

# Document

---

## 本地构建与测试

如需本地预览效果,请确保已经安装( `nodejs >= 18` )环境

shell

```
# 安装依赖
npm install

# 在开发模式中查看效果
npm run docs:dev

# 编译为静态文件
npm run docs:build

# 在开发模式中查看效果
npm run docs:preview
```

---

## tips

文档生成配置所在目录 `doc/.vitepress` 目录下的 `config` 文件或者 `config` 目录 `vitepress` 会自动导入 `config` 文件或者 `config` 文件夹下的 `index.mts` 文件

其中的字段 `sidebar` 表示配置侧边栏, `nav` 表示配置导航栏, 具体细节可以参考 `vitepress` 官方文档

- [nodejs Dist](#)
- [vitepress](#)

## github actions

`deploy.yml` 配置依赖于 `yarn.lock` 文件。若使用 github actions 配合 GitHub 自动生成页面，请保证 github `{path}/settings/actions` 页面内单选项全部选择顶部第一个。

在第一次生成后，将 `gh-pages` 分支(详见配置 `deploy.yml` ) 设为 GitHub pages，后续页面更新由 actions 自动完成。

# XFusion description

My great project tagline

[get-started](#)

[introduction](#)

[\[中文\]](#)

# XFusion 文档中心

[介绍](#)[快速入门](#)

## 多平台

支持 esp32, ws63 等平台。后续将支持 linux, stm32 等等。

## 快速迁移

一次开发，多端部署。  
xfusion 有独立的 HAL（硬件抽象层），实现同一份代码在跨平台时只需简单修改引脚配置。

## 协程

内置无栈协程，协作式调度简单可预测，移植无需汇编。

[\[English\]](#)

XFusion，来自 X(Embedded Kits System) —— 嵌入式套件系统，是一个融合多个嵌入式平台的软件开发工具包(SDK)，为开发者提供统一且便于开发的嵌入式开发环境。

# API 参考

## 提示

### API 参考

本文主要介绍 xfusion 包含 xf\_hal、xf\_utils、xf\_task 等组件在内的系统组件。

---

## 阅读对象：

- 想要深入了解 xfusion 框架的用户以及移植开发者、组件开发者。
- 

## 子文档

# 代码注释指南

本文说明 xfusion c/cpp 代码的注释规范。

## 适用范围：

- 外部库以外的所有 xfusion c/cpp 头文件/源文件。

## 阅读对象：

- 所有 c 语言代码贡献者。

# 简介

xfusion c/cpp 代码的注释使用 doxygen 风格的注释，如果使用 vscode 编写代码，请安装 [cschlosser.doxdocgen](#) 插件，方便生成 doxygen 风格的注释。

vscode 的 doxygen 插件：[Doxygen Documentation Generator](#). doxygen 文档：[Documenting the code](#)

# 原则

注释是帮助读者在阅读和使用代码的信息。

## 注释的位置：

注释可以添加到函数、结构体、类型定义、枚举、宏等任何需要说明代码作用的地方。但是注释只能在需要说明的代码的上方或者右侧。注释在需要说明的代码的右侧时，请注意使用 `/*!<` 注释 `*/` 样式的注释，以表明当前注释说明的对象是左侧的代码。

如：

```
/* ***** 正确示例 ***** */  
  
/* 总线配置 */  
xf_spi_bus_cfg_t bus_cfg = {0};
```

```
xf_spi_bus_cfg_t bus_cfg = {0}; /*!< 总线配置, 注意这个`!<`, 表示当前注释说明的是左边的代码

/* ***** 错误示例 ***** */

/* 1. ↓缺少`!<`, 导致生成文档时注释位置错误 */
xf_spi_bus_cfg_t bus_cfg = {0}; /* 总线配置 */
xf_spi_pin_t spi_pin = {0};

/* 2. ↓注释位置错误 */
xf_spi_bus_cfg_t bus_cfg = {0};
/* 总线配置 */
```

## 注释的重点：

函数中的注释应当概括一段代码的作用，以及解释说明代码中重点以及难以理解的部分。好的代码（拥有规范的命名、操作明确等特征）应当可以说明代码本身在做什么，即代码能够自己解释自己。因此没必要在函数中每一行都解释代码在做什么，而是在**必要情况下说明代码做了什么，为什么这么做**。

## 注释的格式：

通常建议只使用 `/* */` 的注释格式。

## 头文件的注释：

要求尽可能详细以及全面，函数功能参数的注释、结构体的注释、枚举类型的注释等等都要完善，好的头文件能够做到让用户不看源码就可以使用这个模块。

## 示例

## 文件头注释

```
/**  
 * @file xf_log.h  
 * @author catcatBlue (catcatblue@qq.com)  
 * @brief xf_log 日志模块。  
 * @version 1.0  
 * @date 2023-07-26 初版。  
 *         2024-01-14 替换 printf 实现、加变量锁、增加日志输出后端、  
 *                     增加 VERBOSE 等级。  
 *                     由于通过日志输出后端可以实现异步输出，因此 log 不设总锁。  
 */
```

```
* Copyright (c) 2024, CorAL. All rights reserved.  
*  
*/
```

TODO: 文件头注释规范。

## 如何注释

以下代码是函数注释示例，位于 `components/xf_hal/xf_spi/xf_spi.h`。

```
/*  
 * @brief xf_spi 初始化总线并添加设备。  
 *  
 * @param spi_num spi 号。  
 * @param clock_speed_khz spi 时钟速度，单位 kHz。  
 * @param preset 预设值，见 @ref xf_spi_presets_t。  
 * @param mosi mosi(主出从入)引脚号。  
 * @param miso miso(主入从出)引脚号。  
 * @param sclk 时钟引脚号。  
 * @param cs 片选引脚号。  
 *      主机模式时可以为 NC，此时传输前后不会操作 cs 引脚，从机模式时必须填入。  
 * @param[in] p_handle 传出的操作句柄。  
 *      主机模式时可以为 NULL，表示想要稍后添加设备（填入 pre_cb 等参数），  
 *      从机模式时必须填入以接收传出的句柄。也可分为`bus_init`、`bus_add_device`两步。  
 * @return xf_err_t  
 *      - XF_ERR_INVALID_ARG      参数错误  
 *      - XF_FAIL                失败  
 *      - XF_OK                 成功  
 *  
 * @note 该初始化函数中主要完成两个步骤：  
 *      1. 调用 `xf_spi_bus_init()` 初始化 spi 总线；  
 *      2. 调用 `xf_spi_bus_add_device()` 添加 spi 设备。  
 */  
  
xf_err_t xf_spi_init(  
    xf_spi_t spi_num,  
    uint16_t clock_speed_khz, xf_spi_presets_t preset,  
    xf_gpio_t mosi, xf_gpio_t miso, xf_gpio_t sclk, xf_gpio_t cs,  
    xf_spi_handle_t *p_handle);
```

从以上示例中，可以看出一个函数注释由以下部分组成：

1. 特殊注释块(Special comment blocks).

## Special comment blocks

特殊注释块是如下样式的注释块：

```
/**  
 * ... text ...  
 */
```

这是 Javadoc 样式的注释块，当然还有其他样式的注释块，但是为了统一风格，xfusion 中**只使用 Javadoc 样式**的注释块。

## 2. 特殊命令(Special Commands).

### Special Commands

命令是如 `@brief @param @return` 样式的代码，命令样式除了以 `@` 开头外，还有以反斜杠开头的 `\` 样式，xfusion 中**只使用以 @ 开头的命令样式**。

在函数注释中，常用命令有以下几种：

1. 简介 `@brief { brief description }`。`@brief` 命令用于简要介绍文件、函数、变量、结构体等代码的功能。这是最常用的命令。注意！如果注释块中只有一个 `@brief`，**请注意删除多余的空行**；如果有多个命令，注意将 `@brief` 与其余命令间隔一行，如下所示。

```
/* ***** ↓ 正确的 ***** */  
/**  
 * @brief 我是注释。  
 */  
  
/**  
 * @brief 我是注释。  
 *  
 * @param 参数1 我是参数1。  
 * @param 参数2 我是参数2。  
 */  
  
/* ***** ↓ 错误的 ***** */  
/**  
 * @brief 不要这样。  
 *  
 */  
  
/**  
 * @brief 不要这样 ↓，缺少空行。  
 */
```

```
* @param 参数 我是参数。  
*/
```

2. 参数 `@param '['dir']' <parameter-name> { parameter description }` . `@param` 命令通常用于介绍函数或者函数原型(类型定义的函数原型)的参数的作用。 `@param` 命令有一个可选参数, 方向 `dir`, 这个参数紧靠 `@param`, 可以是 `[in]`、`[in, out]` 和 `[out]` . xfusion 中如果不注明方向 `dir`, 则默认为 `[in]` ; 一旦参数涉及传出, 比如函数内会修改指针指向的空间, 则必须标明传出双向 `[in, out]` 和传出 `[out]` . `@param` 命令中的参数名 `<parameter-name>` 在插件生成 doxygen 风格注释时会自动生成。 `@param` 命令中的参数描述 `{ parameter description }` 用于描述该参数的作用, 必要时请加上 `@ref` 或者 `@see` 命令告知读者有用的参考信息。如果函数没有参数, 请跳过该命令。

```
/*  
 * @brief Copies bytes from a source memory area to a destination memory area,  
 * where both areas may not overlap.  
 *  
 * @param[out] dest The memory area to copy to.  
 * @param[in] src The memory area to copy from.  
 * @param[in] n The number of bytes to copy  
 */  
void memcpy(void *dest, const void *src, size_t n);
```

3. 返回值 `@return { description of the return value }` . `@return` 是对返回结果的描述。如果有多种返回结果请用以下格式表示 (\* - 中有 6 个空格实际上是因为按了两次 TAB) ; 如果没有返回值请跳过该命令。

```
/*  
 * @brief xf_spi 反初始化。  
 *  
 * @param spi_num spi 号。  
 * @return xf_err_t  
 * - XF_ERR_INVALID_ARG      参数错误  
 * - XF_ERR_NOT_SUPPORTED    不支持 (未对接)  
 * - XF_FAIL                 失败  
 * - XF_OK                  成功  
 *  
 * @note 会删除该总线下的所有设备。  
 */  
xf_err_t xf_spi_deinit(xf_spi_t spi_num);
```

4. 注意事项 @note { text } 或 @attention { attention text } 以及警告 @warning { warning message } . @note 或 @attention 命令用于告诉读者需要注意的重要信息。 @warning 命令告诉读者必须怎么做或者禁止怎么做，否则会产生什么后果。

再次强调，为了节省时间，请用功能类似于 [cschlosser.doxdocgen](#) 的 vscode 插件生成 doxygen 风格注释的模板。以下代码是生成的注释模板，插件可以帮你节省时间。

```
/*
 * @brief
 *
 * @param spi_num
 * @param clock_speed_khz
 * @param preset
 * @param mosi
 * @param miso
 * @param sclk
 * @param cs
 * @param p_handle
 * @return xf_err_t
 */
xf_err_t xf_spi_init(
    xf_spi_t spi_num,
    uint16_t clock_speed_khz, xf_spi_presets_t preset,
    xf_gpio_t mosi, xf_gpio_t miso, xf_gpio_t sclk, xf_gpio_t cs,
    xf_spi_handle_t *p_handle);
```

以下代码是结构体注释示例，位于 [components/xf\\_hal/xf\\_spi/xf\\_spi\\_types.h](#) 。

```
/*
 * @brief spi 通用配置结构体。
 */
typedef struct _xf_spi_cfg_t {
    uint16_t hosts: 1;                      /*!< spi 是否是主机，见 xf_spi_hosts_t */
    uint16_t data_width: 3;                  /*!< spi 数据宽度，见 xf_spi_data_width_t */
    uint16_t direction: 3;                  /*!< spi 方向，见 xf_spi_direction_t */
    uint16_t clock_phase: 1;                /*!< spi 时钟相位，见 xf_spi_clock_phase_t */
    uint16_t clock_polarity: 1;              /*!< spi 时钟极性，见 xf_spi_clock_polarity_t */
    uint16_t bit_order: 1;                  /*!< spi 比特顺序，见 xf_spi_bit_order_t */
    uint16_t nss: 2;                        /*!< spi NSS 管理方式，见 xf_spi_nss_t */
    uint16_t crc: 1;                        /*!< spi CRC，见 xf_spi_crc_t */
    uint16_t tx_en_dma: 1;                 /*!< spi DMA 通道，见 xf_spi_dma_ch_t */
    uint16_t rx_en_dma: 1;                 /*!< spi DMA 通道，见 xf_spi_dma_ch_t */
    uint16_t reserve: 1;                   /*!< 保留 */
    uint16_t clock_speed_khz;             /*!< spi 时钟速率(单位千赫兹) */
```

```

    uint16_t crc_polynomial;           /*!< 指定用于计算 CRC 的多项式 */
    uint16_t transfer_sz;             /*!< 写: spi 最大传输大小; 读: spi 已接收大小 */
    void *p_ext_cfg;                /*!< 额外配置 */
} xf_spi_cfg_t;

```

## 做得更好

除了 `@brief`, `@param`, `@return` 等等常用命令, 还有 `@details`, `@code`, `@example` 等命令可以让你的注释做得更出色。如以下示例所示。

```

/*
 * @brief xf_spi 设置设备配置。
 *
 * @param handle 设备操作句柄。
 * @param p_dev_cfg 设备配置指针。见 @ref xf_spi_dev_cfg_t。
 * @return xf_err_t
 *         - XF_ERR_INVALID_ARG      参数错误
 *         - XF_ERR_NOT_SUPPORTED    不支持（未对接）
 *         - XF_FAIL                 失败
 *         - XF_OK                  成功
 *
 * @details
 * Example:
 * @code{c}
 * // handle 是通过 xf_spi_bus_add_device() 得到的句柄
 * xf_spi_cfg_t spi_cfg = {0};
 * spi_cfg.data_width = XF_SPI_DATA_WIDTH_8_BITS;
 * xf_spi_dev_cfg_t dev_cfg = {0};
 * dev_cfg.p_spi_cfg = &spi_cfg;
 * BIT_SET1(dev_cfg.target_mask, XF_SPI_CFG_DATA_WIDTH);
 * xf_spi_dev_set_cfg(handle, &dev_cfg); //!!! 设置 spi0 总线配置
 * // 如果设置成功, 函数`xf_spi_dev_set_cfg()`会将对应位设为 0
 * if (BIT_GET(dev_cfg.target_mask, XF_SPI_CFG_DATA_WIDTH) == 0) {
 *     xf_printf("successfully set!\r\n");
 * }
 * @endcode
 */

xf_err_t xf_spi_dev_set_cfg(
    xf_spi_handle_t handle, xf_spi_dev_cfg_t *p_dev_cfg);

```

### 1. 其他命令.

#### [Special Commands](#)

1. 代码块 `@code['{'<word>'}']` . 代码块是注释中被解释为代码的部分，可以显示语法高亮。用于说明在实际代码中如何使用某个函数或其他代码。`@code` 与 `@endcode` 一起出现。`@code` 的可选参数 `['{'<word>'}']` 用于指定语法高亮的语言，在 `xf_spi_dev_set_cfg()` 示例中通过 `@code{c}` 指定为 c 语言的语法高亮。
2. 详细描述 `@details { detailed description }` . 就像 `@brief` 开始一个简短的描述，`@details` 开始详细的描述。你也可以开始一个新的段落（空白行），那么就不需要 `@details` 命令了。`@details` 可以使用 markdown 语法。
3. 示例文件 `@example['{lineno}'] <file-name>` . `@example` 命令可以在生成的注释中链接到示例文件。`@example` 命令可选参数 `{lineno}` 可以启用示例的行号

| 用法参见: [cmdexample](#)

4. 分组.

| [grouping](#)

1. 定义组 `@defgroup <name> (group title)` . `@defgroup` 命令用于表示注释块中包含类、模块、概念、文件或命名空间主题的文档，也就是用于对符号进行分类。您还可以使用组作为其他组的成员，从而建立组的层次结构。`@defgroup` 命令的参数 `<name>` 是唯一的标识符，且不能有空格。这意味着同一个名字不能 `@defgroup` 两次。`@defgroup` 命令的参数 `(group title)` 是组的标题，括号是可选的，中间可以有空格。可以通过组前的开始标记 `@{` 和组后的结束标记 `@}` 将成员分组在一起。注意 `@}` 后面请注释组的标题，告诉告诉读者当前括号的归属。

```
/*
 * @defgroup Unique_ID_of_the_group 组的标题
 * @{
 *
 /* 你希望添加到组里的内容... */

 /**
 * @} // Unique_ID_of_the_group
 * // ↑ 请重复一遍组的标识告诉读者当前括号的归属。
 */
```

2. 添加组 `@addtogroup <name> [(title)]` . `@addtogroup` 命令用于将代码添加到指定组中，如果组不存在时则会创建组。类似与 `@defgroup`，但是 `@addtogroup` 可以重复使用而不会警告，`@addtogroup` 命令的参数 `<name>` 是标识符。`@addtogroup` 命令的参数

[**(title)**] 是组的标题，是可选项。注意 @} 后面请注释组的标题，告诉读者当前括号的归属。 @addtogroup 用法类似于 @defgroup 。此处不在赘述。

# 参考文献

- [Documenting Code](#)
- [Documenting the code](#)
- [Special Commands](#)

# 编码风格指南

本文说明 xfusion 编码所使用的风格。

请遵守 xfusion 编码风格，统一编码风格可以减少因为风格转换带来的阅读成本。

---

**适用范围：**

- 外部库以外的所有 c 头文件/源文件。

TODO: xfusion cpp 语言代码编程规范

**阅读对象：**

- 所有 c 语言代码贡献者。

**非目标：**

- c 语言代码编程规范。本文只说明风格 xfusion 编码风格，编程规范（如如何提高代码安全性）不是本文目标，可以查阅：
    - [Google C++ Style Guide](#)
    - [MISRA C:2012 Amendment 3](#)
- 

## xfusion 头文件/源文件模板

xfusion 目前已经提供了 xfusion 内 c 语言代码头文件/源文件模板，提交到 xfusion 的代码请应用 xfusion 头文件/源文件模板。

空白模板见 [examples/get\\_started/xf\\_template/blank\\_xf\\_template/](#) 目录内的  
`xf_template.h/xf_template.c`。说明见[模板的说明](#)。

模板说明实例见 [examples/get\\_started/xf\\_template/main](#) 内的  
`xf_template_source_detail.h/xf_template_source_detail.c`。详情见下文。

## 使用自动格式化

详情见：[format\\_code/README.md](#)。

请不要忽略该部分，自动格式化能节省大量的排版工作量。

自动格式化会帮你完成在合适的运算符周围添加空格、缩进代码、限制连续空白行数、限制每行字符数等等工作。

## 模板的说明

该小节介绍 xfusion 的 c 语言头文件/源文件模板的组成部分。

只需了解模板各个部分的作用即可，需要使用时请从

[examples/get\\_started/xf\\_template/blank\\_xf\\_template/](#) 目录内复制出来并修改。

所有 c/cpp 代码文件的总体要求如下（自动格式化会帮你完成这些总体要求）：

1. 使用 4 个空格替代 TAB，并且**不要**使用制表符缩进代码；
2. 行尾**不要**尾随空格；
3. 每行字数**推荐不超过 80 个字符**，最多 120 字符（含）；

如果每行少于 80 个字符，在 vscode 中两栏对照查看代码时能全部看完，而不用水平滚动或开启换行显示。

4. 文件编码格式为 **UTF-8** 格式；
5. 使用 Unix 风格的 **LF** 行结束符，而不是 Windows 风格的 **CRLF** 行结束符。

## 头文件模板

头文件模板由以下几个部分组成：

- 文件描述。如以下代码所示，至少需要包含文件名、作者、简介、版本、日期、版权声明这几个部分。

```
/*
 * @file xf_template.h
 * @author your name (you@domain.com)
 * @brief xfusion 头文件空白模板。
 * @version 0.1
 * @date 2023-10-23
 *
 * Copyright (c) 2023, CorAL. All rights reserved.
 */

```

- 防止头文件重复包含的宏。要点：
  1. 修改宏名为当前文件名的大写格式，不含点'.'以及其他无法作为宏定义的字符。
  2. 以双下划线开头和结尾这个宏。
  3. 在宏的 `#endif` 结尾，注释说明该 `#endif` 的归属，如 `#endif /* __XF_TEMPLATE_H__ */`。

```
C
```

```

#ifndef __XF_TEMPLATE_H__ /*!< 修改为文件名对应的定义 */
#define __XF_TEMPLATE_H__

/* ... */

#endif /* __XF_TEMPLATE_H__ */

```

- `extern "C"`。用于 c++ 程序调用时声明该程序是 c 源程序。注意！`extern "C"` 在 `#include` 之后。

```
C
```

```

#include "xf_def.h"

/* 注意`extern "C"`在`#include`之后。 */
#ifndef __cplusplus
extern "C" {
#endif

/* ... */

#ifndef __cplusplus
} /* extern "C" */
#endif

```

- 内容提示符及对应内容。注意头文件与源文件的内容提示符不完全相同。**其他内容应当与内容提示符保持上下各一空行的间隔。** 头文件的内容提示符包含以下内容：

1. 头文件 **Includes**。

如：`#include "xf_def.h" ;`

2. 宏定义(无参宏) **Defines**。

如：`#define FOO (1) ;`

3. 类型定义 **Typedefs**。

如：`typedef int xf_tmpl_int_t; ;`

#### 4. 全局函数原型 Global Prototypes 。

如: `int global_func(int args);` ;

#### 5. 宏函数(有参宏) Macros 。

如: `#define XF_TEMPLATE_MACROS_ADD(_a, _b) ((_a) + (_b))` ;

/\* 头文件包含的内容提示符 \*/

`/* ===== [Includes] ===== */`

`/* ===== [Defines] ===== */`

`/* ===== [Typedefs] ===== */`

`/* ===== [Global Prototypes] ===== */`

`/* ===== [Macros] ===== */`

---

## 源文件模板

源文件模板由以下几个部分组成:

- 文件描述。格式同头文件模板。
- 内容提示符及对应内容。注意头文件与源文件的内容提示符不完全相同。**其他内容应当与内容提示符保持上下各一空行的间隔。** 源文件的内容提示符包含以下内容:

#### 1. 头文件 Includes 。

如: `#include "xf_template.h"` ;

#### 2. 宏定义(无参宏) Defines 。

如: `#define BAR (1)` ;

#### 3. 类型定义 Typedefs 。

如: `typedef int int_t;` ;

#### 4. 静态函数原型 Static Prototypes 。

如: `int _static_func(int args);` ;

## 5. 静态变量 Static Variables 。

如: `int s_val = 0; ;`

## 6. 宏函数(有参宏) Macros 。

如: `#define XF_TEMPLATE_MACROS_SUB(_a, _b) ((_a) - (_b)) ;`

## 7. 全局函数定义 Global Functions 。

如: `int global_func(int args) { return args; }; ;`

## 8. 静态函数定义 Static Functions 。

如: `int _static_func(int args) { return args; }; .`

```
/* 源文件包含的内容提示符 */

/* ===== [Includes] ===== */

/* ===== [Defines] ===== */

/* ===== [Typedefs] ===== */

/* ===== [Static Prototypes] ===== */

/* ===== [Static Variables] ===== */

/* ===== [Macros] ===== */

/* ===== [Global Functions] ===== */

/* ===== [Static Functions] ===== */
```

# 编码风格说明

该部分将详细介绍编码风格。注释风格请见《[代码注释指南](#)》。该文档说明如何写出符合 doxygen 格式要求的注释。

## 缩进风格

xfusion c 源码缩进风格使用"One True Brace Style", 简称 [1TBS](#) 或 [OTBS](#) , 这是一种基于"K&R"风格的变体。 [1TBS](#) 和 [K&R](#) 最大的区别是, [1TBS](#) 不允许单语句分支时省略花括号。此处

不对缩进风格作详细介绍，详细内容请查阅[Indentation style](#)。

xfusion 缩进风格的示例代码如下，详细示例见

[examples/get\\_started/xf\\_template/main/xf\\_template\\_source\\_detail.c](#)。

```
/*
 * @brief 主函数。
 */
int main(int argc, char *argv[])
{
    xf_tmpl_int_t ret = 0;
    int32_t val = 0;

    /**
     * @brief 分支语句哪怕只有一句也必须加花括号。
     */
    if (XF_TEMPLATE_VERSION != XF_TEMPLATE_VERSION_CHECK(1, 0, 0)) {
        XF_TEMPL_PRINTF("error: version check failed\n");
    } else {
        XF_TEMPL_PRINTF("version check: ok\n");
        XF_TEMPL_PRINTF("version is %lu\n", (uint32_t)XF_TEMPLATE_VERSION);
    } /*!< 必要时此处需要添加判断条件，以说明该花括号的来源 */
}

xf_component_func();
xf_template_another_func();
xf_template_init();

ret = _xf_template_func(&s_struct, &val);
/**
 * @brief 常量在前可以避免不必要的逻辑错误。
 * 如 if (XF_TEMPL_FAIL == ret) 时编译会报错。
 */
if (XF_TEMPL_FAIL == ret) {
    XF_TEMPL_PRINTF("error: ret is XF_TEMPL_FAIL\n");
    return XF_TEMPL_FAIL;
}

XF_TEMPL_PRINTF("ret: %d\n", ret);
XF_TEMPL_PRINTF("ok\n");

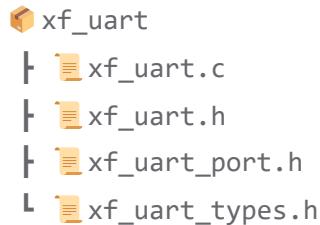
return XF_TEMPL_OK;
}
```

# 命名风格

## 文件及目录命名风格

TODO: 文件及目录命名风格详细说明

如 xf\_uart 所示：



- xf\_uart.h: 当前模块对外提供的功能的头文件。通常提供给用户使用。
- xf\_uart.c: 当前模块功能的的实现。
- xf\_uart\_port.h: 实现当前模块功能所需要的接口。用户通常无需使用。
- xf\_uart\_types.h: 当前模块定义的数据类型。

## 内容(命名及排版)风格

由于头文件内的内容会被别处调用，通过命名空间标识来源是有必要的。

**命名空间：**

对于会被别处调用的代码，除非特殊情况，否则都**要求**加上命名空间（含全局变量，同时**建议不使用**全局变量）。会被别处调用的代码包括但不限于：

1. 宏 (含无参宏与带参宏)；
2. 结构体、共用体、枚举类型等；
3. 类型定义；
4. 函数；
5. 全局变量 (建议不使用)。

## 宏

宏定义(无参宏)是预处理指令中的一种，预处理阶段时会替换宏名为宏对应的替换列表。格式为

#define 宏名 替换列表 。 如：

```
#define XF_TEMPLATE_HELP_STR           "xf_template v0.1"
#define XF_TEMPLATE_HELP_STR_SPLICING   "test" XF_TEMPLATE_HELP_STR "abc123"
```

宏函数(带参宏)。 格式为 `#define` 宏名(参数1,参数2,...,参数n) 替换列表 。 如:

```
/*
 * @brief 带参宏示例。
 */
#define XF_TEMPLATE_MACROS_ADD(_a, _b) ((_a) + (_b))

/*
 * @brief 无需返回参数的宏
 *
 * 1. 通常需要用 do { } while (0) 包围（除非通过宏定义变量等情况）。
 * 2. while (0) 后不要加分号（使用时强制加分号）。
 */
#define XF_TEMPLATE_MACROS_NO_RETURN(_a, _b) \
    do { \
        s_data = (_a) + (_b); \
    } while (0)

/*
 * @brief 需要返回参数的宏
 *
 * 1. 使用 ({ }) 包围。
 * 2. 明显括出返回值。
 */
#define XF_TEMPLATE_MACROS_HAS_RETURN(_x) \
    ({ \
        typeof(_x) __ret = (_x); \
        __ret = __ret + (_x); \
        (__ret); \
    })

/*
 * @brief 关于条件编译
 *
 * 1. 需要在对应的 #else 后追加相应的的条件（如: `!defined(xf_printf)`）,
 * 在 #endif 后标注 #if 的信息（如: `defined(xf_printf)`）。
 *
 * @note 如何宏需要缩进保持美观, 请在'#'号后面缩进。如下缩进所示。
 */
/* xf_template 输出接口 */
#ifndef xf_printf
#define XF_TEMP_PRINTF(_fmt, ...) xf_printf((_fmt), ##__VA_ARGS__)
#else /* !defined(xf_printf) */

```

```

#define XF_TEMPL_PRINTF(_fmt, ...)    printf(_fmt), ##__VA_ARGS__
#endif /* defined(xf_printf) */

#ifndef UNUSED
#define UNUSED(_x)                  ((void)(_x))      /*!< 未使用的变量 */
#endif

```

## 命名要点：

- 根据需要添加命名空间；源文件(.c)内的定义的宏的名字可以不添加命名空间。**推荐**添加。一旦需要被别的文件调用，就**必须**加上命名空间。
- 宏定义(无参宏)的宏名**必须大写**，如有特殊情况不大写宏定义时需要特殊注明；
- 宏函数(带参宏)的宏名**通常大写**，除了非常类似于函数的宏或者编译器特性宏以及其他约定成俗的情况；
- 宏函数(带参宏)的参数对大小写没有要求，但是通常**推荐**添加单下划线 \_ 以说明该参数是宏的参数，并且加以括号，防止表达式传入宏参数时造成逻辑错误；
- 能说明宏的作用。

## 格式要点：

- 宏对应的内容通常建议加括号，如下；

```
#define XF_TEMPLATE_DEFINE (1)
```

- 不要求**对齐多个连续的宏名及其对应的内容，以及宏的继续符 \，除非代码已经稳定很少更改；

```

/* 多个连续的宏，内容不要求对齐 */
#define XF_TEMPLATE_MACROS_ADD(_a, _b)      ((_a) + (_b))
#define XF_TEMPLATE_MACROS_NO_RETURN(_a, _b)   do { s_data = (_a) + (_b); } while (0)
#define XF_TEMPLATE_MACROS_FOO                (1)

/* 宏的继续符`\'对齐的情况，不要求对齐 */
#define XF_TEMPLATE_MACROS_HAS_RETURN(_x) \
{ \
    typeof(_x) __ret = (_x); \
    __ret = __ret + (_x); \
    (__ret); \
}

```

- 如果需要缩进预处理指令，请在在 # 号后面缩进，而不是在 # 号前面缩进。

尽管 astyle 通过 `--indent-preproc-block` 命令可以自动缩进预处理指令，但是实测发现存在错误缩进多行注释的情况，因此暂未使用。

如：

```
/* 正确示例 */
#ifndef UNUSED
#define UNUSED(_x) ((void)(_x))
#endif

/* 错误示例 */
#ifndef UNUSED
#define UNUSED(_x) ((void)(_x))
#endif
```

## 类型定义、结构体、共用体、枚举类型等

xfusion 所有的 c 头文件/源文件都强制使用类型定义去替代结构体、共用体、枚举类型，包括浮点类型、整型等数据类型。

### 命名要点：

#### 1. 类型定义、结构体、共用体。

1. 用**类型定义**替换所有的结构体、共用体、枚举类型。

2. 结构体、共用体、枚举类型的名字用单下划线 `_` 加类型定义的名字。

3. 类型定义的名字需要用 `_t` 后缀结尾，说明该类型是通过类型定义定义的。

4. 通常都需要添加命名空间前缀。

5. 通常使用 `int32_t`、`uint8_t` 等类型定义替换 `int`、`unsigned char` 等数据类型。

6. 如下：

```
/**
 * @brief 结构体示例。
 *
 * @details
 * 1. 结构体**必须**用类型定义。
 * 2. 结构体名字是类型定义名字前加单下划线。如: `_xf_temp_struct_t`。
 *   当然用 xf_temp_struct_s 也可以。
 */
typedef struct _xf_temp_struct_t {
    xf_temp_int_t num;           /*!< 这是一个数字 */
}
```

```

char *p_str; /*!< 这是一个字符串指针, 前缀`p_`强调指针类型 */

union {
    uint8_t all; /*!< 通过这个值可以修改整个共用体 */
    struct {
        uint8_t val_u4: 4; /*!< 这是位域的示例, u4 表示有 4 位, 根据位置可能
                             使用不同的位数 */
        uint8_t val_bit4: 1; /*!< 这是位域的示例, bit4 表示的是从 bit0 起数的
                             第一个位 */
        uint8_t reserved: 3; /*!< 这是位域中未使用的位 */
    } bits; /*!< 如果使用了英文缩写, 应当在此说明缩写的原文 */
} data;
} xf_temp_struct_t;

```

## 7. 命名示例: TODO: 类型定义命名示例。

### 2. 枚举类型。

1. 枚举元素要求和宏定义一样, 无特殊情况都**必须大写**。

2. 不使用枚举类型定义变量 (如以下示例代码中的 `xf_temp_enum_t` ), 通常只使用枚举类型定义的枚举元素。

为了避免不同编译器为枚举类型分配不同大小。如 `components/xf_def/xf_err.h` 内的 `xf_err_code_t` 和 `xf_err_t` 。使用 `xf_err_code_t` 内的枚举值, 但不使用 `xf_err_code_t` 定义变量。

3. 枚举元素通常要求有一个最大值, 并且该最大值通常不作为正常值使用。

4. 除了只做索引的枚举元素以外, 应当用**功能**代替其中的数字。

5. 如:

```

/**
 * @brief 枚举类型示例。
 *
 * @details
 * 1. 枚举类型通过类型定义重命名。
 * 2. 枚举类型命名是类型定义名字前加单下划线。如 "_xf_temp_enum_t"。
 * 3. 枚举类型的值需要大写。
 * 4. 枚举值通常要求有一个最大值, 并且该最大值使用时通常不作为正常值。
 */
typedef enum _xf_temp_enum_t {
    XF_TEMP_ENUM_0 = 0x00, /*!< 枚举值 0, 第一个枚举值通常要求手动赋值 */
    XF_TEMP_ENUM_1, /*!< 枚举值 1 */
    XF_TEMP_ENUM_2, /*!< 枚举值 2 */
    /* 此处保留一行空行, 以区分正常值和最大值 */
    XF_TEMP_ENUM_MAX, /*!< 枚举值最大值 */
}
```

```
XF_TEMPL_ENUM_DEFAULT = XF_TEMPL_ENUM_1,      /*!< 枚举值默认值 */
} xf_temp_t;
```

## 函数、变量、常量、goto 标签等

xfusion c 语言代码都采用 [snake\\_case](#) 风格命名。也就是 [unix like](#) 风格。如 `xf_evt_attach()`、`xf_evt_sys_init()` 等等。如需缩写，请从[缩写词表](#)中选取。

### 命名要点：

#### 1. 函数。

函数命名的总体原则是： {主语}\_{谓语}\_{宾语} 。

1. {主语} 通常由 {前缀}\_{模块} 组成，是发起动作的主体。

1. {主语} **不可省略**。

2. 如 `xf_spi` 表示 xfusion 中的 spi 模块（隐含 xf\_hal）。

3. 如 `xf_spi_dev` 表示发起动作的主语（主体）是 spi 设备(device)。

2. {谓语} 即为动作，动作的承接对象可能是主语，也可能是宾语，取决于是否有宾语。

1. {谓语} **不可省略**。

好的示例如 `xf_uart_get_tx_buffer_free()`，表示在 uart 中获取发送缓冲区剩余空间大小。而如果去除 `get`，`xf_uart_tx_buffer_free()` 的则会产生歧义，可能的含义有，1. 同包含 `get` 的情况；2. 发送缓冲区剩余空间大小。

2. {谓语} 优先使用具有相反含义的词组。词组列表见附录[反义或互斥词组](#)。

1. 具有相反含义的常用词组的有：获取/设置(get/set)、初始化/反初始化(initialize/deinitialize)、获取/释放(acquire/release)、创建/销毁(create/destroy)、添加/移除(add/remove)、上锁/解锁(lock/unlock)等。

2. 其余没有相反含义的常用词组：延时(delay)、是否(is)、检查(check)等。

3. {宾语} 即为动作的承接对象。

1. {谓语} 可以省略。省略时表示动作的对象就是主语自己。如 `xf_delay_ms()` 表示以 ms 为单位延时（隐含主语为系统或当前线程）。如 `xf_systime_init()` 表示初始化 `xf_systime` 模块。

#### 2. 变量、常量。

1. 前缀。前缀有助于使用者在使用时辨别当前变量的类型，目前有以下常用几种前缀。前缀可以组合使用。

1. `s_`，表示静态变量，无论是函数内或者文件内定义的静态变量都必须用 `static` 修饰；

如：`components/xf_log/xf_log.c` 中的 `static xf_log_level_t s_global_level = (xf_log_level_t)XF_LOG_DEFAULT_LEVEL;`。

2. `g_`，表示全局变量，与 `s_` 的区别是能否通过 `extern` 等方式直接访问，只要能被直接访问（不含通过指针访问）到的变量，都必须用 `g_`。

3. `c_`，表示常量。位于只读数据段的数据。对于常量推荐使用 `c_` 前缀。

1. 常量可以在头文件中声明，并在别的文件内调用。

2. 有 `const` 修饰的变量不一定是常量。

3. 对于指针应当特别小心，只有满足：1. 该指针不可改变指向；2. 不可通过该指针修改指向的空间的数据；3. 该指针指向的空间的数据不会改变。才能直接在头文件中声明。以下示例说明了能在头文件中声明的指针的示例。

```
/* 一个变量数组 */
char buf[10] = {0};

/* 以下情况都可能存在数据竞争，都不能在头文件中直接声明 */

/* 指向变量数组的指针变量 */
char *p_buf = buf;
/* 指向变量数组的指针常量 */
char *const pc_buf = buf;
/***
 * @brief 指向变量数组的指针变量，且不可通过当前指针修改指向的空间。
 *
 * @note 尽管不可通过当前指针修改指向的空间，
 * 但是指针的指向可能改变，存在数据竞争风险。
 */
const char *cp_buf = buf;
/***
 * @brief 指向变量数组的指针常量，且不可通过当前指针修改指向的空间。
 *
 * @note
 * 尽管不可通过当前指针修改指向的空间，指针的指向也不能改变，
 * 但是指向的空间是变量数组，在通过 cpc_buf 读取数据过程中，
 * 数据有可能改变，因此存在风险。
 * 如：
 * 初始时，buf[0...9] == "abcdef\0\0\0\0",
 * 通过 cpc_buf 读取 3 个字符后，线程切换到修改 buf 的线程，并修改 buf:
 * buf[0...9] == "012345\0\0\0\0", 线程再切换回通过 cpc_buf 读取数据的线程:
 * 最终读取的数据是: "abc345\0\0\0\0"
 */
```

```

*/
const char *const cpc_buf = buf;

/* 能在头文件中声明的情况是（命名空间假设为 xf_tmp） */

/* .h */
extern const char *const g_xf_tmp_character_literal;
/* .c */
const char *const g_xf_tmp_character_literal = "hello world";

/* 实际上也可以在 .h 中直接 */
// const char *const g_xf_tmp_character_literal = "hello world";

```

4. **p\_**，表示**指针**。**强烈推荐**指针使用 **p\_ 前缀**。二级指针可以用 **pp\_ 或者 p\_xxx\_ptr**。二级指针以上的级别的指针通常不使用。

3. goto 标签。通常**建议**单下划线开头，如 **\_xf\_xxx\_init\_err**，表示 xxx 模块初始化错误时的处理，如释放资源等等。

---

## 内容编排风格

详情见详细示例 [examples/get\\_started/xf\\_template/main/xf\\_template\\_source\\_detail.c](#)。

### 👉 [xf\\_template\\_source\\_detail.c](#) 👈

请根据内容提示符编写代码。

## 头文件顺序

1. 当前组件的公共头文件，也就是当前组件对外提供的功能的头文件；如：`#include "xf_template_header_detail.h"`；
2. 标准库，如 `#include <stdio.h>`；
3. 其他 POSIX 标准标头，如：`#include <sys/time.h>`；
4. 本文件所需要的其他组件的头文件，如：`#include "xf_log.h"`、`#include "xf_spi.h"`；
5. 本文件所需要的当前组件的其他头文件或者私有头文件。

备注：“当前组件的公共头文件”也可放到“本文件所需要的当前组件的其他头文件或者私有头文件”之前 项目中很多历史遗留代码还没修改顺序，请以该顺序为准。

如：

```
/* ===== [Includes] ===== */
#include "xf_template_header_detail.h"

#include <stdio.h>

#if ENABLE_XF_HAL
#include "xf_hal.h"
#endif /* ENABLE_XF_HAL */

#include "xf_component_template.h"
```

## 其余细则

- 不要以空行开始或者结束函数。

```
void function1()
{
    do_one_thing();
    do_another_thing();
    /* 不正确，函数结尾不要空行 */
}

/* 函数和函数之间需要空行 */

void function2()
{
    /* 不正确，函数开始不要空行 */
    int var = 0;
    while (var < SOME_CONSTANT) {
        do_stuff(&var);
    }
}
```

- 在条件和循环关键字后添加单空格（自动格式化会自动完成）。

```
if (condition) { /*!< 正确 */
    /* ... */
}
switch (n) { /*!< 正确 */
    case 0:
    /* ... */
}
for(int i = 0; i < CONST; ++i) { /*!< 不正确，关键词周围缺少空格 */
```

```
/* ... */
```

```
}
```

- 每行最大长度为 120 个字符（超长时自动格式化会自动换行）。
- 在合适的运算符周围添加空格（自动格式化会自动完成）。
- 如果不再需要某些代码，请将其完全删除。如果需要临时禁用，请在注释的代码周围说明原因。`#if 0 ... #endif` 代码块同样。

## 附录

### 缩写词表

中文	英文	缩写
参数	argument	arg
参数	parameter	param
描述符	descriptor	dsc
缓冲区	buffer	buf
命令	command	cmd
配置	configuration	cfg
控制	control	ctrl
比较	compare	cmp
位置	position	pos
错误	error	err
时钟	clock	clk
设备	device	dev
消息	message	msg
字符串	string	str
回调	call-back	cb
分配	allocate	alloc
临时	temp	tmp

中文	英文	缩写
对象	object	obj
同步	synchronize	sync
信号量	semaphore	sem
互斥量	mutex	mtx
注册/寄存器	register	reg
之前	previous	prev
当前	current	curr
最大	maximum	max
最小	minimum	min
增加	increment	inc
减少	decrement	dec
初始化	initialize	init
反初始化	deinitialize	deinit

## 反义或互斥词组

中文	英文
添加/删除	add/remove
添加/删除	add/delete
开始/结束	begin/end
创建/销毁	create/destroy
插入/删除	insert/delete
递增/递减	increment/decrement
锁定/解锁	lock/unlock
旧/新	old/new
源/目标	source/target

中文	英文
源/目标	source/destination
第一个/最后一个	first/last
放入/取出	put/get
打开/关闭	open/close
启动/停止	start/stop
显示/隐藏	show/hide
复制/粘贴	copy/paste
获取/释放	acquire/release
最小/最大	min/max
当前/之前	current/previous
当前/之后	current/next
下一个/之前	next/previous
发送/接收	send/receive
上/下	up/down

## 参考文献

1. [Espressif IoT Development Framework Style Guide](#)
2. [LVGL Coding style](#)
3. [Artistic Style 3.4.12 Documentation](#)
4. [RT-Thread 编程风格](#)

# 文档指南

文档是用于帮助其他人了解你的代码、功能或者构思等信息的载体。

本文说明编写 xfusion 文档需要遵守的格式、内容等规则。

---

## 适用范围：

- 外部库以外的所有文档，目前 xfusion 文档以 markdown 格式为主。

## 阅读对象：

- 所有贡献者。
- 

本文假定读者已经对 markdown 语法有充分的了解，如对参考 markdown 语法，请参考[参考文献](#)中的 [2-3] 。

# 文档格式

xfusion 文档目前只提供中文和英文两种语言，为了管理不同语言文档中的图片，请将文档中使用到的图片统一放到 [doc/public/image](#) 路径下。

编写文档时，请遵循以下**文档规则**：

### 1. 一个段落写在同一行内。

**错误示例如下。**这个示例中将一段或者一句话分成了很多行。

我是一段很长的段落。在这个段落中，我将反复强调一个观点，那就是我是一段很长很长的段落。这个段落的目的是为了达到一定的字数，通过不断地重复“我是一段很长的段落”来实现。这种写作方式可能单调，但它有效地传达了信息，即我是一段很长的段落。总之，这个段落的核心就是它的长度和重复性。

markdown

尽管现在的长度已经缩减，但核心思想仍然不变。这种重复的写作手法，虽然可能显得有些单调，却能够清晰地传达出一个信息：我是一段很长的段落。这就是这个段落存在的意义，它通过简洁的语言和重复的结构，向读者展示了其核心内容。总结来说，这个段落的主旨依然是它的冗长和重复，这是其独特的表达方式。

**正确的示例如下。** markdown 在查看时通常会打开自动换行，不需要手动换行。一段话一行有助于在翻译软件中快速翻译，而不需要手动删除换行。

我是一段很长的段落。在这个段落中，我将反复强调一个观点，那就是我是一段很长很长的段落。这个段

尽管现在的长度已经缩减，但核心思想仍然不变。这种重复的写作手法，虽然可能显得有些单调，却能够

## 2. 对齐中英文文档的行号。

这样做能帮助译者节省翻译的时间，而且通过比较行数也能在一定情况下反应某个语言的文档的领先和落后情况。

## 3. 使用自动格式化，并注意一些格式细节。

推荐使用 vscode 插件 [esbenp.prettier-vscode](#) 来完成格式化与 markdown 预览等功能。通过该插件可以使用 `alt + shift + f` 快捷键来快速格式化 markdown 文档而不用鼠标右键格式化或者手动格式化。

### 1. 在中英文、数字间适当插入空格。

尽管很多渲染器在预览时可以自动插入空格，但还是有很多不支持。插入空格的 markdown 文档在阅读源码时更加美观。例如： [你说的对，但是《STM32》是由意法半导体（ST）推出的一系列 32 位的单片机。](#) 。

### 2. 使用 \* 而不是 \_。

例如需要加粗的部分使用 `**粗体**`、而不是 `_粗体_`。因为 `_` 可能无法被正确识别，而且在输入中文时输入 `_` 需要频繁切换中英文输入。

### 3. 代码块中标注正确的语言种类，以提供语法高亮。

4. 建议在说明复杂的逻辑时使用 `mermaid`、`wavedrom` 绘制图表、时序图辅助说明。

5. 不建议在 markdown 文档内使用注释，如 `<!-- -->`。

## 4. 文件格式：

1. 使用 3 或者 4 个空格替代 `TAB`，并且**不要**使用制表符缩进代码；
2. 行尾**不要**尾随空格；
3. 文件编码格式为 **UTF-8** 格式；
4. 使用 Unix 风格的 `LF` 行结束符。
5. markdown 源码也需要保证一定的美观。 TODO: markdown 源码格式具体规定。

本节介绍文档中应当有什么内容。 目前主要说明 example 的介绍文档以及组件的说明文档应当有哪些内容。

---

## example 的介绍文档

TODO: example 的介绍文档模板。

example 的介绍文档需要包含以下内容：

### 1. 支持情况。

该示例支持的芯片或平台的概况。如： `stm32103c8` , `esp32` 。

### 2. 示例简述。

这个示例是什么？提供了什么功能或者有什么作用？

### 3. 如何使用。

#### 1. 所需的软件和硬件。

1. 软件如平台支持包。示例需要的必要组件，如无特殊情况，必须集成到示例中，而不需要再次克隆子模块等。

2. 硬件包含芯片平台、所需要的片外外设，外设和芯片需要如何连接。

2. 示例提供了什么配置？用户需要修改什么示例配置才能正常运行？

3. 构建和烧录有什么步骤与特殊要点？

### 4. 运行现象。

运行该示例有什么输出和现象？

1. 描述运行现象。

2. 放出运行日志。

### 5. 常见问题。

示例有什么常见问题？需要如何解决？

### 6. 示例平台差异。

该示例在不同平台上运行会有什么不同表现？如果没有差异，该部分可以省略。如果有差异，需要怎么做才能屏蔽区别？如果无法屏蔽，请强调差异。

### 7. 示例详解。 (可以省略)

主要介绍示例中的重点难点代码。

## 8. 测试环境。

需要列出以下信息确保其他人能够复现示例。

1. 芯片型号；
  2. 芯片开发环境的 SDK 版本号；
  3. 工具链名称及版本号；
  4. 平台支持包版本号（如实现 `xf_hal` 的下层硬件驱动）。
- 

## 组件的介绍文档

### Note

组件的介绍文档模板。

## 参考文献

1. [编写文档 - ESP32 - — ESP-IDF 编程指南 latest 文档 \(espressif.com\)](#)
2. [Markdown 教程 | 菜鸟教程 \(runoob.com\)](#)
3. [Markdown 入门基础 | Markdown 官方教程](#)

# 贡献指南

非常感谢您对 xfusion 作出贡献！xfusion 在 Apache-2.0 开源许可证下开源。

本文说明如何对 xfusion 作出贡献。

---

**阅读对象：**

- 所有贡献者。
- 

## 准备工作

提交 Pull Request(以下简称 PR) 前, 请检查以下条目:

- 许可证:
  - 如果提交的代码不是您从零开始写的, 请检查是否与兼容 xfusion 的开源许可证? xfusion 只接受兼容 Apache-2.0 许可证的代码。
  - 如果您是公司的雇员, 通常您代码的版权属于公司, 因此这意味着您提交代码前必须获得公司的授权, 并提交相应的许可协议。
- PR 提交步骤:
  - 做任何重要的贡献之前, 请检查 issue 中是否有相关讨论, 或者提出 issue 询问想法, 避免浪费您的精力。
  - 请参考[PR 提交步骤](#)。
- 代码风格:
  - 是否符合 xfusion 的[编码风格指南](#)?
- 代码注释:
  - 是否符合 xfusion 的[代码注释指南](#)?
  - 代码必要的注释是否充分?
  - 注释描述是否明确无歧义?
  - 是否注明了注意事项?
- 说明文档:

- 如果贡献的是较大的功能块，是否提供了符合 xfusion 的[说明文档指南](#)的说明文档或者使用说明？
- 签署 CLA：如果您是第一次向 xfusion 贡献代码，则需要在 PR 页面签署 Contributor License Agreement(CLA)。

## 子文档

- [代码注释指南](#)
- [编码风格指南](#)
- [文档编写指南](#)
- [PR 提交步骤](#)

# Pull Request 提交步骤

本文说明 xfusion 的 Pull Request 提交步骤。

## 阅读对象：

- 所有需要提交代码的贡献者。

## Pull Request 是什么

Pull Request (以下简称 PR) 和 Merge Request (以下简称 MR) 都是代码协作中**用于请求将代码更改合并到主分支的机制**。

当你想要贡献代码到一个项目时，你通常需要从原项目中 fork 一份副本，然后在你的副本上进行更改。更改完成后，你会向原项目发起一个 Pull Request，请求项目维护者拉取(pull)你的更改并合并到他们的项目。

Pull Request 是在 GitHub 上使用的术语，而 Merge Request 通常与 GitLab 关联，只是 Merge Request 更直接地反映了请求的最终操作，即合并(merge)代码到主分支。因此 **PR 和 MR 在下文中不作区分**。

## xfusion 的 Pull Request 提交步骤

xfusion 不允许直接推送代码到主分支(main)，因此您必须先要 fork 一份副本。以下是具体的操作方法：

本文假设读者已经安装好了 git，并且注册了 GitHub 账户。

### 1. Fork 项目。

访问{xfusion 仓库链接}，并且点击页面右上角的 Fork 按钮，fork 一份 xfusion 副本。

### 2. 克隆仓库。

- 打开您 fork 的 xfusion 仓库副本网页，点击网页上的 [Code](#) 获取 https 克隆链接。
- 然后打开您的本地终端，克隆您 fork 的仓库。

```
# 克隆仓库
git clone --recursive {您 fork 的 xfusion 仓库链接} xfusion
cd xfusion
# 添加上游仓库，即 xfusion 原始仓库
git remote add upstream {xfusion 仓库链接}
```

### 3. 创建新的分支。

在新分支上修改，有几个优势：

1. **能够保持主分支干净。**
2. **易于管理：**如果你在主分支上直接进行开发，那么每次上游仓库更新时，你都需要处理合并冲突。而如果你在不同的分支上工作，就可以更容易地拉取上游的更新，并且在必要时只合并你的特定更改。
3. **并行开发：**创建新分支可以让你同时在多个功能上工作，而不会互相干扰。这对于处理多个问题或添加多个功能特别有用。
4. **代码审查：**在单独的分支上工作可以让其他贡献者更容易地审查你的代码，因为它们只包含相关的更改。

可以通过以下命令创建新的分支。

```
# 切换到新的分支
git checkout -b local-branch
# 推送 local-branch 到远端
git push
```

### 3. 做出修改。

您的修改可以是修复代码或者文档的 bug，提交新的功能等等，xfusion 欢迎您的任何与 xfusion 发展方向相符的贡献。

请注意，xfusion 的**每个 PR 只接受 1 个 commit**，因此每次 PR 不要涉及不同的方面。如需修改多个方面，请创建多个分支，各自修改后再提交 PR。

#### 1. 做出修改。

```
# 此时已经在 local-branch 分支了
# 做出您的修改，此处以 my-file.c 为例子
vim my-file.c
```

#### 2. 检查风格。

您的代码应当符合[贡献指南](#)中提到的[编码风格指南](#)等注意事项。您也可以用格式化脚本先格式化您的代码。

bash

```
python ${您的xfusion路径}/tools/format_code/format.py my-file.c
```

### 3. 推送到 fork 仓库中。

bash

```
# 将 my-file.c 的修改添加到暂存区  
git add my-file.c  
# 提交暂存区到本地仓库, 注意 commit 消息的格式, 这在下文可以找到  
git commit my-file.c  
# 推送到您 fork 的 xfusion 远端仓库  
git push
```

NOTE: commit 消息的格式请见下文: [👉 commit 消息的格式 👈](#)

### 4. 保持同步。

当您测试了代码后就可以准备提交了。在提交前请确保您的 fork 和上游保持同步。

bash

```
git checkout main  
# 拉取上游, 也就是 xfusion 原始仓库  
git fetch upstream  
# 合并上游到本地  
git merge upstream/main  
git push
```

将本地 main 分支与本地新建的分支合并。

bash

```
git checkout local-branch  
git merge main  
git push
```

### 5. 根据需要重复以上步骤。

### 6. 如果你的修改已经完毕, 但是有多个 commit, 再提交前请用 rebase 来压缩他们。

如果您不清楚如何才能压缩他们, 请参考: [将 Github 拉取请求压缩到一个提交中- Eli Bendersky 的网站 --- Squashing Github pull requests into a single commit - Eli Bendersky's website \(thegreenplace.net\)](#)

## 4. 创建 Pull Request。

1. 在 git 仓库中选择需要合并到主分支的分支，这里是 [local-branch](#)，点击 create Pull Request 按钮创建 Pull Request。
2. 请确认提交前的检查清单。
3. 签署 CLA。
4. 创建 Pull Request 成功后，审核人员会审核您的代码，相关意见会在 Pull Request 页面中反馈给您，您需要根据意见修改。一旦审核人员认为您的修改没有问题了，请及时压缩到一个 commit，之后审核人员通过您的贡献。

## commit 消息的格式

xfusion 目前使用 vscode 插件 [redjue.git-commit-plugin](#) 生成 commit 消息。

---

### 格式

格式遵循 [Angular Team Commit Specification](#)，如下所示：

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

### type(类型)

必须是以下之一：

Type	Description
init	项目初始化
feat	添加新特性
fix	修复 bug
docs	仅仅修改文档
style:	不影响代码逻辑的更改（仅仅修空格、格式、缺少分号等）
refactor	既不修复错误也不添加功能的代码更改

Type	Description
perf	优化相关，比如提升性能、体验
test	添加或纠正现有测试
build	依赖相关的内容
ci	ci 配置相关
chore	对构建过程或辅助工具和库（例如文档生成）的更改
revert	回滚到上一个版本

## scope(修改范围)

范围可以是指定提交更改位置的任何内容。

修改范围是**必填**项目，目前使用的格式约定如下：

最外层目录名-修改的模块 .

例如：

```
fix(example-gatt): 延时改xf_task; 修正部分log输出
docs(ports-ws63): 上传readme
fix(components-xf_hal..): 更新日志等级
```

txt

## subject(概述)

概述是对更改的简要描述：

- 使用祈使式、现在时：“change” not “changed” nor “changes”。
- 不要将首字母大写。
- 结尾无点(.)。
- 最多 20 个字符。
- 目前以中文为主，不排除修改为英文的可能。

## body(详情)

用于描述此更改的详情。

## 备注

备注通常是修复 bug 的链接。

**重大变更**应以 `BREAKING CHANGE`: 一词开头，并带有一个空格或两个换行符。

格式详情见：[RedJue/git-commit-plugin: Automatically generate git commit \(github.com\)](https://github.com/RedJue/git-commit-plugin)

## 本文待办事项

TODO: 1. 持续集成 (CI) Continuous Integration (CI)。 TODO: 2. 替换链接 [{xfusion 仓库链接}](#)，给出详细的步骤截图。 TODO: 3. 预 commit。 TODO: 4. git 相关教程链接。 TODO: 5. 使用 vscode 相关插件优化步骤。 TODO: 6. rebase 具体步骤。见[使用 Git 进行更改](#)。 TODO: 7. 提交前的检查清单。

## 参考文献

- [使用 Git 进行更改- NuttX latest 文档 --- Making Changes Using Git — NuttX latest documentation \(apache.org\)](#)
- [NuttX RFC 0001: 代码贡献工作流- NUTTX - Apache 软件基金会 --- NuttX RFC 0001: Code Contribution Workflow - NUTTX - Apache Software Foundation](#)
- [贡献- LVGL 文档 --- Contributing — LVGL documentation](#)
- [投稿指南-ESP 32- - ESP-IDF 编程指南最新文档 --- Contributions Guide - ESP32 - — ESP-IDF Programming Guide latest documentation \(espressif.com\)](#)
- [Angular 提交格式参考表 --- Angular Commit Format Reference Sheet \(github.com\)](#)
- [RedJue/git-commit-plugin: Automatically generate git commit \(github.com\)](https://github.com/RedJue/git-commit-plugin)

# 快速入门

本文介绍 xfusion 的安装步骤、使用方法等基础内容。

## 阅读对象：

- xfusion 用户。

# 环境需求

目前主要使用的开发环境是：Windows + Linux 虚拟机。

通常工作流程是在 Windows 使用 VSCode 通过 ssh 连接到虚拟机，在 Windows 上编辑代码，在 Linux 上编译，后根据目标平台在 Windows(WS63 使用 BurnTool 烧录)或者 Linux 上烧录。

本文假设读者已安装好 VMware Ubuntu20.04 或者 WSL 虚拟机，并且配置好了 SSH 远程连接、VSCode 开发环境、git。如果没有安装，可以参考以下链接安装：

## 1. 安装虚拟机 (VMware 或 WSL 二选一即可)

### 1. VMware Ubuntu 虚拟机。

1. [使用 VMWare 安装 Ubuntu 并使用 vscode 远程连接到虚拟机 | 邱维东 \(qiu-weidong.github.io\)](#)

2. [VMware 安装配置 Ubuntu \(最新版、超详细\) \\_vmware-workstation-full-17.5.1-23298084.exe-CSDN 博客](#)

### 2. WSL 虚拟机。

1. [设置 WSL 开发环境 | Microsoft Learn](#)

2. [开始通过 WSL 使用 VS Code | Microsoft Learn](#)

# 前置准备

如果没有安装 python，请先安装 python 3.8 以上的版本的 python.

## 1. 打开虚拟机终端。

## 2. 安装 python。

```
sudo apt-get install python3 python3-pip
```

# 安装 xfusion

如果您有 xfusion release 压缩包（含子模块的完整 release），您可以使用《[通过压缩包](#)》的步骤安装 xfusion 本体。否则请通过《[通过 git 链接](#)》的步骤安装。

## 通过压缩包

1. 打开虚拟机终端。
2. 解压 xfusion 到你想放到的文件夹。

```
cd ~
# 创建你希望放置的文件夹
mkdir development; cd development
# 通过 ssh 或其他方式将 xfusion.xxxxxxx.tar.gz 复制到虚拟机
# 此处省略
# 解压
tar -xvzf xfusion.xxxxxxx.tar.gz
cd xfusion
# 如果 xfusion 目录内不存在 sdks 文件夹, 请创建它
mkdir sdks
```

## 通过 git 链接

TODO 更新 xfusion 链接

1. 打开虚拟机终端。
2. 克隆仓库。

由于写该文档时还没有公开的 git 仓库链接，下文用 <http://xxx/xfusion/xfusion.git> 替代。

```
cd ~
# 创建你希望放置的文件夹
mkdir development; cd development
# 克隆 xfusion 仓库, 目前请不要递归克隆
# 如果公开的 git 仓库有 feature 分支, 请切换到 feature
```

```
git clone http://xxx/xfusion/xfusion.git -b feature
cd xfusion
# 初始化和更新子模块
git submodule init
git submodule update
# 如果 xfusion 目录内不存在 sdks 文件夹, 请创建它
mkdir sdks
```

# 安装具体平台 SDK

xfusion 本身不含工具链，需要安装对应平台的开发环境才能编译 xfusion 的代码。

xfusion 目前支持以下平台，您可以根据需要安装对应平台的 sdk，请至少选择一个平台安装一次开发环境：

1. 《[从 esp32 开始](#)》(基于 esp-idf v5.0)
2. 《[从 ws63 开始](#)》(HI3863 芯片)

至此您应当可以在某个平台上编译烧录了，您可以看看接下来的[实用技巧](#)帮助你配置 VSCode 的开发环境，或者看看[工程相关](#)了解如何创建带有用户组件的工程，以及如何安装外部组件。

在导出 xfusion 环境变量后可以直接通过 `xf` 命令来查看所支持的命令。

```
xf                                     txt
Usage: xf [OPTIONS] COMMAND [ARGS]...
Options:
  -v, --verbose  Enable verbose mode.
  -r, --rich      Enable rich mode.
  -t, --test      Enable test mode.
  --help          Show this message and exit.

Commands:
  build          编译工程
  clean          清空编译中间产物
  create         初始化创建一个新工程
  export         导出对应sdk的工程（需要port对接）
  flash          烧录工程（需要port对接）
  install        安装指定的包
  menuconfig     全局宏的配置
  search         模糊搜索包名
  uninstall      卸载指定的包
  update         更新对应sdk的工程（需要port对接）
  version        查询当前版本
```

# 实用技巧

如果您使用 VSCode 开发，这些《[实用技巧](#)》可以使你的 VSCode 更好用，比如配置智能感知实现精准的代码提示、配置代码格式化等。

## 子文档

- [从 esp32 开始](#)
- [从 ws63 开始](#)
- [实用技巧](#)
- [工程相关](#)

[plugin:vite:import-analysis] Failed to resolve import  
"/image/practical\_tips-select\_intellisense.png" from  
"doc/zh\_CN/get-started/practical\_tips.md". Does the file exist?

D:/projects/document/doc/zh\_CN/get-started/practical\_tips.md:44:  
60

```
3 | const _sfc_main = {name:"doc/zh_CN/get-started/practical_ti
4 | import { createElementVNode as _createElementVNode, createT
5 | import _imports_0 from '/image/practical_tips-select_intell
|           ^
6 | import _imports_1 from '/image/practical_tips-macro_expande
7 |
```

```
at TransformPluginContext._formatError (file:///D:/projects/document/node_mo
at TransformPluginContext.error (file:///D:/projects/document/node_modules/.
at normalizeUrl (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4_
at async file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4_types+u
at async Promise.all (index 1)
at async TransformPluginContext.transform (file:///D:/projects/document/node_
at async PluginContainer.transform (file:///D:/projects/document/node_module
at async loadAndTransform (file:///D:/projects/document/node_modules/.pnpm/v
```

---

Click outside, press **Esc** key, or fix the code to dismiss.

You can also disable this overlay by setting `server.hmr.overlay` to `false` in `vite.config.js`.

# 工程相关

本文介绍 xfusion 中的工程相关内容，如新建带有用户组件的工程、安装外部组件。

---

**阅读对象：**

- xfusion 用户。
- 

## 子文档

- [新建工程与用户组件](#)
- [安装外部组件](#)

```
[plugin:vite:import-analysis] Failed to resolve import  
"/image/install_external_components-components.png" from  
"doc/zh_CN/get-started/project/install_external_components.md".  
Does the file exist?
```

D:/projects/document/doc/zh\_CN/get-started/project/install\_external\_components.md:22:63

```
3 | const _sfc_main = {name:"doc/zh_CN/get-started/project/inst
4 | import { createElementVNode as _createElementVNode, createT
5 | import _imports_0 from '/image/install_external_components-
|           ^
|           ^

6 | import _imports_1 from '/image/install_external_components-
7 | import _imports_2 from '/image/install_external_components-
```

```
at TransformPluginContext._formatError (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4/lib/index.js:111:11)
at TransformPluginContext.error (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4/lib/index.js:109:11)
at normalizeUrl (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4/lib/index.js:100:11)
at async file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4/_types/index.js:1:1
at async Promise.all (index 1)
at async TransformPluginContext.transform (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4/lib/index.js:111:11)
at async PluginContainer.transform (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4/lib/index.js:109:11)
at async loadAndTransform (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4/lib/index.js:100:11)
```

Click outside, press **Esc** key, or fix the code to dismiss.

You can also disable this overlay by setting `server.hmr.overlay` to `false` in `vite.config.js`.

```
[plugin:vite:import-analysis] Failed to resolve import  
"/image/new_project_and_user_component-menu_user_foo.png" from  
"doc/zh_CN/get-  
started/project/new_project_and_user_component.md". Does the  
file exist?
```

D:/projects/document/doc/zh\_CN/get-started/project/new\_project\_a  
nd\_user\_component.md:159:69

```
3 |   const _sfc_main = {name:"doc/zh_CN/get-started/project/new_  
4 |   import { createElementVNode as _createElementVNode, createT  
5 |   import _imports_0 from '/image/new_project_and_user_compone  
|           ^  
6 |  
7 |
```

```
at TransformPluginContext._formatError (file:///D:/projects/document/node_mo  
at TransformPluginContext.error (file:///D:/projects/document/node_modules/.  
at normalizeUrl (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4_  
at async file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4@types+n  
at async Promise.all (index 1)  
at async TransformPluginContext.transform (file:///D:/projects/document/node  
at async PluginContainer.transform (file:///D:/projects/document/node_module  
at async loadAndTransform (file:///D:/projects/document/node_modules/.pnpm/v
```

---

Click outside, press **Esc** key, or fix the code to dismiss.

You can also disable this overlay by setting `server.hmr.overlay` to `false` in `vite.config.js`.

[plugin:vite:import-analysis] Failed to resolve import  
"/image/starting\_with\_esp32-compilation\_success.png" from  
"doc/zh\_CN/get-started/starting\_with\_esp32.md". Does the file  
exist?

D:/projects/document/doc/zh\_CN/get-started/starting\_with\_esp32.m  
d:42:65

```
3 | const _sfc_main = {name:"doc/zh_CN/get-started/starting_wit  
4 | import { createElementVNode as _createElementVNode, createT  
5 | import _imports_0 from '/image/starting_with_esp32-compilat  
| ^  
6 | import _imports_1 from '/image/starting_with_esp32-example_  
7 |
```

```
at TransformPluginContext._formatError (file:///D:/projects/document/node_mo  
at TransformPluginContext.error (file:///D:/projects/document/node_modules/.  
at normalizeUrl (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4_  
at async file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4@types+n  
at async Promise.all (index 1)  
at async TransformPluginContext.transform (file:///D:/projects/document/node  
at async PluginContainer.transform (file:///D:/projects/document/node_module  
at async loadAndTransform (file:///D:/projects/document/node_modules/.pnpm/v
```

---

Click outside, press **Esc** key, or fix the code to dismiss.

You can also disable this overlay by setting `server.hmr.overlay` to `false` in `vite.config.js`.

[plugin:vite:import-analysis] Failed to resolve import  
"/image/starting\_with\_ws63-menu\_close\_porting\_xf.png" from  
"doc/zh\_CN/get-started/starting\_with\_ws63.md". Does the file  
exist?

D:/projects/document/doc/zh\_CN/get-started/starting\_with\_ws63.md:83:66

```
3 |   const _sfc_main = {name:"doc/zh_CN/get-started/starting_wit
4 |   import { createElementVNode as _createElementVNode, createT
5 |   import _imports_0 from '/image/starting_with_ws63-menu_clos
|           ^
6 |   import _imports_1 from '/image/starting_with_ws63-compilati
7 |   import _imports_2 from '/image/starting_with_ws63-xf_export
```

```
at TransformPluginContext._formatError (file:///D:/projects/document/node_mo
at TransformPluginContext.error (file:///D:/projects/document/node_modules/.
at normalizeUrl (file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4_
at async file:///D:/projects/document/node_modules/.pnpm/vite@5.3.4@types+n
at async Promise.all (index 1)
at async TransformPluginContext.transform (file:///D:/projects/document/node_
at async PluginContainer.transform (file:///D:/projects/document/node_module
at async loadAndTransform (file:///D:/projects/document/node_modules/.pnpm/v
```

---

Click outside, press **Esc** key, or fix the code to dismiss.

You can also disable this overlay by setting `server.hmr.overlay` to `false` in `vite.config.js`.



# 组件开发指南

TIP

组件开发指南

本文介绍 xfusion 的组件（模块）开发规范。

---

**阅读对象：**

- 想要深入了解 xfusion 框架的用户以及组件开发者。
-

# 深入了解

TODO

深入了解

本文介绍 xfusion 的工程、文件夹结构、构建流程、组件开发指南、移植指南等更深入的内容。

---

**阅读对象：**

- 想要深入了解 xfusion 框架的用户以及移植开发者、组件开发者。
- 

## 子文档

- [xfusion 文件夹结构](#)
- [xfusion 构建流程](#)
- [xfusion 运行流程](#)
- [组件开发指南](#)
- [移植指南](#)

# 移植指南

本文说明 xfusion 如何添加新的平台或芯片支持、外设驱动或组件对接支持。

## 阅读对象：

- 想要添加新的平台或芯片支持/外设驱动支持的移植开发者。

## 概述

为 xfusion 移植通常分为两个部分：

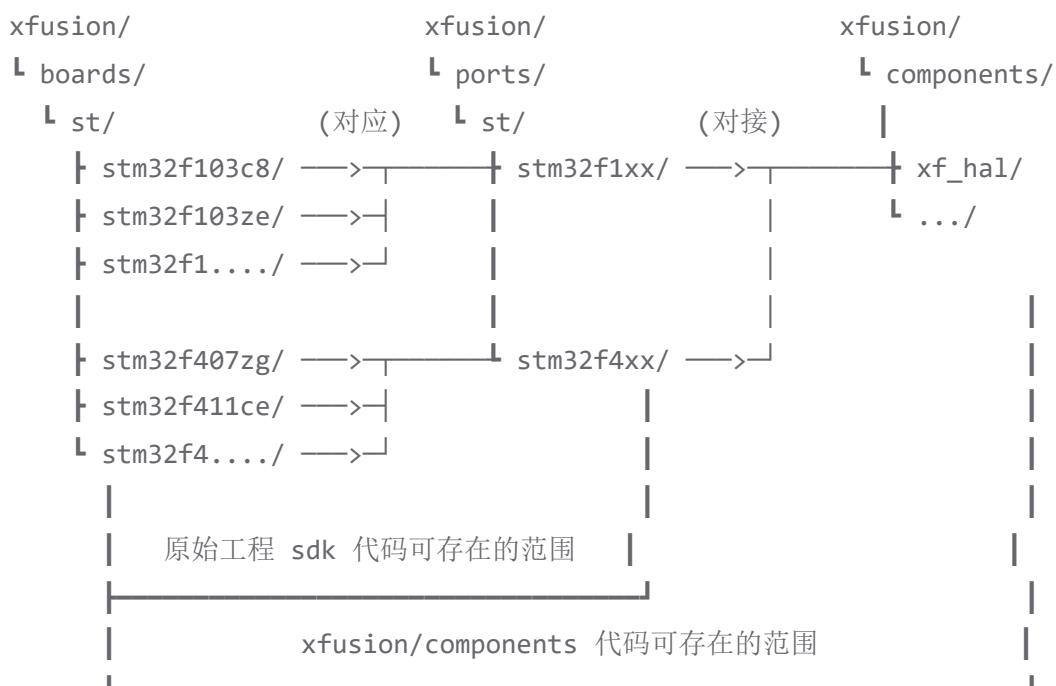
### 1. 移植新的平台或芯片。

比如添加 stm32f103ze、esp32、at32f407vgt7 等新的芯片及其原始工程。

### 2. 移植外设驱动或组件对接。

比如添加 xf\_hal 中驱动的实现。

如图所示：



根据上图，为 xfusion 移植的两个部分可重新解释为：

## 1. 移植新的平台或芯片。

即图中左侧的 `boards` 和 `ports` 部分，一是添加适配 xfusion 环境变量的原始工程，如 `stm32f103c8`；二是创建原始工程 `stm32f103c8/` 和实现对接的代码 `stm32f1xx/`（本质上是原始工程的组件）的关系，即将实现对接的代码加入编译或调用。

## 2. 移植外设驱动或组件对接。

即图中右侧的 `ports` 和 `components` 部分。也就是在 `ports` 内，根据 `components` 内的组件的接口要求实现 `components` 内组件的对接。

## 子文档

- [移植示例 - 添加新平台支持](#)
- [移植示例 - 添加组件支持](#)

# 移植示例 - 添加组件支持

TIP

移植示例 - 添加组件支持

本文说明 xfusion xf\_hal 组件 xf\_xxx 如何移植外设驱动支持。其余组件的对接也有类似的步骤。

---

阅读对象：

- 想要添加新的外设驱动支持的移植开发者。
-

# 移植示例 - 添加新平台支持

TIP

移植示例 - 添加新平台支持

本文将主要以 ws63 芯片为例，介绍 [与移植直接相关的文件或文件夹](#)、[如何添加原始工程](#)、[已移植工程内文件的介绍](#) 和 [原始工程对接步骤](#)。

---

**阅读对象：**

- 想要添加新的平台或芯片支持的移植开发者。

**前置知识：**

- 读者应当配置好了 xfusion 的一种的开发环境（能编译运行），并且知道 xfusion 的编译、运行操作步骤，了解在这个过程中用了什么文件。
  - 读者应当了解 ws63 编程，并且了解 ws63 sdk 工程结构。
-

# xfusion 构建流程

本文简要说明 xfusion 的构建流程。

## 阅读对象：

- 想要深入了解 xfusion 框架的用户以及移植开发者。

## export 阶段

构建之初会使用 `export` 脚本激活 xfusion

windows cmd:

```
.\export.bat <target>
```

windows powershell:

```
.\export.ps1 <target>
```

linux:

```
./export.sh <target>
```

其目的首先是导出 `XF_ROOT`、`XF_TARGET`、`XF_VERSION`、`XF_TARGET_PATH` 四个临时环境变量。关闭当前 `shell` 则环境变量消失。其次，创建 `python` 虚拟环境（如果当前出于 `python` 虚拟环境中，则不创建）。最后安装位于 `tools/xf_build/` 下的 `xf_build` 构建工具

## 前期判断

当我们执行 `xf build` 命令的时候。会自动调用 `tools/xf_build/xf_build/xf_build/cmd/cmd.py` 中的 `build()` 函数。`build()` 函数操作：

1. **检查是否是工程目录。** 此检查是通过当前目录下有无 `xf_project.py` 实现的。后续会创建临时环境变量 `XF_PROJECT_PATH` 保存工程路径。
2. **检查当前目标有无改变。** 这里利用了 `XF_ROOT` 下的 `build/project_info.json` 保存的 `XF_TARGET_PATH` 对比当前环境变量中的 `XF_TARGET_PATH` 是否一致。如果不一致，则调用 `xf clean` 命令进行清除
3. **检查当前工程有无改变。** 这里利用了 `XF_ROOT` 下的 `build/project_info.json` 保存的 `XF_PROJECT_PATH` 对比当前环境变量中的 `XF_PROJECT_PATH` 是否一致。如果不一致，则调用 `xf clean` 命令进行清除
4. **执行 `xf_project.py` 脚本, 完成收集编译信息任务**

## 收集阶段

`xf_project.py` 被执行后，其内容大致如下：

```
python
import xf_build

xf_build.project_init()
xf_build.program()
```

`project_init()` 方法位于 `tools/xf_build/xf_build/xf_build/__init__.py` 文件中。主要完成默认 `project` 对象的创建，以及简化其方法的调用

`program()` 方法位于 `tools/xf_build/xf_build/xf_build/build.py` 文件中。 `program()` 主要的作用是：

1. 将 `XF_ROOT` 下的 `components` 文件夹下的所有文件夹视为一个个组件。
2. 将 `XF_PROJECT_PATH` 下的 `components` 文件夹下的所有文件夹视为一个个组件。
3. 将 `XF_PROJECT_PATH` 下的 `main` 视为一个组件。
4. 执行所有组件的 `xf_collect.py` 文件
5. 最终将所有的构建信息收集到 `XF_PROJECT_PATH` 下的 `build/build_environ.json` 中

其中 `xf_collect.py` 文件大致为：

python

```
import xf_build
```

```
xf_build.collect()
```

`collect()` 方法的 `srcs` 默认为 `["*.c"]`、`inc_dirs` 默认为 `["."]`。如果不设置具体内容，则默认收集该文件夹内的所有 `.c` 文件。可以自定义，直接采取默认参数。

还有个参数是 `requires`。主要涉及到组件之间的依赖关系。如果 A 需要 B 组件里面的函数则在 A 的 `xf_collect.py` 文件中的 `collect()` 改为 `collect(requires=[B])` 组件名为文件夹名。

TODO: 后续将会添加更多的指令收集，如：`cflag` 等编译参数收集。

---

## 插件编译部分

上个阶段的末期会调用 `XF_ROOT` 下的 `plugins` 下的 `XF_TARGET` 插件。这部分需要移植者针对不同的 `target` 进行对应的编译插件开发。

插件开发需要完成以下几个功能：

1. 创建你所需要的 `target` 文件夹
2. 在 `target` 文件夹下创建 `__init__.py` 文件。该文件内容如下：

python

```
from .build import *
```

只有该文件存在，才会被识别为一个 `python` 包

3. 在 `target` 文件夹下创建 `build.py` 文件。该文件内容如下：

python

```
import xf_build

hookimpl = xf_build.get_hookimpl()

class esp32():
    @hookimpl
    def build(self, args):
        """
```

这里对接编译的内容。

通过 `XF_PROJECT_PATH` 下的

`build/build_environ.json` 文件

生成对应的sdk构建脚本

启动sdk的编译命令

....

```
@hookimpl
```

```
def clean(self, args):
```

....

这里对接清除编译命令

....

```
@hookimpl
```

```
def flash(self, args):
```

....

这里对接烧录命令

....

```
@hookimpl
```

```
def export(self, args):
```

....

这里对接导出命令

....

```
@hookimpl
```

```
def update(self, args):
```

....

这里对接导出更新命令

....

```
@hookimpl
```

```
def menuconfig(self, args):
```

....

这里sdk的menuconfig命令。

....

# xfusion 运行流程

TODO

xfusion 运行流程

本文简要说明 xfusion 的运行流程。

---

**阅读对象：**

- 想要深入了解 xfusion 框架的用户以及移植开发者、组件开发者。
-

# xfusion 文件夹结构

TODO

xfusion 文件夹结构

本文简要说明 xfusion 的文件夹结构。

---

**阅读对象：**

- 想要深入了解 xfusion 框架的用户以及移植开发者。
-

# 介绍

本文简要 xfusion 的特性与硬件要求。

---

## 阅读对象：

- 想要添加新的平台或芯片支持/外设驱动支持的贡献者。
- 

# xfusion

**XFusion**, 来自 **X(Embedded Kits System)** —— 嵌入式套件系统，是一个融合多个嵌入式平台的软件开发工具包(SDK)，**为开发者提供统一且便于开发的嵌入式开发环境**。

开发者基于 XFusion 开发应用时，无需花过多时间及精力在移植（像 RTOS），基础驱动、基础功能的实现等与平台底层相关的工作，可以更专注于应用功能的设计与实现，并且，在其上开发的应用，可以在多平台上快速迁移、切换。**(一次开发，多端部署)**

Fusion，意为融合、联合，且有核聚变的意思，表达了 XFusion 的愿景：让分散的平台融合在一起，凝聚出更大的能量，更好地支持开发者实现他们的想法。

TODO 这里放 xfusion 整体框架图

# 特性

## TODO

可考虑增加特征集的描述

<https://nuttx.apache.org/docs/latest/introduction/about.html#feature-set>

- 丰富强大的组件库；
- xf\_build 跨平台构建工具；
- xf\_log log 打印调试工具；
- xf\_task 协作式调度器；
- xf\_hal 基础硬件抽象层；
- xf\_heap 内存管理工具；
- xf\_osal 系统抽象层；

- xf\_utils 基础功能;
- 全部由 C 编写完成, 遵从 C99 语法;
- 丰富详实的例程;
- 采用 Apache2.0 开源协议;
- 支持导出原生工程, 可以使用 keil 等原生 IDE 开发调试;
- 支持模拟器仿真, 可以无硬件依托进行开发 (待开发)。

## 硬件要求

- 16、32 或 64 位微控制器或处理器。
- 建议使用 >16 MHz 时钟速度。
- 闪存/ROM:
  - > 64 kB 用于非常重要的组件 (> 建议使用 180 kB)。
- RAM:
  - 静态 RAM 使用量: ~2 kB, 取决于使用的功能和对象类型。
  - 堆: > 2kB (> 建议使用 8 kB 以上)。
- C99 或更新的编译器。

以下结果为 xfusion 基础组件大小, 排除了对接部分的大小, 对接部分大小见 [基础组件大小详情](#)。

描述	优化等级	flash(bytes)	ram(bytes)
空间优先	Os	4,209	385
	Og	4,742	385
调试	Og	15,043	4,484

### 空间优先

- 关闭 log
- 关闭 xf\_heap 静态数组(静态内存池设为 1 byte)

### 调试

- 开启 log
- xf\_heap 静态数组大小设为 4KB

## 目前支持的平台

1. esp32 (基于 esp-idf v5.0)

2. ws63 (HI3863 芯片)

后续计划支持：

2. linux

3. esp32c3

4. stm32

## 开源证书

xfusion 使用 Apache License 2.0 开源许可协议开源。

xfusion 中用到的其他开源项目声明如下：

TODO

其他开源项目声明

## 存储库布局

TODO

存储库布局

## 版本说明

TODO

补全版本说明

## FAQ

TODO

补全 FAQ