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`) (`pnpm >= 9.2.0`) 环境。

nodejs 安装详见： https://nodejs.org/zh-cn/download/package-manager 包管理器安装详见： https://nodejs.org/zh-cn/download/package-manager/all

```shell
# 安装依赖
npm install

# 热更新预览
npm run dev

# 编译静态文件
npm run build

# 预览编译后的文件
npm run preview
```

使用 yarn 平替 npm 请留意小于 (`yarn 4.5`) 版本不要提交 `yarn.lock` 文件。

```shell
# 安装依赖
yarn

# 热更新预览
yarn dev

# 编译静态文件

```

```
yarn build

# 预览编译后的文件
yarn preview
```

---

tips

文档生成配置所在目录 `.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

具体配置细节请参考 `vitepress` 官方文档

- vitepress

github actions

`deploy.yml` 配置依赖于 `yarn.lock` 文件，(`yarn 1.2x`) 与 (`yarn 4.5`) 版本 lock 文件差异较大，请避免使用 yarn 导致提交不同的 lock。 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。

样式表配置

样式表在路径：.vitepress/theme/assets/index.css 如需要改变样式，自行修改

修改左侧目录结构

通过修改路径 .vitepress/sidebar.ts

最后更新于: 2024/11/29 16:13:19

Next page

欢迎来到 XFusion 文档！

快速入门

快速上手 XFusion

API参考

接口的查询手册

深入了解

了解 XFusion 的运作原理

移植指南

移植到一个全新的芯片

贡献指南

给 XFusion 提供贡献

FAQ

常见的小问题

最后更新于: 2024/12/2 22:53:03

FAQ

作者

kirto

Q1: 哪里可以提问?

A1: 您可以通过以下方式联系我们:

- QQ 群聊：加入 QQ 群
- 邮件支持：kirto@pthyidh.com
- github 支持：xfusion issue
- gitee 支持：xfusion issue

Q2: XFusion 的每个部分都能拆开来单独使用吗?

A2: 没错，XFusion 的每个模块都能单独移植并使用。如果您担心整体移植的体量会过大，完全可以采用分模块移植。 包括我们的组件库，也是采用中间件的写法。可以单独移植到您的 SDK 环境中。 详细的可以参考我们的 B 站视频:

最后更新于: 2024/11/29 16:13:19

Previous page
组件贡献指南

API 参考

作者

ccb5

本文主要介绍 XFusion 的应用层接口和移植层的接口

目录

- 开发 API 参考

最后更新于: 2025/2/8 19:14:26

代码注释指南

作者

ccb5

本文说明 XFusion c/cpp 代码的注释规范。

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 */` 样式的的注释，以表明当前注释说明的对象是左侧的代码。

如：

```c
/* ********** 正确示例 ********** */

/* 总线配置 */
```

```c
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

/* ********** 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

```c
/**
 * @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` 。

```c
/**
 * @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[out] 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。

```

```c
 */
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：

   ```c
   /**
    * ... 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。

      ```c
      /* ********* ↓ 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。

```c
/**
 * @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。

```c
/**
 * @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。

```c
/**
 * @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(
```

```c
    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` 。

```c
/**
 * @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_
    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
    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。

```c
/**
 * @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
```

```c
 *      - 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。

```c
/**
 * @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

最后更新于: 2024/12/2 20:04:27

Previous page
贡献指南

Next page
编码风格指南

移植示例 - 添加组件支持

作者

kirto

本文说明如何给 xf 包管理仓库添加新的包

仓库说明

仓库地址：https://github.com/x-eks-fusion/xf_components

该仓库用于保存组件贡献者贡献的仓库。由于我们并不希望组件仓库强依赖 XF 本身的框架，所以，组件分为解耦部分和 xfusion 移植部分。当您使用本仓库的组件且正在使用 xfusion 的时候。可以通过

```shell
xf search <组件名>
```

进行模糊搜索功能。 通过：

```shell
xf install <组件名>
```

则可以将组件下载并解压到当前工程的 components 文件夹中，提供给您使用 通过：

```shell
xf uninstall <组件名>
```

则可以删除卸载对应的组件

贡献说明

对移植者的要求比较高。组件的提供必须要完成中间层、移植层和示例等。

中间层

中间层要求：

- 禁止依赖底层的操作部分需要自行抽象成对阶层，对接部分可以是宏定义，可以是函数声明，可以是函数指针，可以是弱定义等方式。
- 尽量直接依赖标准库，依赖标准库需要提前宏定义，方便替换
- 可以依赖第三方符合我们标准的解耦中间层，需要在 config.json 中说明
- 解耦中间层的配置可以依赖移植层的 xxx_config.h 文件
- 这部分可以是单独仓库（子模块），也可以是和移植部分在一起

移植层

移植层要求：

- 依赖 xfusion 层的基础操作，将解耦中间层的对接部分简化
- 对 xfusion 层需要的抽象操作需要写明依赖
- 尽量不要依赖特定的底层功能，如有需要，例如：硬件加速图像计算等需求。应当提供对应的宏予以选择
- 移植层需要对接解耦中间层的配置，修改成 XFConfig 的操作，方便用户使用 menuconfig 进行配置

示例

移植者需要提供至少一个示例，提供在移植了 xfusion 后，如何调用的完整流程。说明文件要大致说明示例目标，menuconfig 的配置，编译过程，运行结果。

README.md

说明文件，需要简单介绍仓库的功能

---

config.json

```json
{
  "name": "cJSON", // 组件名称
  "author": "DaveGamble", // 组件作者
  "version": "1.7.18", // 组件版本
  "license": "Apache-2.0", // 组件开源协议
  "keywords": [], // 关键词
  "description": "A collection of useful cmake utilities", // 组件简要描述
  "url": "https://github.com/DaveGamble/cJSON", // 组件链接
  "dependencies": {}, // 依赖组件以及版本
  "links": {
    "Homepage": "https://www.json.org/json-en.html", // 主页链接
    "Issues": "https://github.com/DaveGamble/cJSON/issues" // issue链接
  }
}
```

其它

移植者如果是移植别人的中间层，则需要标明别人中间层的仓库来源链接，仓库的作者以及开源协议。 如果需要配置的文件，可以使用 XFConfig 进行配置。

贡献过程

1. fork 文档仓库
2. 添加自己的仓库文件夹
3. 添加 config.json
4. 添加XFConfig
5. 提交 PR 合并到主仓库

最后更新于: 2024/12/2 16:13:34

Previous page
Pull Request 提交步骤

Next page
FAQ

编码风格指南

作者

ccb5

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。

前置准备：

- 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。

```c
/**
 * @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。 */
#ifdef __cplusplus
extern "C" {
#endif

/* ... */

#ifdef __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_templ_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))`；

```c
/* 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; };`。

```c
/* 源文件包含的内容提示符 */

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

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

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

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

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

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

```

```c
/* ==================== [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`。

```c
/**
 * @brief 主函数。
 */
int main(int argc, char *argv[])
{
    xf_templ_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();

```

```c
    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
├ xf_uart.c
├ xf_uart.h
├ xf_uart_port.h
└ xf_uart_types.h
```

- 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:

```c
#define XF_TEMPLATE_HELP_STR            "xf_template v0.1"
#define XF_TEMPLATE_HELP_STR_SPLICING   "test" XF_TEMPLATE_HELP_STR "abc123"
#define XF_TEMPLATE_DEFINE              (1)
```

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:

```c
/**
 * @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）。
 */
```

```c
#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 */
#ifdef xf_printf
#   define XF_TEMPL_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：

1. 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。

2. 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；
3. 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；
4. 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；
5. 能说明宏的作用。

格式要点：

1. 宏对应的内容通常建议加括号，如下；

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

2. **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；

```c
/* 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
#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);                            \
    })
```

3. 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：

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

```c
#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：

```c
/**
 * @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_templ_struct_t`。
 *    当然用 xf_templ_struct_s 也可以。
 */
typedef struct _xf_templ_struct_t {
    xf_templ_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
    union {                         /*!< 结构体内的联合体或结构体等可以匿名 */
        uint8_t all;               /*!< 通过这个值可以修改整个共用体 */

```

```c
        struct {                    /*!< 结构体内的联合体或结构体等可以匿名 */
            uint8_t val_u4: 4;      /*!< 这是位域的示例，u4 表示有 4 位，根据
            uint8_t val_bit4: 1;    /*!< 这是位域的示例，bit4 表示的是从 bit0
            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_templ_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_templ_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：

```c
/**
 * @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_templ_enum_t"。
 *    用 xf_templ_enum_e 也可以。
 * 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_templ_enum_t {
    XF_TEMPL_ENUM_0 = 0x00,         /*!< 枚举值 0，第一个枚举值通常要求手动赋
    XF_TEMPL_ENUM_1,                /*!< 枚举值 1 */
    XF_TEMPL_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_TEMPL_ENUM_MAX,                  /*!< 枚举值最大值 */

    XF_TEMPL_ENUM_DEFAULT = XF_TEMPL_ENUM_1,     /*!< 枚举值默认值 */
} xf_templ_enum_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。

```c
/* 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
 * 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：

```c
/* ==================== [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。

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

```c
                                /* 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）。

```c
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

中文	英文	缩写
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
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

中文	英文	缩写
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
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

中文	英文
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/之前	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

最后更新于: 2024/12/2 20:04:27

Previous page
代码注释指南

Next page
文档编写指南

文档指南

作者

ccb5, dotc

本文描述编写 XFusion 文档需要遵守的内容、格式等规范。

前置准备：

1. 对 markdown 语法有充分的了解，如需参考 markdown 语法，请参考 Markdown 入门基础 | Markdown 官方教程，Markdown-教程 | 菜鸟教程。
2. 对 vitepress 部分补充语法，详参: VitePress 内置的 Markdown 扩展 。

文档仓库的获取及前置准备

---

1. 文档仓库的获取

> **WARNING**
>
> 如需对文档系统部分进行贡献 (需要提 PR)，请勿直接克隆主仓库进行修改，请严格按以下步骤执行。

- 文档主仓库地址: https://github.com/x-eks-fusion/xf_docs

1. fork 主仓库到自己的账户下。
2. 在本地克隆自己 fork 的仓库。
3. 根据将修改的内容，创建合适的分支名，最后基于这个分支进行修改。

- 详参:Pull Request 提交步骤

2. 环境的搭建

- 如需本地预览效果,请确保已经安装 (`nodejs >= 18`) (`pnpm >= 9.2.0`) 环境。
  - nodejs 安装详见： https://nodejs.org/zh-cn/download/package-manager
  - 包管理器安装详见： https://nodejs.org/zh-cn/download/package-manager/all

---

3. 根据需要执行以下指令

```shell
# 安装依赖
npm install (必需)

# 热更新预览（看实际需要，通常使用此命令进行边编写边查看实际渲染效果的操作）
npm run dev

# 编译静态文件（看实际需要）
npm run build

# 预览编译后的文件（看实际需要）
npm run preview
```

关于文档修改的一些操作

- 文档修改一般有如下操作:
  - 需要在侧边栏新增大纲:
    - 在文件 ".vitepress/sidebar.ts" 下的 "function sidebarTOC()" 中增加对应的大纲信息以及大纲条目描述。
    - 例 :

    ▶ **增加大纲 "深入了解"，其中有两条目: "XFusion 目录结构", "XFusion 构建流程"**

  - 需要在现有大纲中新增条目:

- 在文件 ".vitepress/sidebar.ts" 下的 "function sidebarTOC()" 中找到对应大纲项的大纲描述，然后在大纲描述中增加对应的条目信息，以及对应文档的路径。
- 例：

  ▶ **在大纲 "深入了解" 中增加 1 个条目: "xf build 构建脚本"**

文档内容

- 目前一般的文档主要有如下内容：
  1. 开头标明作者。如下：

     ```markdown
     > [!NOTE] 作者
     > <作者名称>
     ```

  2. 概述。请使用正文格式进行描述。
  3. 内容。
  4. 参考文献。

文档格式规范

XFusion 文档目前只提供中文，为了管理不同文档中的图片，**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。

   ```markdown
   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。

   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。

```markdown
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

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
```

2. **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 `<!-- -->` 。

3. 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。

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
4. VitePress 内置的 Markdown 扩展

最后更新于: 2024/12/2 20:04:27

Previous page
编码风格指南

Next page
Pull Request 提交步骤

贡献指南

作者

ccb5

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。

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
- 移植示例 - 添加组件支持
- 编码风格指南
- 文档编写指南
- PR 提交步骤

最后更新于: 2025/2/8 19:14:26

Previous page
其他对接

Next page
代码注释指南

Pull Request 提交步骤

作者

ccb5

本文说明 XFusion 的 Pull Request 提交步骤。

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。**

   1. 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。

   2. 然后打开您的本地终端，克隆您 fork 的仓库。

```bash
# 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。

```bash
# 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。

```bash
# 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
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：

```txt
🐞 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
```

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)]

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)

最后更新于: 2024/12/2 20:04:27

Previous page
文档编写指南

Next page
组件贡献指南

添加自己的代码

作者

kirto

本文主要介绍如何在一个 XFusion 工程上添加自己的源代码和头文件。

前提

当我们准备了一份工程并激活好 XFusion 后

简单的添加独立文件

---

main 文件夹中添加文件

当我们需要添加一些应用层代码 .c .h 文件的时候。 我们可以考虑在工程内的 main 文件夹添加。 由于 main 文件夹中工程收集脚本 `xf_collect.py` 中的 xf_build.collect() 方法自动收集 main 文件下的源文件。 所以，如果在 main 文件夹下面添加文件是不用修改 `xf_collect.py` 的。

```bash
hello
├ main
│ ├ my_code.c
│ ├ my_code.h
│ ├ xf_collect.py
│ └ xf_main.c
└ xf_project.py
```

main 文件夹中添加子文件夹

当我们的文件较多的时候，会考虑用文件夹管理文件。 此时将文件夹加入 main 文件夹中。

```bash
hello
├ main
│ ├ my_code
│ │ ├ my_code.c
│ │ └ my_code.h
│ ├ xf_collect.py
│ └ xf_main.c
└ xf_project.py
```

由于新增的文件是在子文件夹中，所以我们需要修改 `xf_collect.py` 进行手动添加

```python

import xf_build

srcs=["*.c", "my_code/*.c"]
inc_dirs = [".", "my_code"]
xf_build.collect(srcs, inc_dirs)
```

除了上述方法外，你还可以在你的子文件夹中复制一份 xf_collect.py

```bash
hello
├ main
│ ├ my_code
│ │ ├ my_code.c
│ │ ├ my_code.h
│ │ └ xf_collect.py
│ ├ xf_collect.py
│ └ xf_main.c
└ xf_project.py
```

xf 指令会扫描 main 下面的 xf_collect.py 运行，但是不会添加子文件夹的 xf_collect.py 。所以，子文件夹的 xf_collect.py下 需要添加一个 `import my_code.xf_collect`

```python

import xf_build
import my_code.xf_collect

xf_build.collect()
```

添加一个自己的组件

当我们要添加一个独立的组件时候，不是很希望和 main 文件夹挤在一起。 于是，我们可以通过创建 `components` 文件夹。 xf 指令会扫描 main 下面的 xf_collect.py 。components的子文件夹的 xf_collect.py 也会被收集。 所以，我们只需要加一个 xf_collect.py 就可以。

```bash
hello
├ components
│ └ my_code
│ │ ├ my_code.c
│ │ ├ my_code.h
│ │ └ xf_collect.py
├ main
│ ├ xf_collect.py
│ └ xf_main.c
└ xf_project.py
```

用户文件夹的 xf_collect.py 内容和 main 的 xf_collect.py 保持一致就行。 但是一定要是 `components` 的子文件夹。

```python

import xf_build

xf_build.collect()
```

添加一个独立的文件夹

如果你的文件夹不希望放到 components 里面。那么也没关系，你可以放到工程中你喜欢的文件夹中。

```bash
hello
├ main
│ ├ xf_collect.py
│ └ xf_main.c
├ my_code
│ ├ my_code.c
│ ├ my_code.h
│ └ xf_collect.py
└ xf_project.py
```

但是需要在 `xf_project.py` 中写清楚文件夹的路径。

```python
import xf_build

user_dirs = ["my_code"]

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

最后更新于: 2024/11/30 02:39:02

Previous page
编译第一个工程

Next page
安装一个组件

编译第一个工程

作者

kirto

文本档主要帮助用户创建第一个工程，以及如何指导用户进行编程，编译，烧录

创建第一个 XFusion 工程

1. 激活自己的 SDK 环境

激活环境

有些 SDK ，例如：esp32 每次打开终端需要使用 export 激活一次环境

2. 激活 XFusion

⚠️ **注意**

get_xf 的命令是执行 export.sh 的别名。具体参考：

linux 环境搭建

windows 环境搭建

```bash
get_xf <target>
```

3. 使用命令创建一个新的 XFusion 工程

```bash
xf create hello
```

此时创建出来的文件夹结构如下：

```bash
hello
├ main
│ ├ xf_collect.py
│ └ xf_main.c
└ xf_project.py
```

- **xf_project.py**: 工程构建脚本，xf 命令通过识别这个文件来确认是不是 XFusion 。文件内部调用 xf_build 初始化工程。
- **xf_collect.py**：工程收集脚本，xf 通过该脚本收集所有编译的信息。文件内部调用 xf_build 的 collec() 方法收集编译信息。默认收集同级文件夹的所有 .c 。以及将当前所在文件夹加入include_path 。
- **xf_main.c**：代码入口文件。主要识别 xf_main() 函数作为 xf_build 的代码入口。

修改第一个 XFusion 工程

1. 添加自己的打印

```c
#include "xf_log.h"

void xf_main(void)
{
    xf_log_printf("hello XFusion\n");
}
```

编译第一个 XFusion 工程

1. 烧录代码

```bash
xf flash
```

2. 查看串口打印

```bash
xf monitor <port>
```

按下ctrl+]退出串口

导出第一个 XFusion 工程

> ⚠️ **注意**
>
> 有些芯片不是命令行编译，需要通过 IDE。比如：keil。 这时上述编译步骤就没用了

1. 导出 IDE 工程

```bash
xf export <工程名>
```

   通过上述命令，可以导出生成一个完整的 IDE 工程。然后打开工程，即可在 IDE 的环境中开发

2. 更新 IDE 工程

```bash
xf update <工程名>
```

   当已有导出工程后，希望能更新工程的时候，可以使用上述指令进行工程更新。

最后更新于: 2024/11/29 16:13:19

Previous page
windows 环境搭建

Next page
添加自己的代码

快速入门

作者

kirto

本文档旨在指导用户搭建 XFusion 开发的软件环境，通过一个简单的示例展示如何使用 XFusion 配置菜单，并编译、下载固件至开发板等步骤。

> ⚠️ **注意**
>
> 这是 XFusion master 分支（最新版本）的文档，该版本在持续开发中。其他版本的文档 XFusion 版本简介供参考。

目录

- 介绍
- 准备工作
  - linux 环境搭建
  - windows 环境搭建
- 选择一个平台开始
  - 从 esp32 开始
  - 从 ws63 开始
- 编译第一个自己的工程
- 添加自己的代码
- 安装一个组件
- 使用 vscode 插件
- xf 命令参考

最后更新于: 2025/2/8 19:14:26

Next page
介绍

安装一个组件

作者

kirto

本文内容主要帮助大家使用 xf 指令安装和下载一个组件。

前提

当我们准备了一份工程并激活好 XFusion 后

快速在工程中添加一个组件

在我们的工程中通过以下指令可以快速添加一个 cjson 组件

```bash
xf install cjson
```

这时 cjson 就会被我们安装到组件中

```bash
hello
├ components
│ └ cjson
│ │ ├ cJSON
│ │ ├ example
│ │ ├ README.md
│ │ ├ config.json
│ │ └ xf_collect.py
├ main
│ ├ xf_collect.py


```

```
│   └ xf_main.c
└ xf_project.py
```

在 xf_main.c 中写一个简单的示例

```c
#include "cJSON.h"
#include "cJSON_Utils.h"
#include "xf_log.h"

void xf_main(void)
{
    // 定义一个复杂的 JSON 字符串
    const char *json_string = "{\"name\":\"John\", \"age\":30, \"address\":{\"city\"

    // 解析 JSON 字符串
    cJSON *root = cJSON_Parse(json_string);
    if (root == NULL) {
        xf_log_printf("Failed to parse JSON\n");
        return;
    }

    // 定义指向目标数据的 JSON 指针路径
    const char *pointer_path = "/contacts/1/value";

    // 使用 cJSONUtils_GetPointer 获取路径中的 JSON 项
    cJSON *target_item = cJSONUtils_GetPointer(root, pointer_path);
    if (target_item == NULL) {
        xf_log_printf("Failed to find target item\n");
        cJSON_Delete(root);
        return;
    }

    // 打印目标项的值
    char *target_value = target_item->valuestring;
    if (target_value != NULL) {
        xf_log_printf("Target value: %s\n", target_value);
    } else {
        xf_log_printf("Failed to print target value\n");
    }


```

```
    // 清理
    cJSON_Delete(root);

}
```

如何删除组件

本质上，xf 就是通过帮你从服务器拉取组件，然后解压放到 components 下面。 删除组件可以通过直接删除指定文件夹。 也可以通过以下指令来快速移除：

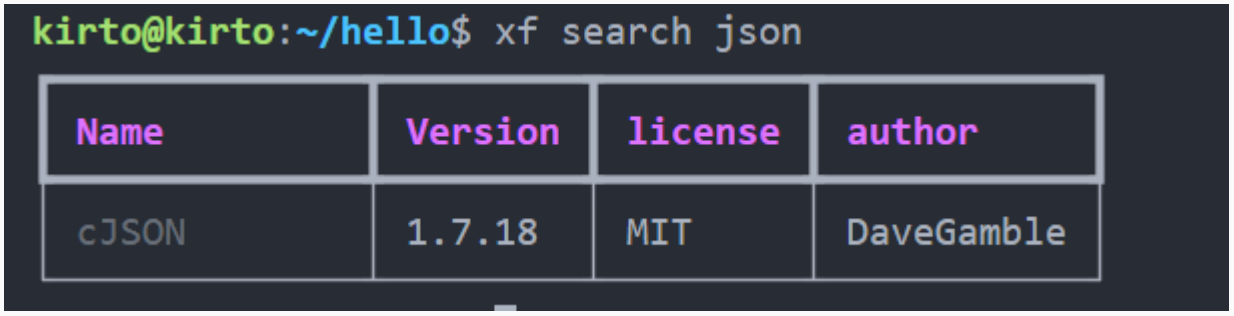
```bash
xf uninstall cjson
```

如何查询组件

如果我们只知道我们需要一个 json 库，但是不知道名字。 可以通过以下指令来搜索我们的库

```bash
xf search json
```

xf 会通过模糊查询，查询到可能匹配的包名



在网页上查看组件

有时候我们想要了解一下有哪些包可以用。 此时可以考虑看看我们的[网页版组件库]

最后更新于: 2024/11/29 16:13:19

Previous page
添加自己的代码

Next page
使用 vscode 插件

介绍

作者

ccb5

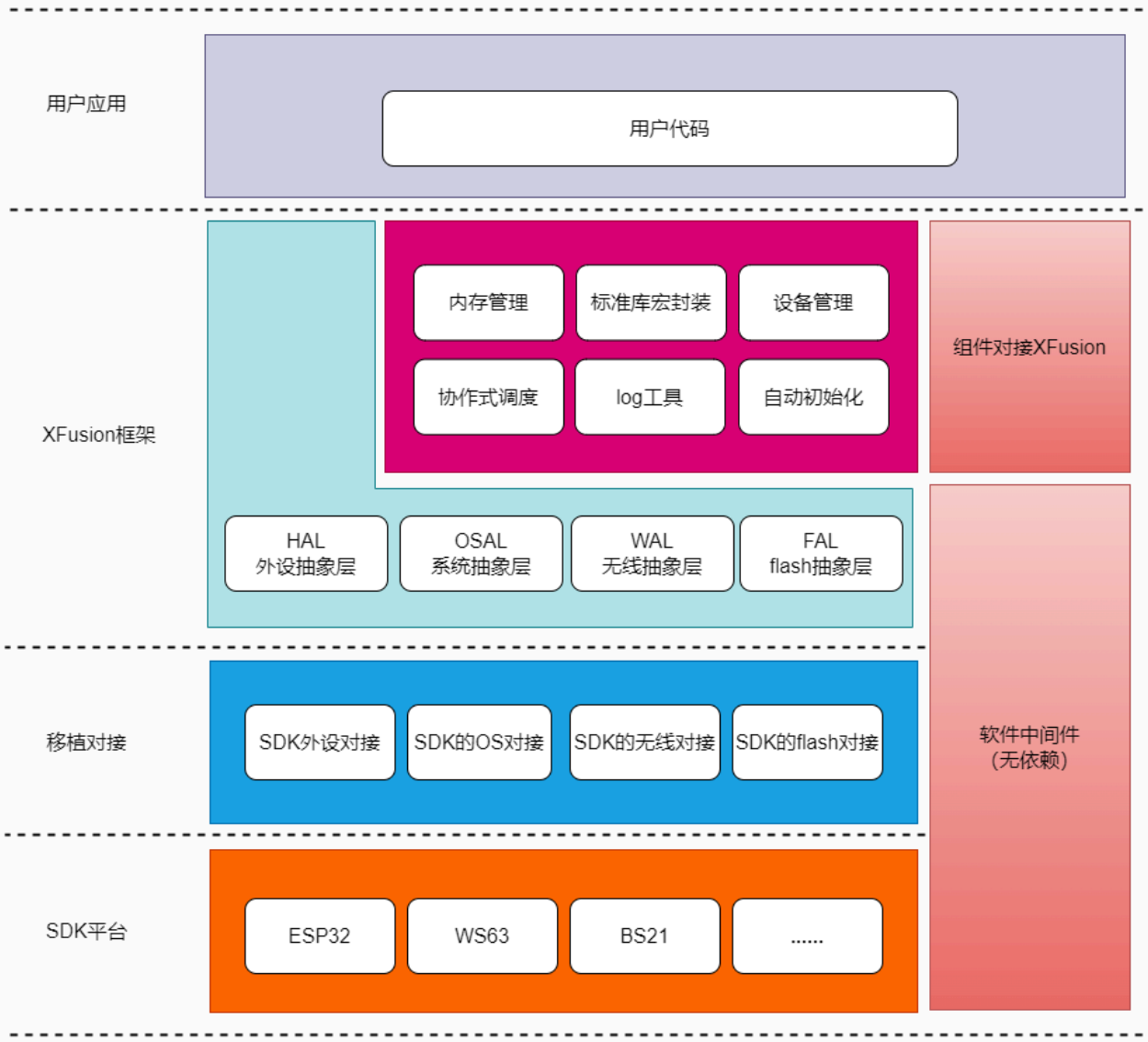
本文主要介绍 XFusion 相关内容

什么是 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。



特性

- 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；
- 所有子模块可作为软件中间件独立移植
- 全部由 C 编写完成，遵从 C99 语法；

- 丰富详实的例程；
- 丰富强大的组件库；
- 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；

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)
3. bs21 (HI2821 芯片)

后续计划支持:

4. linux
5. esp32c3
6. stm32

开源证书

> **Apache License 2.0**
>
> XFusion 及其子模块全部使用 Apache License 2.0 协议

存储库布局

> **重要**
>
> 所有仓库均能独立移植

- github:
  - XFusion: XFusion 主仓库
  - xf_utils: 工具库，有标准库的宏封装，基础的log等级封装，lock锁，链表等功能
  - xf_log：log库，可以对接多个底层。还有过滤器，可以根据你想要的内容过滤log。
  - xf_init：初始化库，提供不同自动初始化实现
  - xf_hal：外设管理库，提供硬件抽象层，管理硬件设备。
  - xf_heap：内存管理库，提供基础的内存管理。也可以对接底层的内存管理
  - xf_task：协作式调度器库，提供协作式多任务功能
  - xf_osal：操作系统抽象库，提供统一的操作系统API
- gitee:

- XFusion: XFusion 主仓库
- xf_utils 工具库，有标准库的宏封装，基础的log等级封装，lock锁，链表等功能
- xf_log：log库，可以对接多个底层。还有过滤器，可以根据你想要的内容过滤log。
- xf_init：初始化库，提供不同自动初始化实现
- xf_hal：外设管理库，提供硬件抽象层，管理硬件设备。
- xf_heap：内存管理库，提供基础的内存管理。也可以对接底层的内存管理
- xf_task：协作式调度器库，提供协作式多任务功能
- xf_osal：操作系统抽象库，提供统一的操作系统API

最后更新于: 2024/12/2 20:04:27

Previous page
快速入门

Next page
从 esp32 开始

Linux 环境搭建

作者

kirto

本文主要介绍在 Linux 环境中如何搭建 XFusion 开发环境

环境搭建

具体搭建linux还是windows环境，需要看SDK可以在windows还是linux上开发

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.

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

如果环境中有 python3， 但是没有 python , 可以创建 python 的软连接

```bash
sudo ln -s /usr/bin/python3 /user/bin/python
```

还需要安装 git 方便拉取源码

```bash
sudo apt-get git
```

安装 XFusion

1. 克隆仓库

```bash
git clone --recurse-submodules https://github.com/x-eks-fusion/xfusion.git
```

2. 激活 XFusion

```bash
cd xfusion              # 进入 XFusion 文件夹
. ./export.sh <target> # 激活指定的芯片
```

> ⚠️ **注意**
>
> 每次打开一个新的终端，如果想要用 XFusion 都需要激活一次

由于这个操作比较常用，所以我们可以通过在 .bashrc 中通过 alias 命令，设置一个别名。

```bash
vim ~/.bashrc # 根据不同 shell ，zsh 就是 .zshrc
```

进入 vim 界面后，输入 `Shift+g`，跳转到最后一行。 按下 `o` 插入自己的命令。

```bash
alias get_xf=". ~/xfusion/export.sh"   # 双引号后面是 XFusion 路径
```

然后通过 `Esc` 退出编辑。 最后通过 `:+w+q` 再加上回车确认保存。

至此，重启终端后。每次激活只需要输入：

```bash
get_xf
```

最后更新于: 2024/12/2 20:04:27

Next page
快速入门

Windows 环境搭建

作者

kirto

本文主要介绍在 Windows 环境中如何搭建 XFusion 开发环境

环境搭建

具体搭建linux还是windows环境，需要看SDK可以在windows还是linux上开发

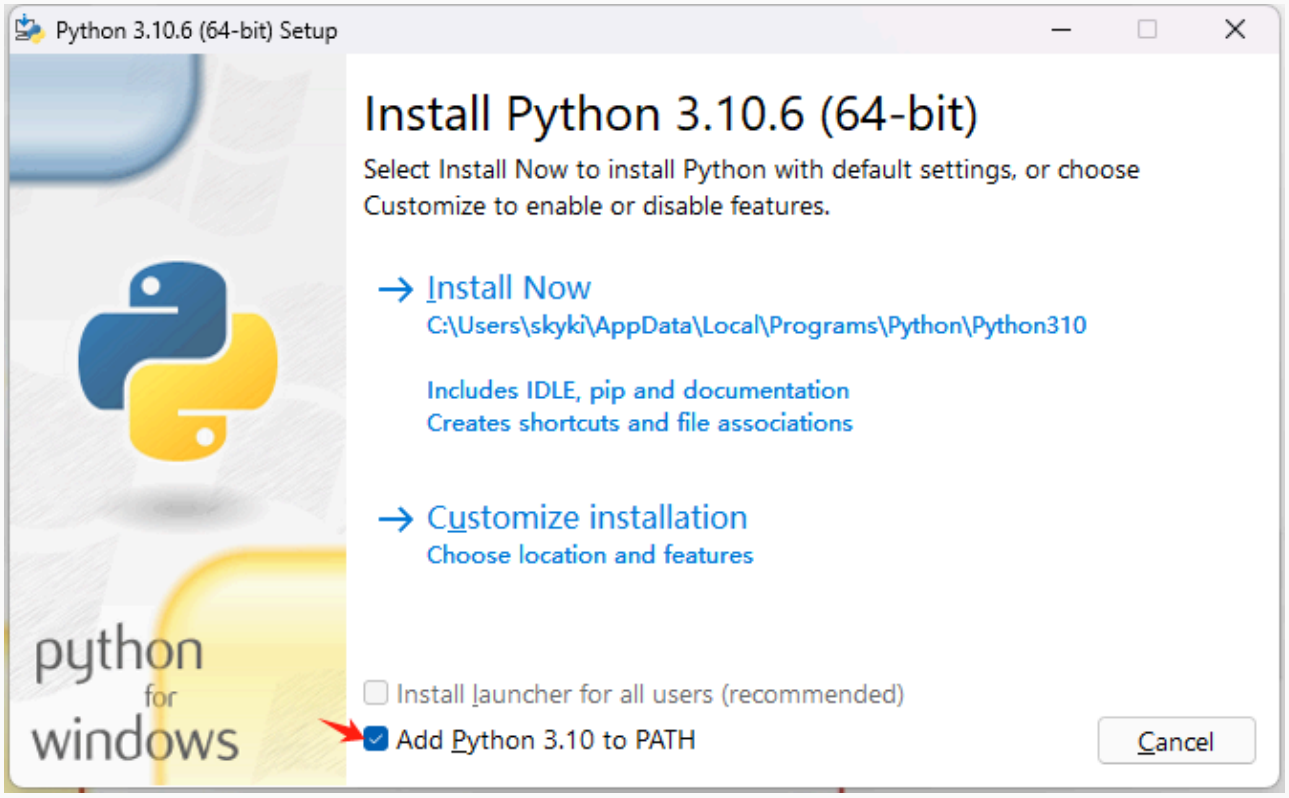
前置准备

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

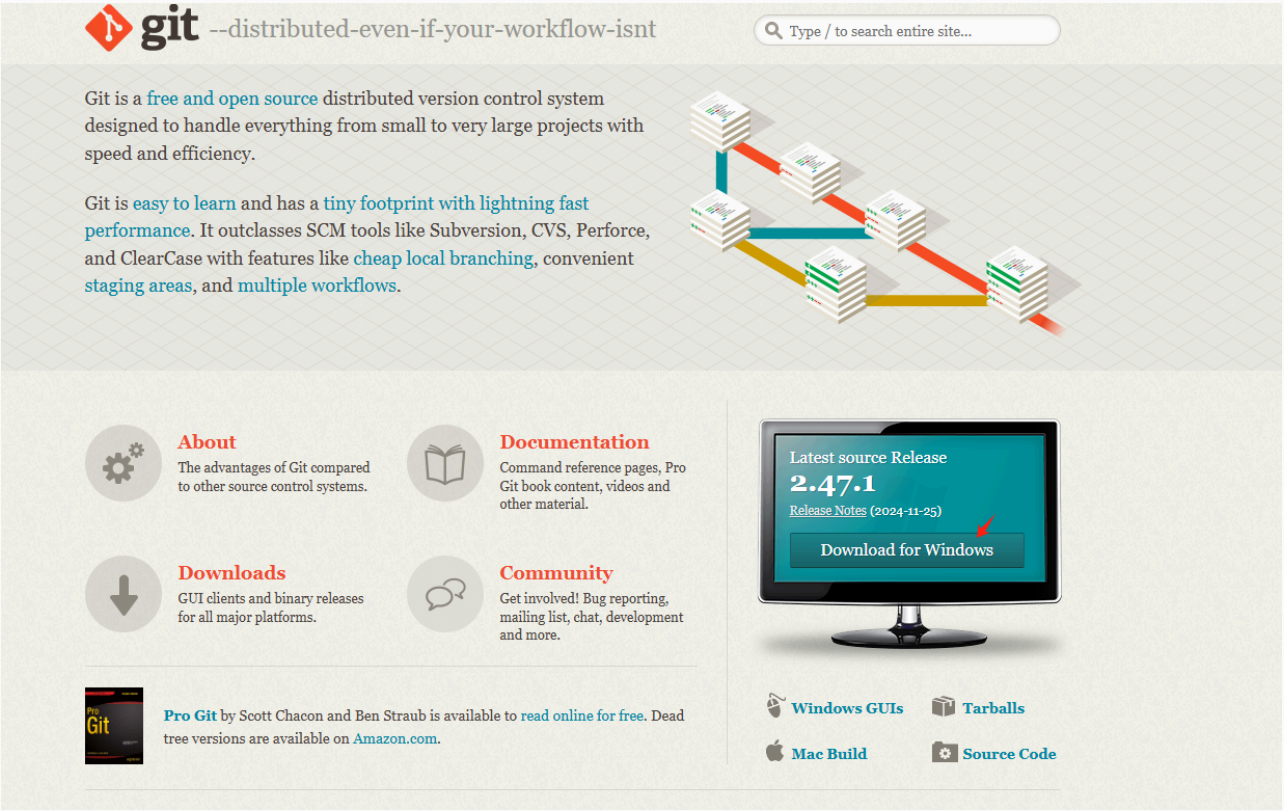
Files

Version	Operating System
Gzipped source tarball	Source release
XZ compressed source tarball	Source release
macOS 64-bit universal2 installer	macOS
Windows installer (64-bit)	Windows
Windows installer (32-bit)	Windows
Windows help file	Windows
Windows embeddable package (64-bit)	Windows
Windows embeddable package (32-bit)	Windows

> ⚠️ **注意**
>
> 安装时勾选 Add Python 3.10 to PATH 方便命令行调用



还需要安装 git 方便拉取源码



安装 XFusion

1. 克隆仓库

```bash
git clone --recurse-submodules https://github.com/x-eks-fusion/xfusion.git
```

2. 激活 XFusion 如果是 powershell：

```powershell
cd xfusion              # 进入 XFusion 文件夹
./export.ps1 <target> # 激活指定的芯片
```

如果是 cmd：

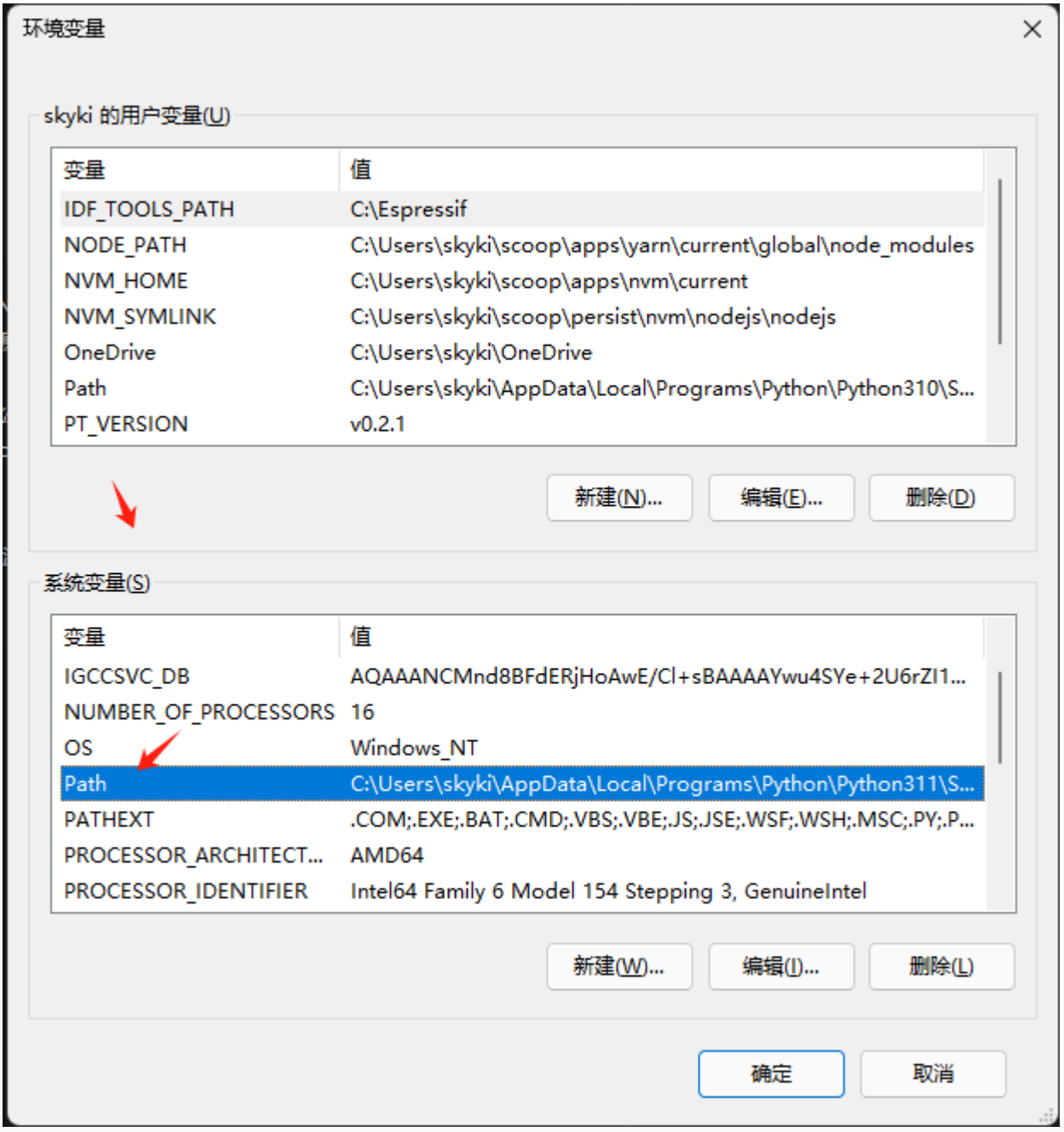
```cmd
cd xfusion              # 进入 XFusion 文件夹
.\export.bat <target> # 激活指定的芯片
```

> ⚠️ **注意**
>
> 每次打开一个新的终端，如果想要用 XFusion 都需要激活一次

由于这个操作比较常用，所以我们可以通过在添加系统环境变量让这个脚本可以在任意路径下使用。

1. 复制当前 XFusion 的绝对路径
2. `win+s` 打开微软搜索
3. 搜索 `编辑系统环境变量`
4. 选择环境变量

5. 选择 `系统变量` 中的 `Path`，点击编辑



6. 点击 `新建`，然后粘贴我们之前复制的 XFusion 的绝对路径
7. 点击 `确定`
8. 打开一个新的 cmd 或者 powershell，现在我们可以在任意路径下使用 export.bat（powershell是export.ps1）了。

最后更新于: 2024/11/29 16:13:19

Next page
快速入门

从 bs21 开始

作者

kirto

本文主要介绍如何搭建 BS21 的 XFusion 开发环境

关于 BS21

bs21 是一款高度集成 2.4GHz SoC BLE&SLE 芯片，集成 BLE5.4/SLE1.0 和 RF 电路，RF 包含功率放大器 PA、低噪声放大器、TX/RX Switch、集成电源管理等 模块，支持 1M/2M/4M 3 种带宽，最大支持 12Mbit/s 速率。

bs21 集成高性能 32bit 微处理器（MCU），硬件安全引擎以及丰富的外设接 口，外设接口包括 SPI、UART、I2C、PWM、GPIO、USB2.0、NFC Tag、PDM、 I2S/PCM、QDEC、KEYSCAN 键盘扫描电路，支持 8 路 13bit 分辨率 ADC、ADC 支 持对接音频 AMIC，内置 SRAM 和合封 Flash，并支持在 Flash 上运行程序。

bs21 支持 LiteOS，并配套提供开放、易用的开发和调试运行环境。

bs21 适应于 PC 配件，IOT 等物联网智能终端领域。

bs21 的主要特性

- 支持 BLE 5.4。
- 支持 LE1M、LE2M、Long Range。
- 支持内置 PA，集成 TX/RX switch。
- 灵敏度
  - LE1M：-97dBm。
  - LE2M：-94dBm。
  - LR125K：-103dBm。

- 发射功率支持 8dBm。
- 支持最多连接 8 条链路，8 条链路 BLE 和 SLE 共享。
- 支持 BLE 白名单、可解析。
- 支持 HID 人机接口设备。
- 支持 BLE 业务间隙扫频功能。
- 支持 BLE AFH 跳频。

激活 XFusion

通过之前在 .bashrc 中保存的激活命令的别名来激活

```bash
get_xf bs21
```

使用 `xf target -s` 命令，可以帮助我们确认导出的 target 是否是 bs21

安装 BS21 SDK

获取源码

激活后，我们仅仅需要一个命令。XFusion 便会自动的根据 `boards/nearlink/bs21` 下的 `target.json` 文件，来下载对应版本的sdk。

```bash
xf target -d
```

安装 BS21 SDK 环境依赖

bs21 的 SDK 是使用 python + cmake 编译，其中依赖一些软件，需要安装

配置 Shell

配置默认使用 bash。打开Linux终端，执行命令“sudo dpkg-reconfigure dash”， 选择 no。

```bash
sudo dpkg-reconfigure dash
```

安装 Cmake

打开 Linux 终端，执行命令“sudo aptinstall cmake”，完成Cmake 的安装。

```bash
sudo apt install cmake
```

安装pycparser

```bash
pip3 install pycparser==2.21
```

最后更新于: 2024/11/29 16:13:19

Previous page
从 ws63 开始

Next page
linux 环境搭建

从 esp32 开始

作者

kirto

本文主要介绍如何搭建 ESP32 的 XFusion 开发环境

关于 ESP32

ESP32 是由乐鑫科技（Espressif Systems）推出的一款高性能 IoT SoC（系统级芯片），支持 Wi-Fi 和 BLE 双模通信，广泛应用于智能家居、工业自动化和可穿戴设备等物联网场景。ESP32 系列芯片内置强大的计算能力和多样化的外设接口，能够满足复杂 IoT 应用的需求。

主要功能与特性 无线通信：集成 IEEE 802.11 b/g/n Wi-Fi 基带和 RF 电路，支持 2.4GHz 频段，提供高达 150Mbps 的物理层速率，支持长距离稳定通信。 BLE 功能：支持 BLE 4.2 和 BLE 5.0 协议，提供高达 2Mbps 的空口速率，兼容 Mesh 网络与 BLE 网关功能，满足多种无线场景。 强大的计算能力：采用双核 Tensilica Xtensa LX6 处理器，主频最高达 240MHz，可流畅运行多任务程序。 丰富的外设接口：集成 UART、SPI、I2C、I2S、PWM、ADC 和 DAC 等多种外设，满足多样化硬件设计需求。 低功耗设计：支持多种低功耗工作模式，包括深度睡眠模式，适合电池供电设备的长时间使用。 先进的硬件集成 集成电路：包括功率放大器（PA）、低噪声放大器（LNA）、RF balun 和天线开关，提供稳定的无线性能。 内置存储：集成 SRAM 和 Flash，部分型号支持外部扩展存储。 安全功能：提供硬件加密引擎（AES、SHA 等），支持安全启动（Secure Boot）和闪存加密功能。

如需更多详细技术资料，请参考：[ESP32 系列芯片 | 乐鑫官网 (espressif.com)](#)。

安装 ESP-IDF

⚠️注意

XFusion 目前对接的是 esp-idf v5.0 版本。

详细步骤见 ESP-IDF 官方文档：

《快速入门 - ESP32 - — ESP-IDF 编程指南 v5.0.6 文档 (espressif.com)》。

安装 XFusion

如果安装 esp-idf 选用 linux 环境。详情参考：linux 环境搭建

使用 XFusion 编译

1. 激活 ESP-IDF 环境 我们在每次打开一个新的终端时，需要激活 ESP-IDF 环境

   - **普通激活**

   ```bash
   cd esp-idf      # 进入 esp-idf 文件夹
   . ./export.sh   # 导出 esp-idf 相关环境变量
   ```

   由于激活需要在每次打开新终端的时候都需要执行，如果按照上述操作，则每次都需要切换文件夹。 以下通过 alias 指令在 .bashrc 中定义别名，简化了激活的方式。

   - **便捷激活**

   ```bash
   vim ~/.bashrc # 根据不同 shell ，zsh 就是 .zshrc
   ```

   进入 vim 界面后，输入 `Shift+g`，跳转到最后一行。 按下 `o` 插入自己的命令。

   ```bash
   alias get_idf=". ~/esp-idf/export.sh"   # 双引号后面是 esp-idf 路径
   ```

   然后通过 `Esc` 退出编辑。 最后通过 `:+w+q` 再加上回车确认保存。

   至此，重启终端后。每次激活只需要输入：

   ```bash
   get_idf
   ```

2. 激活 XFusion 环境 通过之前在 .bashrc 中保存的激活命令的别名来激活

```bash
get_xf esp32
```

3. 至此，就可以使用 xf 命令了

```bash
xf --help
```

最后更新于: 2024/11/29 16:13:19

Previous page
介绍

Next page
从 ws63 开始

从 ws63 开始

作者

kirto

本文主要介绍如何搭建 WS63 的 XFusion 开发环境

关于 WS63

WS63 系列是 2.4GHz Wi-Fi 6 BLE 星闪多模 IoT SoC 芯片，其中增强款芯片 WS63 支持 2.4GHz 的雷达人体活动检测功能，适用于大小家电、电工照明及对人体出没检测有需求的常电类物联网智能场景。

- 集成 IEEE 802.11 b/g/n/ax 基带和 RF 电路，包括功率放大器 PA、低噪声放大器 LNA、RF balun、天线开关以及电源管理模块等；
- 支持 20MHz 频宽，提供最大 114.7Mbps 物理层速率，支持更大的发射功率和更远的覆盖距离；
- 支持 BLE 1MHz/2MHz 频宽、BLE4.0/4.1/4.2/5.0/5.1/5.2 协议、BLE Mesh 和 BLE 网关功能，最大空口速率 2Mbps；
- 支持星闪 SLE 1MHz/2MHz/4MHz 频宽、SLE1.0 协议、支持 SLE 网关功能，最大空口速率 12Mbps。

WS63 系列芯片采用 QFN40（5mm x 5mm）封装，匹配不同场合的应用，细分为下列两种：

- Hi3863：合封 4MB Flash，支持 WiFi、SLE、BLE 多模并发，支持单天线通道
- Hi3863E：支持雷达人体活动检测，合封 4MB Flash，支持 WiFi、SLE、BLE 多模并发，支持双天线通道

见：《WS63 芯片 | 海思官网 (hisilicon.com)》。

激活 XFusion

通过之前在 .bashrc 中保存的激活命令的别名来激活

```bash
get_xf ws63
```

使用 `xf target -s` 命令，可以帮助我们确认导出的 target 是否是 ws63

安装 WS63 SDK

获取源码

激活后，我们仅仅需要一个命令。XFusion 便会自动的根据 `boards/nearlink/ws63` 下的 `target.json` 文件，来下载对应版本的sdk。

```bash
xf target -d
```

安装 WS63 SDK 环境依赖

ws63 的 SDK 是使用 python + cmake 编译，其中依赖一些软件，需要安装

配置 Shell

配置默认使用 bash。打开Linux终端，执行命令“sudo dpkg-reconfigure dash”， 选择 no。

```bash
sudo dpkg-reconfigure dash
```

安装 Cmake

打开 Linux 终端，执行命令“sudo aptinstall cmake”，完成Cmake 的安装。

```bash
sudo apt install cmake
```

安装pycparser

```bash
pip3 install pycparser==2.21
```

最后更新于: 2024/11/29 16:13:19

Previous page
从 esp32 开始

Next page
从 bs21 开始

使用 vscode 插件

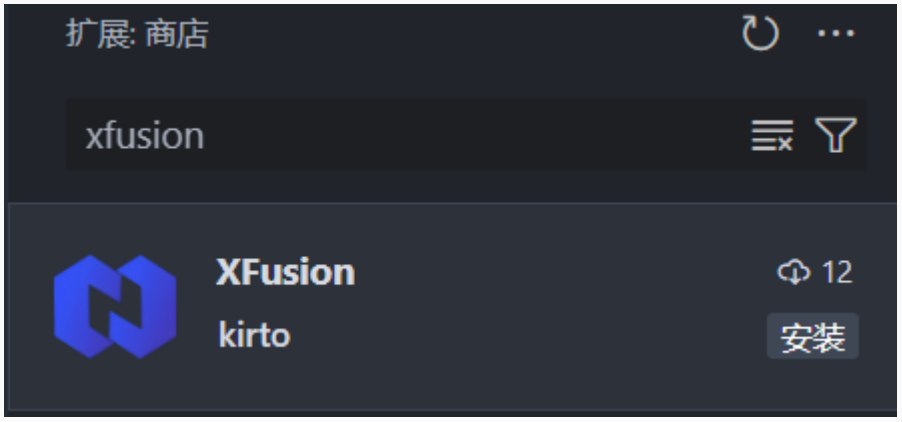
作者

kirto

本文主要介绍如何使用 vscode 插件简化开发流程

安装 XFusion vscode 插件

在 vscode 的拓展商店搜索 xfusion



点击安装即可安装插件

安装完插件左侧边栏会出现一个xfusion图标。点击图标会出现下面的启动页面。

当然，也可以在插件设置中配置路径。 点击`选择路径`，找到 XFusion 路径后点击 `以后不再提示` 然后关闭这个页面。 就完成了 vscode 插件安装

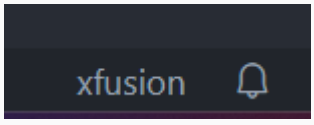
如何使用 XFusion 插件

> ⚠️ **注意**
>
> 执行以下操作前至少要打开一个终端 插件的本质是可视化的调用指令

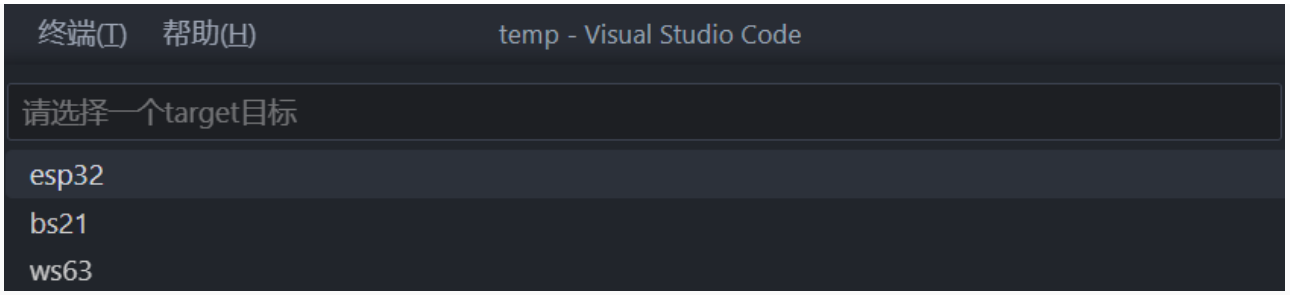
---

export 的过程

点击右下角的 xfusion 标签



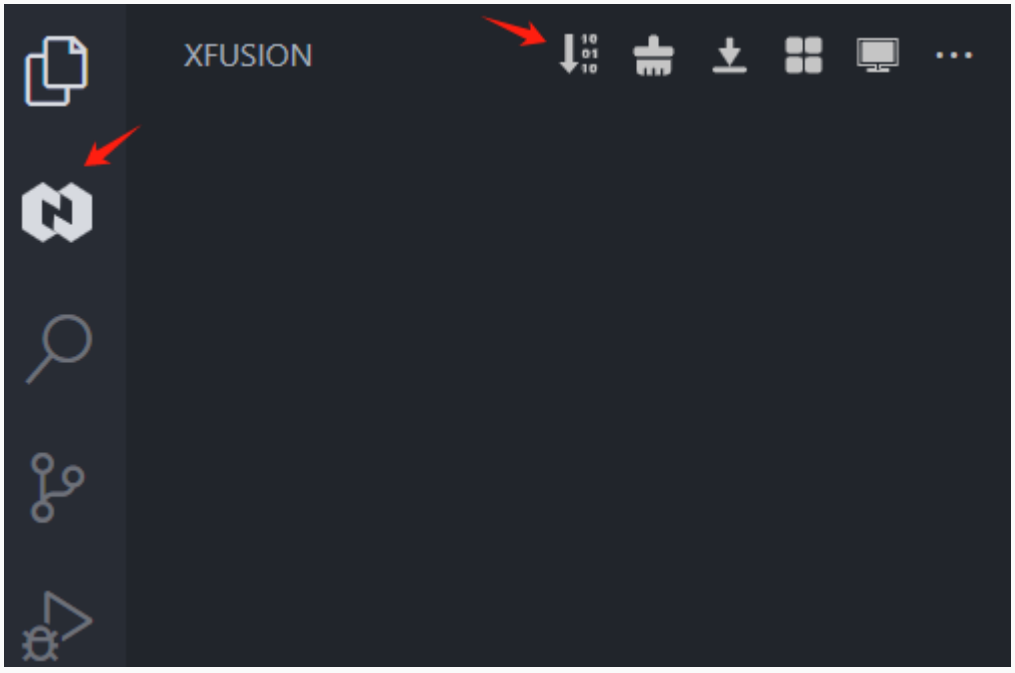
选择一个 target 。这里对应着命令行中的 `激活 xfusion` 操作。



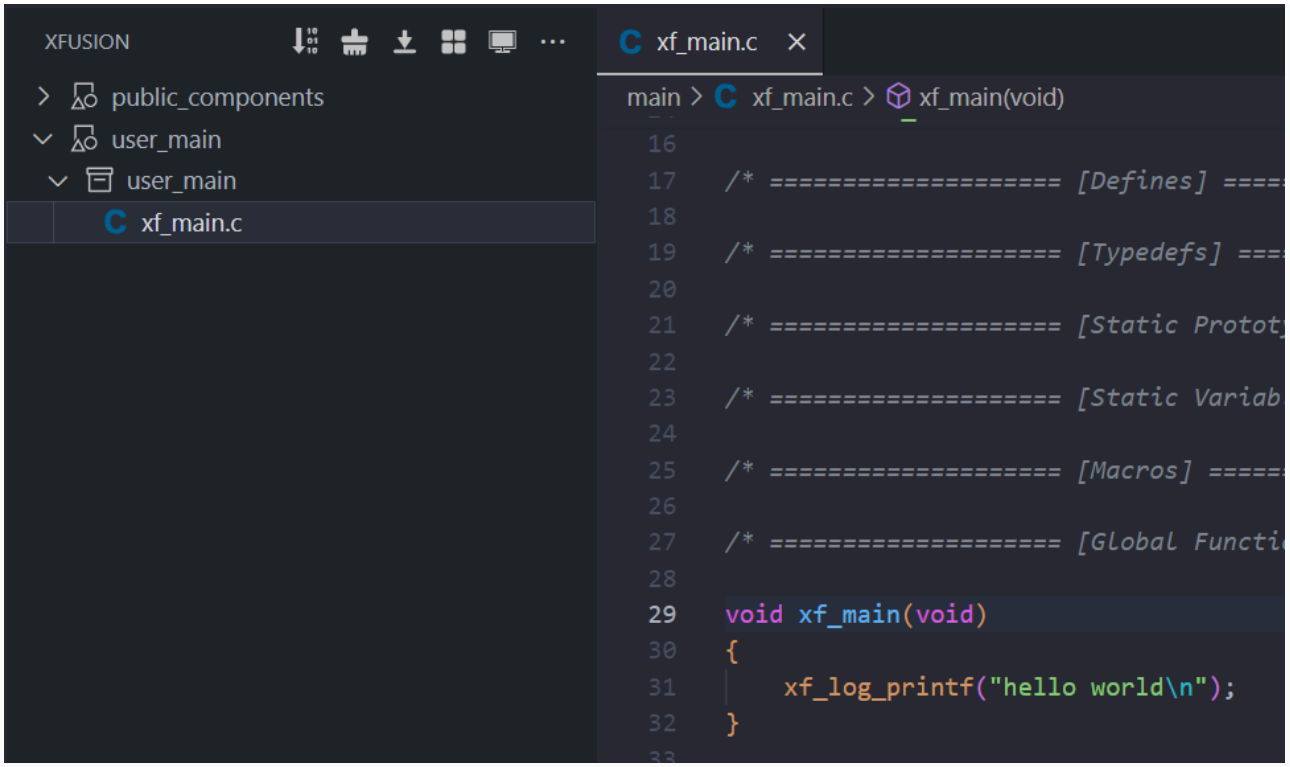
此时，插件就会自动发送 export 命令到终端了。

插件编译

点击左侧边栏的 logo。点击 `编译` 图标。这一步对应命令行的操作为[xf build](#)。



这时会出现文件目录树。目录树主要展示你调用组件目录和源文件。

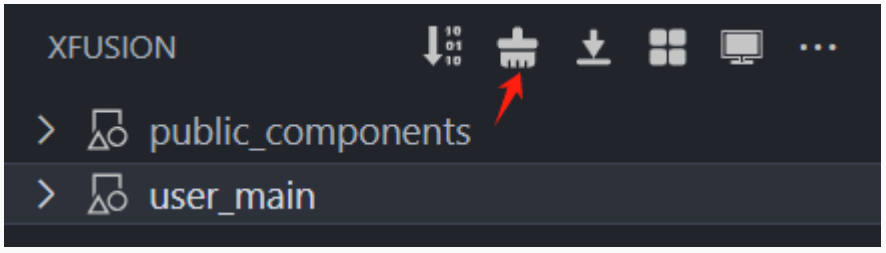


目录树会有以下几种目录：

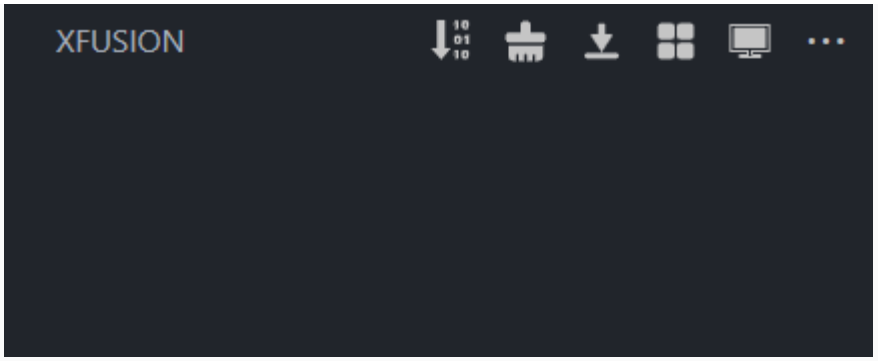
- public_components (公共组件库)
- user_components (用户组件库)
- user_dirs (用户文件夹)
- user_main (用户主文件夹)

插件清除

点击 清除 图标。这一步对应命令行的操作为xf clean。

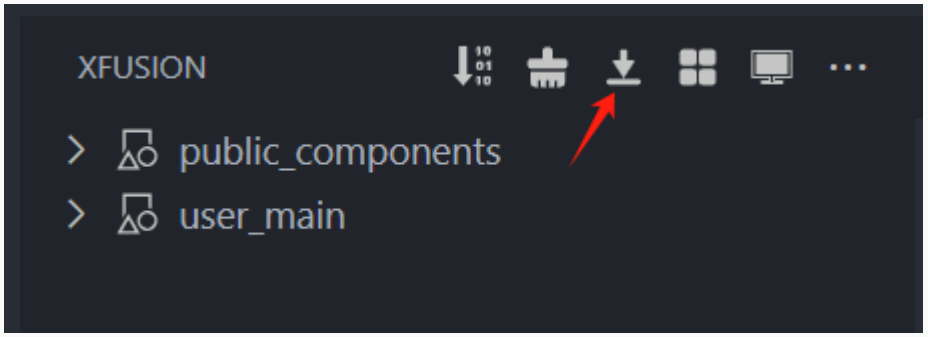


则清除编译产生的中间产物，并清空目录树的显示



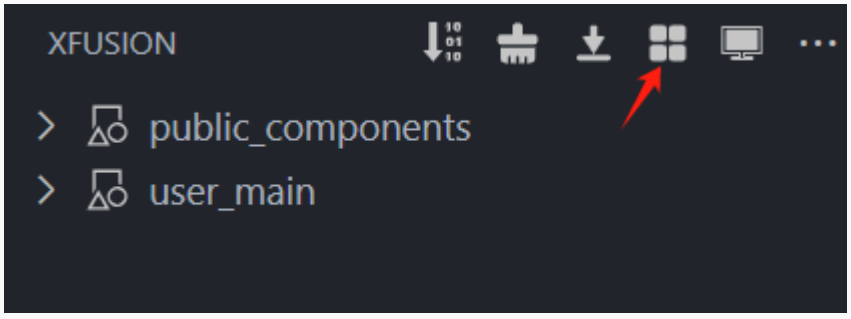
插件烧录

点击 `下载` 图标。这一步对应命令行的操作为xf flash。



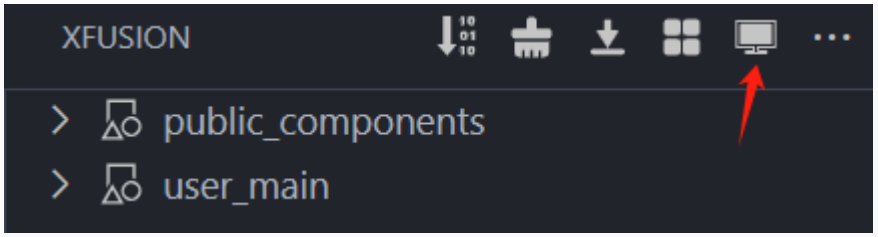
插件打开 menuconfig

点击 `菜单` 图标。这一步对应命令行的操作为xf menuconfig。

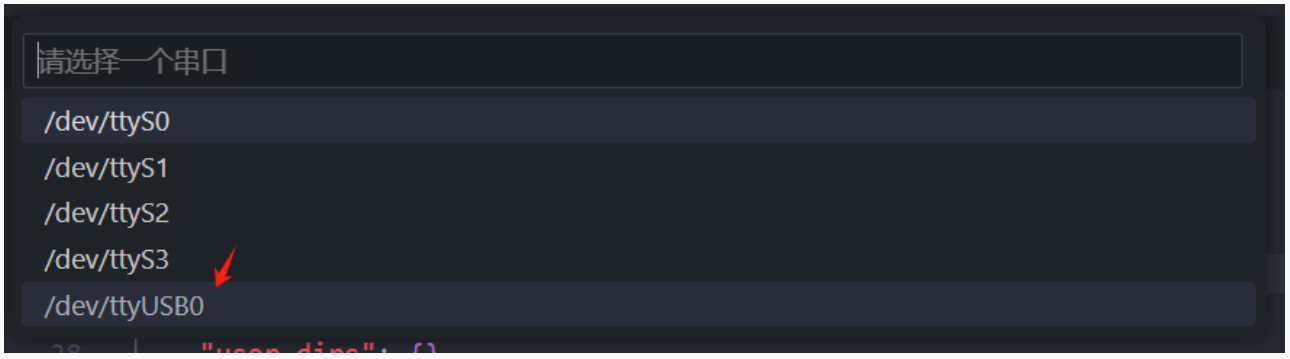


插件打开串口监视器

点击 `监视器` 图标。这一步对应命令行的操作为xf monitor [串口号]。



然后，选择串口。就可以在命令行中查看了。



最后更新于: 2024/11/29 16:13:19

Previous page
安装一个组件

Next page
xf 命令参考

XF 命令手册

作者

kirto

本文主要介绍 xf 命令的具体功能和用法

命令名称: `help`

功能: 展示 xf 所有命令

使用方法:

```
xf --help
```

参数:

无

命令名称: `build`

功能: 编译 XFusion 工程（需对接）。

使用方法:

```
xf build
```

参数:

无

命令名称: `clean`

功能: 清空编译中间产物（需对接）。

使用方法:

```
xf clean
```

参数:

无

命令名称: `create`

功能: 创建一个 XFusion 新工程。

使用方法:

```
xf create [工程名称]
```

参数:

无

命令名称: `export`

功能: 导出 SDK 源工程（需对接）。

使用方法:

```
xf export [工程名称]
```

参数:

无

命令名称: `update`

功能：更新对应 SDK 的工程（需对接）。

使用方法：

```
xf update [工程名称]
```

参数：

无

命令名称：`flash`

功能：烧录固件（需对接）。

使用方法：

```
xf flash
```

参数：

无

命令名称：`install`

功能：安装指定的包。

使用方法：

```
xf install [包名] [参数]
```

参数：

- `-v, --version [版本]`：安装指定版本包。
- `-g, --glob`：安装到全局。

命令名称：`uninstall`

功能：卸载指定的包。

使用方法：

```
xf uninstall [包名]
```

参数：

- `-g, --glob`：卸载全局的包。

命令名称：`search`

功能： 模糊搜索包名。

使用方法：

```
xf search [包名]
```

参数：

无

命令名称：`menuconfig`

功能： 图形化配置 XFusion。

使用方法：

```
xf menuconfig
```

参数：

无

命令名称：`monitor`

功能： 命令行串口监视器。

使用方法：

```
xf monitor [串口号]
```

参数:

无

命令名称: `target`

功能: target相关操作。

使用方法:

```
xf target [参数]
```

参数:

- `-s, --show`: 展示目标和目标路径。
- `-d, --download`: 下载SDK(依赖target.json)。

最后更新于: 2024/11/29 16:13:19

Previous page
使用 vscode 插件

Next page
API 参考

深入了解

作者

dotc

本文介绍 XFusion 的工程、文件夹结构等更深入的内容。

目录

- XFusion 目录结构
- XFusion 构建流程
- xf build 构建脚本
- xf task 协程调度器详解

最后更新于: 2025/2/8 19:20:14

xf build 构建脚本

作者

kirto

本文主要说明，xf build的功能和相关源码。

xf build 构思来源

在开发 XFusion 的时候，我意识到在 c 语言编译中由于各种构建脚本不同一，导致我们做一个中间件的时候不得不去适配多个构建脚本。 在这种情况下，我们考虑能不能做一个脚本，该脚本用于生成一个含有各种编译信息的 json 文件。再由生成器生成不同的构建脚本，或者工程。

上述的思路就是 xf build 的制作初衷。

再基础的优化和功能添加后，xf build 具有以下的功能：

- 生成含有各种编译信息的 json 文件
- 可以拓展的各种生成器插件
- 支持导出原生工程
- 支持 menuconfig 可视化裁剪配置工程
- 支持基于 pyserial 的命令行串口监视器
- 支持基于 kconfiglib 的 menuconfig 配置工具
- 包管理工具

仓库地址

github: https://github.com/x-eks-fusion/xf_build

gitee: https://gitee.com/x-eks-fusion/xf_build

xf 命令参考

参考

详见：[xf 命令参考]

构建相关 API

这部分是在编译的时候，xf_project.py xf_collect.py 所调用的 API 。

project_init

功能： 初始化一个 XFusion 工程。

参数：

- `user_dirs` (`list`): 用户添加的额外文件夹。

返回值：

无

program

功能： 开始构建工程。

参数：

- `cflags` (`list`): 全局的 cflags 参数。

返回值：

无

collect

功能： 收集编译信息。

参数：

- `srcs` (`list`): 收集源文件。
- `inc_dirs` (`list`): 收集头文件路径。

- `requires` (`list`): 模块依赖关系。
- `cflags` (`list`): 模块 cflags 参数。

返回值:

无

get_define

功能: 从 menuconfig 中获取宏定义的值。

参数:

- `define` (`str`): 需要获取的宏定义。

返回值:

返回宏的值，如果是bool类型返回y或者n

插件对接 API

这部分是在对接的时候，插件需要对接的 API 。

build

功能: 根据收集的编译信息，生成 SDK 构建脚本，并启动编译。

参数:

- `args` (`list`): 顶层调用命令传递下来的参数。

返回值:

无

clean

功能: 清除编译的中间产物。

参数:

- `args` (`list`): 顶层调用命令传递下来的参数。

返回值:

无

flash

功能： 调用命令烧录。

参数：

- `args` (`list`): 顶层调用命令传递下来的参数。

返回值：

无

menuconfig

功能： 调用底层 menuconfig。

参数：

- `args` (`list`): 顶层调用命令传递下来的参数。

返回值：

无

export

功能： 根据收集的编译信息，导出原始 sdk 工程。

参数：

- `args` (`list`): 顶层调用命令传递下来的参数。

返回值：

无

update

功能： 根据收集的编译信息，更新已有的 sdk 工程。

参数：

- `args` (`list`): 顶层调用命令传递下来的参数。

返回值:

无

插件调用 API

exec_cmd

功能: 在终端中执行命令行。

参数:

- `command` (`str|list`): 需要执行的命令。

返回值:

无

apply_template

功能: 应用模板文件。

参数:

- `temp` (`str`): 模板文件的路径。
- `save` (`str`): 保存应用模板后的文件路径。
- `replace` (`dict`): 保存应用模板后的文件路径。

返回值:

无

apply_components_template

功能: 处理并生成基于模板的文件，按照配置数据自动生成对应的目录和文件。

参数:

- `temp` (`str`): 模板文件的名称，用于加载模板内容。

- `save` (`str`): 生成文件的后缀或文件名。如果以 `.` 开头，表示后缀；否则为完整文件名。

返回值:

无

get_define

功能: 获取 menuconfig 的宏。

参数:

- `define` (`str`): menuconfig 中的定义。

返回值:

无

cd_to_root

功能: 切换到 xfusion 的根目录下。

参数:

无

返回值:

无

cd_to_target

功能: 切换到当前 target 目录下。

参数:

无

返回值:

无

cd_to_project

功能： 切换到当前工程目录下。

参数：

无

返回值：

无

get_sdk_dir

功能： 获取 SDK 所在的路径。（只有配置了 target.json 才可以获取）

参数：

无

返回值：

返回 SDK 的路径

get_XF_ROOT

功能： 获取 XFusion 路径

参数：

无

返回值：

返回 XFusion 的路径

get_XF_PROJECT_PATH

功能： 获取当前 target 目录。

参数：

无

返回值：

返回当前 target 目录

get_XF_PROJECT_PATH

功能： 获取当前工程路径。

参数：

无

返回值：

返回当前工程路径

get_PROJECT_BUILD_PATH

功能： 获取当前工程的 build 路径。

参数：

无

返回值：

返回当前工程的 build 路径

get_ROOT_PLUGIN

功能： 获取当前 target 插件路径。

参数：

无

返回值：

返回当前 target 插件路径。

get_PROJECT_CONFIG_PATH

功能： 获取当前工程配置路径。

参数：

无

返回值:

返回当前工程配置路径。

最后更新于: 2024/12/2 20:23:23

Previous page
XFusion 构建流程

Next page
xf task 协程调度器详解

xf task 协程调度器详解

作者

kirto

本文主要说明，xf task的功能、移植和使用。

xf_task 介绍

背景

在嵌入式系统中，任务调度是系统运行的核心。场景的调度方式有抢占式和协作式。 抢占式调度是大多数实时操作系统（RTOS）采用的方式，能够中断正在运行的任务切换到优先级更高的任务，适用于对实时性要求高的场景。 但实现较为复杂，容易引发资源竞争，每个任务的开销大也是其无法适用于更加极限的嵌入式环境中。 相比之下，协作式调度在任务主动让出 CPU 控制权时进行切换。其实现简单、切换开销小，更容易避免资源竞争问题。 尽管市面上绝大多数 RTOS 以抢占式调度为主，但在许多低功耗和资源受限的场景中，协作式调度的优势往往更加重要。

介绍

xf_task 是 XFusion 的一个核心功能。其本身是一个轻量级协作式调度器。可以跑在多线程的环境下也可以跑在裸机的环境中。 相比于RTOS，其本身的占用仅仅需要 2.5k （keil ac5 编译得出）左右，且创建一个 ntask 任务仅仅需要 96B 的大小。 xf_task 实现了三种任务：ctask（有栈协程）、 ntask（无栈协程）和 ttask （定时器任务） ctask 提供了上

下文保存，使用体验上无限接近于线程。 ntask 提供了非常轻量的任务方式，非常适合资源受限的嵌入式设备。 ttask 提供了软件定时器的拓展。

功能

xf task支持非常丰富的功能，这些功能后续会一一进行详细介绍：

1. 协作式调度，支持裸机移植，支持多线程中构造多个调度器 裸机和多线程中移植
2. 支持三种种任务，ttask （定时任务）、ntask（无栈协程）、ctask （有栈协程）三种任务用法
3. 移植简单，ttask、ntask 仅需对接时间戳的获取。ctask 对接上下文的创建和切换函数（可通过宏屏蔽）xf_task 配置项详解
4. 支持协作式优先级 优先级机制
5. 支持两种消息队列，queue和ctask_queue 消息队列机制
6. 支持任务触发机制 时间触发和事件触发
7. 支持紧急任务 紧急任务机制
8. 支持任务饥饿度机制 任务饥饿与饥饿度
9. 支持 mbus 发布订阅机制 同步/异步的发布订阅机制
10. 支持任务池机制 任务池和任务回收机制
11. 仅依赖 xf_utils, 支持 c99

开源地址

> github 开源地址：https://github.com/x-eks-fusion/xf_task
> gitee 开源地址：https://gitee.com/x-eks-fusion/xf_task

原理介绍

调度器原理

调度流程图

```mermaid
graph LR
    F["任务创建"]
    A["就绪态（链表数组）"]
    B["阻塞态（链表）"]
    C["运行态（指针）"]
    D["删除态（链表）"]
    E["挂起态（链表）"]

    F --> B
    subgraph Box [调度器]
        direction LR
        A --> B
        B --> C
        C --> A
        E -.-> B
        D
    end
```

调度流程解析

源码位置

调度部分的代码主要是在 `src/kernel/xf_task_manager.c` 文件的 `void xf_task_manager_run(xf_task_manager_t manager)` 函数中。

调度流程

1. 任务创建：
   - 不同的任务创建过程并不相同。但是它们共同继承 xf_task_base_t 基类
   - 无论什么任务，创建后会进入阻塞态
2. 阻塞态：
   - 通过handle的虚函数，不同的任务有不同的阻塞条件处理函数。
   - ctask和ntask由于无阻塞条件，所以自动进入就绪态。后续遇到delay等函数才会带着阻塞条件（时间或事件）再次进入阻塞态
   - ttask会经过一次delay时间后，从阻塞态过度到就绪态

- 此外这里会对比所有时间条件的最小值，这将为后续的空闲任务提供最大的休息时间

3. 就绪态：
   - 就绪态会从高到低不同优先级（数值越小优先级越高）找到可执行的任务并让其进入运行态。
   - 就绪态找到了可以运行的任务后就不会继续遍历
   - 如果没有就绪任务，则会再次更新阻塞态中时间最小值，并将其作为空闲任务最大休息时间传递给空闲任务。后运行空闲任务。空闲任务运行完毕进入下一次调度
4. 运行态：
   - 运行态中的不同任务进行运行不同的exec的虚函数
   - 运行完毕则再度进入下一次调度
5. 挂起态：
   - 挂起态时用户自己主动挂起不希望运行的任务。当挂起任务出于阻塞态时，挂起的时间将不会算入阻塞时间。
   - 挂起的任务不会进入调度器，除非用户手动释放。
   - 释放后的任务进入阻塞态，再次参与调度。
6. 删除态：
   - 当任务被删除时，不会立即删除。第一时间将会被放入删除态中。
   - 删除态真正的处理会在调度器进入空闲之前，统一被释放掉。
   - 在内存池中，任务的删除态被替换为内存池回收函数。以便后续重复利用。

对象及继承关系

继承关系图

mermaid

```mermaid
classDiagram
    %% 虚函数表结构
    class xf_task_vfunc_t {
      +xf_task_create_t constructor
      +xf_task_reset_t reset
      +xf_task_update_t update
      +xf_task_exec_t exec
    }

```

```
%% 任务基类
class xf_task_base_t {
  +xf_list_t node
  +xf_task_manager_t manager
  +xf_task_func_t func
  +void *arg
  +uint32_t type
  +uint32_t state
  +uint32_t flag
  +uint32_t signal
  +uint32_t priority
  +uint32_t delay
  +xf_task_time_t wake_up
  +xf_task_time_t suspend_time
  +int32_t timeout
  +const xf_task_vfunc_t *vfunc
  +xf_task_delete_t delete
}

%% 循环任务
class xf_ttask_handle_t {
  +uint32_t count
  +uint32_t count_max
}

%% 条件任务
class xf_ntask_handle_t {
  +xf_ntask_compare_func_t compare
  +xf_ntask_status_t status
  +xf_list_t lc_list
  +xf_list_t args_list
}

%% 协作任务
class xf_ctask_handle_t {
  +size_t stack_size
  +xf_task_context_t context
  +void *stack
  +xf_list_t queue_node
}

%% 任务调度器
```

```
    class xf_task_manager_handle_t {
      +xf_task_t current_task
      +xf_task_t urgent_task
      +xf_list_t ready_list
      +xf_list_t blocked_list
      +xf_list_t suspend_list
      +xf_list_t destroy_list
      +xf_task_on_idle_t on_idle
      +xf_list_t hunger_list
      +xf_task_context_t context
    }

    %% 继承关系
    xf_ttask_handle_t --|> xf_task_base_t : inherits
    xf_ntask_handle_t --|> xf_task_base_t : inherits
    xf_ctask_handle_t --|> xf_task_base_t : inherits

    %% 关联关系
    xf_task_base_t --> xf_task_vfunc_t : uses
    xf_task_base_t --> xf_task_manager_handle_t : belongs to
```

不同对象的具体结构

这里采用了观察者模式来实现该功能，调度器对象 task_manager 不依赖具体的任务对象，转而依赖其基类 task_base 。这使得具体的 task 子类可以随意拓展，解耦了整体的逻辑结构。

task_base 对象

```c
typedef struct _xf_task_base_t {
    xf_list_t node;                 /*!< 任务节点，挂载在 manager 上 */
    xf_task_manager_t manager;      /*!< 保存 task 所属的 manager ，以便更快访问 mar
    xf_task_func_t func;            /*!< 每个任务所执行的内容 */
    void *arg;                      /*!< 任务中用户定义参数 */
    uint32_t type:      2;          /*!< 任务类型，见 @ref xf_task_type_t */
    uint32_t state:     3;          /*!< 任务状态，见 @ref xf_task_state_t */
    uint32_t flag:      8;          /*!< 任务标志位，外部设置的标志位，内部只会读取
    uint32_t signal:    9;          /*!< 任务间信号，内部传递消息使用，外部无法设置，
                                     *   见 XF_TASK_SIGNAL_* 宏 */
    uint32_t priority:  10;         /*!< 任务优先级，具体最大值参考 @ref XF_TASK_PRI
```

```c
    uint32_t delay;                 /*!< 对类型于有上下文是延时时间，对于没有上下文
    xf_task_time_t wake_up;          /*!< 唤醒时间，通过延时时间计算而来 */
    xf_task_time_t suspend_time;    /*!< 挂起时间，挂起期间内的时间不会算入延时时间
    int32_t timeout;                /*!< 超时时间，正数为超时时间，负数则属于提前唤
    const xf_task_vfunc_t *vfunc;   /*!< 虚函数指针，由子对象实现具体操作。
                                     *   虚函数指针是实现不同类型任务统一调度的关键
    xf_task_delete_t delete;        /*!< 虚函数指针，其内容通常为回收任务内存
                                     *   task pool 中通过替换它实现任务池回收任务 */

#if XF_TASK_HUNGER_IS_ENABLE
    xf_list_t hunger_node;          /*!< 饥饿节点，挂载在 manager 上的 hunger_list
                                     *   以便更快速的遍历饥饿任务 */
    uint32_t hunger_time;           /*!< 任务饥饿度，单位为 ms。超过该时间，任务爬升
#endif

#if XF_TASK_USER_DATA_IS_ENABLE
    void *user_data;                /*!< 用户传递的参数 */
#endif // XF_TASK_USER_DATA_IS_ENABLE

} xf_task_base_t;
```

提供了所有 task 的公共部分。其中比较重要的是这里定义了一个虚函数表 constructor, reset, update, exec。这就是每个子对象需要实现的虚函数。对于 task_manager 来说，操作的也是这 task 的公共部分。其余非公共部分通过虚函数来间接实现。

ttask 对象

```c
typedef struct _xf_ttask_handle_t {
    xf_task_base_t base;    /*!< 继承父对象 */
    uint32_t count;         /*!< 记录 ttask 剩余循环次数 */
    uint32_t count_max;     /*!< 记录 ttask 循环次数上限 */
} xf_ttask_handle_t;
```

继承于 task_base 对象。 除了这个对象之外，还需要实现一个注册函数 void xf_xxx_vfunc_register(void) 。 这里会通过 task 注册表 xf_task_reg.inc 自动生成。注册需要对接父函数的虚函数。定时器支持任意次数的循环和无限循环。

ntask 对象

```c
typedef struct _xf_ntask_handle_t {
    xf_task_base_t base;                /*!< 继承父对象 */
    xf_ntask_compare_func_t compare;    /*!< 直到这个函数返回 0，会通过事件信号触发
    xf_ntask_status_t status;           /*!< 记录 ntask 退出状态 */
    xf_list_t lc_list;                  /*!< 记录 ntask 上下文 */
    xf_list_t args_list;                /*!< 参数收集器 */
} xf_ntask_handle_t;
```

继承于 task_base 对象。 除了这个对象之外，还需要实现一个注册函数 void xf_xxx_vfunc_register(void) 。 这里会通过 task 注册表 xf_task_reg.inc 自动生成。注册需要对接父函数的虚函数。 无栈协程实现，借鉴了 protothread 的实现。 通过 switch case 的封装实现了循环以及 delay 的功能。 支持任意层级的嵌套。

ctask 对象

```c
typedef struct _xf_ctask_handle_t {
    xf_task_base_t base;        /*!< 继承 task_base 父对象 */
    size_t stack_size;          /*!< 任务上下文堆栈大小 */
    xf_task_context_t context;  /*!< 任务上下文对象 */
    void *stack;                /*!< 任务上下文堆栈地址 */
    xf_list_t queue_node;       /*!< 队列等待 */
} xf_ctask_handle_t;
```

继承于 task_base 对象。相比无栈协程，该对象需要对接保存上下文和切换上下文的函数。而且用户需要提供 XF_TASK_CONTEXT_TYPE 上下文的对象类型。当然如果你觉得移植困难，可以通过配置文件屏蔽 ctask 只用 ttask 和 ntask。

task_manager 对象

```c
typedef struct _xf_task_manager_handle_t {
    xf_task_t current_task;                        /*!< 当前执行任务 */
    xf_task_t urgent_task;                         /*!< 紧急任务 */
    xf_list_t ready_list[XF_TASK_PRIORITY_LEVELS]; /*!< 任务就绪队列 */
    xf_list_t blocked_list;                        /*!< 任务阻塞队列 */
    xf_list_t suspend_list;                        /*!< 任务挂起队列，挂起任务不参
    xf_list_t destroy_list;                        /*!< 任务销毁队列，进行异步销毁
    xf_task_on_idle_t on_idle;                     /*!< 空闲任务回调 */
#if XF_TASK_HUNGER_IS_ENABLE
```

```c
    xf_list_t hunger_list;                          /*!< 任务饥饿队列，达到其指定值
#endif // XF_TASK_HUNGER_IS_ENABLE
#if XF_TASK_CONTEXT_IS_ENABLE
    xf_task_context_t context;                      /*!< 调度器上下文 */
#endif // XF_TASK_CONTEXT_IS_ENABLE
} xf_task_manager_handle_t;
```

位于 kernel 文件夹内，用于调度任务的运行。默认有 default_manager。不同线程中可以创建独立的 task_manager , 对应的函数都有 xxx_with_manager 版本。

功能详解

三种任务的用法

ttask、ntask、ctask三种任务本质上都是异步的，只不过为了更方便用户的同步使用习惯而封装更像是写“裸机”。

异步调用 ttask

```c
#include "xf_task.h"
#include <stdio.h>

static void task(xf_task_t task)
{
    printf("hello ttask");
}

void app_main()
{
    // 创建任务
    xf_ttask_create(task, (void *)1, 1, 1000, 5);
    // 创建无限循环任务
    xf_ttask_create_loop(task, (void *)2, 1, 2000);
}
```

```c

/**
 * @brief 创建 ttask 循环任务。
 *
 * @param func ttask 任务执行的函数。
 * @param func_arg 用户自定义执行函数参数。
 * @param priority 任务优先级。
 * @param delay_ms 任务延时周期，单位为毫秒。
 * @return xf_task_t 任务对象，返回为 NULL 则表示创建失败
 */
xf_task_t xf_ttask_create_loop(xf_task_func_t func, void *func_arg, uint16_t priorit

/**
 * @brief 创建指定次数的 ttask 任务。
 *
 * @param func ttask 任务执行的函数。
 * @param func_arg 用户自定义执行函数参数。
 * @param priority 任务优先级。
 * @param delay_ms 任务延时周期，单位为毫秒。
 * @param count 任务循环的次数上限。
 * @return xf_task_t 任务对象，返回为 NULL 则表示创建失败
 */
xf_task_t xf_ttask_create(xf_task_func_t func, void *func_arg, uint16_t priority, ui
```

优点:

- 周期性任务很方便
- 创建任务的开销非常小
- 可以选择多次循环还是无限循环
- 移植难度简单

缺点:

- 遇到非周期任务，使用起来很麻烦。异步的逻辑构思很麻烦。
- 不允许出现 delay 等阻塞函数。

同步调用 ctask

```c
#include "xf_task.h"
#include <stdio.h>

static void task(xf_task_t task)
{
    // 获取任务参数
    while (1) {
        printf("hello ctask");
        // 使用 ctask 专属延时函数
        xf_ctask_delay(1000);
    }
}
void app_main()
{
    // 创建任务
    xf_ctask_create(task, (void *)1, 1, 1024);
}
```

```c
/**
 * @brief 创建 ctask 任务。
 *
 * @param func ctask 任务执行的函数。
 * @param func_arg 用户自定义执行函数参数。
 * @param priority 任务优先级。
 * @param stack_size 任务上下文堆栈大小。
 * @return xf_task_t 任务对象，返回为 NULL 则表示创建失败
 */
xf_task_t xf_ctask_create(xf_task_func_t func, void *func_arg, uint16_t priority, si
```

优点：

- 创建大循环，并且可以调用 delay 等阻塞函数。
- 嵌套非常方便，不需要修改原有函数。
- 同步调用思维，更加贴近裸机的调用习惯。

缺点：

- 单个任务的开销比较大，不适合多次创建。
- 需要额外移植保存上下文，移植难度较大。

更加轻量级的同步调用 ntask

```c
#include "xf_task.h"
#include <stdio.h>

static void task(xf_task_t task)
{
    XF_NTASK_BEGIN(task);

    while (1) {
        printf("hello ntask\n");
        // ntask 专属延时
        xf_ntask_delay(1000);
    }

    XF_NTASK_END();
}

void app_main()
{
    // 创建任务
    xf_ntask_create(task, (void *)1, 1);
}
```

```c
/**
 * @brief 创建无栈协程
 *
 * @param manager 任务调度器
 * @param func 任务执行函数
 * @param func_arg 任务参数
 * @param priority 任务优先级
 * @return xf_task_t 任务对象，NULL 表示创建失败
 */
xf_task_t xf_ntask_create(xf_task_func_t func, void *func_arg, uint16_t priority);
```

优点：

- 创建大循环，并且可以调用 delay 等阻塞函数。
- 同步调用思维，更加贴近裸机的调用习惯。

- 单个任务开销很小。
- 移植难度简单。

缺点：

- 不能保存临时变量，需要借助到变量池。
- 可以嵌套，但是需要修改原有的函数。

---

task_manager 的函数及其用法

task_manager 任务管理器，主要的作用就是参与调度。其 xf_task_manager_run 函数需要被放到无限循环中。 裸机中，我们一般只有一个 task_manager ，于是，封装了一个 default_task_manager 简化了调用的管理器参数的传入。 任务管理器支持 Tickless 模式，可以通过空闲函数的参数（最大空闲时间）进行休眠。休眠器件 SysTick 会不起作用。所以我们提供了补偿函数，用户通过 RTC 等手段记录休眠真实时间，然后通过调用补偿函数，对现有的任务进行补偿。以此达到空闲时休眠的效果。

API 展示

任意任务管理器操作函数

- **创建任务管理器：**

```c
xf_task_manager_t xf_task_manager_create(xf_task_on_idle_t on_idle);
```

- **设置 manager 的空闲回调函数：**

```c
xf_err_t xf_task_manager_set_idle(xf_task_manager_t manager, xf_task_on_idle_t on_i
```

- **启动任务管理器调度任务：**

```c
void xf_task_manager_run(xf_task_manager_t manager);
```

- **为任意任务管理器紧急任务：**

```c
xf_err_t xf_task_set_urgent_task_with_manager(xf_task_manager_t manager, xf_task_t t
```

- **为任意任务管理器设置时间补偿：**

```c
xf_err_t xf_task_manager_set_compensation_time(xf_task_manager_t manager, xf_task_ti
```

默认任务管理器操作函数

- **创建默认的任务管理器：**

```c
xf_err_t xf_task_manager_default_init(xf_task_on_idle_t on_idle);
```

- **置默认任务管理器的空闲回调函数：**

```c
xf_err_t xf_task_manager_set_default_idle(xf_task_on_idle_t on_idle);
```

- **获取默认的任务管理器：**

```c
xf_task_manager_t xf_task_get_default_manager(void);
```

- **启动任务管理器调度任务：**

```c
void xf_task_manager_run_default(void);
```

- **为默认任务管理器紧急任务：**

```c
xf_err_t xf_task_set_urgent_task(xf_task_t task, bool force);
```

- **为默认任务管理器设置时间补偿：**

```c
xf_err_t xf_task_manager_set_compensation_time_default(xf_task_time_t time_ms);
```

其它操作函数

- **获取管理器运行的任务：**

```c
xf_task_t xf_task_manager_get_current_task(xf_task_manager_t manager);
```

- **设置当前任务为就绪态：**

```c
xf_err_t xf_task_manager_task_ready(xf_task_manager_t manager, xf_task_t task);
```

- **设置当前任务为挂起态：**

```c
xf_err_t xf_task_manager_task_suspend(xf_task_manager_t manager, xf_task_t task);
```

- **设置当前任务为删除态：**

```c
xf_err_t xf_task_manager_task_destory(xf_task_manager_t manager, xf_task_t task);
```

- **设置当前任务为阻塞态：**

```c
xf_err_t xf_task_manager_task_blocked(xf_task_manager_t manager, xf_task_t task);
```

---

task_base 的函数及其用法

task_base 是 ttask、ntask、ctask的基类，所以它的函数方法可以给这三个子类使用。 但是子类的函数方法是不能通用的。

API 展示

任务调度相关函数

- **删除任务：**

```c
void xf_task_delete(xf_task_t task);
```

- **重置任务：**

```c
void xf_task_reset(xf_task_t task);
```

- **挂起任务：**

```c
xf_err_t xf_task_suspend(xf_task_t task);
```

- **恢复任务：**

```c
xf_err_t xf_task_resume(xf_task_t task);
```

- **触发任务：**

```c
xf_err_t xf_task_trigger(xf_task_t task);
```

查询任务

- **查询任务状态：**

```c
xf_task_state_t xf_task_get_state(xf_task_t task);
```

- **查询任务类型：**

```c
xf_task_type_t xf_task_get_type(xf_task_t task);
```

- **查询任务优先级：**

```c
uint16_t xf_task_get_priority(xf_task_t task);
```

- **查询任务超时：**

```c
int32_t xf_task_get_timeout(xf_task_t task);
```

- **查询任务所属调度器：**

```c
xf_task_manager_t xf_task_get_manager(xf_task_t task);
```

- **查询任务传入参数：**

```c
void *xf_task_get_arg(xf_task_t task);
```

设置任务

- **设置任务延时：**

```c
xf_err_t xf_task_set_delay(xf_task_t task, uint32_t delay_ms);
```

- **设置任务优先级：**

```c
xf_err_t xf_task_set_priority(xf_task_t task, uint16_t priority);
```

- **设置任务执行函数：**

```c
xf_err_t xf_task_set_func(xf_task_t task, xf_task_func_t func, void *arg);
```

饥饿值机制相关

这部分需要设置 XF_TASK_HUNGER_ENABLE 为 1

- **启用并设置任务饥饿值：**

```c
void xf_task_feel_hungery_enable(xf_task_t task, uint32_t hunger_time);
```

- **禁用任务饥饿值：**

```c
void xf_task_feel_hungery_disable(xf_task_t task);
```

user_data 相关

这部分需要设置 XF_TASK_USER_DATA_ENABLE 为 1

- **设置任务 user_data：**

```c
xf_err_t xf_task_set_user_data(xf_task_t task, void *user_data);
```

- **查询任务 user_data：**

```c
void *xf_task_get_user_data(xf_task_t task);
```

---

ttask 的函数及其用法

ttask 的本身函数是形如 xxx_with_manager 结构。在多线程多协程的环境中，可以跑在不同的任务管理器中。 多线程环境下，一个线程跑一个任务管理器，跨任务管理器的通讯需要用线程间通讯 详见：多线程移植 但在裸机环境或者多线程中只用一个线程跑协程的环境下，就可以直接使用不带 _with_manager 的函数，简化了函数的调用。 ttask 的延时时间为 0 的时候，ttask将会变成一个纯事件触发的任务。只有调用 xf_task_trigger() 函数才能被执行。

API 展示

任意任务管理器的函数操作

- **创建指定次数的定时器任务：**

```c
xf_task_t xf_ttask_create_with_manager(xf_task_manager_t manager,
                                       xf_task_func_t func,
                                       void *func_arg,
                                       uint16_t priority,
                                       uint32_t delay_ms,
                                       uint32_t count);
```

- **创建无限循环的定时器任务：**

```c
xf_task_t xf_ttask_create_loop_with_manager(xf_task_manager_t manager,
                                            xf_task_func_t func,
                                            void *func_arg,
                                            uint16_t priority,
                                            uint32_t delay_ms);
```

默认任务管理器的函数操作

- **创建指定次数的定时器任务：**

```c
xf_task_t xf_ttask_create(xf_task_func_t func, void *func_arg, uint16_t priority, ui
```

- **创建无限循环的定时器任务：**

```c
xf_task_t xf_ttask_create_loop(xf_task_func_t func, void *func_arg, uint16_t priorit
```

其它 ttask 专属操作函数

- **设置 ttask 循环次数：**

```c
xf_err_t xf_ttask_set_count(xf_task_t task, uint32_t count);
```

- **查询 ttask 剩余循环次数：**

```c
uint32_t xf_ttask_get_count(xf_task_t task);
```

- **设置 ttask 总循环次数：** （此函数需要配合任务复位才能生效）

```c
xf_err_t xf_ttask_set_count_max(xf_task_t task, uint32_t count_max);
```

当任务初始化的时候，count_max 和 count 会被设置成循环次数。此后 count_max 记录一开始设置的循环次数，用于复位时重置循环次数。count 则是实际剩余循环次数。 当任务的 count 到达 0 后，任务会被自动丢入删除态进行销毁。

ctask 的函数及其用法

ctask 保存了上下文。任务在创建后即可进入就绪态运行。如果遇到 xf_ctask_delay 或者是 xf_ctask_queue_xxx 相关的 API 则进入阻塞态。 ctask 的专属消息队列相比普通消息队列多了一个超时的功能。使得消息队列在等待的情况下不耽误其余任务的运行。

API 展示

任意任务管理器的函数操作

- **创建有栈协程：**

```c
xf_task_t xf_ctask_create_with_manager(xf_task_manager_t manager,
                                       xf_task_func_t func,
                                       void *func_arg,
                                       uint16_t priority,
                                       size_t stack_size);
```

- **ctask 延时函数：**

```c
void xf_ctask_delay_with_manager(xf_task_manager_t manager, uint32_t delay_ms);
```

- **ctask 消息队列创建：**

```c
xf_ctask_queue_t xf_ctask_queue_create_with_manager(xf_task_manager_t manager,
                                                    const size_t size,
                                                    const size_t count);
```

默认任务管理器的函数操作

- **创建有栈协程：**

```c
xf_task_t xf_ctask_create(xf_task_func_t func, void *func_arg, uint16_t priority, si
```

- **ctask 延时函数：**

```c
void xf_ctask_delay(uint32_t delay_ms);
```

- **ctask 消息队列创建：**

```c
xf_ctask_queue_t xf_ctask_queue_create(const size_t size, const size_t count);
```

其它 ctask 专属操作函数

- **ctask 消息队列发送：**

```c
xf_err_t xf_ctask_queue_send(xf_ctask_queue_t queue, void *buffer, uint32_t timeout)
```

- **ctask 消息队列接收：**

```c
xf_err_t xf_ctask_queue_receive(xf_ctask_queue_t queue, void *buffer, uint32_t timec
```

---

ntask 的函数及其用法

ntask 与 ctask 不同的是，它需要在任务函数中开头结尾加入 XF_NTASK_BEGIN(task) 和 XF_NTASK_END()。 此外，ntask 支持嵌套：

```c
#include "xf_task.h"
#include <stdio.h>

xf_async_t test(xf_task_t task, int a)
{
    XF_NTASK_BEGIN(task);

    xf_ntask_delay(1000);
    printf("hello ntask\n");

    XF_NTASK_END();
}

```

```c
static void task(xf_task_t task)
{
    XF_NTASK_BEGIN(task);

    while (1) {
        xf_await(test(task, 1));
    }

    XF_NTASK_END();
}

void app_main()
{
    // 创建任务
    xf_ntask_create(task, (void *)1, 1);
}
```

如上述例程，如果希望嵌套的函数调用 ctask 专属函数，需要这样才能使用。

API 展示

任意任务管理器的函数操作

- **创建无栈协程任务：**

```c
xf_task_t xf_ntask_create_with_manager(xf_task_manager_t manager,
                                       xf_task_func_t func,
                                       void *func_arg,
                                       uint16_t priority);
```

默认任务管理器的函数操作

- **创建无栈协程任务：**

```c
xf_task_t xf_ntask_create(xf_task_func_t func, void *func_arg, uint16_t priority);
```

无栈协程变量池

由于无栈协程无法保留临时变量，除了使用静态和全局变量之外。ntask提供一个由键值对设置的变量池机制。 用于在 ntask 里面保存变量。

- **变量池中获取整数：**

```c
int xf_ntask_args_get_int(xf_task_t *task, const char *name);
```

- **变量池中获取浮点数：**

```c
float xf_ntask_args_get_float(xf_task_t *task, const char *name);
```

- **变量池中获取数组：**

```c
void *xf_ntask_args_get_array(xf_task_t *task, const char *name);
```

- **变量池中保存整数：**

```c
xf_err_t xf_ntask_args_set_int(xf_task_t *task, const char *name, int value);
```

- **变量池中保存浮点数：**

```c
xf_err_t xf_ntask_args_set_float(xf_task_t *task, const char *name, float value);
```

- **变量池中保存数组：**

```c
xf_err_t xf_ntask_args_set_array(xf_task_t *task, const char *name, void *value, uns
```

无栈协程专用宏函数

很多 API 实际上是通过宏进行封装，不建议自己进行调用。我们可以通过以下宏进行调用。

- **无栈协程开始宏：**

```c
XF_NTASK_BEGIN(ntask)
```

- **无栈协程结束宏：**

```c
XF_NTASK_END()
```

- **暂时跳出协程：**

```c
xf_ntask_yield()
```

- **阻塞直到条件达成：**

```c
xf_ntask_until(compare_cb)
```

- **ntask 专属延时：**

```c
xf_ntask_delay(delay_ms)
```

- **阻塞直到条件达成或超时：**

```c
xf_ntask_until_timeout(compare_cb, timeout_ms)
```

- **结束并退出无栈协程：**

```c
xf_ntask_exit()
```

- **嵌套调用无栈协程：**

```c
xf_await(func)
```

xf_task 配置项详解

基本规则

文件夹结构规则

xf_task 的源码文件夹中，拥有四个子文件夹。其最外面有个 xf_task_config_internal.h 文件，在子文件夹也有对应的xxx_config.h，移植者需要自己写一个 xf_task_config.h 的配置文件，其依赖关系如下：

> 子文件夹源码 → 子文件夹的 xxx_config.h → xf_task_config_internal.h → xf_task_config.h

子文件夹的 xxx_config.h 中包含了当前子文件夹所需要的宏的默认值。如果有不知道的配置，可以自行去这些文件中找到默认值，一般会有注释。

xf_task_config_internal.h 一方面方便集中引入用户的配置文件 xf_task_config.h。另一方面，如果有跨不同子文件夹的公共配置，默认值将会被放到这里。

如果你希望所有配置都是默认值，则创建一个空的 xf_task_config.h 即可。配置的项目会覆盖掉默认值。

基础配置宏规则

基础配置宏

```c
// 相关功能注释
#ifndef XF_BUFEER_SIZE
#define XF_BUFEER_SIZE  (10)
#endif
```

- 一般的配置，我们会在 xxx_config.h 文件中配置一个默认值。以便在用户不配置该选项的情况下，代码能运行。
- 在其上方也会有解释型注释。方便用户的查询。

开关类配置宏

```c
// 希望默认值为1
// 外部配置使用XF_ADC_ENABLE，内部调用使用XF_ADC_EN
#if !defined(XF_ADC_ENABLE) || (XF_ADC_ENABLE)
#define XF_ADC_IS_ENABLE (1)    /* 默认配置 */
#else
#define XF_ADC_IS_ENABLE (0)
#endif


// 希望默认值为0
// 外部配置使用XF_ADC_ENABLE，内部调用使用XF_ADC_EN
#if !defined(XF_ADC_DISABLE) || (XF_ADC_DISABLE)
#define XF_ADC_IS_ENABLE (0)    /* 默认配置 */
#else
#define XF_ADC_IS_ENABLE (1)
#endif
```

- 开关宏，我们通常会隔离外部的配置。内部使用 XXX_IS_ENABLE 宏进行判断，外部使用 XXX_ENBALE 或 XXX_DISABLE 进行配置。
- 当默认值为 0 的时候，对外使用的是 XXX_DISABLE。默认值为 1 的时候，对外使用的是 XXX_ENBALE。用户可以很明显的了解到内部的默认配置。

数据类型类配置宏

```c
// 外界配置 XF_HEAP_INTPTR_TYPE宏
#ifndef XF_HEAP_INTPTR_TYPE
#define XF_HEAP_INTPTR_TYPE intptr_t
#endif

// 内部 typedef 可以帮忙检查宏是不是类型
// 同时在书写中 typedef 修饰的类型在代码提示中会被识别成类型而不是宏
typedef XF_HEAP_INTPTR_TYPE xf_heap_intptr_t;
```

- 数据类型的配置一般会形如 XXX_TYPE 。

数据限制宏

```c
// 限定用户配置在 0-10 之间
#ifndef XF_TASK_LEVEL
```

```c
#define XF_TASK_LEVEL (5)
#else
#if (XF_TASK_LEVEL < 0) || (XF_TASK_LEVEL > 10)
#error "task must be 0-10"
#endif
#endif
```

- 当配置的为具体数值时，如果希望用户的配置是一个范围，则会按照如上方式进行限制

选项型配置

```c

// XF_LOG_NEWLINE_LF 优先级大于XF_LOG_NEWLINE_CRLF 大于 XF_LOG_NEWLINE_NONE
// 都没定义则是"\n"
#ifdef XF_LOG_NEWLINE
#error "user cant define this macro"
#endif
#if defined(XF_LOG_NEWLINE_LF)
#define XF_LOG_NEWLINE "\n"
#elif defined(XF_LOG_NEWLINE_CRLF)
#define XF_LOG_NEWLINE "\r\n"
#elif defined(XF_LOG_NEWLINE_NONE)
#define XF_LOG_NEWLINE ""
#else
#define XF_LOG_NEWLINE "\n"  // 什么都没定义的情况下默认配置
#endif
```

- 选项类型的宏，如上述例子，用户可以通过定义，XF_LOG_NEWLINE_LF 或者 XF_LOG_NEWLINE_CRLF 或者 XF_LOG_NEWLINE_NONE 其中一个（多定义按照优先级来确定）。从而定义 XF_LOG_NEWLINE 的内容
- 用户不能直接定义 XF_LOG_NEWLINE 。防止用户定义奇怪的内容。

配置项

kernel 配置

> 配置宏：`XF_TASK_PRIORITY_LEVELS`
>
> 功能：配置任务最高优先级，1 ~ 1024 之间

默认值：24

配置宏：`XF_TASK_HUNGER_ENABLE`

功能：是否启用任务饥饿机制

默认值：1

配置宏：`XF_TASK_USER_DATA_ENABLE`

功能：是否启用 user_data

默认值：1

port 配置

配置宏：`XF_TASK_TICKS_FREQUENCY`

功能：对接定时器的滴答频率，单位为 Hz

默认值：1000

配置宏：`XF_TASK_CONTEXT_TYPE`

功能：设置上下文类型

默认值：无默认值，开启了 ctask 必须配置该选项

utils 配置

配置宏：`XF_TASK_MBUS_ENABLE`

功能：是否打开 MBUS 功能

默认值：1

配置宏：`XF_TASK_POOL_ENABLE`

功能：是否打开任务池功能

默认值：1

内部公共配置

> 配置宏：`XF_TASK_CONTEXT_DISABLE`
>
> 功能：是否打开 ctask
>
> 默认值：0

> 配置宏：`XF_TASK_TIME_TYPE`
>
> 功能：时间戳类型
>
> 默认值：uint64_t

优先级机制

与 RTOS 的优先级不同，由于 xf_task 是协作式调度。故不存在 RTOS 中的抢占优先级。xf_task 优先级机制是一种协作式优先级，每次进入调度的时候，优先级提供给调度器一个选择哪个就绪任务运行的依据。

> stm32 中有个功能叫优先级分组。其中有两种优先级，一个叫抢占优先级（Preemptive Priority）一个叫子优先级（Sub Priority）。抢占优先级类似 RTOS 里面的优先级，主要的特点是高优先级的会打断低优先级。而子优先级就类似我们说的优先级，只决定同时触发时谁先执行。

- 优先级决定了当不同任务都处于就绪状态时，谁先运行
- 即使优先级再高也要等到当前任务主动让出CPU才能运行
- 优先级的数字越小优先级越高
- 优先级可以通过配置项进行配置
- 优先级可以由用户在运行时动态改变
- 同优先级的任务遵循先入先出的原则

消息队列机制

虽然 xf_task 是协作式调度，这就意味着他可以随便在不同任务间使用全局变量和静态变量，但是消息队列有时候会很好用。于是，我们的 xf_task 支持了两种消息队列机制。

普通消息队列

普通的消息队列在任意的任务都能使用。这部分的代码与核心是解耦的。 所以可以单独拿出来使用。

API 展示

- **消息队列初始化：**

```c
xf_err_t xf_task_queue_init(xf_task_queue_t *const queue, void *data, const size_t s
```

- **重置消息队列：**

```c
xf_err_t xf_task_queue_reset(xf_task_queue_t *const queue);
```

- **判断是否为空：**

```c
bool xf_task_queue_is_empty(const xf_task_queue_t *const queue);
```

- **获取数据个数：**

```c
size_t xf_task_queue_count(const xf_task_queue_t *const queue);
```

- **偷看第一个元素：**

```c
void *xf_task_queue_peek(const xf_task_queue_t *const queue);
```

- **获取剩余空间：**

```c
size_t xf_task_queue_available(const xf_task_queue_t *const queue);
```

- **发送数据：**

```c
xf_err_t xf_task_queue_send(xf_task_queue_t *const queue, void *item, const xf_task_
```

- **删除第一个元素：**

```c
xf_err_t xf_task_queue_remove_front(xf_task_queue_t *const queue);
```

- **接收一个元素：**

```c
xf_err_t xf_task_queue_receive(xf_task_queue_t *const queue, void *const buffer);
```

ctask 消息队列

ctask 的消息队列，由于 ctask 可以保存上下文。所以在发送和接受又会有 timeout。 阻塞期间不会耽误别的任务执行。 其 API 参考：[ctask 的函数及其用法]

时间触发和事件触发

xf task 基本上是可以基于时间触发和事件触发。

时间触发

时间触发在不同的任务中的表现不同：

- ctask：ctask中 的 delay 就经典的时间触发，当然还包括 ctask 版本的消息队列的超时部分
- ntask：ntask 中的 delay 也是时间触发。
- ttask：ttask 的定时器周期是时间触发。

事件触发

相比于时间触发，事件触发则有一个明显的函数：

```c
xf_err_t xf_task_trigger(xf_task_t task);
```

该函数是异步的，调用后不会立即执行该任务，等当前任务主动释放 CPU 的时候才会被触发，触发后任务从阻塞态进入就绪态。 用户可以通过该函数触发任何任务，除此之外还有些隐藏的调用：

- ctask：ctask 中消息队列接收到消息后会通过事件触发被阻塞的任务
- ntask: ntask 中 xf_ntask_yield() 会短暂让出 CPU。

另外，ttask 如果主动调用 xf_task_trigger() 函数，不会消耗其循环次数。

紧急任务机制

紧急任务不是一个新的任务。是由原先的任务通过函数：

```c
xf_err_t xf_task_set_urgent_task_with_manager(xf_task_manager_t manager, xf_task_t t
// or
xf_err_t xf_task_set_urgent_task(xf_task_t task, bool force);
```

- 将已有的任务设置为紧急任务，该紧急任务会在下次调度的时候被优先执行(非优先级)。
- 紧急任务运行完毕后，紧急状态则会被解除。
- 当有多个任务被设置为紧急任务时，如果启用 force ，则会使用最设置任务为紧急任务。如果不启用 force，则会返回 XF_ERR_BUSY 告诉设置失败。

任务饥饿与饥饿度

任务饥饿

由于协作式任务的优先级不能进行抢占。所以，从理想的角度，每个任务不能占用过长的时间。这会导致很多进入就绪任务无法进行执行，以 ntask 为例子。遇到某种算法占用过长时间，我们需要利用：

```c
xf_ntask_yield()
```

短暂让出 CPU。 即使这样，也会出现低优先级任务就绪中迟迟无法执行。全都是高优先级任务在执行。 这种情况下，我们称低优先级任务为“饥饿”状态。

饥饿值机制

当任务开启 XF_TASK_HUNGER_ENABLE 时，任务可以使用：

```c
void xf_task_feel_hungery_enable(xf_task_t task, uint32_t hunger_time);
```

函数开启饥饿值机制。当任务的超时时间大于指定的饥饿时间，则饥饿的任务会自动上升一个临时优先级直至任务得以执行。该机制主要保证任务不会永远出于饥饿状态。 如果取消紧急任务可以使用如下函数：

```c
void xf_task_feel_hungery_disable(xf_task_t task);
```

同步/异步的发布订阅机制

虽然 xf_task 是协作式调度，这就意味着他可以随便在不同任务间使用全局变量和静态变量，但是发布订阅机制有时候确实很方便。于是，我们的 xf_task 支持了同步和异步两种发布订阅机制。

同步发布订阅

```c
#define TOPIC_ID 1 // 随便给一个id号

void bus_cb(const void *const data, void *user_data)
{
    int num = *(int *)data;
    printf("data:%d\n", num);
}

void task_pub(xf_task_t task)
{
    static int num = 1;

```

```c
    num++;
    // 同步发送（快，但是会阻塞本任务）
    xf_task_mbus_pub_sync(TOPIC_ID, &num);
}

void app_main()
{
    // 注册 topic
    xf_task_mbus_reg_topic(TOPIC_ID, sizeof(int));
    // 订阅 topic
    xf_task_mbus_sub(TOPIC_ID, bus_cb, NULL);
    // 创建一个任务，定时发布消息
    xf_ttask_create_loop(task_pub, (void *)1, 1, 2000);
}
```

同步发布订阅是在消息发布的时候立即去执行订阅者这边的回调。 它的优势就是处理速度会很及时，但是如果订阅者过多，或者处理时间过长。此时就会阻塞当前任务的执行。

异步发布订阅

```c
#define TOPIC_ID 1 // 随便给一个id号

void bus_cb(const void *const data, void *user_data)
{
    int num = *(int *)data;
    printf("data:%d\n", num);
}

void task_pub(xf_task_t task)
{
    static int num = 1;
    num++;
    // 异步发送（慢，但是不会阻塞本任务）
    xf_task_mbus_pub_async(TOPIC_ID, &num);
}

void app_main()
{
    // 注册 topic
    xf_task_mbus_reg_topic(TOPIC_ID, sizeof(int));
```

```c
    // 订阅 topic
    xf_task_mbus_sub(TOPIC_ID, bus_cb, NULL);
    // 创建一个任务，定时发布消息
    xf_ttask_create_loop(task_pub, (void *)1, 1, 2000);
}
```

异步发布订阅在发送的时候并不会直接进行触发，而是将发送的消息保存起来后触发内部的 ttask 任务。 然后，等待当前任务执行完。则进行异步回调的执行。

> 注意：目前的发布订阅机制仅服务于默认的任务管理器。自定义的任务管理器尚且不支持

API 展示

- **注册 topic：**

```c
xf_err_t xf_task_mbus_reg_topic(uint32_t topic_id, uint32_t size);
```

- **注销 topic：**

```c
xf_err_t xf_task_mbus_unreg_topic(uint32_t topic_id);
```

- **异步发布消息至 topic：**

```c
xf_err_t xf_task_mbus_pub_async(uint32_t topic_id, void *data);
```

- **同步发布消息至 topic：**

```c
xf_err_t xf_task_mbus_pub_sync(uint32_t topic_id, void *data);
```

- **订阅指定 topic：**

```c
xf_err_t xf_task_mbus_sub(uint32_t topic_id, xf_task_mbus_func_t mbus_cb, void *user
```

- **取消订阅指定 topic：**

```c
xf_err_t xf_task_mbus_unsub(uint32_t topic_id, xf_task_mbus_func_t mbus_cb);
```

任务池和任务回收机制

当我们大量且多次的创建和删除任务，有可能会导致内存的碎片化。所以，我们对现有的 xf_task 设计了任务池机制。

```c

xf_task_pool_t tpool = NULL;

static void ttask(xf_task_t task)
{
    static int num = 0;
    num++;
    // 不断的创建任务（由于任务池的存在，所以不会申请内存，反而会回收已经使用完的任务
    printf("ttask:%d\n", num);
    if (num < 5) {
        xf_task_init_from_pool(tpool, ttask, (void *)1, 1);
    } else { // 创建五次后删除任务池
        xf_err_t err = xf_task_pool_delete(tpool);
        if (err == XF_OK)
        {
            printf("delete tpool\n");
        }
    }
}

void app_main()
{
    tpool = xf_ttask_pool_create(MAX_WORKERS, 1000, 1);

    // 通过任务池创建 ttask 任务
    xf_task_init_from_pool(tpool, ttask, (void *)1, 1);
}
```

任务池有以下功能：

- 任务池可以创建 ctask、ntask 和 ttask 任务池。
- 任务池创建好后，可以分别从不同的任务池里初始化任务
- 当任务删除时，任务进入回收
- 任务池中没有任务则会返回 NULL

API 展示

默认任务管理器创建任务池

- **创建 ttask 任务池：**

```c
xf_task_pool_t xf_ttask_pool_create(uint32_t max_works, uint32_t delay_ms, uint32_t
```

- **创建 ntask 任务池：**

```c
xf_task_pool_t xf_ntask_pool_create(uint32_t max_works);
```

- **创建 ctask 任务池：**

```c
xf_task_pool_t xf_ctask_pool_create(uint32_t max_works, size_t stack_size);
```

任意任务管理器创建任务池

- **创建 ttask 任务池：**

```c
xf_task_pool_t xf_ttask_pool_create_with_manager(uint32_t max_works, xf_task_manager
```

- **创建 ntask 任务池：**

```c
xf_task_pool_t xf_ntask_pool_create_with_manager(uint32_t max_works, xf_task_manager
```

- **创建 ctask 任务池：**

```c
xf_task_pool_t xf_ctask_pool_create_with_manager(uint32_t max_works, xf_task_manager
```

任务池操作函数

- **删除任务池：**

```c
xf_err_t xf_task_pool_delete(xf_task_pool_t pool);
```

- **从任务池中初始化任务：**

```c
xf_task_t xf_task_init_from_pool(xf_task_pool_t pool, xf_task_func_t func, void *fur
```

裸机和多线程中移植

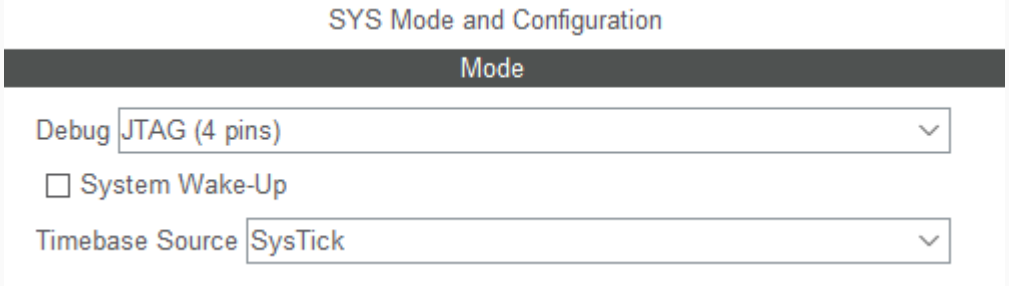
在 XFusion 中，我们可以随意使用 xf_task 。 另外，xf_task 也支持独立移植。接下来，介绍如何在裸机和多线程中独立的移植 xf_task

---

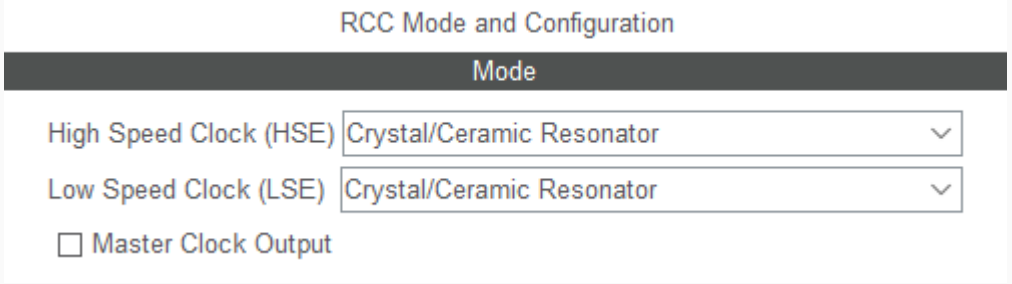
手把手教你裸机移植（stm32）

1. 提供一个基础移植工程

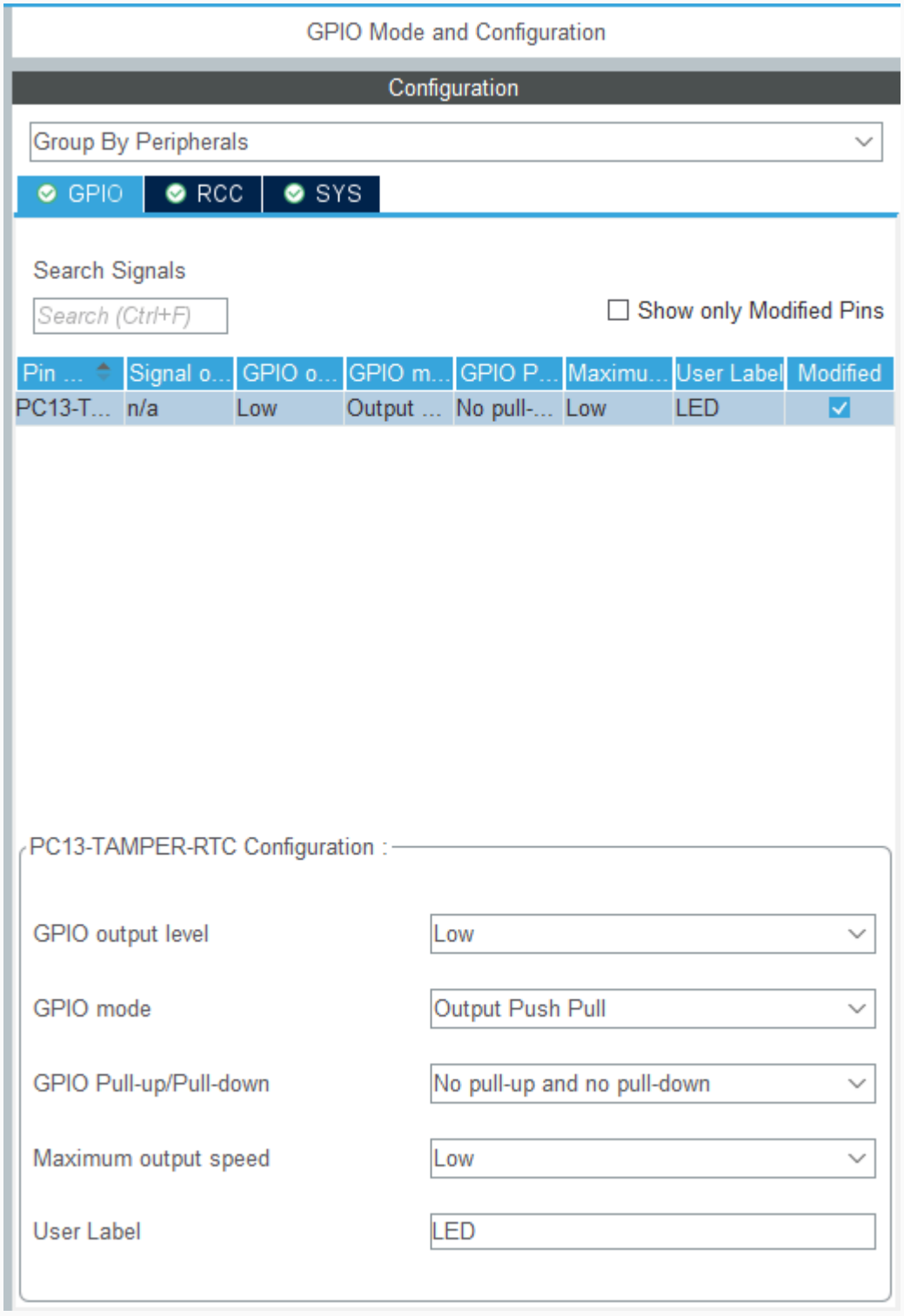
我们利用 STM32CUBEMX 配置一个 HAL 库工程。



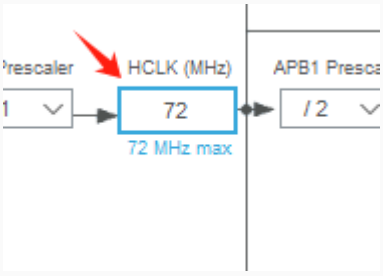
这里配置 debug （根据自己手上调试器不同自行更改）



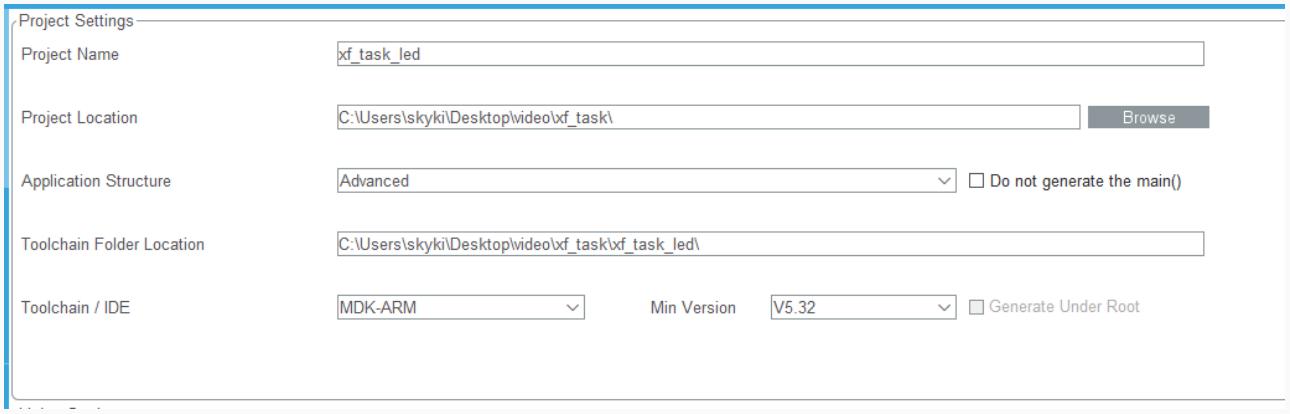
这里配置时钟为外部时钟



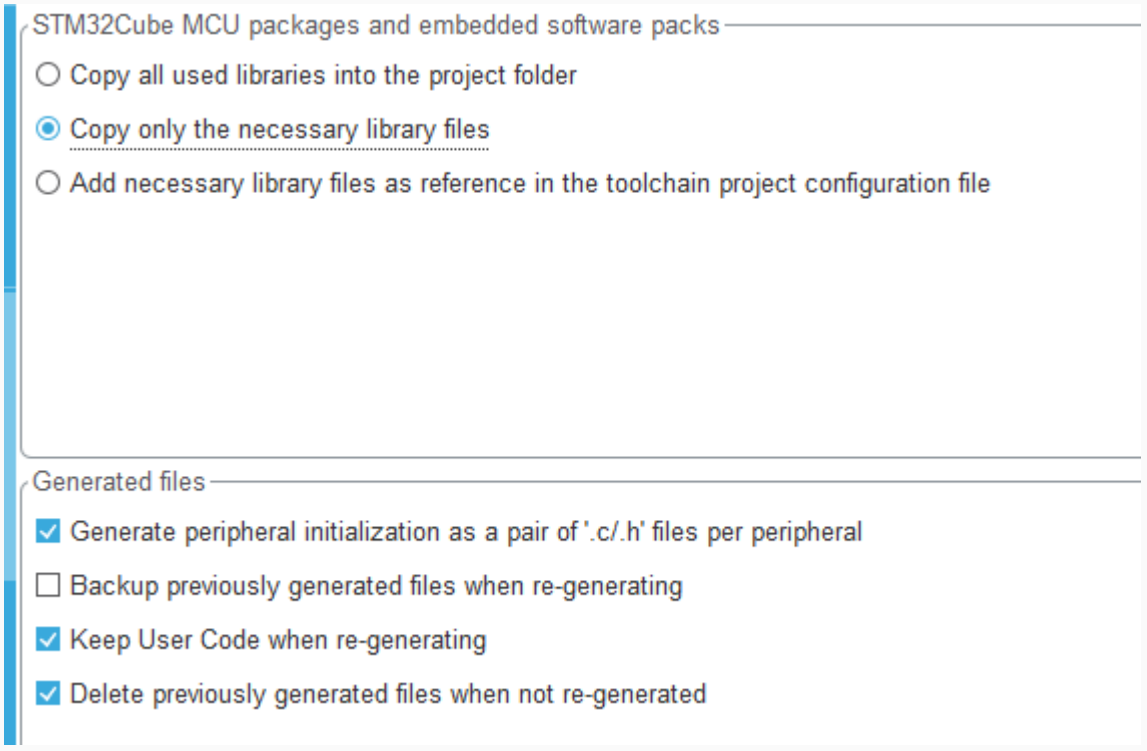
配置 PC13 为推挽输出，方便后续用它做 led 闪烁验证移植完成



配置 cpu 主频为 72 MHz 。



设置导出工程，这里导出 mdk 工程。



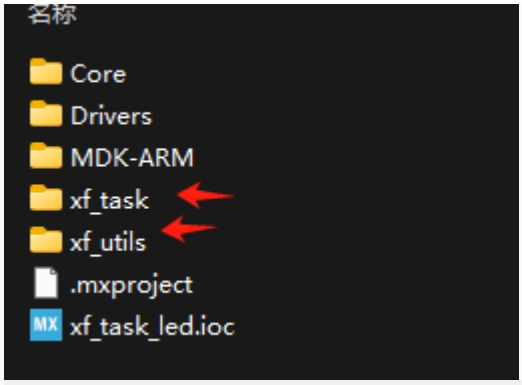
让生成的工程更加简洁。

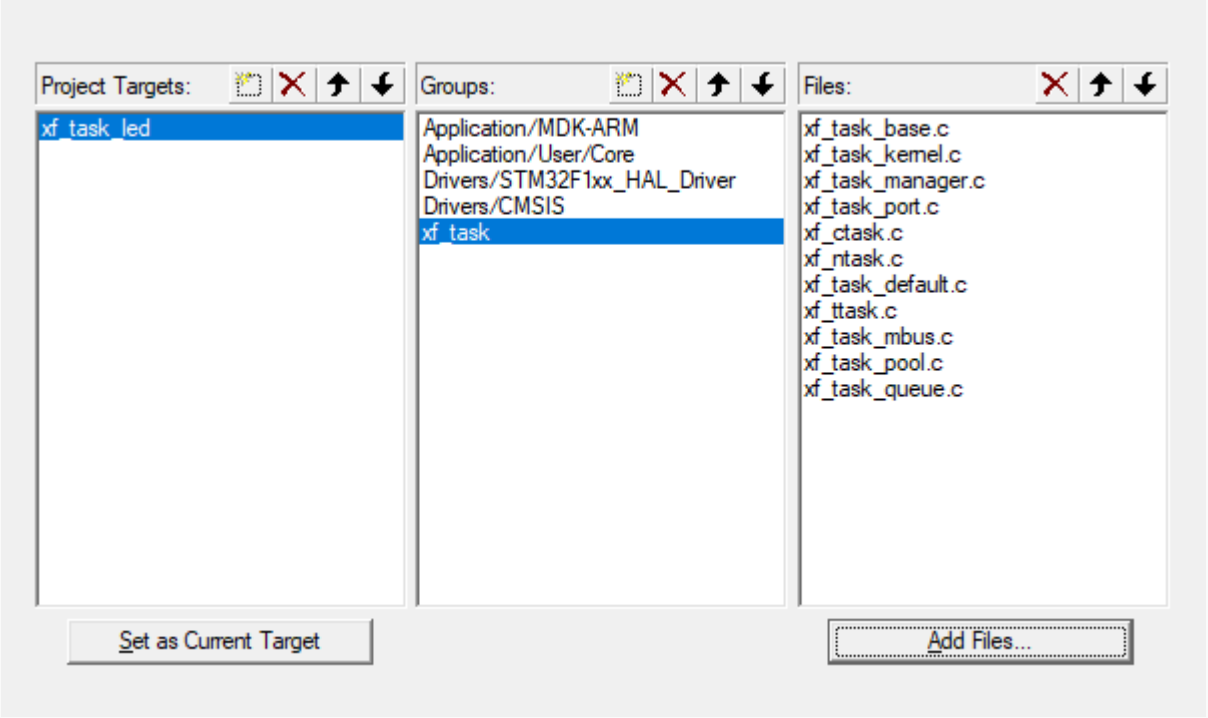
编译后，无报错，无警告，表示成功。

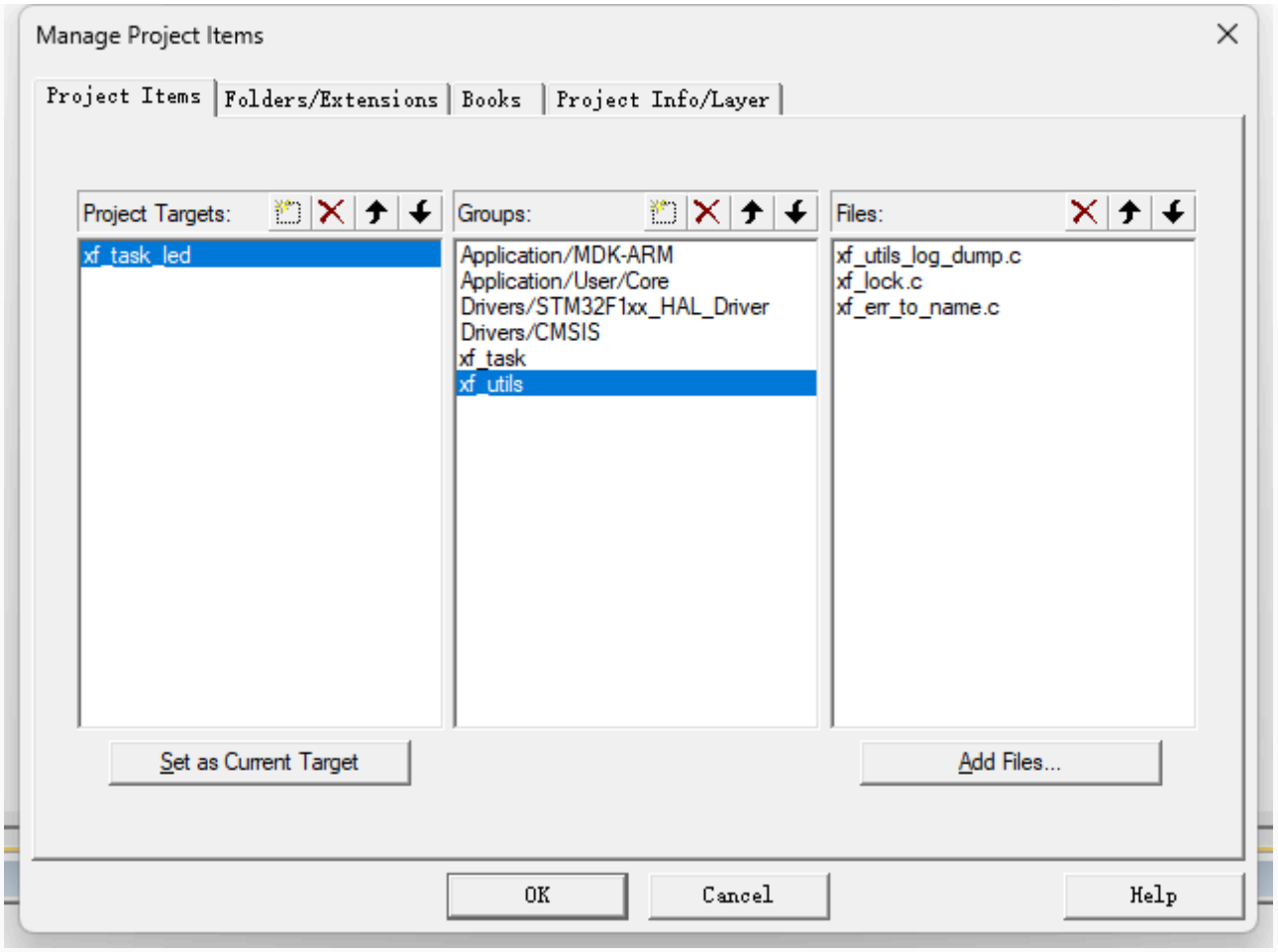
2. 将 xf_task 加入工程

找到你导出的工程，进入工程目录。利用 git clone 下载 xf_task 和 xf_utils

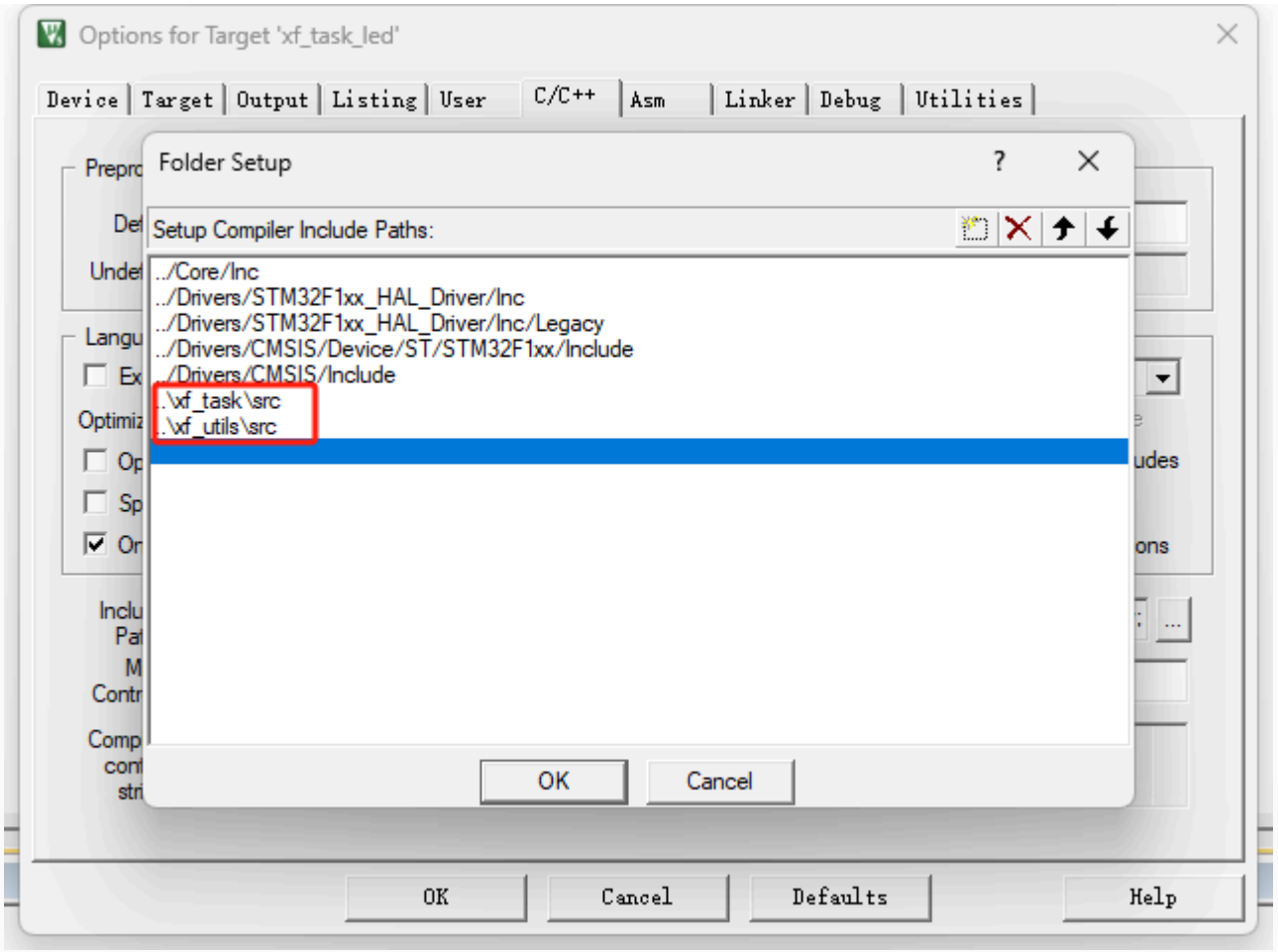
```shell
git clone https://github.com/x-eks-fusion/xf_task.git
git clone https://github.com/x-eks-fusion/xf_utils.git
```


名称
Core
Drivers
MDK-ARM
xf_task
xf_utils
.mxproject
xf_task_led.ioc

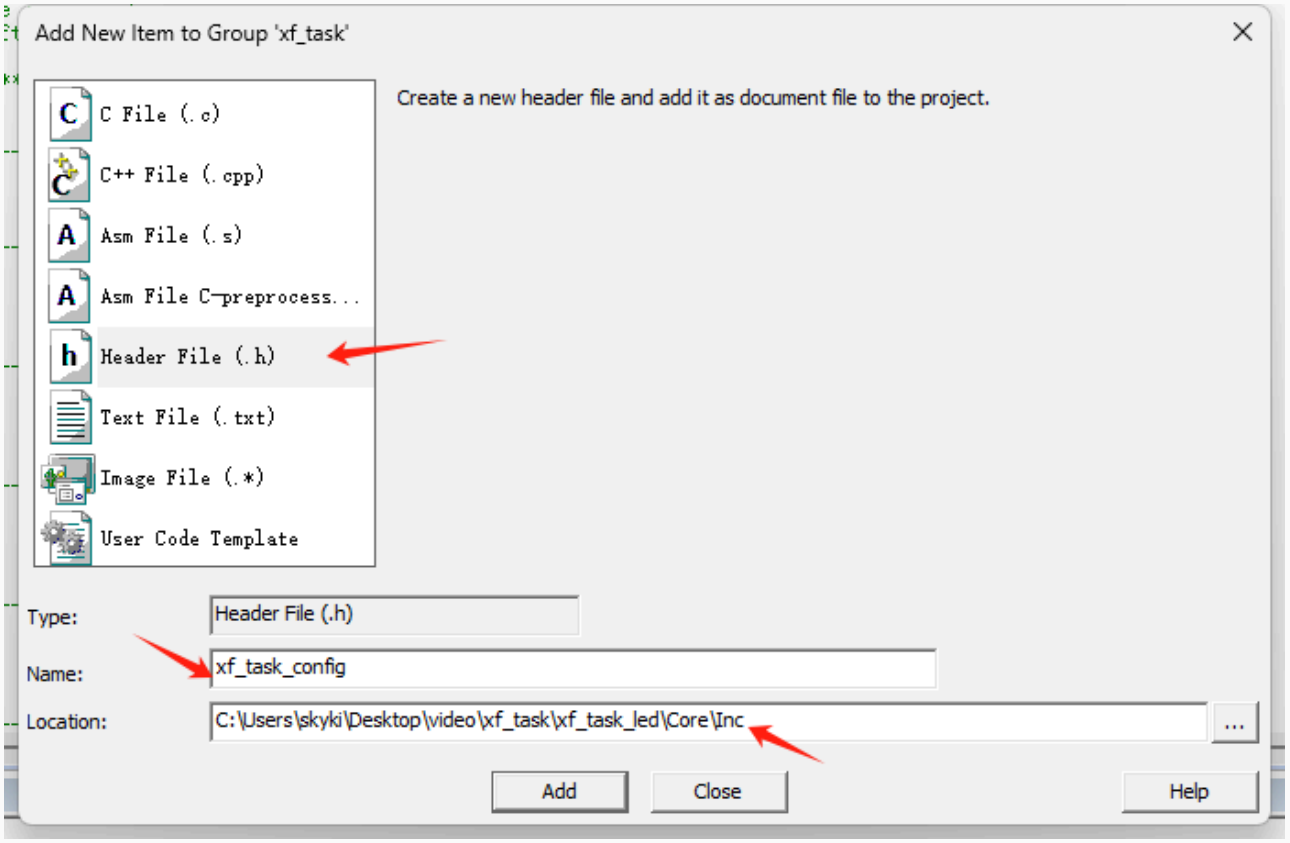

Project Targets:
xf_task_led
Groups:
Application/MDK-ARM
Application/User/Core
Drivers/STM32F1xx_HAL_Driver
Drivers/CMSIS
xf_task
Files:
xf_task_base.c
xf_task_kernel.c
xf_task_manager.c
xf_task_port.c
xf_ctask.c
xf_ntask.c
xf_task_default.c
xf_ttask.c
xf_task_mbus.c
xf_task_pool.c
xf_task_queue.c
Set as Current Target
Add Files...

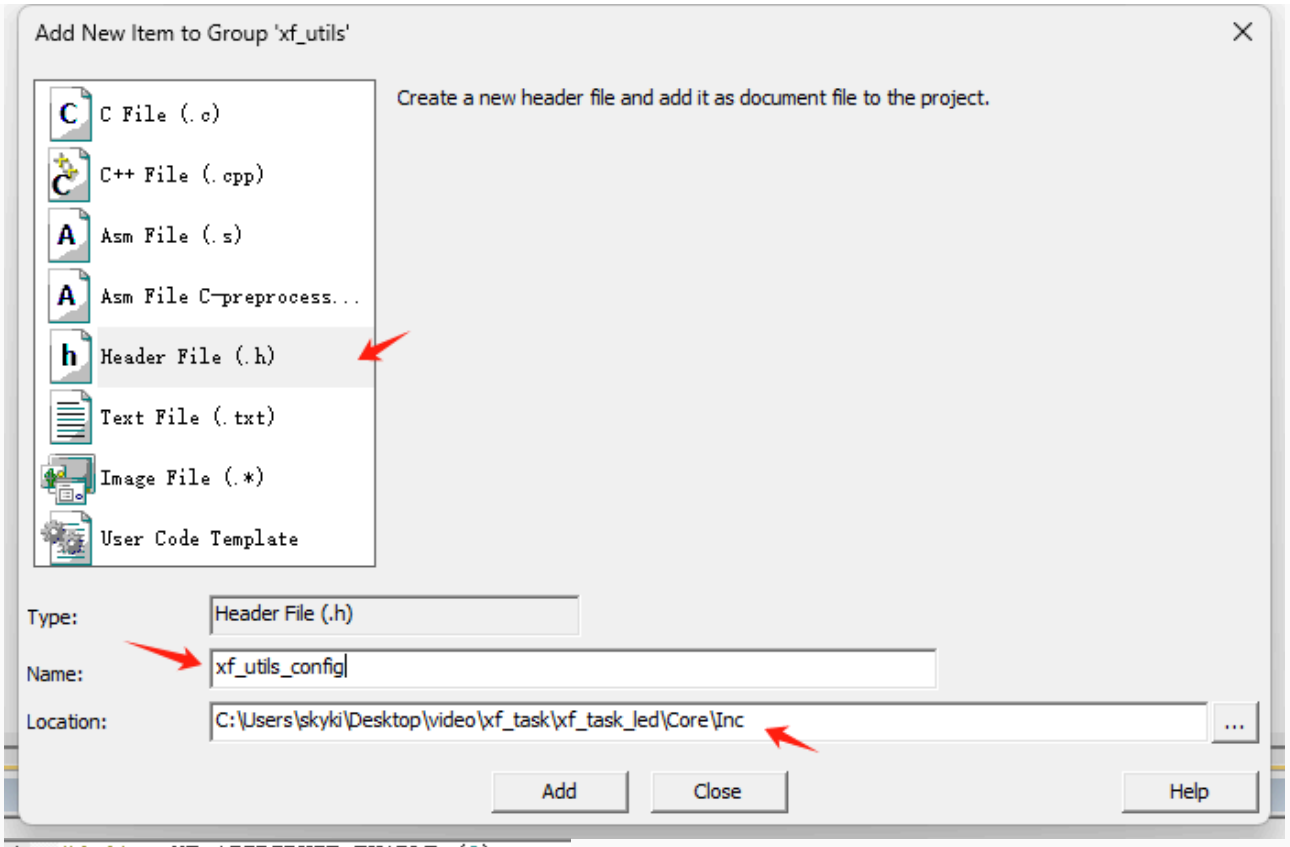


添加 xf_task 到工程里面去



添加 src 到 include path 中






```
#define XF_ATTRIBUTE_ENABLE (0)
#define XF_LOG_LEVEL (XF_LOG_NONE)
```

添加用户配置项, xf_task 配置项可以保持默认。xf_utils 的配置项关闭 XF_ATTRIBUTE_ENABLE （改功能需要 gnu 且 xf_task 中没用到）和 XF_LOG_LEVEL （减少占用）。

3. 添加移植代码

```
/* USER CODE BEGIN Includes */

#include "xf_task.h"

/* USER CODE END Includes */
```

首先，添加头文件。

```
/* USER CODE BEGIN 0 */

static xf_task_time_t task_get_tick(void)
{
  return HAL_GetTick();
}
```

然后，添加对接 ms 级时间戳的函数。

```
/* USER CODE BEGIN SysInit */

xf_task_tick_init(task_get_tick);
xf_task_manager_default_init(NULL);

/* USER CODE END SysInit */
```

调用函数对接时间戳并初始化默认的任务管理器。

```
while (1)
{
  xf_task_manager_run_default();
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
```

调用默认管理器进行调度

4. 添加 ttask 任务翻转 LED

```
static void task(xf_task_t task)
{
    HAL_GPIO_TogglePin(LED_GPIO_Port, LED_Pin);
}

/* USER CODE END 0 */
```

写一个任务用于循环翻转 LED

```
/* USER CODE BEGIN 2 */
xf_ttask_create_loop(task, NULL, 1, 1000);
/* USER CODE END 2 */
```

在 main 中创建一个 ttask

多线程移植

在多线程中，不同的线程可以运行不同的任务管理器。需要注意如下几个点：

- 每个线程都可以创建属于自己的协程任务管理器
- 不同任务管理器之间需要通过协程间通讯
- 目前发布订阅只能在默认的调度器中使用 [同步异步的发布订阅机制]
- 除了默认任务管理器，其余创建则需要 xxx_with_manager 版本的函数

最后更新于: 2025/2/8 19:14:26

Previous page
xf build 构建脚本

Next page
移植指南

XFusion 构建流程

作者

kirto

本文简要说明 `XFusion` 的构建流程。

---

export 阶段

构建之初会使用 `export` 脚本激活 `xfuison`

windows cmd:

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

windows powershell:

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

linux:

```shell
. ./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。 当关闭当前终端则环境变量消失。

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，通过 `pip` 安装 `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。

`requires` 参数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。

`cflag` 则是涉及到的编译标志位参数收集。

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。
        """
```

最后更新于: 2024/12/2 20:23:23

Previous page
XFusion 目录结构

Next page
xf build 构建脚本

XFusion 目录结构

作者

dotc

本文简要说明 XFusion 的目录结构。

---

xfusion/ : 目录结构

```bash
xfusion/
 ├ .vscode/
 ├ boards/ ........................ 各平台原生工程及与 XF 构建相关的对接
 ├ build/ .........................
 ├ components/ ..................... XF 内部组件
 ├ docs/ ........................... XF 文档
 ├ examples/ ....................... XF 例程
 ├ plugins/ ........................ 平台 SDK 与 XF 构建相关的对接
 ├ ports/ .......................... 平台 SDK 与 XF 功能相关的对接
 ├ sdks/ ........................... 各平台 sdk及工具
 ├ tools/ .......................... XF 相关的工具
 ├ .editorconfig
 ├ .gitignore
 ├ .gitmodules
 ├ LICENSE ......................... XF 许可证
 ├ README.md
 ├ XFKconfig
 ├ export.bat ...................... XF 环境激活脚本 （win端）
 ├ export.ps1
```

```
├ export.sh ....................... XF 环境激活脚本 （linux端）
└ requirements.txt ................ XF 依赖描述
```

boards/： 各平台原生工程及与 XF 构建相关对接。

```Bash
boards/
├ espressif/ <──────── 厂商或平台
│  └ esp32/ <<═══════════ │ ═══ 编译目标的原生工程（包含与XF构建相关对接）
│    ├ ...                │              ‖
│    ├ target.json <──── │ ─────────────‖── 环境激活时，会在 boards 目录下被
│    └ ...                │              ‖
├ nearlink/ <──────────┘              ‖
│  ├ bs21/ <<═══════════════════════ ┤
│  └ ws63/ <<═══════════════════════ ┘
├ README.md
└ XFKconfig  ..................... 平台描述配置文件（自动生成）
```

components/： XF 内部组件。

```Bash
components/
├ xf_fal/ ................ XF FAL （ Flash 抽象层）
│  └ ...
├ xf_hal/ ................ XF HAL （硬件抽象层）
│  └ ...
├ xf_heap/ ............... XF Heap （堆内存管理）
│  └ ...
├ xf_init/ ............... XF Init （初始化管理）
│  └ ...
├ xf_log/ ................ XF Log （日志）
│  └ ...
├ xf_nal/ ................ XF NAL (网络抽象层)
│  └ ...
├ xf_net_apps/ ........... XF Net APP (网络相关应用)
│  └ ...
├ xf_osal/ ............... XF OSAL (操作系统抽象层)
│  └ ...
├ xf_sys/ ................ XF SYS (系统功能)
│  └ ...
```

```
├ xf_task/ ............... XF Task (XF 协作式调度任务)
│  └ ...
├ xf_utils/ ............... XF Utils (XF 通用功能 (工具) 集)
│  └ ...
└ xf_wal/ ................ XF WAL (无线功能抽象层)
   ├ xf_ble/ ............... XF BLE (BLE 功能)
   ├ xf_sle/ ............... XF SLE (SLE 功能)
   ├ xf_wifi/ ............... XF WIFI (WiFi 功能)
   └ ...
```

docs/： XF 文档。

examples/： XF 例程。

```bash
examples/
├ example_components/ ............. 组件例程
│  ├ ex_easy_wifi/
│  └ README.md
├ get_started/ ................... 快速开始例程
│  ├ template_project/
│  └ xf_template/
├ osal/ ........................... OSAL 例程
│  ├ event/
│  ├ kernel/
│  ├ mutex/
│  ├ notify/
│  ├ queue/
│  ├ semaphore/
│  ├ thread/
│  └ timer/
├ peripherals/ ................... 外设例程
│  ├ adc/
│  ├ dac/
│  ├ gpio/
│  ├ i2c/
│  ├ pwm/
│  ├ spi/
│  ├ timer/
│  └ uart/
```

```
├ protocols/ ..................... 协议例程
│  ├ http_request/
│  ├ icmp_echo/
│  ├ iperf/
│  └ sockets/
├ system/ ........................ 系统功能例程
│  ├ heap/
│  ├ init/
│  ├ log/
│  └ sys/
├ task/ .......................... 协作式调度任务（XF Task）例程
│  ├ mbus/
│  ├ ntask/
│  ├ ntask2/
│  ├ task_pool/
│  └ trigger/
└ wireless/ ...................... 无线功能例程
   ├ ble/
   ├ sle/
   └ wifi/
```

ports/：平台 SDK 与 XF 功能相关的对接。

```Bash
ports/
├ espressif/ <──────────── 厂商或平台
│  └ esp32/ <<═══════════════ │ ═══ 编译的目标与 XF 功能相关的对接（此处目录名
└ nearlink/ <──────────────┘              ‖
   └ ws63/ <<═══════════════════════════════╝
```

sdks/：各平台 sdk 及工具

tools/：XF 相关的工具。

```Bash
tools/
├ export_script/ ..................... XF 环境激活相关脚本
│  ├ README.md
│  ├ check_virtualenv.py ............... XF 环境激活相关脚本
```

```
6    │  ├ gen_kconfig.py
7    │  └ get_path.py
8    └ format_code/ ....................... 代码格式化相关
       └ ...
```

最后更新于: 2024/12/2 20:04:27

Previous page
深入了解

Next page
XFusion 构建流程

移植指南

作者

dotc

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等的支持。

目录

- 平台工程移植
- 构建对接
- 基础功能对接
- 外设对接
- 其他对接

最后更新于: 2024/12/2 20:04:27

基础功能对接（XF_SYS）

作者

dotc

本章节介绍如何对接 XFusion 基础功能部分

前置准备：

- 了解 XFusion 的构建命令及其参数的作用。
- 了解 XFusion 基础功能的作用及 XFusion 的例程。
- 熟悉将要对接的 XFusion 基础功能的原理、将对接的平台的处理流程等。
- 了解 XFusion 基础功能的对接要求。

目前需要对接的基础功能有

1. xf_init
2. XFusion 的调用
3. xf_log
4. xf_sys

对接流程

---

1. xf_init 对接：

> 目前 xf_init 的自动初始化已实现调用的方法统一，即用户只需调用同名的方法即可

- 目前 xf_init 的自动初始化实现方法有 3 种，仅需选一种方式进行对接，然后 menuconfig 配置成对应的方法即可：

1. **(section 属性)(GNU 特性)**：通过 `section` 将自动初始化的函数的符号导出指定的段，实现依赖倒置。
2. **(constructor 属性)(GNU 特性)**：将自动初始化的函数的符号，通过 `constructor` 挂载到内置初始化链表，实现在调用时初始化(延迟初始化)，同时也实现依赖倒置。
3. **(显式注册表)**：显式调用注册函数，此时需要手动修改注册表。此时 `xf_init` 也会依赖需要初始化的组件，通常不推荐使用。

section 方法的对接步骤

1. 需在平台侧的链接脚本中，找到 text 段
2. 在 text 段中加入 `xf_auto_init` 段，如下：

```c

 /* 省略 */

/* 在此插入 xf_auto_init 段 */
. = ALIGN(4);                    /* 32 位使用 4 字节对齐方式，64 位使用 8 字节对
/* .xf_auto_init* : 通配 .xf_auto_init* 的符号 ; SORT: 对符号排序; KEEP : 确保这
KEEP(*(SORT(.xf_auto_init*)))
. = ALIGN(4);                    /* 32 位使用 4 字节对齐方式，64 位使用 8 字节对

 /* 省略 */
```

constructor 方法的对接步骤

- 略，仅需平台侧编译器支持 constructor 属性方法即可

显式注册表 方法的对接步骤

- 略，不推荐使用

---

2. XFusion 的调用

- XFusion 需由平台侧工程调用才能正常运行，目前需要被调用的方法有 2 个：

1. **void xfusion_init(void)** : 初始化 XFusion ， 包含 log 初始化，自动初始化等。需要放在较早被调用的位置，且在 "xfusion_run" 的调用前。

2. **void xfusion_run(void)** : 运行 XFusion 。需要将该函数放到循环里面调用。

- 例 : main 函数中调用 "xfusion_init" 与 "xfusion_run" 方法。 (也可自行创建的线程进行调用)

```c

void main(void)
{
    xfusion_init();
    while (1)
    {
        xfusion_run();
    }
    return;
}
```

---

3. xf_log 对接:

- 目前 xf_log 仅对接一个方法:

1. ***int xf_log_register_obj(xf_log_out_t out_func, void *user_args)***

- 描述：注册 log 的后端 (log 最终输出到哪里)，其最大值受到 XF_LOG_OBJ_MAX 的限制。

- 参数说明:

  1. ***out_func*** : 后端输出函数， 如果减少 IO 操作，可以考虑使用异步缓冲

     - 类型 : typedef void(*xf_log_out_t)(const char *str, size_t len, void *arg);

  2. ***user_args*** : 传入的参数，会在 out_func 中被调用。

- 对接流程 :

> 如对接了 xf_sys 中的时间戳功能，则 xf_log 会正常输出时间戳；否则不会正常输出时间戳。

1. 实现 **xf_log_out_t** 类型的后端函数 (如 printf 、串口 等)。
2. 实现另一个函数 (假设为 : "port_log_init")，其实现为 : 通过 "**xf_log_register_obj**" 将实现后端函数注册至 xf_log 实现对接。
3. 通过 **XF_INIT_EXPORT_SETUP** 将 "port_log_init" 加入自动初始化列表 (注意：需包含初始化的头文件 "xf_init.h" ， 否则以上操作无效)。

- 例 "port_xf_log.c" 文件内容 :

```c

#include "xf_log.h"
#include "xf_init.h"
#include "stdio.h"

static void xf_log_out(const char *str, size_t len, void *arg)
{
    if ((NULL == str) || (0 == len)) {
        return;
    }
    print("%.*s", (int)len, str);
}

static int port_log_init(void)
{
    xf_log_register_obj(xf_log_out, NULL);
    return 0;
}
XF_INIT_EXPORT_SETUP(port_log_init);
```

- 验证 : 编译带有 xf_log 日志输出，且日志等级设置正常的工程，运行查看输出日志结果进行验证。

---

xf_sys 对接

- xf_sys 目前可对接的功能有：(看情况进行对接)

1. 系统时间 (xf_sys_time) (强烈建议对接)
2. 系统看门狗 (xf_sys_watchdog)
3. 系统重启
4. 中断开启与关闭

系统时间对接

- 目前只需要调用 "**xf_sys_time_init**" 来注册系统时间微妙级 (us)的时间戳获取的方法即可。
- 对接流程：
  1. 实现 "**xf_us_t (*get_us)(void)**" 类型的微妙级时间戳获取函数。
  2. 实现另一个函数 (假设为："port_sys_init")，其实现为：通过 "**xf_sys_time_init**" 将实现的微妙级时间戳获取函数注册至 xf_sys 实现对接。
  3. 通过 **XF_INIT_EXPORT_BOARD** 将 "port_sys_init" 加入自动初始化列表 (注意：需包含初始化的头文件 "xf_init.h"， 否则以上操作无效)。
- 例：

```c

#include "xf_sys.h"
#include "xf_init.h"

#include <sys/time.h>
#include <time.h>

static xf_us_t _port_xf_sys_get_us(void)
{
    struct timespec current_time;
    clock_gettime(CLOCK_MONOTONIC, &current_time);

    return current_time.tv_sec*(1000*1000) + current_time.tv_nsec/1000;
}

```

```c
static int port_sys_init()
{
    xf_sys_time_init(_port_xf_sys_get_us);
    return XF_OK;
}

XF_INIT_EXPORT_BOARD(port_sys_init);
```

系统看门狗对接

- 目前可对接的系统看门狗接口有:
  1. ***xf_err_t xf_sys_watchdog_enable(void)*** : 系统看门狗开启
  2. ***xf_err_t xf_sys_watchdog_disable(void)*** : 系统看门狗关闭
  3. ***xf_err_t xf_sys_watchdog_kick(void)*** : 系统看门狗喂狗操作

系统重启对接

- 目前可对接的系统重启接口有 :
  1. ***void xf_sys_reboot(void)*** : 软件系统重启

系统中断对接

- 目前可对接的系系统中断接口有 :
  1. ***xf_err_t xf_sys_interrupt_enable(void)*** : 系统中断开启
  2. ***xf_err_t xf_sys_interrupt_disable(void)*** : 系统中断关闭

至此，基础功能对接完成。

- 后面根据需要可对接 xf_osal 、 xf_ble 、 xf_sle 、xf_wifi 、 xf_netif 等其他部分。

最后更新于: 2025/2/8 19:33:07

Previous page
构建对接

Next page
外设对接

构建对接

作者

dotc

本章节介绍如何对接 XFusion 构建部分

(可简单理解为：如何将 XFusion 的工程的构建 (编译) 信息给到平台侧的工程，一起进行构建)

前置准备：

- 了解 XFusion 的构建命令及其参数的作用。
- 会使用 python
- 简单了解 jinja 用法
- 对将要对接的平台构建流程较为熟悉
- 已完成 平台工程移植 的步骤 (本章节默认已经完成该步骤)

目前需要对接的基础功能有

对接流程

---

1. 对接构建方法

1. 在 **xfusion/plugins** 下 新增一个目录，目录名需要与目标目录名 (***xfusion/boards/ ... /[target_name]***) 一致（或者说与环境激活时 (. ./export.sh [target_name])，传入的目标对象名一致）。
   - 原因：xfusion 在进行构建时 (如： xf build)，会在 xfusion/plugins 下 搜寻环境激活的目标对象名的同名目录，执行其下的构建插件脚本。

2. 在 xfusion/plugins/[target_name]/ 下，创建 "**__init__.py**" 文件，其内容如下：

```python
from .build import *
```

3. 在 xfusion/plugins/[target_name]/ 下，创建 "**build.py**" 文件，并按要求及需要实现各个回调方法。

- build.py 中回调方法的对接：
  - build : 对接编译方法
    - 可简单理解为：将 XFusion 下的工程构建信息便捷转换为平台侧的构建信息，然后一起参与编译。
    - build 方法对接通常有以下步骤 (简述):
      1. 提供模板文件（*.j2，普通模板与组件模板）（基于 jinja 模板引擎）
      2. 调用模版系统解析方法，配合模板文件（*.j2），解析 xf 侧工程的编译信息 (**build_environ.json**，详情浏览下面 "xf 执行编译时的执行过程 (xf 工程目录下)："的第 1 点)， 生成平台侧可用的构建信息文件（如 xx.mk、CMakeLists.txt、xx.cmake 等）
      3. 将新生成的构建信息文件（平台侧可用的）加入到平台侧工程进行编译
    - xf 编译时的执行过程 (xf 工程目录下)：
      1. xf 构建系统会收集 xf 侧的编译相关信息参与编译的文件、includepath、编译选项、CFLAGS、组件信息等），并生成至 **xf 工程目录下** 的 **build 目录下 build_environ.json** 文件中。
      2. xf 构建系统执行此插件文件的 **build** 方法。
    - build 方法一般为以下步骤（详情）（需由移植开发者编写）：

      > 关于模板文件如何编写参考 jinja 文档或已存在的其他平台的模板文件

      - 根据需要调用对应的模版解析方法及传入对应的模板文件， 对 xf 工程 build 目录下的 **build_environ.json** 文件进行解析。 方法目前有两种：
        1. 普通模板解析方法：

```python
def apply_template(temp, save, replace=None):
temp：普通模板文件
save：解析完后保存的文件路径（包含名字）
replace（可选）：
```

1. 读取 build_environ.json 文件
2. jinja 模板引擎加载传入的模板文件进行解析
3. 接收解析结果，转存至指定的文件（传入的 save 参数）

2. 组件模板解析方法（非必须，也可自行通过 普通模板解析方法 实现组件解析）

```python
def apply_components_template(temp, suffix):
    temp：组件模板文件
    suffix：解析完后保存的文件的名字或后缀。
```

1. 读取 **build_environ.json** 文件
2. jinja 模板引擎加载传入的组件模板文进行解析：
   - 遍历 **public_components** 、 **user_components** 组件集合下的各组件项，每项单独根据组件模板进行解析生成单独的文件，存储的位置会根据 **build_environ.json** 中的层次结构进行目录创建，文件名会根据传入参数 suffix 进行生成：
     - 如果是 suffix 是类似 “xxx” 的文件名形式，则会生成 “xxx” 的文件
     - 如果是 suffix 是类似 “.xxx” 的后缀形式，则会生成组件名（又或者说是所在目录名）同名的后缀是 “.xxx” 的文件。
   - 例如：public_components 下有组件 xf_hal， 解析生成文件后会 xf 工程 build 目录下，会出现 build/public_components/xf_hal/ 的目录结构，在 xf_hal 下：
     - 如果传入 suffix 是 “CMakeLists.txt”，则会生成 CMakeLists.txt 文件；
     - 如果传入 suffix 是 “.mk”，则会生成 xf_hal.mk 文件；

- clean : 对接清除方法。(略)

- flash : 对接下载方法。(略)
- export : 对接工程导出方法。(略)
- update : 对接（导出的）工程更新方法。(略)
- menuconfig : 对接目标平台的 menuconfig。(略)
- clean : 对接清除方法。(略)

▶ **build.py 内容的模板及注解如下：**

---

2. 验证

- 在任意普通的 xf 工程下，激活目标环境，然后执行 "xf build"，查看编译信息等是否显示编译成功。

至此，构建对接完毕

- 后面需对接的是 xf_sys 的部分

最后更新于: 2024/12/2 20:04:27

Previous page
平台工程移植

Next page
基础功能对接

外设对接（XF_HAL）

作者

dotc

本章节介绍如何对接 XFusion 外设 (HAL) 部分

前置准备：

- 了解 XFusion 的构建命令及其参数的作用。
- 了解 xf_hal 的作用及 XFusion 的例程。
- 熟悉将要对接的外设的原理、将对接的平台的处理流程等。
- 了解 xf_hal 对将要对接的外设的对接要求。

目前可对接外设功能有

1. xf_adc
2. xf_dac
3. xf_gpio
4. xf_i2c
5. xf_pwm
6. xf_spi
7. xf_tim
8. xf_uart

对接流程

1. 实现对接 xf 驱动操作集的注册函数。一般流程为：

1. 对接 xf_driver_ops_t 通用操作集的各个方法：（具体作用及要求详参：[xf_driver_ops_t 操作集的各方法作用及对接要求]）

```c
typedef struct _xf_driver_ops_t {
    xf_err_t (*open)(xf_hal_dev_t *dev);
    xf_err_t (*ioctl)(xf_hal_dev_t *dev, uint32_t cmd, void *config);
    int (*read)(xf_hal_dev_t *dev, void *buf, size_t count);
    int (*write)(xf_hal_dev_t *dev, const void *buf, size_t count);
    xf_err_t (*close)(xf_hal_dev_t *dev);
} xf_driver_ops_t;
```

2. 将对接好的操作集注册至 xf_hal 中，调用如“xf_hal_xxx_register”形式的函数，xxx 是外设类型，如 gpio 则调用 xf_hal_gpio_register。

---

2. 将 ① 中实现的注册函数加入自动初始化工作序列中。一般流程为：

调用自动初始化标记宏 "XF_INIT_EXPORT_PREV" 对注册函数进行标记。如：" XF_INIT_EXPORT_PREV(port_xf_gpio_register); ":

> **IMPORTANT**
>
> 注意需要加上 " ***#include "xf_init.h"*** " XF 初始化头相关的文件包含，否则注册无效，编译也不会报错只有警告，这一步容易被忽略。

例: gpio 对接 (操作集填充及注册部分)：

```c
int port_xf_gpio_register(void)
{
    xf_driver_ops_t ops = {
        .open = port_gpio_open,
```

```
        .close = port_gpio_close,
        .ioctl = port_gpio_ioctl,
        .write = port_gpio_write,
        .read = port_gpio_read,
    };
    return xf_hal_gpio_register(&ops);
}

XF_INIT_EXPORT_PREV(port_xf_gpio_register);
```

至此，外设对接完毕

---

xf_driver_ops_t 操作集的各方法作用及对接要求

1. xf_err_t (*open)(xf_hal_dev_t *dev)

- 通常是对应外设应用侧的 init 操作被调用时被调用，进行对象构造。如 gpio 是 xf_hal_gpio_init 被调用时，会调用对接时注册到操作集下的 open 方法。
- 返回值类型： **xf_err_t** ： 正常返回 XF_OK。
- 参数说明：

  ▶ **xf_hal_dev_t *dev：xf hal 基类对象：**

- open 函数一般会执行的流程如下：
  1. 有效性检查。如 GPIO、I2C 资源索引号是否有效、是否超出可用的范围等。
  2. （非必要）初始化（创建）平台侧的外设对象，句柄记录至基类对象（参数 dev）中的 platform_data 成员。一般包含对接时需要再各个操作中流转的一些数据，如外设的部分状态、平台侧读写的方法（如 I2C 主机与从机记录为不同的读写方法）。
  3. （非必要）对平台侧的外设对象成员进行初始化。

2. xf_err_t (*close)(xf_hal_dev_t *dev)

- 通常是对应外设应用侧的 deinit 操作被调用时被调用，进行对象析构。
- 返回值类型： **xf_err_t** ： 正常返回 XF_OK。
- 参数说明：
  - ***xf_hal_dev_t *dev***：xf hal 基类对象。

    ▶ **xf_hal_dev_t *dev**

- close 函数一般会执行的流程如下：
  1. 有效性检查。如 GPIO、I2C 资源索引号是否有效、是否超出可用的范围等。
  2. （非必要）外设相关的反初始化处理
  3. （非必要）反初始化（销毁、回收内存）平台侧的外设对象。

3. xf_err_t (*ioctl)(xf_hal_dev_t *dev, uint32_t cmd, void *config)

- 通常是对应外设应用侧相关配置被设置时会被调用。
- 常见的配置如 timeout、IO num、速率、外设开启与关闭等，不同外设可能有不同的可配置项，详情查阅如 "xf_hal_xxx_cmd_t" 的类型（xxx 为外设类型，如 GPIO 则为 xf_hal_gpio_cmd_t）。
- 返回值类型： **xf_err_t** ： 正常返回 XF_OK。
- 参数说明：
  - ***xf_hal_dev_t *dev***：xf hal 基类对象。

    ▶ **xf_hal_dev_t *dev**

  - ***uint32_t cmd***：配置命令集，每个命令各占一个位（bit），从第 0 为向上递增，当某位为 1 时则表示该位命令触发。详情查阅如 "xf_hal_xxx_cmd_t" 的类型 (xxx 为外设类型，如 GPIO 则为 xf_hal_gpio_cmd_t)。
  - ***void *config***：hal 层配置记录。记录着 hal 层该外设的配置信息。此处是 void 类型，通常需要强转类型为对应外设的 hal 层配置类型使用，类型形式

如“xf_hal_xxx_config_t”（xxx 为外设类型，如 GPIO 则为 xf_hal_gpio_config_t）。

- 所有外设都有两个通用控制命令： **XF_HAL_XXX_CMD_DEFAULT** 与 **XF_HAL_XXX_CMD_ALL。**
  - **XF_HAL_XXX_CMD_DEFAULT (0x0)**：用于表示需要将默认配置加载至 hal 层配置记录 config 中。一般需要单独且放在最前面处理。
  - **XF_HAL_XXX_CMD_ALL (0x7FFFFFFF)**：因为置 1 了所有配置位，所以表示所有命令生效。一般无需单独处理，只需按命令位逐个处理即可。

  ▶ **这两个命令都是在用户侧调用外设 init 函数时，在调用 open 后，进行下发的，调用过程如下：**

  ▶ **ioctl 一般处理流程参考（也可自行按要求用其他方式进行处理）：**

4. int (*read)(xf_hal_dev_t *dev, void *buf, size_t count)

- 在 hal 层外设读相关的操作被执行时会被调用。
- 返回值类型： **int**：正常则返回实际读取到的数据量；**异常报错时，需要注意返回的必须是负值错误码。**
  - 返回值处理情况：
    1. 如果是非阻塞异步读且未完成的情况一般返回的是 0。
    2. 一般错误码均是正值（如 xf_err_t 中除 XF_FAIL 外的错误码），则需要取反处理，以确保负值返回。
    3. 可能有些平台某些错误返回本身就是负值，则直接返回即可。如果有需要，也可对这种负值错误情况进行特殊提示。
- 参数说明：
  - ***xf_hal_dev_t *dev***：xf hal 基类对象。

    ▶ **xf_hal_dev_t *dev**

- ***void *buf***：读出的数据的缓存。
- ***size_t count***：期望读出的数据量

5. int (*write)(xf_hal_dev_t *dev, const void *buf, size_t count)

- 在 hal 层外设写相关的操作被执行时会被调用。
- 返回值类型： **int** ：正常则返回实际写入的数据量；**异常报错时，需要注意返回的必须是负值错误码。**
  - 返回值处理情况：
    1. 如果是非阻塞异步写且未完成的情况一般返回的是 0。
    2. 一般错误码均是正值（如 xf_err_t 中除 XF_FAIL 外的错误码），则需要取反处理，以确保负值返回。
    3. 可能有些平台某些错误返回本身就是负值，则直接返回即可。如果有需要，也可对这种负值错误情况进行特殊提示。
- 参数说明：
  - ***xf_hal_dev_t *dev***：xf hal 基类对象。

    ▶ **xf_hal_dev_t *dev**

  - ***const void *buf***：写入的数据的缓存。
  - ***size_t count***：需要写入的数据量

最后更新于: 2024/12/2 20:04:27

其他对接

作者

dotc

本章节介绍如何对其他功能的内容。(如 xf_osal 、xf_ble 、 xf_sle 等)

前置准备：

- 了解 XFusion 的构建命令及其参数的作用。
- 了解对接的功能的作用及 XFusion 的例程。
- 熟悉将要对接的功能的原理、将对接的平台的处理流程等。
- 了解 XFusion 对将要对接的功能的对接要求。

目前可对接其他功能有

1. xf_osal
2. xf_ble
3. xf_sle
4. xf_wifi
5. xf_netif

对接流程

略

最后更新于: 2024/12/2 20:04:27

Previous page
外设对接

Next page
贡献指南

平台工程移植

作者

dotc

本章节介绍如何将平台 SDK 与 XFusion 关联

(可简单理解为：如何创建一个关联 XFusion 的平台侧工程)

前置准备：

- 了解 XFusion 的构建命令及其参数的作用。
- 会使用 python
- 简单了解 jinja 用法
- 对将要对接的平台构建流程较为熟悉

对接流程

---

1. 新增目标对象及目标目录

- 在 xfusion/boards 下新增一个目录，再在其下创建一个名为 "**target.json**" 的文件，此时，**该目录名**则为 xfusion 下平台环境激活以及构建时可以被选定的**目标对象名**。
  - 原因：xfusion 在进行环境激活时（. ./export.sh xxx），递归检索并收集 boards 目录下的存在 "target.json" 文件的目录的名字，将其所在的目录名加入到可激活、可构建的目标对象列表中。
- 例：将要新增 星闪芯片 ws63 的支持，目标名为 "ws63"

  1. 可以在 boards 下递归创建目录 nearlink/ws63

2. 其后在 boards/nearlink/ws63 下创建 "**target.json**" 文件

3. 此时不指定目标平台去执行环境激活命令（**. ./export.sh**），即可看到提示需要传入目标名，其后紧跟着当前支持的目标名，就能看的 "ws63" 在支持列表中了，如下：

```bash
user@host:. ./export.sh
user@host:You need to choose one of the following targets: ws63 bs21 esp32
```

- export 激活环境过后，以下变量将会被设置：

  1. "XF_ROOT" 设置成激活的 XFusion 的跟目录。
  2. "XF_TARGET" 设置成激活的目标对象名。
  3. "XF_TARGET_PATH" 设置成激活的目标目录的路径。
  4. "XF_VERSION" 设置成激活的 XFusion 的版本。
  - 更多可查看激活命令执行时的输出信息。

---

2. 提供平台 SDK 至 XFusion

- 准备好要对接的平台的 SDK，放至 xfusion/sdks 下。
- 如果可以且需要进行版本控制，可在**目标目录**下的 "**target.json**" 文件中添加版本信息。(***xfusion/boards/ ... /[target_name]***，如何创建目标目录见: 新增目标对象)
  - "**target.json**" 版本控制条目及说明如下：

```json
{
    "sdks":{
        "url": "https://github.com/xxxxx",
        "commit": "xxxxxx",
        "dir": "平台 SDK 在 xfusion/sdks 下的路径 (目录名)",
        "branch": "分支名"
    }
}
```

3. 提供平台侧工程 至 XFusion (平台工程关联至 XFuison)

> **IMPORTANT**
>
> 此步通用但不是必须的。（如：平台侧的普通工程无法存在于 SDK 外，或者难度过大，则可能没有此步骤，不提供平台侧工程至 XFusion ，而是在源 SDK 下的工程直接关联至 XFusion，不过此情况下也可参考本步骤进行对接）

1. 准备一份平台侧最简的、包含构建文件、能正常编译的平台侧原生工程，迁移或拷贝至**目标目录**下 (如：xfusion/boards/../["target_name"]，如何创建目标目录见: 新增目标对象)
2. 后面将以此工程为基础，来对接 (关联) 到 XFusion，浏览下一章节 构建对接

最后更新于: 2024/12/2 20:04:27

Previous page
移植指南

Next page
构建对接