain.com
    smtpserver = smtp.domain.com
    smtpserverport = 465
    smtpencryption = ssl
```

See the Git send-email documentation for more details.

The patches should be sent to `dev@dpdk.org`. If the patches are a change to existing files then you should send them TO the maintainer(s) and CC `dev@dpdk.org`. The appropriate maintainer can be found in the `MAINTAINERS` file:

```
git send-email --to maintainer@some.org --cc dev@dpdk.org 000*.patch
```

Script `get-maintainer.sh` can be used to select maintainers automatically:

```
git send-email --to-cmd ./devtools/get-maintainer.sh --cc dev@dpdk.org 000*.patch
```

New additions can be sent without a maintainer:

```
git send-email --to dev@dpdk.org 000*.patch
```

You can test the emails by sending it to yourself or with the `--dry-run` option.

If the patch is in relation to a previous email thread you can add it to the same thread using the Message ID:

```
git send-email --to dev@dpdk.org --in-reply-to <1234-foo@bar.com> 000*.patch
```

The Message ID can be found in the raw text of emails or at the top of each Patchwork patch, for example. Shallow threading (`--thread --no-chain-reply-to`) is preferred for a patch series.

Once submitted your patches will appear on the mailing list and in Patchwork.

Experienced committers may send patches directly with `git send-email` without the `git format-patch` step. The options `--annotate` and `confirm = always` are recommended for checking patches before sending.

6.12.1 Backporting patches for Stable Releases

Sometimes a maintainer or contributor wishes, or can be asked, to send a patch for a stable release rather than mainline. In this case the patch(es) should be sent to `stable@dpdk.org`, not to `dev@dpdk.org`.

Given that there are multiple stable releases being maintained at the same time, please specify exactly which branch(es) the patch is for using `git send-email --subject-prefix='PATCH 16.11' ...` and also optionally in the cover letter or in the annotation.

6.13 The Review Process

Patches are reviewed by the community, relying on the experience and collaboration of the members to double-check each other’s work. There are a number of ways to indicate that you have checked a patch on the mailing list.

6.13.1 Tested, Acked and Reviewed by

To indicate that you have interacted with a patch on the mailing list you should respond to the patch in an email with one of the following tags:

- Reviewed-by:
- Acked-by:
- Tested-by:
- Reported-by:
- Suggested-by:

The tag should be on a separate line as follows:

```
tag-here: Name Surname <email@address.com>
```

Each of these tags has a specific meaning. In general, the DPDK community follows the kernel usage of the tags. A short summary of the meanings of each tag is given here for reference:

`Reviewed-by:` is a strong statement that the patch is an appropriate state for merging without any remaining serious technical issues. Reviews from community members who are known to understand the subject area and to perform thorough reviews will increase the likelihood of the patch getting merged.

`Acked-by:` is a record that the person named was not directly involved in the preparation of the patch but wishes to signify and record their acceptance and approval of it.

`Tested-by:` indicates that the patch has been successfully tested (in some environment) by the person named.

`Reported-by:` is used to acknowledge person who found or reported the bug.

`Suggested-by:` indicates that the patch idea was suggested by the named person.

6.13.2 Steps to getting your patch merged

The more work you put into the previous steps the easier it will be to get a patch accepted. The general cycle for patch review and acceptance is:

1. Submit the patch.
2. Check the automatic test reports in the coming hours.
3. Wait for review comments. While you are waiting review some other patches.
4. Fix the review comments and submit a `v n+1` patchset:

   ```
   git format-patch -3 -v 2
   ```

5. Update Patchwork to mark your previous patches as "Superseded".
6. If the patch is deemed suitable for merging by the relevant maintainer(s) or other developers they will `ack` the patch with an email that includes something like:

   ```
   Acked-by: Alex Smith <alex.smith@example.com>
   ```

   Note: When acking patches please remove as much of the text of the patch email as possible. It is generally best to delete everything after the `Signed-off-by:` line.
7. Having the patch `Reviewed-by:` and/or `Tested-by:` will also help the patch to be accepted.
8. If the patch isn't deemed suitable based on being out of scope or conflicting with existing functionality it may receive a `nack`. In this case you will need to make a more convincing technical argument in favor of your patches.
9. In addition a patch will not be accepted if it doesn't address comments from a previous version with fixes or valid arguments.
10. It is the responsibility of a maintainer to ensure that patches are reviewed and to provide an `ack` or `nack` of those patches as appropriate.
11. Once a patch has been acked by the relevant maintainer, reviewers may still comment on it for a further two weeks. After that time, the patch should be merged into the relevant git tree for the next release. Additional notes and restrictions:

    - Patches should be acked by a maintainer at least two days before the release merge deadline, in order to make that release.

- For patches acked with less than two weeks to go to the merge deadline, all additional comments should be made no later than two days before the merge deadline.
- After the appropriate time for additional feedback has passed, if the patch has not yet been merged to the relevant tree by the committer, it should be treated as though it had, in that any additional changes needed to it must be addressed by a follow-on patch, rather than rework of the original.
- Trivial patches may be merged sooner than described above at the tree committer's discretion.

CHAPTER

SEVEN

DPDK VULNERABILITY MANAGEMENT PROCESS

7.1 Scope

Only the main repositories (dpdk and dpdk-stable) of the core project are in the scope of this security process (including experimental APIs). If a stable branch is declared unmaintained (end of life), no fix will be applied.

All vulnerabilities are bugs, but not every bug is a vulnerability. Vulnerabilities compromise one or more of:

- Confidentiality (personal or corporate confidential data).
- Integrity (trustworthiness and correctness).
- Availability (uptime and service).

If in doubt, please consider the vulnerability as security sensitive. At worst, the response will be to report the bug through the usual channels.

7.2 Finding

There is no pro-active security engineering effort at the moment.

Please report any security issue you find in DPDK as described below.

7.3 Report

Do not use Bugzilla (unsecured). Instead, send GPG-encrypted emails to security@dpdk.org. Anyone can post to this list. In order to reduce the disclosure of a vulnerability in the early stages, membership of this list is intentionally limited to a small number of people.

It is additionally encouraged to GPG-sign one-on-one conversations as part of the security process.

As it is with any bug, the more information provided, the easier it will be to diagnose and fix. If you already have a fix, please include it with your report, as that can speed up the process considerably.

In the report, please note how you would like to be credited for discovering the issue and the details of any embargo you would like to impose.

If the vulnerability is not public yet, no patch or information should be disclosed publicly. If a fix is already published, the reporting process must be followed anyway, as described below.

7.4 Confirmation

Upon reception of the report, a security team member should reply to the reporter acknowledging that the report has been received.

The DPDK security team reviews the security vulnerability reported. Area experts not members of the security team may be involved in the process. In case the reported issue is not qualified as a security vulnerability, the security team will request the submitter to report it using the usual channel (Bugzilla). If qualified, the security team will assess which DPDK version are affected. A bugzilla ID (allocated in a reserved pool) is assigned to the vulnerability, and kept empty until public disclosure.

The security team calculates the severity score with CVSS calculator based on inputs from the reporter and its own assessment of the vulnerability, and agrees on the score with the reporter.

An embargo may be put in place depending on the severity of the vulnerability. If an embargo is decided, its duration should be suggested by the security team and negotiated with the reporter. Embargo duration between vulnerability confirmation and public disclosure should be between **one and ten weeks**. If an embargo is not required, the vulnerability may be fixed using the standard patch process, once a CVE number has been assigned.

The confirmation mail should be sent within **3 business days**.

Following information must be included in the mail:

- Confirmation
- CVSS severity and score
- Embargo duration
- Reporter credit
- Bug ID (empty and restricted for future reference)

7.5 CVE Request

The security team develops a security advisory document. The security team may, at its discretion, include the reporter (via "CC") in developing the security advisory document, but in any case should accept feedback from the reporter before finalizing the document. When the document is final, the security team needs to request a CVE identifier from a CNA.

The CVE request should be sent to secalert@redhat.com using GPG encrypted email (see contact details).

7.5.1 CVE Request Template with Embargo

```
A vulnerability was discovered in the DPDK project.
In order to ensure full traceability, we need a CVE number assigned
that we can attach to private and public notifications.
Please treat the following information as confidential during the embargo
until further public disclosure.

[PRODUCT]:
[VERSION]:
[PROBLEMTYPE]:
[SEVERITY]:
```

```
[REFERENCES]: { bug_url }
[DESCRIPTION]:

Thanks
{ DPDK_security_team_member }, on behalf of the DPDK security team
```

7.5.2 CVE Request Template without Embargo

```
A vulnerability was discovered in the DPDK project.
In order to ensure full traceability, we need a CVE number assigned
that we can attach to private and public notifications.

[PRODUCT]:
[VERSION]:
[PROBLEMTYPE]:
[SEVERITY]:
[REFERENCES]: { bug_url }
[DESCRIPTION]:

Thanks
{ DPDK_security_team_member }, on behalf of the DPDK security team
```

7.6 Fix Development and Review

If the fix is already published, this step is skipped, and the pre-release disclosure is replaced with the private disclosure, as described below. It must not be considered as the standard process.

This step may be started in parallel with CVE creation. The patches fixing the vulnerability are developed and reviewed by the security team and by elected area experts that agree to maintain confidentiality.

The CVE id and the bug id must be referenced in the patch.

Backports to the identified affected versions are done once the fix is ready.

7.7 Pre-Release Disclosure

When the fix is ready, the security advisory and patches are sent to downstream stakeholders (security-prerelease@dpdk.org), specifying the date and time of the end of the embargo. The communicated public disclosure date should be **less than one week**

Downstream stakeholders are expected not to deploy or disclose patches until the embargo is passed, otherwise they will be removed from the list.

Downstream stakeholders (in security-prerelease list), are:

- Operating system vendors known to package DPDK
- Major DPDK users, considered trustworthy by the technical board, who have made the request to techboard@dpdk.org

The *OSS security private mailing list mailto:distros@vs.openwall.org>* will also be contacted one week before the end of the embargo, as indicated by *the OSS-security process <https://oss-security.openwall.org/wiki/mailing-lists/distros>* and using the PGP key listed on the same page, describing the details of the vulnerability and sharing the patch[es]. Distributions and major vendors follow this private mailing list, and it functions as a single point of contact for embargoed advance notices for open source projects.

The security advisory will be based on below template, and will be sent signed with a security team's member GPG key.

7.7.1 Pre-Release Mail Template

```
This is an advance warning of a vulnerability discovered in DPDK,
to give you, as downstream stakeholders, a chance to coordinate
the release of fixes and reduce the vulnerability window.
Please treat the following information as confidential until
the proposed public disclosure date.

{ impact_description }

Proposed patches are attached.
Unless a flaw is discovered in them, these patches will be merged
to { branches } on the public disclosure date.

CVE: { cve_id }
Severity: { severity }
CVSS scores: { cvss_scores }

Proposed public disclosure date/time: { disclosure_date } at 15:00 UTC.
Please do not make the issue public (or release public patches)
before this coordinated embargo date.
```

If the issue is leaked during the embargo, the same procedure is followed with only a few days delay between the pre-release and the public disclosure.

7.8 Private Disclosure

If a vulnerability is unintentionally already fixed in the public repository, a security advisory is sent to downstream stakeholders (security-prerelease@dpdk.org), giving few days to prepare for updating before the public disclosure.

7.8.1 Private Disclosure Mail Template

```
This is a warning of a vulnerability discovered in DPDK,
to give you, as downstream stakeholders, a chance to coordinate
the deployment of fixes before a CVE is public.

Please treat the following information as confidential until
the proposed public disclosure date.

{ impact_description }

Commits: { commit_ids with branch number }

CVE: { cve_id }
Severity: { severity }
CVSS scores: { cvss_scores }

Proposed public disclosure date/time: { disclosure_date }.
Please do not make the vulnerability information public
before this coordinated embargo date.
```

7.9 Public Disclosure

On embargo expiration, following tasks will be done simultaneously:

- The assigned bug is filled by a member of the security team, with all relevant information, and it is made public.
- The patches are pushed to the appropriate branches.
- For long and short term stable branches fixed, new versions should be released.

Releases on Monday to Wednesday are preferred, so that system administrators do not have to deal with security updates over the weekend.

The security advisory is posted to announce@dpdk.org and to *the public OSS-security mailing list <mailto:oss-security@lists.openwall.com>* as soon as the patches are pushed to the appropriate branches.

Patches are then sent to dev@dpdk.org and stable@dpdk.org accordingly.

7.9.1 Release Mail Template

```
A vulnerability was fixed in DPDK.
Some downstream stakeholders were warned in advance
in order to coordinate the release of fixes
and reduce the vulnerability window.

{ impact_description }

Commits: { commit_ids with branch number }

CVE: { cve_id }
Bugzilla: { bug_url }
Severity: { severity }
CVSS scores: { cvss_scores }
```

7.10 References

- A minimal security response process
- fd.io Vulnerability Management
- Open Daylight Vulnerability Management
- CVE Assignment Information Format

CHAPTER

EIGHT

DPDK STABLE RELEASES AND LONG TERM SUPPORT

This section sets out the guidelines for the DPDK Stable Releases and the DPDK Long Term Support releases (LTS).

8.1 Introduction

The purpose of the DPDK Stable Releases is to maintain releases of DPDK with backported fixes over an extended period of time. This provides downstream consumers of DPDK with a stable target on which to base applications or packages.

The Long Term Support release (LTS) is a designation applied to a Stable Release to indicate longer term support.

8.2 Stable Releases

Any release of DPDK can be designated as a Stable Release if a maintainer volunteers to maintain it and there is a commitment from major contributors to validate it before releases. If a release is to be designated as a Stable Release, it should be done by 1 month after the master release.

A Stable Release is used to backport fixes from an `N` release back to an `N-1` release, for example, from 16.11 to 16.07.

The duration of a stable is one complete release cycle (3 months). It can be longer, up to 1 year, if a maintainer continues to support the stable branch, or if users supply backported fixes, however the explicit commitment should be for one release cycle.

The release cadence is determined by the maintainer based on the number of bugfixes and the criticality of the bugs. Releases should be coordinated with the validation engineers to ensure that a tagged release has been tested.

8.3 LTS Release

A stable release can be designated as an LTS release based on community agreement and a commitment from a maintainer. The current policy is that each year's November (X.11) release will be maintained as an LTS for 2 years.

After the X.11 release, an LTS branch will be created for it at https://git.dpdk.org/dpdk-stable where bugfixes will be backported to.

A LTS release may align with the declaration of a new major ABI version, please read the *ABI Policy* for more information.

It is anticipated that there will be at least 4 releases per year of the LTS or approximately 1 every 3 months. However, the cadence can be shorter or longer depending on the number and criticality of the backported fixes. Releases should be coordinated with the validation engineers to ensure that a tagged release has been tested.

For a list of the currently maintained stable/LTS branches please see the latest stable roadmap.

At the end of the 2 years, a final X.11.N release will be made and at that point the LTS branch will no longer be maintained with no further releases.

8.4 What changes should be backported

Backporting should be limited to bug fixes. All patches accepted on the master branch with a Fixes: tag should be backported to the relevant stable/LTS branches, unless the submitter indicates otherwise. If there are exceptions, they will be discussed on the mailing lists.

Fixes suitable for backport should have a `Cc: stable@dpdk.org` tag in the commit message body as follows:

```
doc: fix some parameter description

Update the docs, fixing description of some parameter.

Fixes: abcdefgh1234 ("doc: add some parameter")
Cc: stable@dpdk.org

Signed-off-by: Alex Smith <alex.smith@example.com>
```

Fixes not suitable for backport should not include the `Cc: stable@dpdk.org` tag.

Features should not be backported to stable releases. It may be acceptable, in limited cases, to back port features for the LTS release where:

- There is a justifiable use case (for example a new PMD).
- The change is non-invasive.
- The work of preparing the backport is done by the proposer.
- There is support within the community.

8.5 The Stable Mailing List

The Stable and LTS release are coordinated on the stable@dpdk.org mailing list.

All fix patches to the master branch that are candidates for backporting should also be CCed to the stable@dpdk.org mailing list.

8.6 Releasing

A Stable Release will be released by:

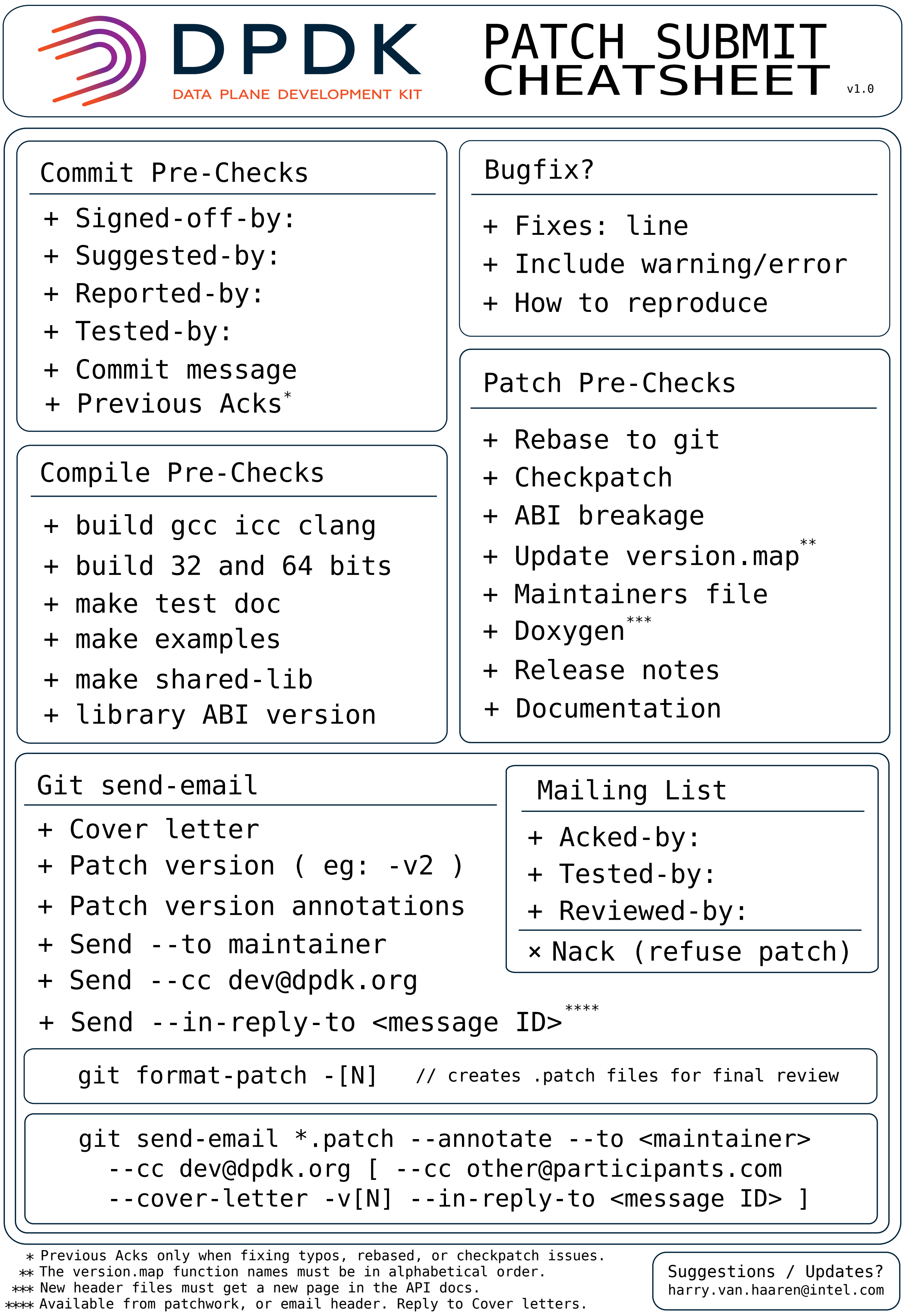
- Tagging the release with YY.MM.n (year, month, number).
- Uploading a tarball of the release to dpdk.org.
- Sending an announcement to the announce@dpdk.org list.

Stable releases are available on the dpdk.org download page.

CHAPTER

NINE

PATCH CHEATSHEET



DPDK
DATA PLANE DEVELOPMENT KIT

PATCH SUBMIT
CHEATSHEET v1.0

Commit Pre-Checks

+ Signed-off-by:
+ Suggested-by:
+ Reported-by:
+ Tested-by:
+ Commit message
+ Previous Acks*

Bugfix?

+ Fixes: line
+ Include warning/error
+ How to reproduce

Compile Pre-Checks

+ build gcc icc clang
+ build 32 and 64 bits
+ make test doc
+ make examples
+ make shared-lib
+ library ABI version

Patch Pre-Checks

+ Rebase to git
+ Checkpatch
+ ABI breakage
+ Update version.map**
+ Maintainers file
+ Doxygen***
+ Release notes
+ Documentation

Git send-email

+ Cover letter
+ Patch version (eg: -v2)
+ Patch version annotations
+ Send --to maintainer
+ Send --cc dev@dpdk.org
+ Send --in-reply-to <message ID>****

Mailing List

+ Acked-by:
+ Tested-by:
+ Reviewed-by:

× Nack (refuse patch)

```
git format-patch -[N]    // creates .patch files for final review
```

```
git send-email *.patch --annotate --to <maintainer>
  --cc dev@dpdk.org [ --cc other@participants.com
  --cover-letter -v[N] --in-reply-to <message ID> ]
```

∗ Previous Acks only when fixing typos, rebased, or checkpatch issues.
∗∗ The version.map function names must be in alphabetical order.
∗∗∗ New header files must get a new page in the API docs.
∗∗∗∗ Available from patchwork, or email header. Reply to Cover letters.

Suggestions / Updates?
harry.van.haaren@intel.com

Fig. 9.1: Cheat sheet for submitting patches to dev@dpdk.org