文件编号：XXX

外部供应商的固件开发规范

（Meta,RL Accessories,FW）

修订历史记录

修订版本	日期	作者	修改描述
0.1	7/1/2023	M.Sompel	初稿

目录

修订历史记录 2
目录 3
目的 4
范围 4
参考文档 4
开发工具 4
固件释放 4
固件测试/调试 5
固件设计实践 6
C 语言编码实践 7
C 语言编码约定 9

目的

本文档的目的是为外部供应商提供固件开发需求。RL 附件团队采用联合开发模式（JDM）由第三方供应商生产我们的产品。JDM 的合作商有时需要编写符合 Meta 和工业标准的固件。

范围

本文档作为附件 JDM 合作商固件开发时的参考文档。JDM 将评审并更新固件。此文档不涉及来自于第三方微控制器的库和 SDK。所有的库应该保持原样，不能有任何改变。

参考文档

MISRA C:2012

开发工具

供应商需要提供用于固件开发，编译，调试和编程的所有工具，以及这些工具的详细操作指令。Meta 会进行审查，编译并且将二进制镜像与供应商的二进制镜像进行比较。

操作指令可以作为工具和硬件的链接进行提供。

为编程和调试而设计的专用夹具需要提供，除非另有约定。

固件释放

固件需要每周释放或者按照约定好的时间表进行释放。固件释放将是正式的。任何固件在释放前必须:

- 所有的代码将会上传至 Meta 创建的 Github
- 用修订版本号打标签
- 无编译错误
- 通过同行评审和 Meta 的代码评审

评审需要提供一些设计文档例如框图，流程图，状态机图。

代码评审将会评审以下内容:

- 编码准则
- 设计的实现
- 解决的问题

当代码评审通过，它就可以添加到释放版本中

固件测试/调试

测试程序和测试步骤以及测试设备由供应商提供。任何专用夹具由供应商提供。所有测试程序在构建前需要 Meta 进行评审和批准。

所有的 bug 应该记录在 Github 的 bug 追踪系统中。符合所有 Github 关于 bug 追踪的规则和约定。

固件文档

Meta 附件合作商开发的所有固件在每一次构建后需要将完整的文档资料发送给 Meta。每一个文档必须包含修订控制。包括但不限于以下文档:

- 系统设计（流程图）
    - 完整的系统流程图
    - 进入/退出电源状态（深度睡眠，运行，充电）
    - 错误状态（FOD,OTP,OCP,SCP）
    - 校准
    - 认证
- 生产/测试的命令和用法
- FW 中的特殊示例
- 电池充电/使用的示例（如果是适用的）
- UX
- 初始化
- 测试程序
    - ERS HW 验证
    - 功能验证
    - 系统验证
    - FW 回归验证

固件设计实践

设计

不要立刻开始写代码。考虑需要实现的东西。规划设计，充分考虑问题，忽略状态机，算法，设计组件，然后开始写代码。

设计评审

当设计完成时，启动设计评审。进行修改是必要的。

代码评审

在代码释放前，需要通过同行评审。这是为了检查错误，编码约定等。

模块化

将你的代码分解成逻辑模块或者函数去提升可读性，可重用性以及易维护性。

层次结构

将关键点清晰地分离以实现层次结构，例如将硬件特定代码与应用程序逻辑分离。

事件驱动设计

使用事件驱动设计以响应异步事件并且可以提高响应性。允许有限的 busy loop。创建事件计时器替代。

功耗管理

设计固件要对功耗进行优化，包括睡眠模式，时钟分频，和高效地使用外设。

将功耗管理功能设计到模块中。

C 语言编码实践

可读性

简洁的书写，使用有意义的变量名和函数名以及结构良好的代码。使用适当的缩进和格式化的公约。保持一致的风格。

注释

使用注释以解释代码背后的复杂的逻辑，假设以及基本原理。有据可查的代码能帮助其他人理解和维护它。

代码可重用性

代码的可重用性通过设计具有单一职责的功能和模块并避免不必要的代码重复来促进代码的可重用性。

可移植性

通过尽量减少与平台相关的代码并尽可能使用标准 C 库，编写易于适应不同硬件平台或操作系统的可移植代码。

无用代码

不应该存在不可访问或注释掉的代码。

代码简化

保持代码简洁，避免不必要的复杂性。简化控制流程，减少嵌套条件，消除冗余代码。保持函数短小，例如避免 1200 行函数。将它分解。

代码组织

- 将相关的功能和数据结构组合在逻辑模块或文件中
- 利用头文件来声明接口，并使关键点清晰地分离
- 以一致的方式组织文件中的函数(例如，按字母顺序，按功能)

错误处理

实现适当的错误处理机制，例如返回代码或错误标志，并优雅地处理错误。**所有的错误必须有一个唯一的标识符。**

- 在整个代码库中正确和一致地处理错误
- 返回错误代码或使用适当的错误处理机制(例如，指针错误时，返回 NULL)
- 在适当的位置，提供清晰和有意义的错误信息或日志记录
- 不应该假设成功-必须检查所有错误条件
- 在可行的情况下，使用唯一标识符将所有错误记录到可捕获的接收器或可检索的存储位置

内存管理

注意内存使用情况，避免内存泄漏和过多的堆栈使用。使用动态明智地分配内存，确保正确的内存回收。

尽量避免使用 malloc/free。

类型安全

在编译时有效地利用 C 的强类型系统来捕获与类型相关的错误。避免过度强制类型转换，确保不同数据类型之间的兼容性。

编译器警告

启用并处理编译器警告，以捕获潜在问题并确保遵守标准。

释放的代码必须没有错误和警告。

优化

优化代码的关键部分以提升速度或大小，但要在代码可读性和可维护性之间进行平衡。在优化之前分析代码以识别瓶颈。

配置

使用包含配置信息的文件。

例如，触发各种电压和定时的状态机应该在配置文件中定义这些值。

ISRs

ISR 处理程序应该简单而快速。获取需要的东西，设置适当的事件，然后退出。不允许在 ISRs 中进行循环。

单一的入口/出口

- 尽量在函数中只有一个入口/出口
- 如果有必要，使用提前的 return 语句来避免 If 语句的深度嵌套

C 语言编码约定

这些编码约定将用于编写 C 代码。

C 语言标准

遵循 C11 标准

命名约定

- 对变量、函数、文件、目录和常量使用有意义和描述性的名称
- 使用 **snake_case**
- 枚举和宏(**MAX_VALUE**)使用大写字母
- 静态常量(**c_constant**)

功能组织

- 以一致的方式组织文件中的函数
- 按字母顺序、功能、…

数据类型

为可移植代码使用特定大小的数据类型(例如 int32_t, uint64_t)。

常量与宏

使用已命名的常量、枚举而不是硬编码值(魔鬼数字)来增强代码的可维护性。

在可能的情况下，选择 const 变量而不是宏，以有利于类型检查和范围限制。

预处理器

避免不必要地使用预处理器指令。

缩进

空格用于缩进。每缩进一次 4 个空格。

行宽

尽量控制在 80 个字符以内。谨慎行事。

空格和括号

- 使用圆括号来澄清复杂的表达式和操作符优先级。
- 所有逗号和分号后面加空格(如果不是在一行的末尾)
  for (int i = 0; i < number; i++)
- 保留字后加空格(sizeof 除外)
- 控制语句的括号前加空格。
  if (foo), while (foo)
- 数学运算符和二进制运算符前后加空格。
  (2 + 7 / x)不是(2+7/x)

空行

使用它们来分隔函数内部的逻辑块。

括号风格

大括号使用线性模式

if (foo) {

while (true) {

控制结构(if, while, for，…)

- 使用清晰和简洁的控制流结构(例如，if-else, switch-case)来提高代码的可读性
- 通过将复杂的逻辑分解成更小的函数或提前使用 return 语句，避免控制流结构的深度嵌套。
- 总是使用大括号。即使对于单个语句也是如此。
- 总是将单个语句放在下一行。

  If (verbose) {
  log(“There are too many discussions about style guides!”);
  }
- 对于计数器变量，使用 **if (count != 0)**或 **if (count > 0)**代替 if (count)。
- if ()条件应该始终是显式布尔表达式。例如，使用 **if (ptr !=NULL)**而不是 if (ptr)。另一方面，如果变量已经是布尔值，则应该执行 if (x)而不是 if (x == true)或 if (!x)而不是 if (x == false)。
- 使用 if (0 == x)而不是 if (x == 0)。如果有 if (0 = x)的拼写错误，编译器将抛出错误。
- 无限循环是用 while(true) 来完成的

  while (true) {}
- 避免在循环中修改循环变量。

Switch-case

使用 FALLTHROUGH 和 NOTREACHED 注释来标记相关的情况。Lint(1)可以理解它们，一些代码检查器(coverity)也可以理解它们，并且它允许人类读者识别何时显式完成而不是错误地完成。然而，不要过于聪明。应该为有明显运行时收益的情况保留 FALLTHROUGH。

```
switch (ch) {          /*Indent the switch. */
    case 'a':         /*Indent the case. */
        aflag = 1;
        /* FALLTHROUGH */
    case 'b':bflag = 1;
        Break;
    case '?':
    default:
```

```
        usage();
        /*NOTREACHED*/
}
```

变量

使用表达性强的变量名。不要不必要地缩写，并始终优化可读性。例如，用 rectangle 代替 rect，用 VirtualAddress 代替 VA 等等。

不要在一行中声明多个变量(这有助于实现其他准则，促进注释，有利于标记)。

如果可能的话，在声明变量时初始化变量，例如，如果常量被赋值给变量。

避免像 **int* a = (int*)param; int b= a->field;**在初始化中，因为它们经常导致意外的 NULL 指针引用。在这种情况下，只需声明 a 和 b，并推迟初始化。

在尽可能小的范围内声明变量。避免使用全局变量。

在给定范围内尽可能靠近使用点声明变量。将变量声明放在代码之前，但要靠近它们被使用的域，即不要在函数范围内对所有变量进行分组。

如果变量或函数形参是指针类型，则将*紧贴类型 **int* number**。

零， NULL，常量和魔鬼数字#

尽可能使用 **const** 作为变量和参数。在指针和指向的位置使用它　　。例如:**foo_t const* const param, static uint32_t const c_num_pages = 1024;**基本原理:向读者传达意图，捕捉错误，并为编译器提供信息以获得更好的代码。

不要在数组定义中使用变量，即使是 **const** 变量。而是使用宏或枚举。基本原理:使用变量定义了一个可变长度的数组，这是本风格指南不允许的。即使变量是 **const** 也是如此。

```
static const unit_t c_num_foo = 32;

void
```

```
function(void)
{
    foo_t many_foo[c_num_foo];   // DON'T DO THIS –varible-length array
}
enum {
    NUM_FOO = 32,
};

void
function(void)
{
    foo_t many_foo[NUM_FOO];   //OK – constant lengtharray
}
```

如果需要一个表示空指针的常量，请使用 **NULL**。使用 **0** 表示数值。

不要使用不必要的强制类型转换。在编译器知道类型的上下文中(例如赋值)使用 **NULL** 而不是**(type*) 0** 或**(type*) NULL**。在其他上下文中使用**(type*) NULL**，特别是对于所有函数参数。(强制类型转换对于可变参数是必要的，如果函数原型可能不在作用域中，则强制类型转换对于其他参数也是必要的。)

不要在代码中使用魔鬼数字(除了作为常量的初始化项之外，不要使用 **0**、**1**、**-1** 以外的任何数字)。在某些情况下，如果无法在其他地方使用该值，那么使用内联魔鬼数字(如果附有解释该值的注释)可能会更具可读性。例如，在循环中乘以 2 以尝试找到最合适的，等等。

对于浮点数，永远不要省略小数点前的初始 **0**(总是 **0.5f**，而不是**.5** 或**.5f**)。不要省略后面的后缀 **0**(总是 **1.0f** 不是 **1** 或 **1.0**)。在定义浮点数时不要省略末尾的 **f**(总是 **1.0f** 而不是 **1.0**)。定义双精度时使用 **1.0**。

函数定义

函数定义应包含返回类型，后跟新行，后跟函数名。

函数名和参数应该在同一行，除非参数不符合 80 个字符的限制。如果参数不符合 80 个字符的限制，则每个参数应在从函数名开始缩进两个块(4 个空格)的新行上。

开始和结束花括号应该在新行上。

如果使用声明，静态函数应该在声明和实现上都有静态标签。

例子

```
int
intinitialize_rectangle (uint32_t top_x, uint32_t top_y)
{
    // Function body goes here
}

int
map_vmo_into_address_space (
    virtual_memory_object_t* vmo,
    virtual_address_region_specifier_t* vmars,
    map_vmo_flags_t* flags)
{
    // Function body goes here
}

int
initialize_rectangle (
    uint32_t top_x,         // x coordinate of the top left corner
    uint32_t top_y,         // Y coordinate of the top left corner
    uint32_t bottom_x,   // X coordinate of the bottom right corner
    uint32_t bottom_y)   // Y coordinate of the bottom right corner
{
  // Function body goes here
}

static const xr_ipc_service_handler_t*
echo_ipc_handler(flatbuffers_thash_t type_hash, const void* context);

// some other code....

static const xr_ipc_service_handler_t*
echo_ipc_handler(flatbuffers_thash_t type_hash, const void* context)
{
  // Function body goes here
}
```

头文件中的函数声明

- 头文件中的函数声明应该在同一行包含返回类型、函数名和参数。
- 如果声明超过 80 个字符的限制，则参数将在下一行继续缩进两个块。
- 不要在其他函数中声明函数;ANSI C 说，不管声明的嵌套如何，这样的声明都具有文件作用域。

例子

```
int initialize_rectangle (int top_x, int top y, int bottom_x, intbottom_y) ;
int map_vmo_into_address_space (virtual_memory_object_t* vmo,
    virtual_address_region_specifier_t* vmars, map_vmo_flags_t*flags);
```

类型定义

- 字面类型的类型定义使用单行
- 使用多行来定义结构体和枚举的类型
- 所有类型定义后加_t 后缀
- 主要结构体应该在使用它们的文件的顶部声明
- 结构体或枚举的每个成员都应独立一行
- 对结构体和枚举使用 typedefs。使用匿名结构体和枚举，除非无法避免。
- 不要重载 struct/enum 和其他标识符，也就是说，不要对结构体和枚举使用相同的名称，唯一的区别是变量的类型声明为 struct Foo 和 enum Foo。
- 使结构名与 typedef 匹配，例如，如果结构不是匿名的，则为 typedef struct foo foo_t。
- 注释没有限制。注释通常应该解释为什么要做某事，而不是正在做什么，除非它不是很明显。避免 i++; //increment i。
- 对于布尔类型使用 bool (stdbool.h)和 true/false。
- 使用 int64_t, uint64_t, int32_t, uint32_t 和其他精确宽度的整数类型，而不是 int (stdint .h)。
- 如果不是很明显，结构中的每个元素都应该有注释。
- 如果行很长，将注释放在每个条目之前。

- 有些人可能会在有注释的元素之间留下空白行。

例子

```
typedef uint64_t virtual_address_t;
typedef struct {
    int top_x;          // x coordinate of the top left corner
    int top_y;          // Y coordinate of the top left corner
    int bottom_x;   // x coordinate of the bottom right corner
    int bottom_y;   // Y coordinate of the bottom right corner
} rectangle_t;

/// A single line Doxygen comment about the structure.
typedef struct {

    /// A Doxygen comment about the following line.

    int really_long_name_that_leaves_no_room;

    /// A Doxygen comment about the second element.
    int another_really_long_name_that_leaves_even_less_room;

    /**
     * A multi line Doxygen comment about following element
     * that gives a lot of information.
     */
    int another_really_long_name_that_leaves_no_room;
} my_example_t
```

宏

避免使用函数式宏。如果可能的话，最好使用静态内联函数。基本原理:静态内联函数可以进行类型检查，避免宏参数展开时产生的双重求值和语法问题。

类表达式宏的定义必须加括号，除非宏被括号括起来之后不能正常的工作。基本原理:用圆括号括住定义可以防止与周围代码的意外交互。但是，有些宏可能不能用圆括号括起来，比如那些展开为类型的宏，或者那些其结果与 stringize(#)操作符一起使用的宏。

例子:

```
#define NUM_VPNS 0x1000
```

```
// CORRECT -- With surrounding parentheses
#define END VPN (VPN + 1)
release_vpns (NUM_VPNS - END_VPN);     // Releases (0x1000-VPN-1) VPNs

// INCORRECT - Without surrounding parentheses
#define END_VPN VPN + 1
release_vpns (NUM_VPNS - END_VPN);     // Releases (0x1000 - VPN +1) VPNs -- two more than intended
```

然而，在本例中，最好的解决方案是使用 **static const** 变量而不是宏。静态常量变量更安全，在可能的情况下应该优先使用。使用 **static const**，可以将示例重写为如下形式:

```
// Use static const variables instead of macros.
static uint64 t const c_num_vpns = 0x1000;
static uint64 t const c_end_vpn = VPN + 1;

release_vpns (c_num_vpns - c_end_vpn); // Releases (0x1000 - VPN - 1) VPNs
```

定义中使用的宏参数必须用圆括号括起来，除非宏不能正确地使用圆括号。基本原理:这可以防止在展开由多个标记组成的参数时出现意外结果。但是，使用 stringize 操作符#、连接操作符##或相邻字符串的隐式连接的宏不能使用带圆括号的参数。

例子:

```
/*
 * CORRECT -- all parameter uses are parenthesized
 *
 * XR USEC (100 + 50) expands to XR DURATION (UINT64 C(1000) * (100 + 50)), which is 150 usec.
 * /
#define XR_USEC(_n) XR_DURATION (UINT64_C(1000) * (_n))

/*
 * INCORRECT -- a parameter use is not parenthesized
 *
 * XR_USEC (100 + 50) expands to XR_DURATION (UINT64_C (1000) * 100 +50), which is 100.050 usec.
 * /
#define XR_USEC(_n) XR_DURATION (UINT64 C(1000) * n)
```

在#define 和宏名之间放一个空格字符。基本原理:一致的代码风格可以提高可读性。

如果宏是函数的内联展开，则函数名全部为小写，而宏的使用相同名称且全部大写。基本原理:一致的代码风格可以提高可读性。

要定义一个无操作或空的类函数宏，请将其定义为((void) 0)。不要使用空的宏定义。基本原理:将宏定义为((void) 0)使其表现得更像一个真正的函数，包括需要一个分号作为语句结束符，从而使宏在不为空时更有可能被正确使用。

在多行宏定义中，垂直对齐连续行反斜杠。基本原理:这使得宏更容易阅读。

在定义寄存器、地址或其他密切相关的东西的块中，对可读性的值进行校验。如果这些寄存器属于同一逻辑实体，则按地址对它们进行排序，或按功能对它们进行分组。不要把它们乱放。

```
#define PERI CRG RSTEN4       0x90
#define PERI_CRG_RSTDIS4      0x94
#define PERI_CRG_ISODIS       0x148
```

如果宏展开为复合语句，则在编译器支持的情况下将其括在语句表达式中(https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html)中(例如 GCC 和 clang)。基本原理:这允许在 if 语句和循环中安全地使用宏，而不需要完整的括号，并且可以绕过 clang 限制。常用的语句 do { … } while(0)有类似的目的，但是 clang 在使用它时不能正确地分析不可访问的代码。

```
#define ASSERT (cond)                                   \
  ({                                                    \
    if (unlikely(! (cond))) {                           \
       PANIC("ASSERT FAILED at: %$", #cond);      \
    }                                                   \
  })
```

宏定义不能以分号结尾。语句终止分号应该在宏调用点提供，而不是由宏提供。理由:包含分号会导致宏定义在语法上的行为与函数调用不同，如果宏调用在无括号循环或条件中后跟分号，则可能导致令人惊讶的结果。

```
#define MARCO(x, y)             \
```

```
({                                  \
   variable = (x) + (y);            \
   (y) += 2;                        \
})
```

Sizeof

选择 sizeof(type)而不是 sizeof(*var)。

理由:使用 sizeof (type)遵循了为阅读代码而不是编写代码而优化的原则，因为它不需要读者找到变量的类型。

未使用的变量

XROS 开发要求主分支在几个不同的构建上既编译又不积累静态分析错误。这会导致一些变量出现问题，这些变量是为了帮助在开发构建中进行调试而引入的。接下来的两节描述了如何处理有效的未使用变量的主要来源:断言和其他所有内容。

断言

断言是一种推荐的机制，用于验证内部调用路径代码的不变量。它们有助于记录函数所做的假设。然而，它们不是一种避免公共 api(例如 library, sycall)的参数检查的机制。例如，不应该使用断言来验证参数是否为 NULL，而应该使用公共 api 返回适当的错误代码。但是，可以使用断言来验证 api 的内部状态是否有效。

不要在断言中包装带有副作用的表达式。如，ASSERT(important_function())或 ASSERT(important = vImportant())。这会在释放和开发人员构建时导致不同的行为。

```
void important_callback (xr_important_object_t* modifiable)
{
   xr_error_t assert_error;
   assert_error = modify_input_with_side_effects_one (modifiable);
   ASSERT_XR_OK (assert_error);
   assert_error = modify_input_with_side_effects_two (modifiable);
   ASSERT (assert_error == XR_ERR_NOT_FOUND);
```

```
}
```

如果变量仅由 ASSERT_MSG 的消息使用，则必须将该变量声明为_ASSERT_ONLY 将其标记为在非调试构建中不使用。

```
void verify_header(header_t* header)
{
  _ASSERT_ONLY xr_uuid_formatted_t uuid_read;
  ASSERT_MSG (header->magic == c_stress_crc_magic
     && header->data_length == length
     && xr_uuid_equals (uuid, &header->uuid),
     "header: magic=%lx length=%lx uuid=%s", header->magic,
     header->data_length, xr_uuid_format (&header->uuid, &uuid_read));
}
```

VLAs (可变长度数组)

不允许使用可变长度数组(https://clang.llvm.org/compatibility.html#vla)。静态大小的数组或堆分配应该在使用可变长度数组的地方使用。其基本原理在 T57940410 中有完整的记录，但可以归结为:

1. 到目前为止，VLAs 是意外导致堆栈溢出的最简单方法，在最坏的情况下，它可能为攻击者提供崩溃代码或泄漏内存的途径。

2. 任何“安全”的 VLA 必须具有静态确定的最大大小，因此，如果必须在栈而不是堆上分配数组，则总是可以分配最大大小数组而不是变长数组。

3. VLAs 生成的代码至少与静态大小的数组一样慢，甚至更慢。

灵活的数组成员

允许使用灵活的数组成员。在具有多个成员的结构体中，结构体的最后一个成员可以是不完整数组类型。这样的成员称为结构体的灵活数组成员，它使人们能够模仿 C 中的动态类型规范，因为您可以将数组大小的指定推迟到运行时。灵活数组成员必须是最后一个成员，并且必须指定为 a[ ]，而不是 a[0]，这样编译器才能检测到误用。下面是一个具有灵活数组成员 a 的结构体示例:

```
Typedef struct {
```

```
    int len;
    char a[ ];
} my_struct_t;
```

然后可以通过以下方式分配内存:

```
my_struct_t *s = xr_malloc(offsetof (My_struct_t, a[n]));
s->len = n;
```

版权

我们正在摆脱使用版权声明的历史惯例。在源代码中使用以下机密布告。

```
/*
 * This software contains information and intellectual property
 * that is confidential and proprietary to Facebook, Inc. and its affiliates.
 */
```

当对拥有第三方版权的文件进行重大更改时，请将 Facebook 版权添加到原始版权声明中:

```
// Copyright 2017 The Fuchsia Authors. All rights reserved.
// Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.
```

当对拥有第三方版权的文件进行重大更改时，请将 Facebook 版权添加到原始版权声明中:

```
// Copyright 2017 The Fuchsia Authors. All rights reserved.
// Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.
```

当基于第三方代码贡献新代码时，请在现有区块中添加 Facebook 版权。例如:

```
/*
 * Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved.
 * Copyright (C) 2017-2018 Hilisicon Electronics Co., Ltd.
 *                    http://www.huawei.com
 *
 * Authors: Yu Chen <chenyu56@huawei.com>
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 of
```

```
 * the License as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 * /
```

在所有这些情况下，过滤器都应该检测并提出修复建议。Facebook 版权的相对位置并不重要，尽管出于可读性的考虑，人们倾向于把它放在首位。

请登录 https://our.intern.facebook.com/intern/wiki/Copyright_Headers/查看所有角落案例。

注释

代码不是自我记录的。

包括注释来解释功能、代码块和复杂逻辑的目的、行为和假设。特别是当有一个特定的方式去做某事的原因。

记录函数原声明、输入/输出参数、返回值和任何错误条件。

避免过多或多余的注释，不能增加有意义的信息。

对于单行代码注释，总是使用//(注释前加空格分隔符)。

```
// This is a single-line comment
```

对于多行代码注释，使用/*…*/。注释必须与开始和结束符号在不同的行上。

```
/*
 * This is a
 * multi-line
 * comment.
 * /
```

文档

头文件应该使用 Doxygen (带有 at 符号前缀)来记录函数、类型和接口的声明和定义。多行注释的开头/**和结尾*/应该在各自的行上。

对单行注释使用///。理由:与其他单行注释的一致性。

使用/**…*/用于多行注释。理由:与其他多行注释一致。

```
/**
 * Brief description which ends with a '.'.
 * Detailed part of the comment started here.
 * /
```

Doxygen 指令使用@符号。不得使用\符号。例如，使用@file 和@param，而不是 file 和 param。

文件文档

源文件必须包含@ file 标签，置于版权声明之后。@file 默认情况下不应该被指定，只有当源文件名在代码库中不是唯一的时候才应该被指定。基本原理:Doxygen 需要@file 标签，如果指令为空，默认使用给定源文件的名称。

示例文件文档，直接位于版权声明之后:

```
/** @file [file name, with partial directory if needed]
 * This is a brief description of my module.
 *
 * The detailed description starts here. It contains a link to design
 * docs created that cover the implementation in this module.
 */
```

对于任何程序，包含主例程(入口点)的源文件的文件文档必须描述程序的功能以及如何调用它。基本原理:这是理解和使用程序的基本文档。

函数的文档

所有函数在定义时都必须具有 Doxygen 格式的文档。公共头文件中的函数声明也必须有 Doxygen 格式的文档。其他函数声明不能有文档块。基本原理:在大多数情况下，只有一点的文档是足够的。但是，公共头文件这样做需要编写 API 文档，供无法访问源代码的开发人员使用。

此外，同一函数的公共文档和内部文档可能具有不同级别的详细信息，特别是当公共文档强调抽象而内部文档讨论该抽象的实现细节时。

示例函数文档:

示例 1:

```
/**
 * Create a uni-directional IPC receiver channel.
 * Setup the kernel representation of a uni-directional IPC channel.
 * The pages belonging to the IPC channel are getting mapped into the
 * kernel address space.
 * The IPC channel acquires references to the underlying physical pages
 * used to back the channel, and to the port. The channel needs to get
 * destroyed before those resources can be freed.
 *
 * To be able to guarantee that the kernel can post a notification to the
 * port to signal that user space should check the IPC channel for new
 * messages, the IPC channel takes a reservation on the port.
 *
 * @note To close the provided port, the IPC channel needs to be closed
 * first.
 *
 * @param[ in ] setup Struct to initialize IPC channel.
 * @param[ in ] produce_mem_object_header Object header to produce memory.
 * @param[ in ] consume_mem_object_header Object header to consume memory.
 * @param[ in ] msg_buffer_mem_object_header Object header to message buffermemory.
 * @param[ in ] control_mem_object_header Object header to control information memory.
 * @param[out] ipc_channel_object_header Object header for new IPC channel.
 *
```

```
 * @retval XR_OK On success.
 * @retval XR_ERR_INVALID_ARGS One of the input arguments is invalid.
 * @retval XR_ERR_NO_VIRTUAL_ADDRESS_SPACE No virtual address space is
 * available to map in pages.
 * @retval XR_ERR_NO_MEMORY A memory allocation failed to allocate the data
 * structure which tracks the IPC channel.
 * @retval XR_ERR_UNAVAILABLE An associated memory region is intended to be
 * freed.
 */
xr_error_t ipc_channel_create (const xr_ipc_channel_setup_t* setup,
      object_header_t* produce_mem_object_header,
      object_header_t* consume_mem_object_header,
      object_header_t* msg_buffer_mem_object_header,
      object_header_t* control_mem_object_header,
      object_header_t** ipc_channel_object_header);
```

示例 2:

```
/**
 * Check whether the message buffer can store the additional message.
 *
 * A message may span across multiple buffer split regions. However, a message
 * may not wrap around, and must always be virtually contiguous.
 * If a new message would wrap around, we restart it at offset o of the buffer.
 * If a message doesn't start at the beginning of a split buffer area,we know
 * that the split region must have been fully read at the time when the initial
 * message was added to the buffer, which means we can just add a new message
 * into the split region.
 * If a message crosses into a new split region, it may only do so, if the
 * corresponding counter is zero, which means that user space has consumed
 * all messages previously associated with that split region.
 *
 * @param[ in ] channel IPC channel for which to enqueue a new message.
```

```
 * @param[ in ] msg_length Length of the message including handle space.
 * @param[ out ] buf_pos [optional] Buffer position from which the new message
 * starts from. Can be set to O if the message would have wrapped around the
 * buffer.
 *
 * @return True if the message can be enqueued, false otherwise.
 */
static bool
ipc_channel_check_msg_buffer (
        ipc_channel_t* channel,
        uint32_t msg_length,
        uint32_t* buf_pos)
```

函数文档由以下元素组成:

1. 对函数效果的描述。基本原理:这是最基本的基本文档。

2. 所有函数参数的范围限制，这些参数在其整个通用范围内是否无效。基本原理:这是安全使用函数的基本信息，特别是对于可能不进行范围检查的内部函数。

3. 所有函数参数及其含义。基本原理:这是使用该函数的重要信息。

4. 函数返回值的含义。基本原理:使用该函数的重要信息。

5. 正确或安全地使用该功能所需的任何说明或警告。这些文件应该记录任何可能被其他使用该函数的工程师产生意外的行为。基本原理:只要有可能，设计功能就不要有产生意外的行为。当这种行为无法避免时，要提前明确通知对方，尽量减少意外。

@brief 指令不能用于函数描述。基本原理:@brief 不会影响函数详细文档的呈现。当出现时，它只是将带注释的段落(不管它跨越的行数或包含的句子数)添加到页面顶部的目录中。避免使用@brief 可以减少额外的工作和视觉干扰。

当一个函数声明被文档化时，文档必须包含正确使用该函数所必需的所有信息，但可以省略使用该函数所不必要的实现细节。

所有在公开头文件的函数文档，以及所有已发布或拟在团队间或外部使用的函数文档，均应包括功能描述。对于其他函数，强烈建议效果描述。但是，如果函数的名称使其效果清晰明了，则可以省略它。基本原理:在需要的地方要求描述，在不需要的地方避免劳作。

函数描述(如果有的话)必须概述构建对函数的理解所需的主要细节，以及任何可能的陷阱或其他怪异行为。理由:基本级别需要的文档。

如果函数实现了一个复杂的、棘手的或非常不寻常的算法，那么函数描述必须记录调用函数的程序员需要了解的关于算法的信息，以便安全、正确地使用它。这样的函数还必须有足够的文档，以便工程师阅读和修改代码;该文档可以放在函数描述中，也可以作为函数内的注释。基本原理:为客户和维护人员提供必要的理解水平。

对于不在公共头文件中且不打算对外使用的函数文档，如果在编写函数时使用了已出版的书籍、论文等，并且所使用的算法超出了典型工程师的知识范围，则应在函数描述中包含对该书籍、论文等的引用。基本原理:认为维护人员具有必要的理解水平。

函数参数必须有文档记录，除非函数签名使所有参数的含义非常清楚和明确。基本原理:在需要的地方要求文档，在不需要的地方避免劳作。

如果一个函数的任何参数被记录下来，那么该函数的所有参数都应该被记录下来。基本原理:避免歧义和不完整。

函数参数应该用@param 指令记录。参数文件还应包括参数的单位和/或范围，如果这些不是非常清楚和明确的话。@param 指令应该指示每个参数的数据流方向，如下所述。

- in-参数由函数读取，标量/结构体/等永远不会被修改。
  - 例如.str in size_t strlen (const char *str);
- out-仅由函数写入。
  - 例如：dest in void* memcpy (void *dest, const void *src, size_t n);
- In,out-参数由函数读取并可能修改。

- 例如：size 在 compute_initial_stack_pointer (uintptr_t base，size_t * size)作为堆栈区域的初始大小，计算后会重置以反映实际大小。
- In,out 不应该用于函数接受参数并改变内部状态的地方。

- optional - Doxygen 不支持可选参数。因此，在参数描述之前添加[optional]关键字。

@param 行在@param 和方括号之间不能有空格，在括号和参数名之间有一个空格，在参数名和参数描述之间有一个空格。不应增加垂直对齐的额外空格。如果参数描述衍生出多行，则参数描述应继续在下一行的开头遵循标准 Doxygen 格式。现有代码可能不遵循这些规则;它不需要更新，直到它被触及。基本原理:这种格式具有足够的可读性，并且比保持垂直对齐更节省时间。

如果对省略的内容是否清晰明了存在分歧，则该内容在定义上是不明确的，不应被省略。理由:错误倾向于过度记录。

如果显式地调用返回值，则各个返回值需要在各自的行上，并且必须以@retval 开头。描述不应该以“If…”开头，而应该只说明在什么条件下可以观察到返回值。如果没有显式的返回值被调用，或者形成一个简单的语句，应该使用@return。
示例@return:

```
/**
 ...
 * @return Pointer to an ipc_channel_t on success, NULL otherwise.
 * /

/**
 ...
 * @return True if the message can be enqueued, false otherwise.
 */
```

允许使用其他标记来增强函数文档的可用性，例如使用@ref 来交叉引用相关文档，但不是必需的。

结构体和联合文档

每个结构体或联合体声明必须记录。文档必须描述其目的或用途。基本原理:在阅读代码时，理解结构体或联合的整体含义是有帮助的。只有结构体或联合体成员的文档记录通常是不够的。

结构体或联合的每个成员都必须记录。但是，如果一个成员的目的和用途与它的声明是非常清楚和明确的，则可以省略该成员的文档。基本原理:允许使用、理解和修改结构体。

如果对省略的内容是否清晰明了存在分歧，则该内容在定义上是不明确的，不应被省略。理由:错误倾向于过度记录。

头文件

Doxygen 接口注释在开发人员使用的 exports .h 头文件中也是强制性的(例如 SDK 中的头文件)，但在内部或私有头文件中则不是。根据本文档中定义库和构建系统约定的部分，这些文件的标识应该是明确的。

导出头文件中的文档应该编写以供公众使用，其详细程度可达手册页。它应包括使用实例。由于受众不同，因此本文档与 C 文件中相同函数的文档不同，也期望与之不同。

由于保持这些头文件同步需要人力，包括技术作者专业编写的内容，因此在开发期间推迟公共头文件文档是可以接受的。所有的文档化延迟的文件应包含文件特定的 TODO 任务注释 例如：// TODO (T123456): Add public header Doxygen comments

特殊的注释

当某些东西需要修复时，使用// TODO(<task id>): <description>注释。不要使用用户名代替任务 id。包括一个简短的描述，给读者提供足够的背景。

代码组织

- .h 文件应与相应的.c 文件在同一目录下

- 单元测试文件应该位于同一目录下名为“test”的子目录下。

包含文件

- 避免使用不必要的 include。
- 对本地包含使用双引号，对系统范围的包含使用尖括号。
- 在包含头文件时，请按范围对头文件进行排序，以便首先出现系统包含文件，其次是项目范围的包含文件，最后是本地包含文件。对于同一作用域的头文件，按名称排序。
- 头文件必须是自包含的，这样就可以在不首先包含其他头文件的情况下包含它们。如果它们依赖于不透明类型，例如，对于在头文件中声明的函数的参数类型，它们应该包含声明不透明类型的头文件，而不是包含完整类型定义的头文件。
- 对所有新头文件使用#pragma once，以防止重复定义。

% cat hikey960-hw.h

#pragma once

…

…

- …
- 不要使用相对路径。例如，不要写#include "…/…/foo.h"