应用指南

AT32F413固件库BSP&Pack应用指南

前言

这篇应用指南对如何使用AT32F413固件库BSP(Board Support Package)以及如何安装AT32 Pack进行了简单的描述，对用户起到引导性的作用。

目录

表目录

图目录

1 简介

为了让用户高效快速的使用Artery MCU，雅特力官方提供了一套完整的BSP&Pack用于开发。主要包括：外设驱动库、内核相关文件、完整的应用例程以及能够支持Keil_v5、Keil_v4、IAR_v6和IAR_v7、IAR_v8等多种开发环境的Pack文件。

本应用指南会介绍BSP&Pack具体的使用方法。

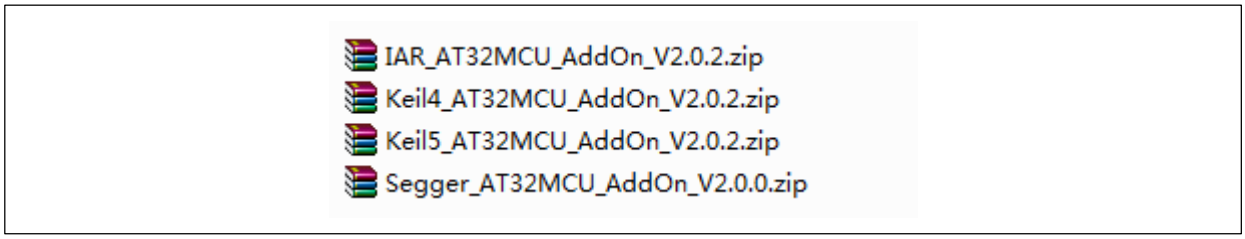
2 Pack 安装步骤

ArteryTek提供了支持Keil_v5、Keil_v4、IAR_v6、IAR_v7和IAR_v8等多种开发环境的Pack文件，对应的Pack采用‘双击’完成一键式安装。

注意：本章节主要以 AT32F403A 做举例说明，AT32 MCU 其他型号的 Pack 安装步骤是类似的，不再累述。

Pack安装文件如下图（具体版本信息按实际情况为准）。

图 1. Pack 安装包



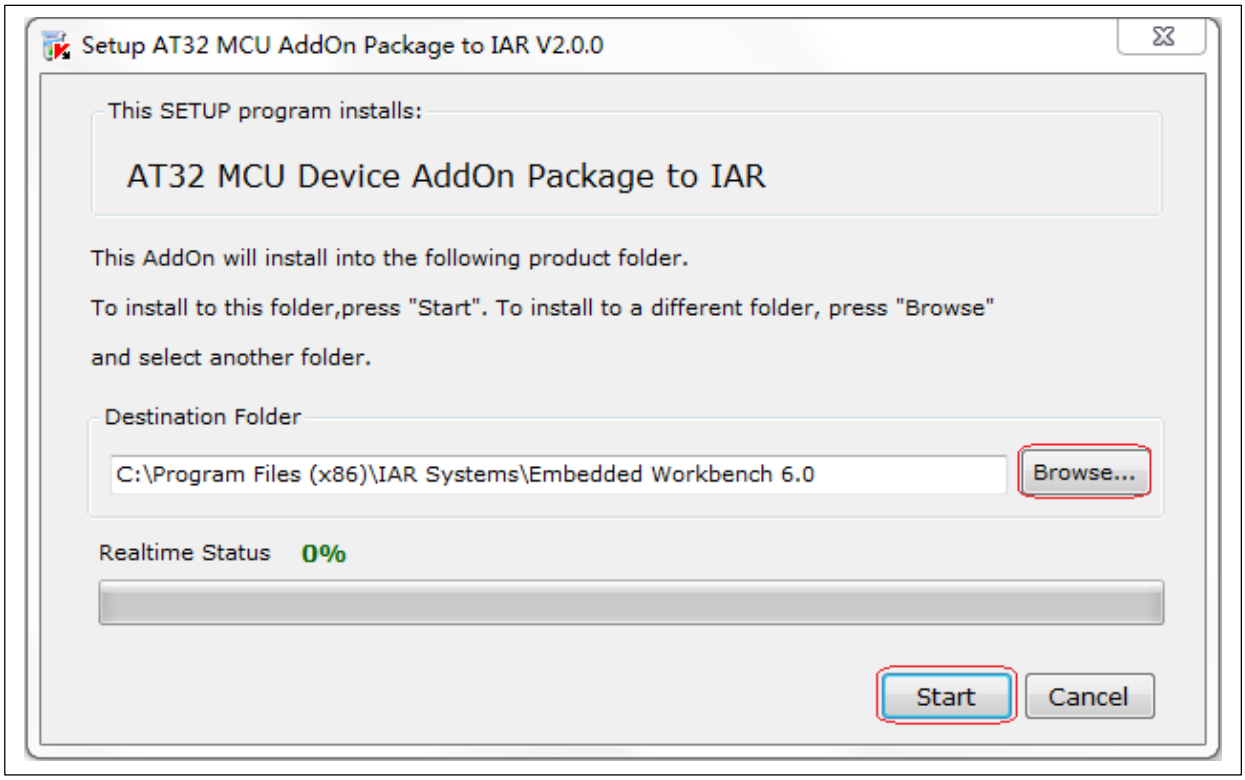
2.1 IAR Pack 安装

IAR_AT32MCU_AddOn.zip：支援 IAR_V6、IAR_V7 和 IAR_V8 的压缩包，安装步骤如下：

① 解压 IAR_AT32MCU_AddOn.zip。

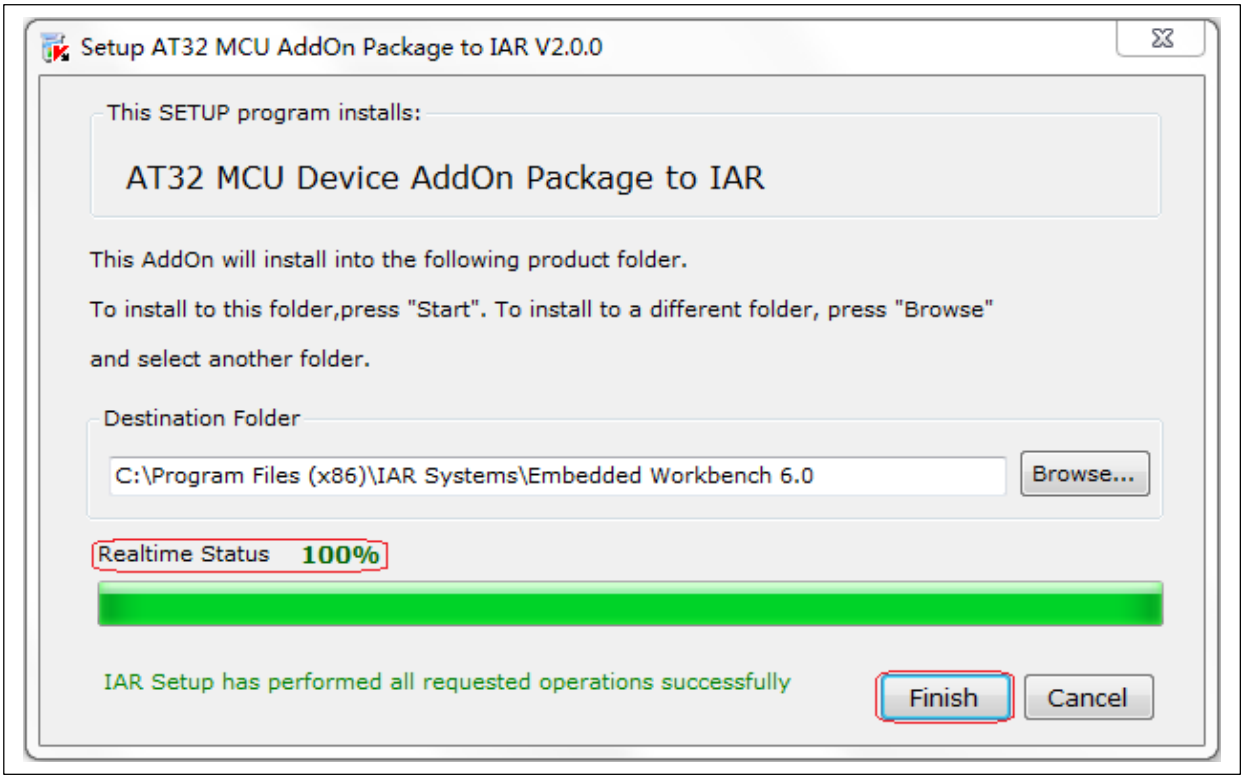
② 双击 IAR_AT32MCU_AddOn.exe，弹出如下界面（具体版本信息按实际情况为准）。

图 2. IAR Pack 安装界面



注意：如果 IAR 的实际安装路径与 “Destination Folder”对话框内的路径不一致，点击“Browse”选择实际安装路径。然后点击“Start”启动安装过程，如下图。

图 3. IAR Pack 安装流程

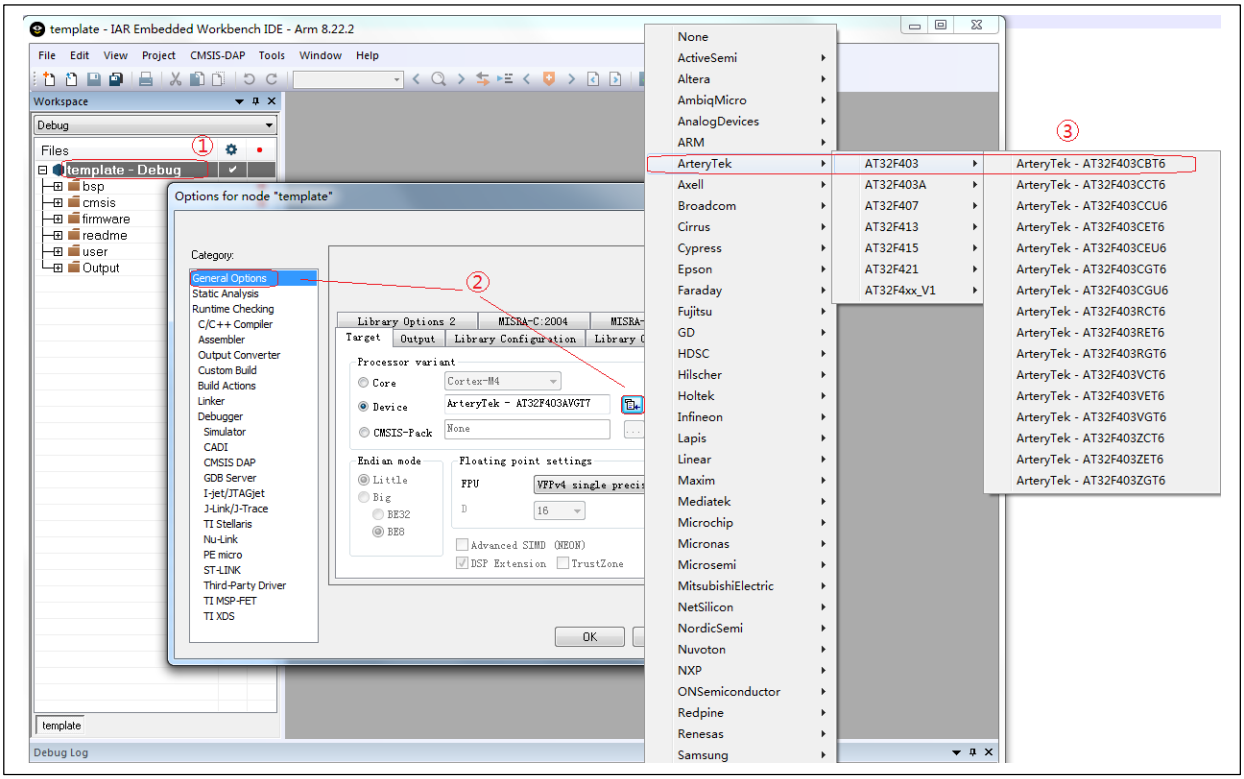


③ 点击“Finish”完成安装。

④ 查看 IAR Pack 是否安装成功。任意打开一个 IAR 工程，按如下步骤操作和查看：

- 鼠标右键点击工程名，并选择 Options…
- 选择 General Options，并点选复选框。
- 查看 ArteryTek 以及 ArteryTek – AT32F…相关的型号信息。

图 4. 查看 IAR Pack 安装情况



2.2 Keil_v5 Pack 安装

Keil5_AT32MCU_AddOn.zip：支援 Keil_v5 的 pack 压缩包，具体版本见包内实际内容，安装步骤如下：

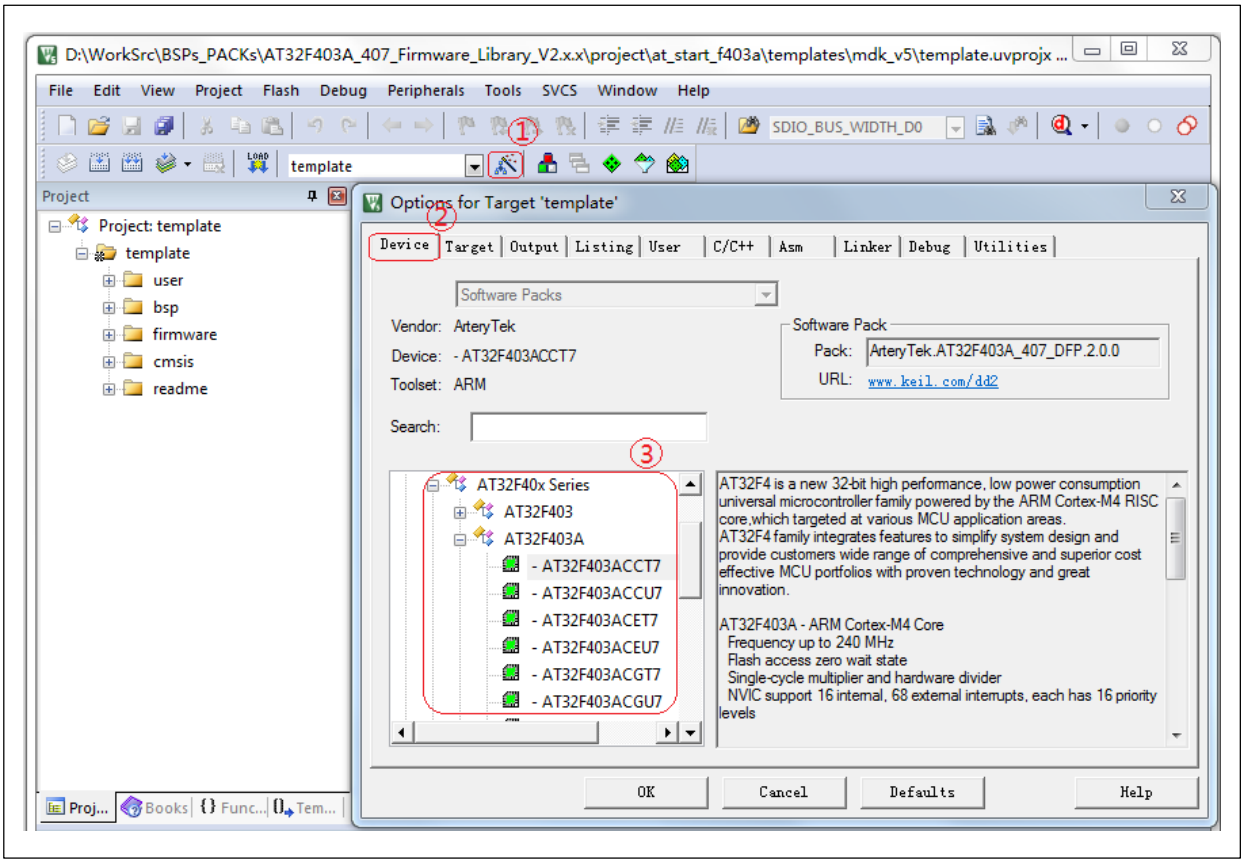
① 解压 Keil5_AT32MCU_AddOn.zip，里面包含了所有目前支持的 Keil5 pack 安装包，都是标准的 Keil_v5 DFP 安装文件。

② 选择所需系列的安装包，双击 ArteryTek.AT32xxxx_DFP.2.x.x.pack 完成一键式安装。

⑤ 查看 Keil_v5 Pack 是否安装成功。按如下步骤操作和查看：

- 点击魔术棒。
- 选择 Device 选项卡。
- 出现 ArteryTek 及相关型号信息。

图 5. 查看 Keil_v5 Pack 安装情况



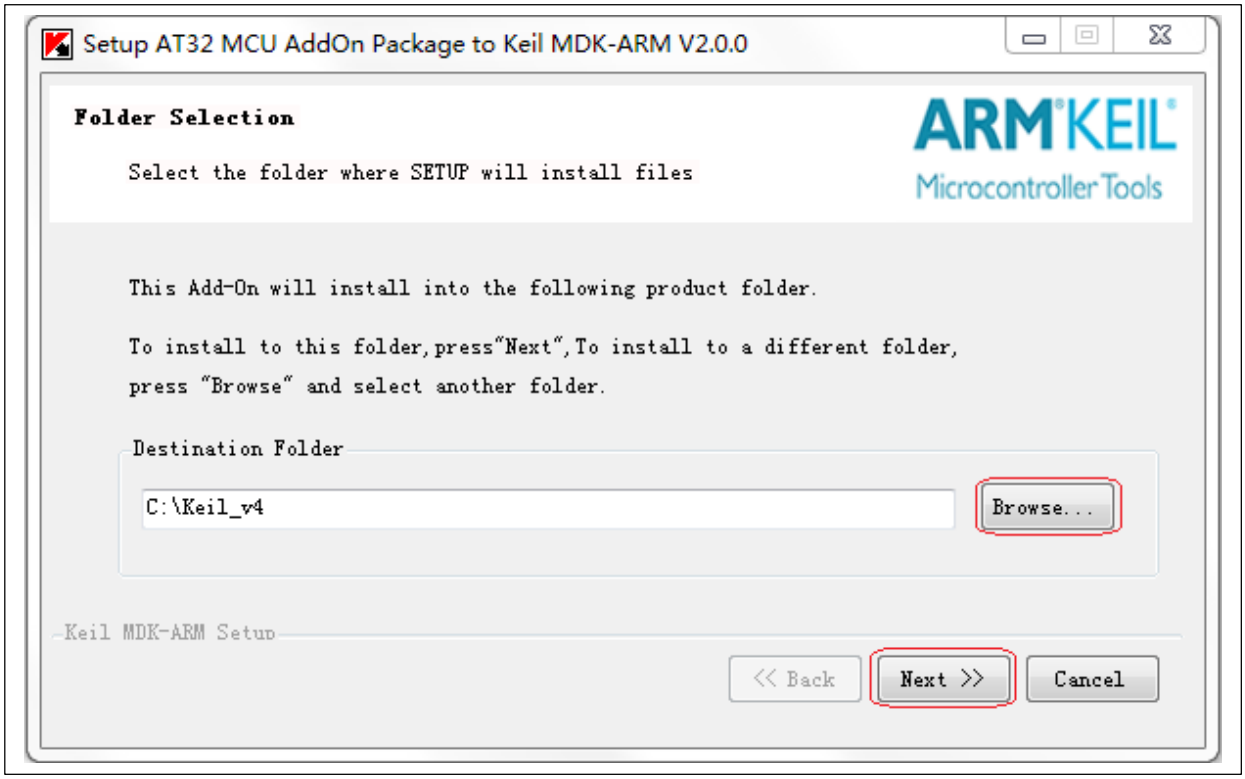
2.3 Keil_v4 Pack 安装

Keil4_AT32MCU_AddOn.zip：支援 Keil_v4 的压缩包，安装步骤如下：

① 解压 Keil4_AT32MCU_AddOn.zip。

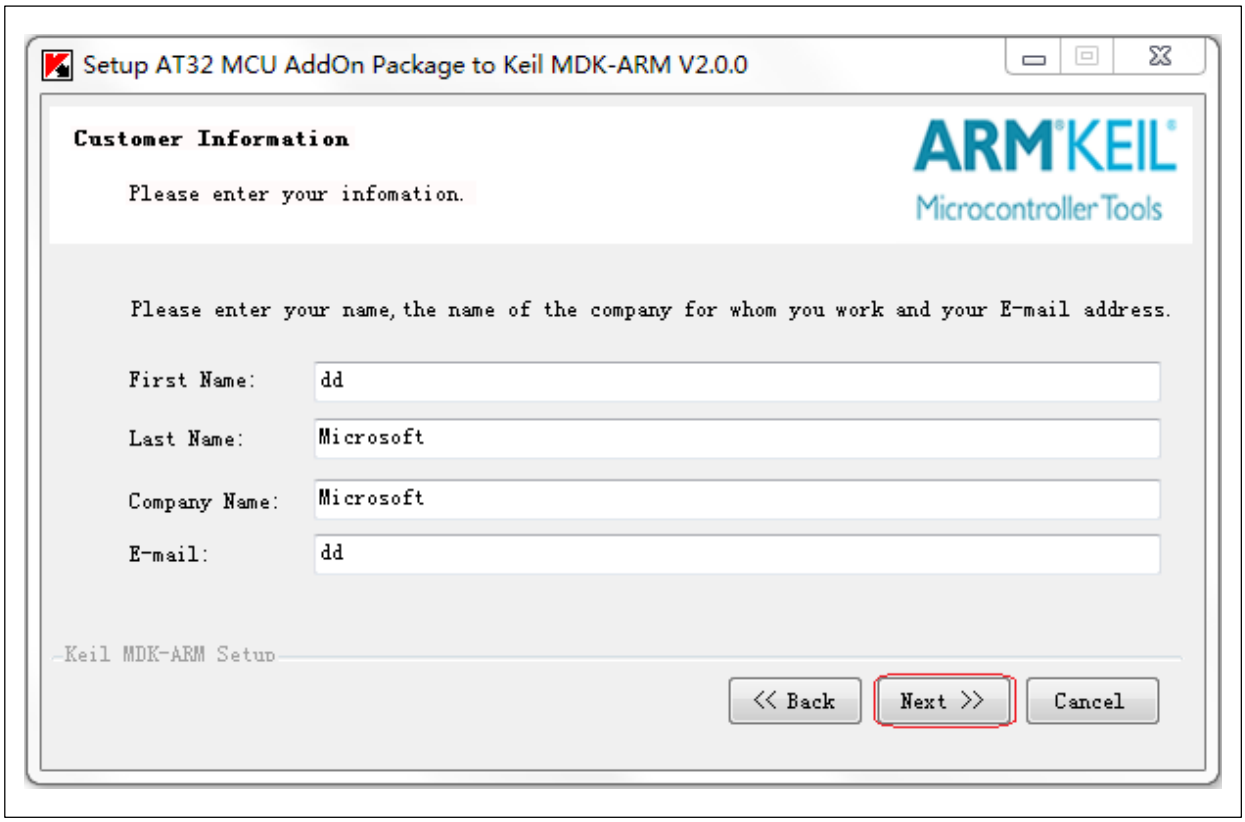
② 双击 Keil4_AT32MCU_AddOn.exe，弹出如下界面（具体版本信息按实际情况为准）。

图 6. Keil_v4 Pack 安装界面



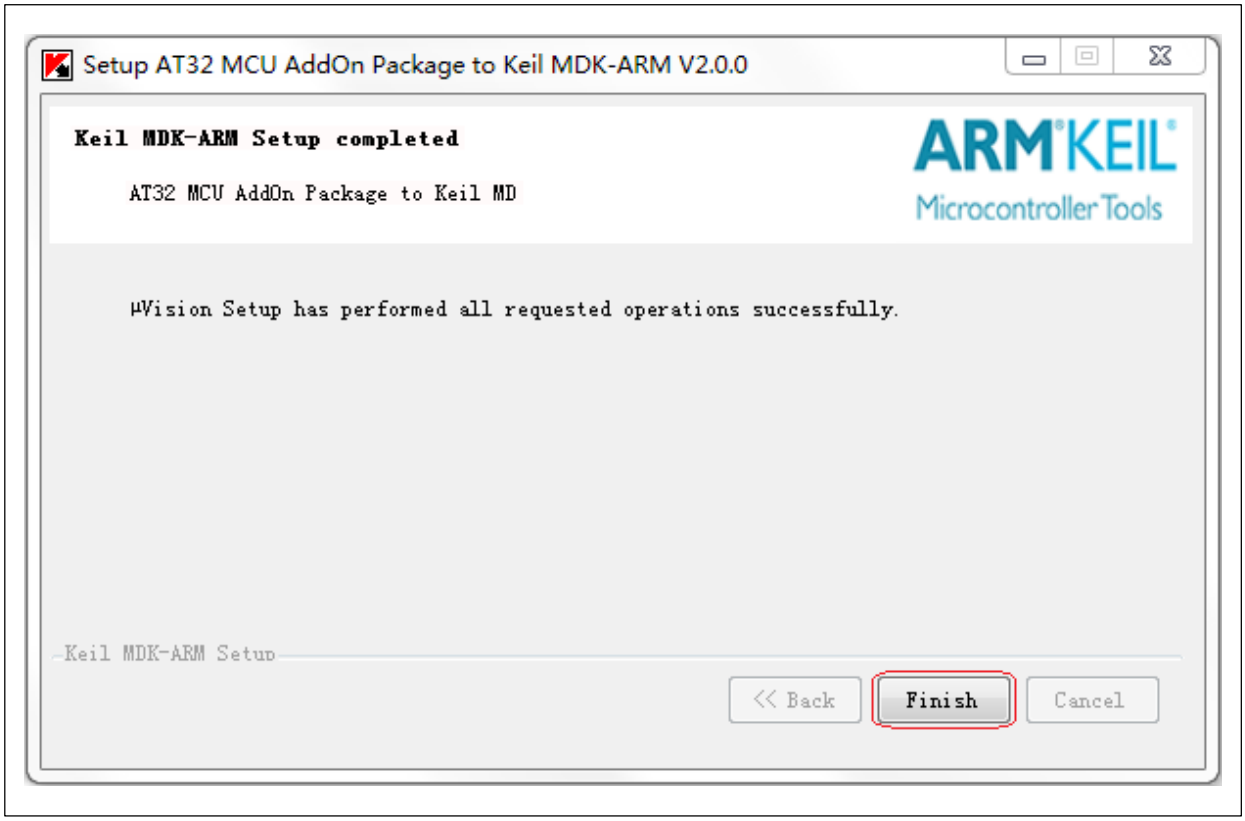
③ 如果 Keil_v4 的实际安装路径与“Destination Folder”对话框内的路径不一致，点击“Browse”选择实际安装路径。然后点击“Next”，弹出如下界面。

图 7. Keil_v4 Pack 安装流程



④ 在上图的界面中修改“Customer Information”，一般不需要修改此类信息。然后点击“Next”启动安装过程，安装结果如下图：

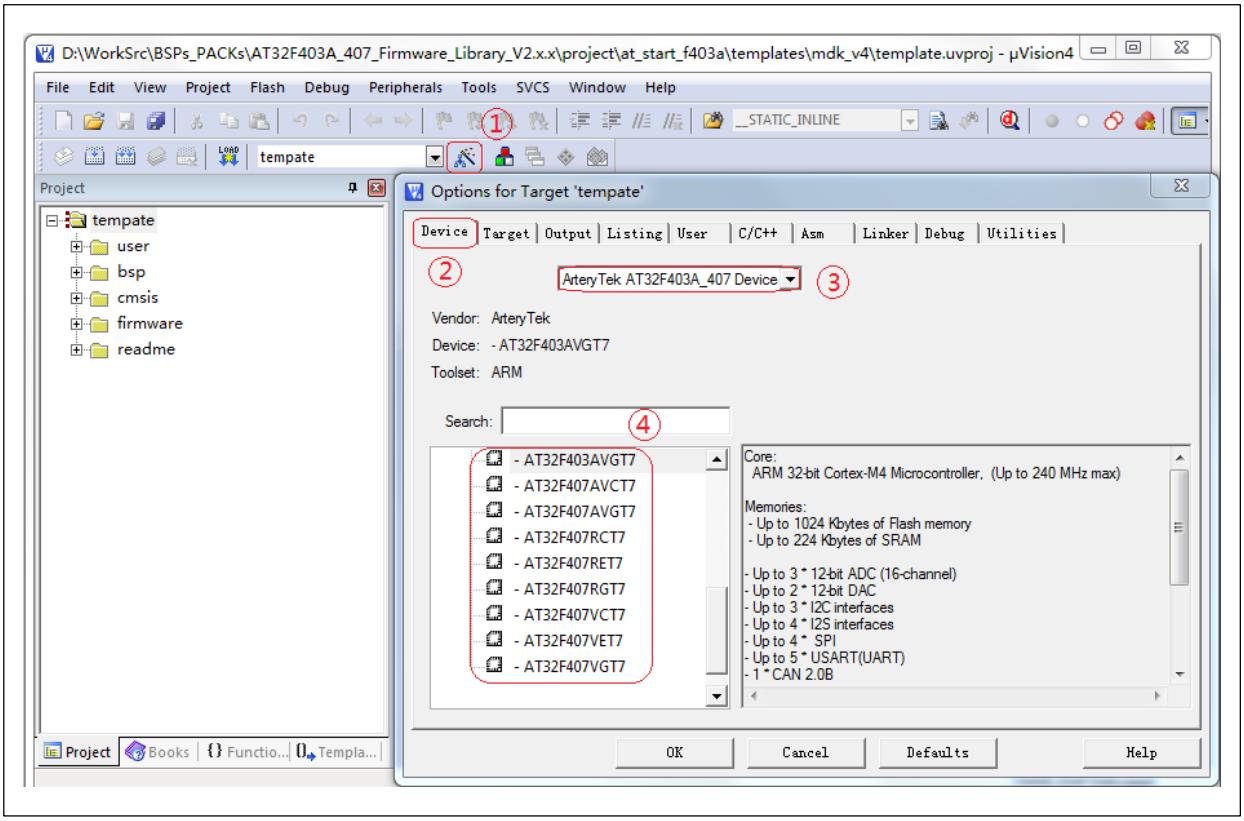
图 8. Keil_v4 Pack 安装完成



⑤ 点击“Finish”完成安装。查看 Keil_v4 Pack 安装是否成功。请按如下步骤进行操作和查看：

- 点击魔术棒。
- 点选 Device 选项卡。
- 选择 ArteryTek 提供的对应系列的型号包文件。
- 出现 ArteryTek 信息及芯片型号。

图 9. 查看 Keil_v4 Pack 安装情况



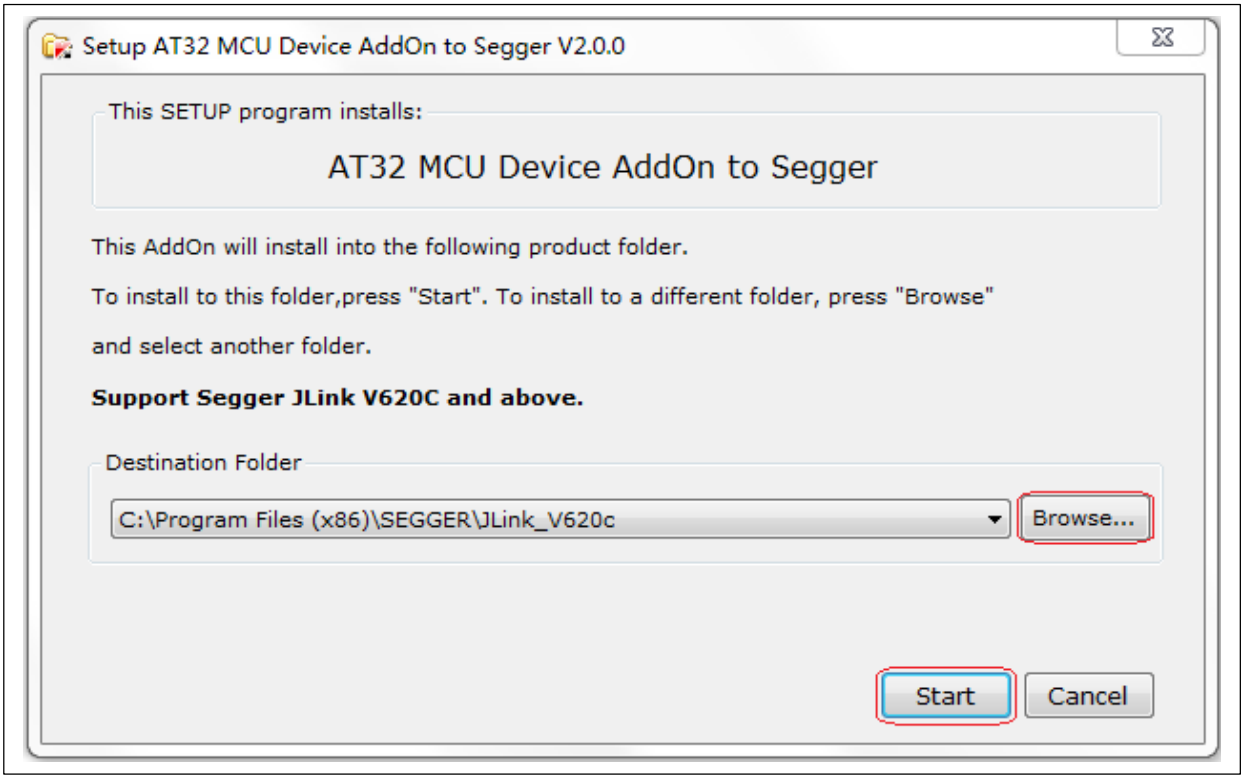
2.4 Segger Pack 安装

Segger_AT32MCU_AddOn.zip：支援 J-Flash 下载的压缩包，安装步骤如下：

① 解压 Segger_AT32MCU_AddOn.zip。

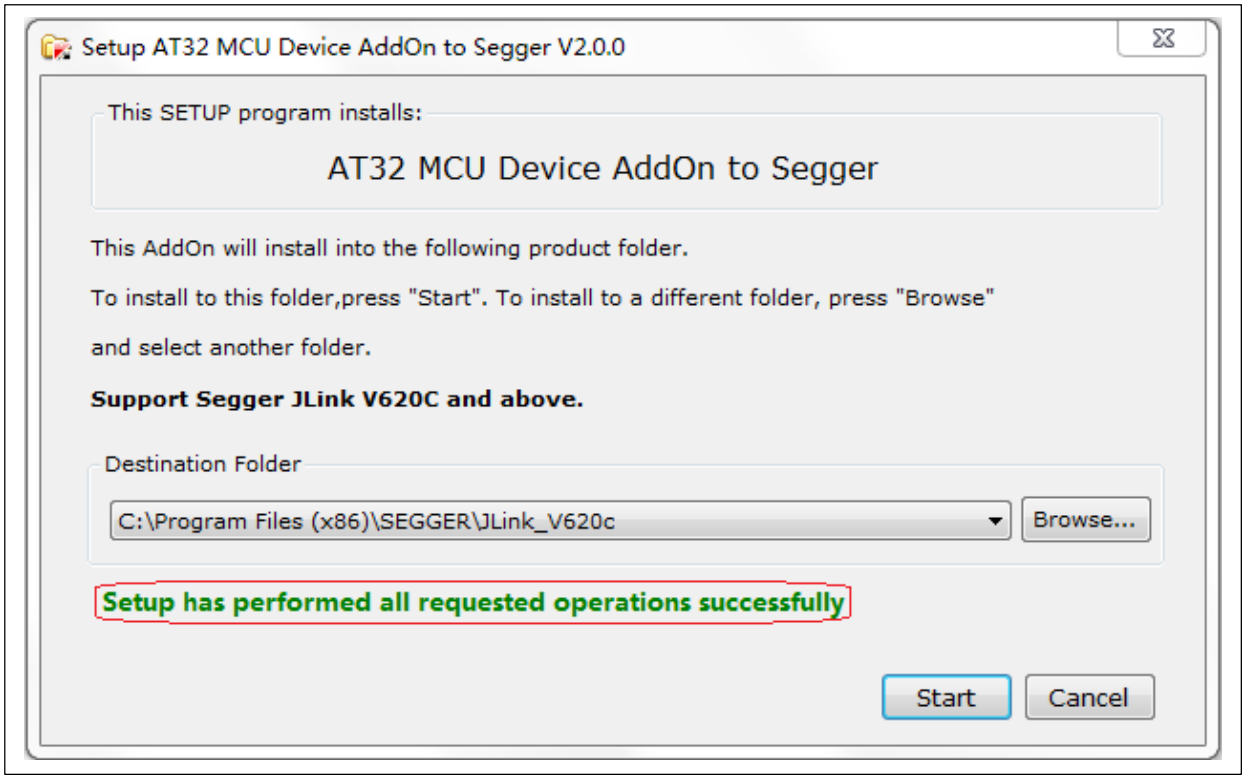
② 双击 Segger_AT32MCU_AddOn.exe，弹出如下界面（具体版本信息按实际情况为准）。

图 10. Segger 包安装界面



注意：如果 Segger 的实际安装路径与 “Destination Folder”对话框内的路径不一致，点击“Browse”选择实际安装路径。然后点击“Start”，弹出如下界面。

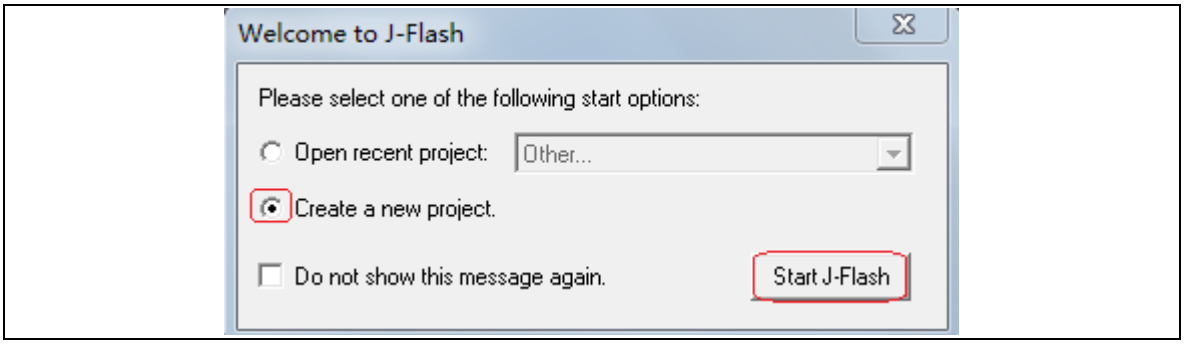
图 11. Segger 包安装流程



③ 出现“Setup has performed all requested operations successfully”则表示已安装成功。查看是否安装成功，请按如下步骤进行操作和查看:

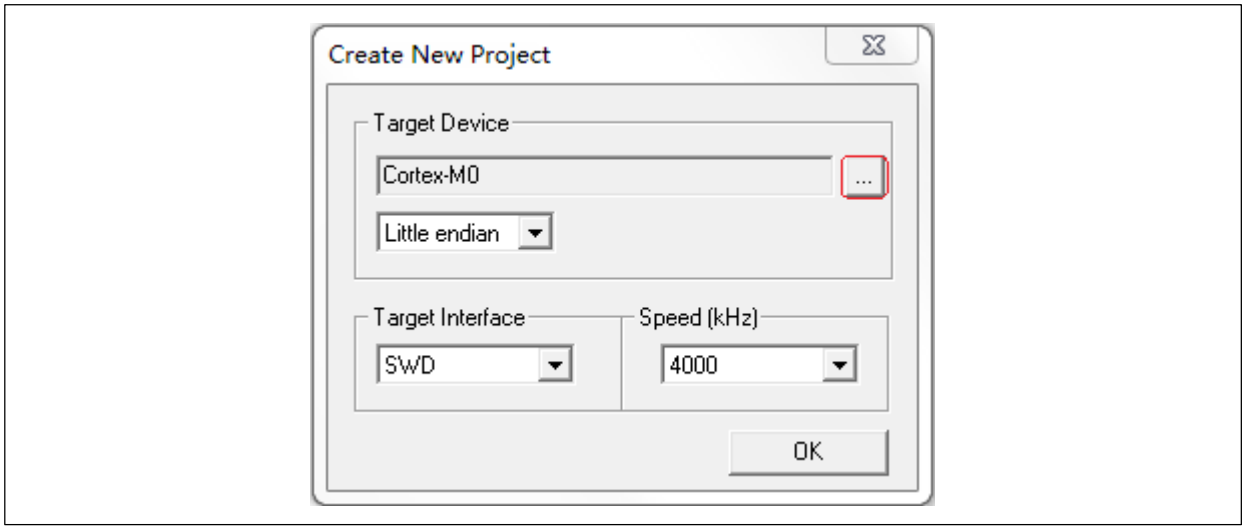
- 打开 J-Flash.exe，出现如下对话框则选择 Create a new project 并点击 Start J-Flash 按钮，如下图：

图 12. 打开 J-Flash



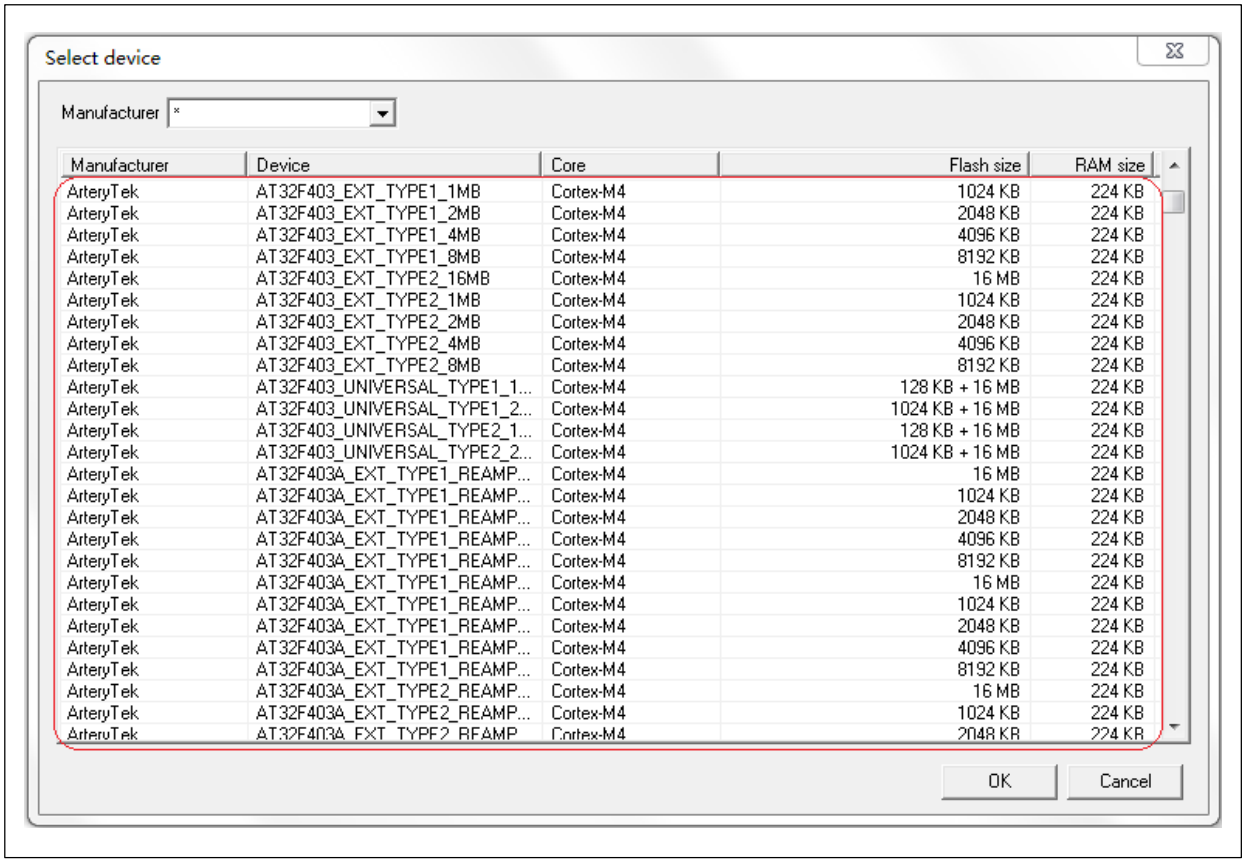
- 启动 J-Flash 后，点击 Target Device 栏后的复选按钮，如下图：

图 13. J-Flash 创建新工程



- 在复选框中上下拉动滚动条如查找到 ArteryTek 相关信息及算法文件则表示安装成功，如下图：

图 14. 查看 Device 信息



3 Flash 算法文件说明

对于Artery MCU，我们都有在对应发布的Pack文件中整合了相关型号的Flash算法文件以供如KEIL/IAR等IDE工具进行在线code下载。虽各IDE工具对于算法文件的使用方法大致都一样，以下还是对算法文件的使用方法进行简单的说明。

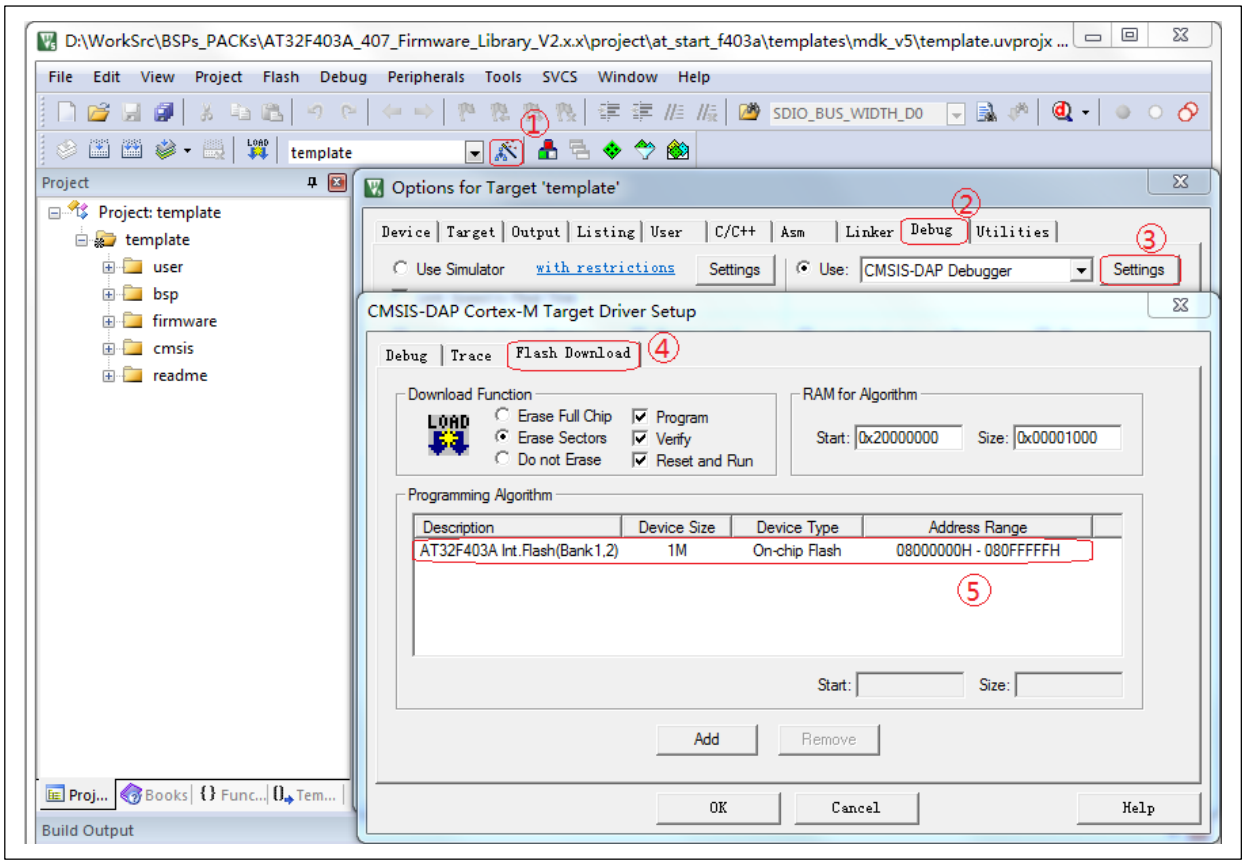
注意：本章节主要以 AT32F403A 做举例说明，AT32 MCU 其他型号的 Flash 算法说明是类似的，不再累述。

3.1 Keil 算法文件的使用方法

因常用的Keil_v4和Keil_v5 IDE开发环境在算法文件选择方法和使用上基本一样，以下对应Keil_v5环境的使用来进行说明。

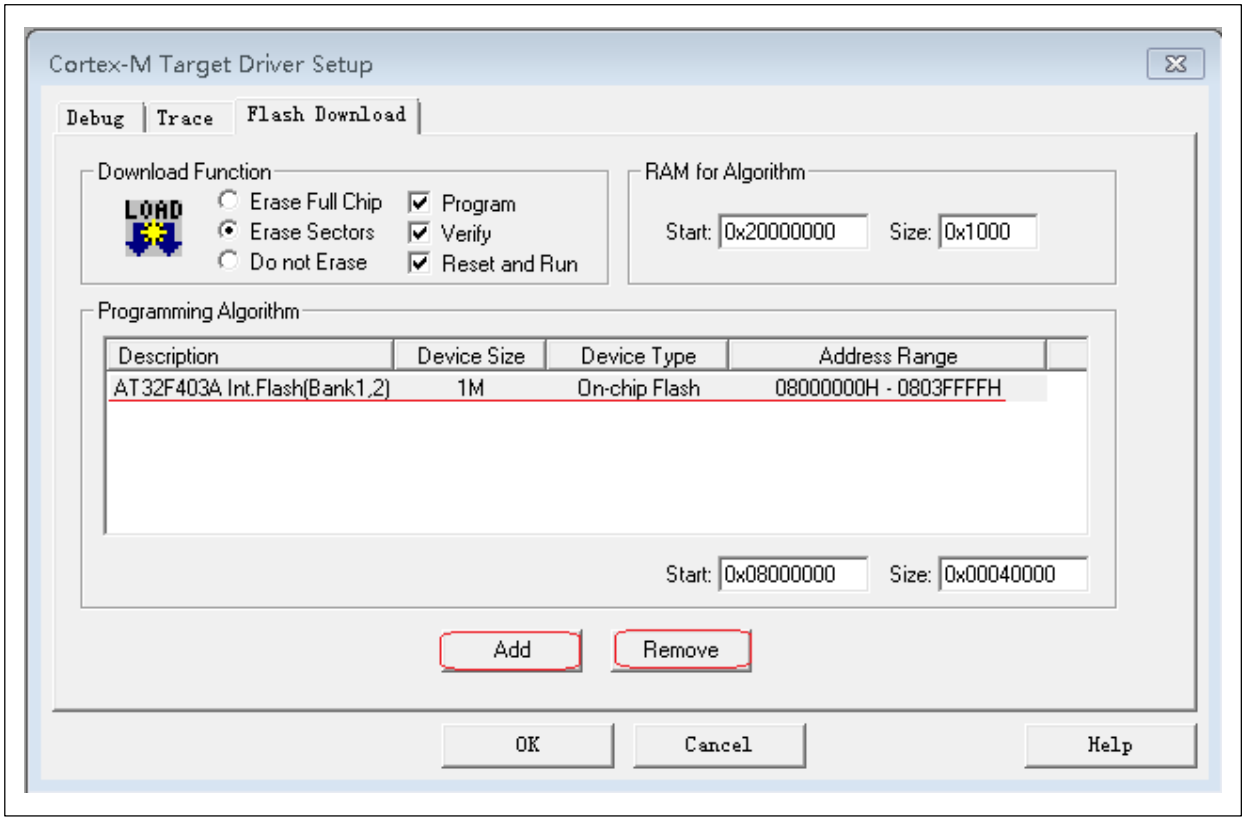
当在Keil IDE开发工具工程建立起来之后即可进行Debug方式配置和flash算法文件的选择。在开发工具内依次点击：配置魔术棒—>Debug选项卡—>Settings—>Flash Download，流程如下图：

图 15. Keil 算法文件设置



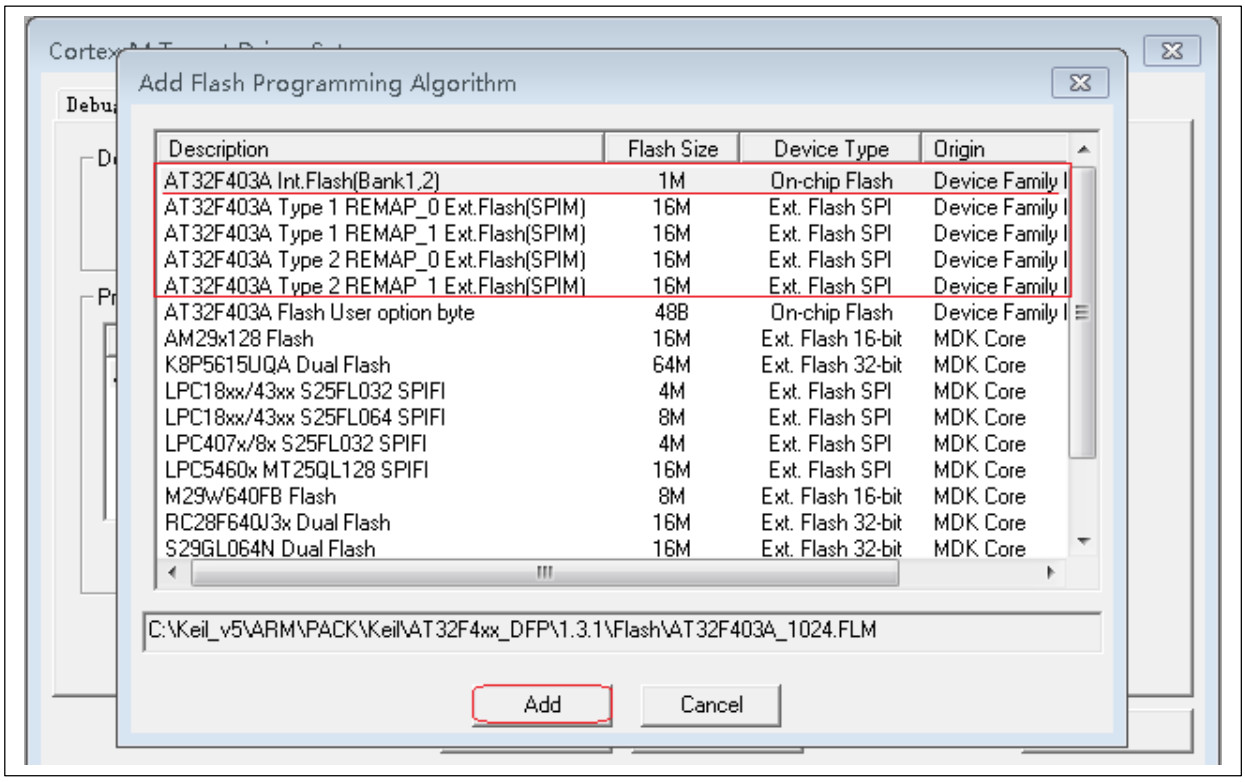
此处示例可看到所选择的Flash算法文件为默认的Flash算法文件，如需更改和移除可自行配置，点击到算法文件后可看到Add和Remove按钮可选择，如所选算法和实际MCU不匹配可使用以下方法重新配置

图 16. Keil 算法文件配置栏



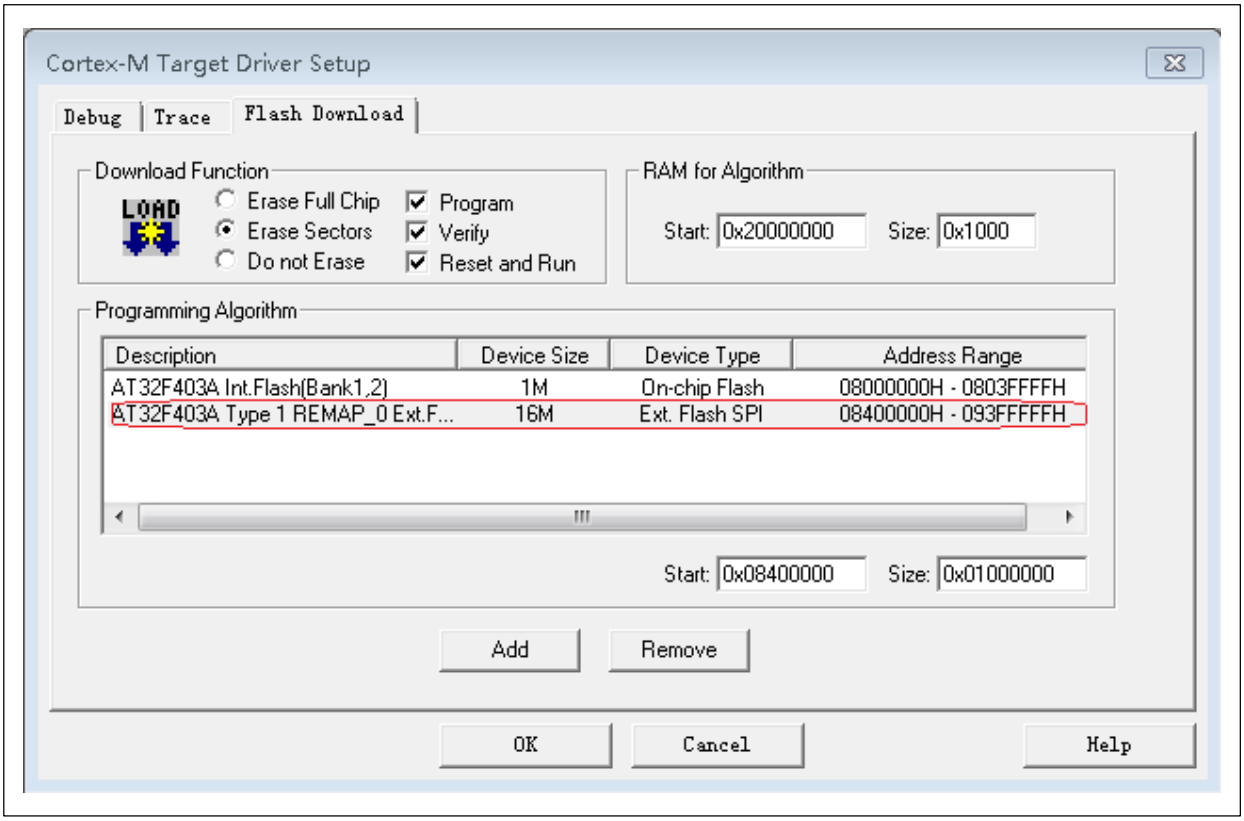
点击Remove可将当前选择到的算法文件从工程配置中移除，点击Add可查看支持此型MCU的算法文件并进行选择，示例如下：

图 17. Keil 选择算法文件



当选择到相应的算法文件后点击Add即可将新算法文件加入到当前工程配置，如下示例是新增SPIM算法到工程中：

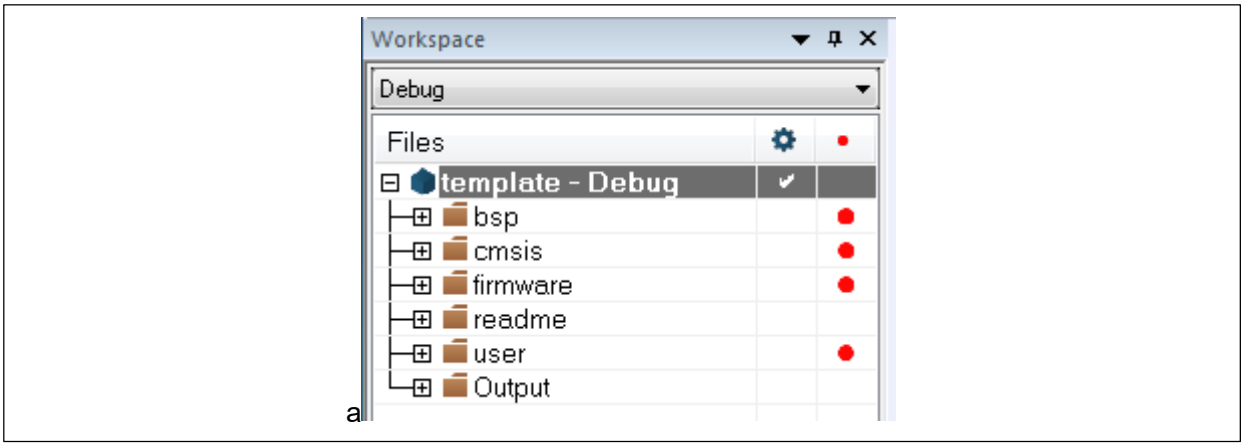
图 18. Keil 新增算法文件



3.2 IAR 算法文件的使用方法

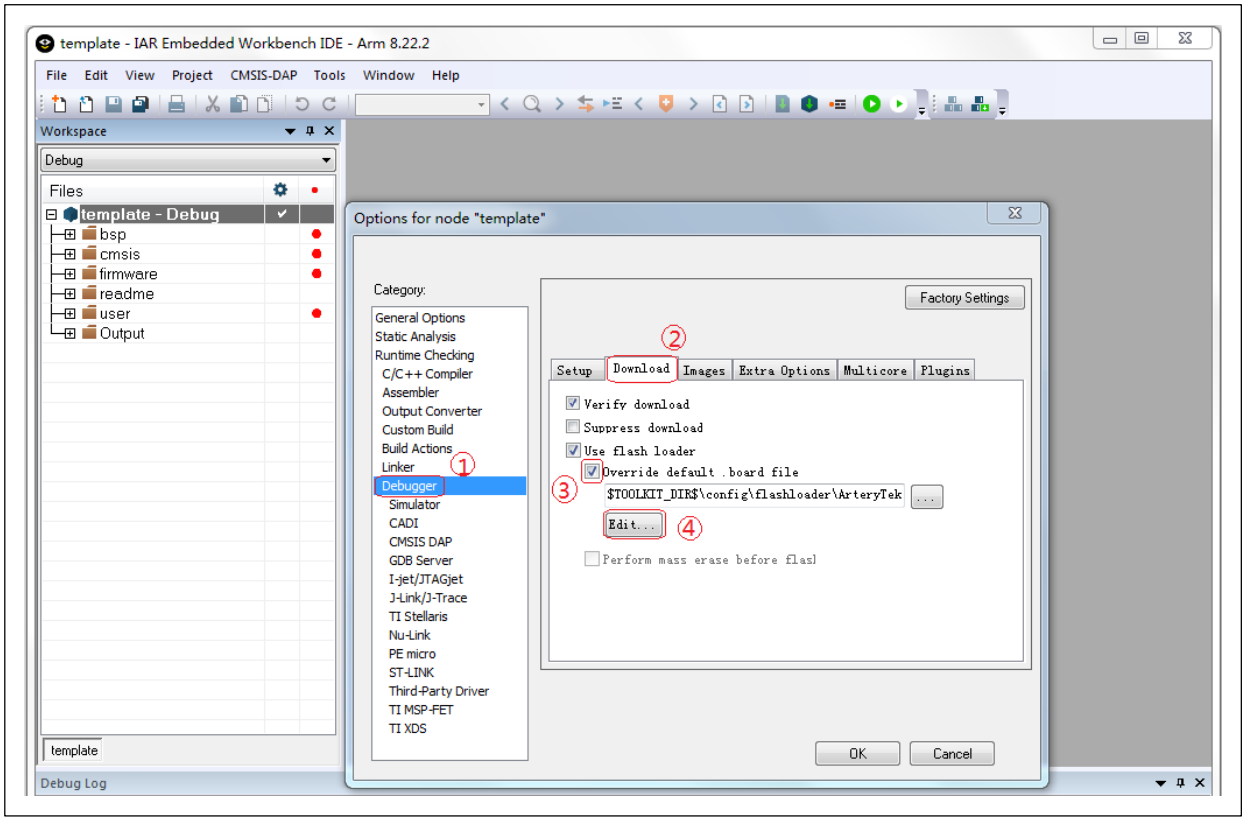
IAR开发环境对算法文件的选择方法是在当新建工程的配置中选定指定的MCU型号后自动选定的对应的默认flash算法文件。如需手动去进行算法文件配置，可在IAR工程建立起来之后，鼠标右击如下灰色选框位置的工程名：

图 19. IAR 工程名



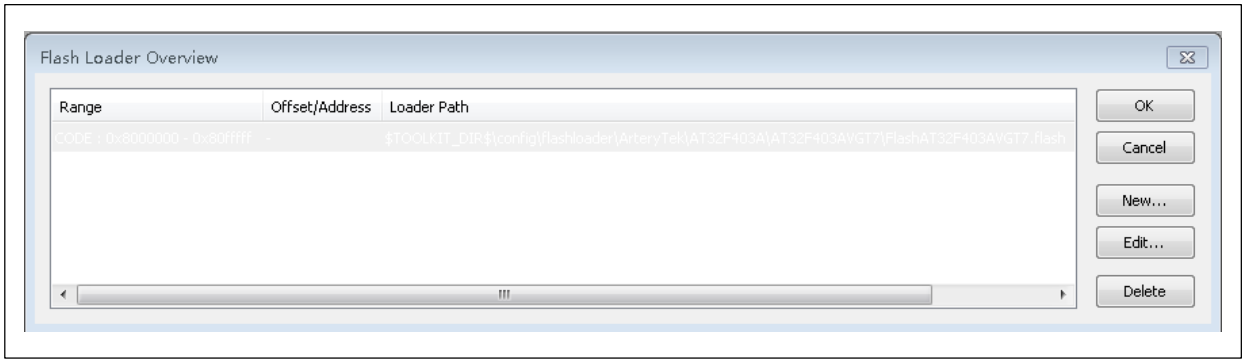
选择Options—>Debugger—>Download—>勾选Override default .board file—>点击Edit，流程如下图所示：

图 20. IAR 算法文件配置



进入后可看见如下的配置界面：

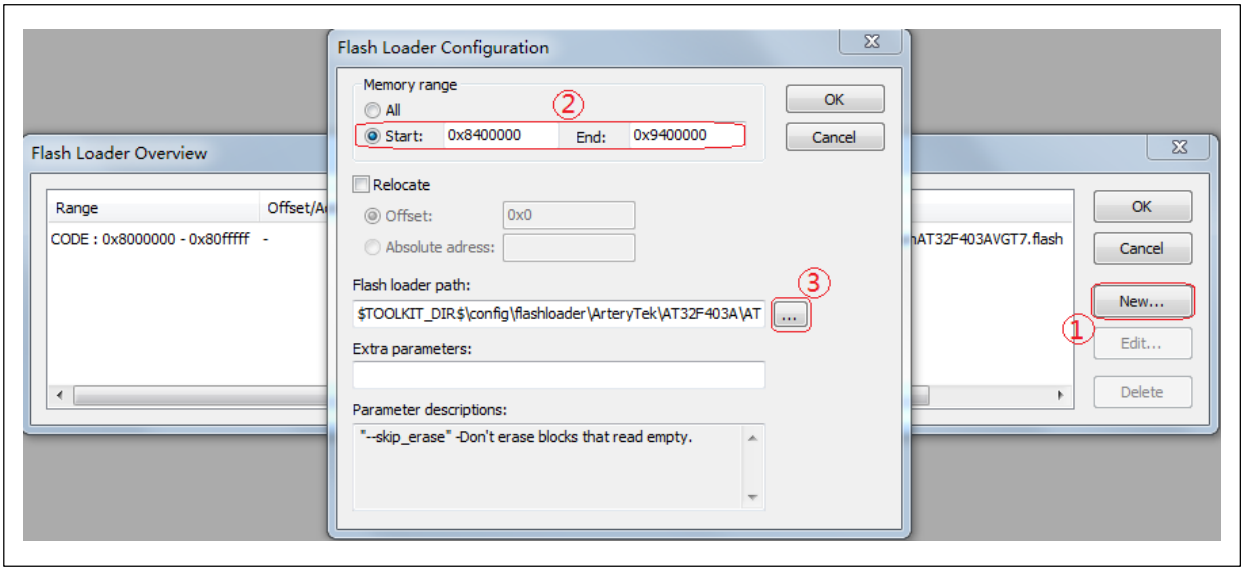
图 21. IAR Flash Loader 新增



其中的flash算法配置方法是选定MCU芯片型号后默认指定，如需手动进行修改可点击旁边的New/Edit/Delete三个选项进行修改。

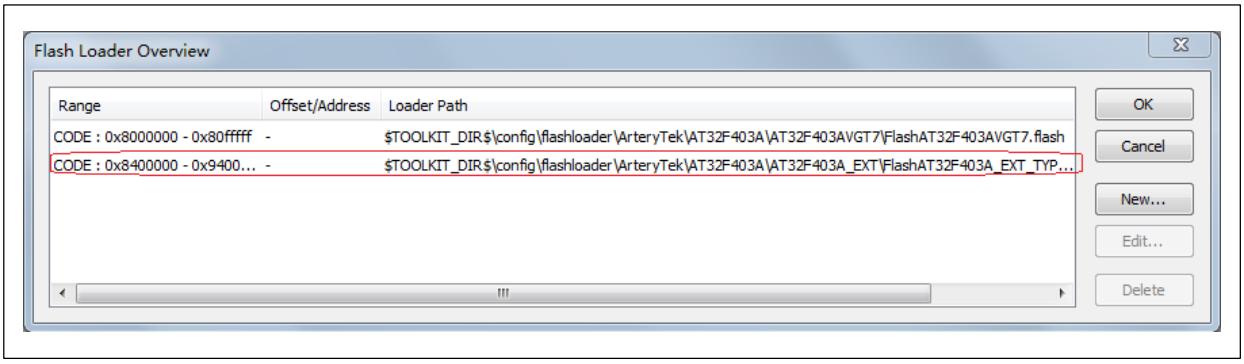
以点击New新增配置Flash算法文件举例。1.点击New—>2.配置Flash范围—>3.选择对应的Flash算法配置文件。流程如下图所示：

图 22. IAR Flash Loader 配置



此处示例是新增SPIM flash算法文件举例。需选择对应型号且正确的Flash算法文件进行配置。被选择的flash算法配置文件是由IAR_AT32MCU_AddOn工具安装到IAR开发环境内。示例新增的SPIM flash算法完成配置后如下图：

图 23. IAR Flash Loader 配置成功



1. SPIM 算法文件说明

Artery部分MCU 支持Bank3（详情请参考官方Reference Manual或DataSheet），其接口外挂flash可作为内部flash不足或特殊应用需求情况下的flash存储介质的扩充，当软件程序中部分code或数据指定编译链接地址在SPIM存储空间时，IDE工具在线下载的过程中需要使用到此算法文件进行外部flash编程。Artery SPIM算法文件的命名方式如下：AT32F4xxTypeNREMAP_P Ext.Flash。

N=1,2

P=0,1

TYPEN：外接的SPI Flash类型，按外接flash类型和型号进行选择。详细信息请参考对应MCU Reference Manual的FLASH_SELECT寄存器描述。

REMAP_P：MCU SPIM PIN脚的复用选择，按连接外部flash的硬件电路PIN脚连线方式进行选择。详细信息请参考对应MCU Reference Manual的外部SPIF重映射章节。

REMAP0：EXT_SPIF_GRMP=000

REMAP1：EXT_SPIF_GRMP=001

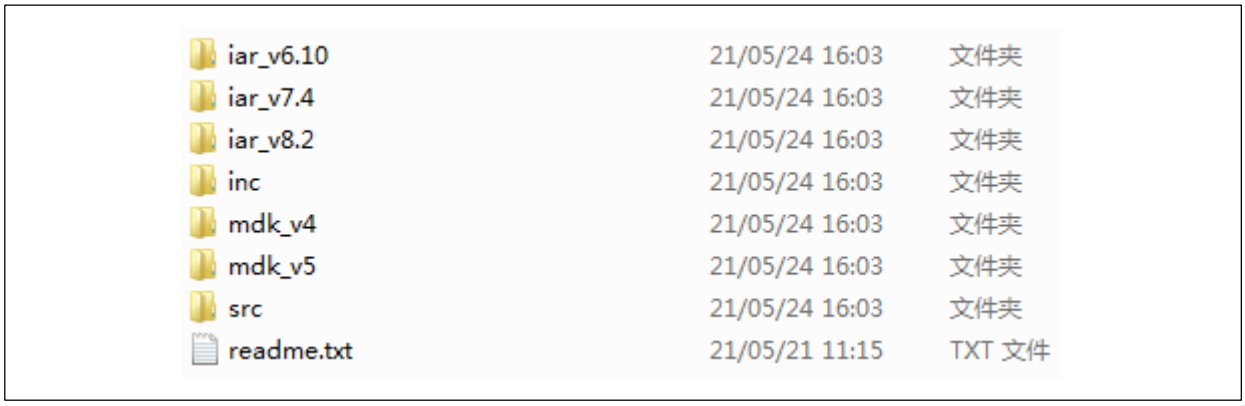
4 BSP 使用简述

4.1 BSP 快速使用

4.1.1 模板工程介绍

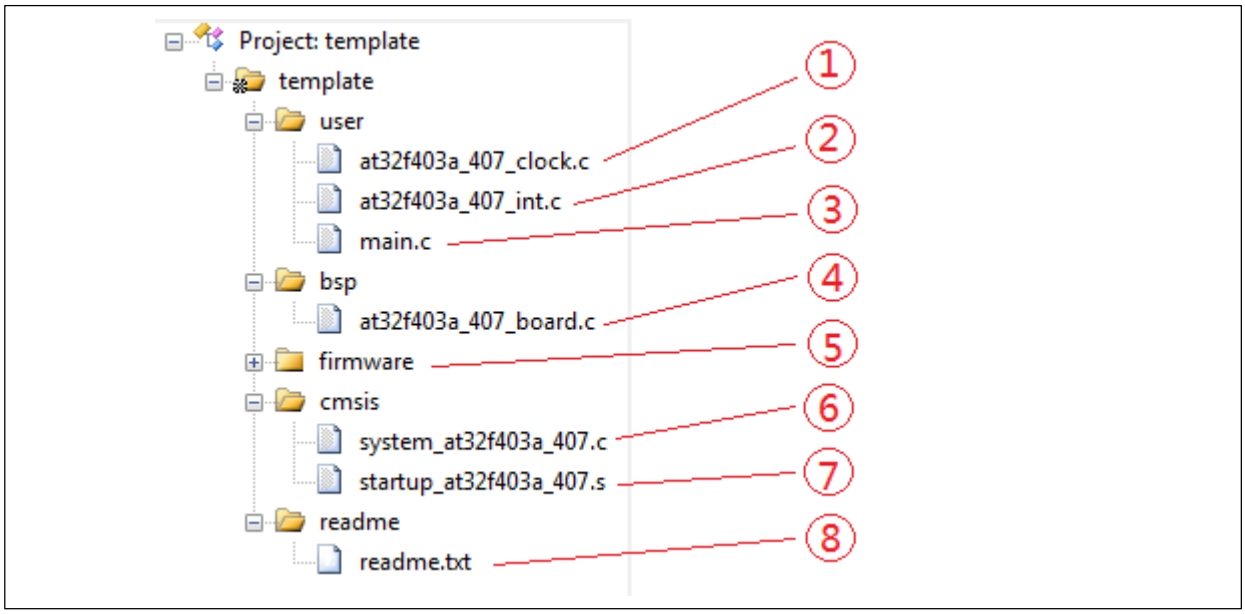
在 ArteryTek 提供的固件库 BSP 中都默认建立好了 Keil 和 IAR 常用版本下的模板工程。以 AT32F403A/407 系列为例，其存放目录在 AT32F403A_407_Firmware_Library_V2.x.x/project/at_start_xxx/templates 中，内容如下：

图 24. templates 文件内容



在此创建了 Keil_v5、Keil_v4、IAR_6.10、IAR_7.4 和 IAR_8.2 版本的模板工程。inc 和 src 文件夹分别保存了模板工程中所用到的应用部分的头文件及源码文件。打开对应工程的文件夹并点击工程文件即可打开对应的 IDE 工程。如下是 Keil_v5 工程示例（具体内容及版本以实际固件包内容为准）：

图 25. Keil_v5 模板工程示例



工程内添加的内容描述如下（以 AT32F403A/407 系列举例，其他系列与此类似）：

① at32f403a_407_clock.c 时钟配置文件，设置了默认的时钟频率及时钟路径。

② at32f403a_407_int.c 中断文件，默认编写了部分内核中断函数的代码流程。

③ main.c 模板工程的主代码文件。

④ at32f403a_407_board.c 板级配置文件，设置了 AT-START 上的按键和 LED 等常用硬件配置。

⑤ firmware 下的 at32f403a_407_xx.c 是各片上外设的驱动文件。

⑥ system_at32f403a_407.c 系统初始化文件。

⑦ startup_at32f403a_407.s 启动文件。

⑧ readme.txt 工程的说明文件，记录了模板工程的一些应用功能及设置方式等信息。

注意：本章节主要以 AT32F403A 做举例说明，AT32 MCU 其他型号的 BSP 使用简述是类似的，不再累述。

4.1.2 BSP 相关宏定义

① 在创建工程时，需要导入启动代码（startup_at32f403a_407.s）到工程，Code 编译之前，还需要根据 MCU 型号，开启对应的宏定义，MCU 型号与宏定义的对应关系如下表

表 1. 型号宏定义对应表

MCU型号	宏定义	PINs	Flash大小(KB)
AT32F403ACCT7	AT32F403ACCT7	48	256
AT32F403ACET7	AT32F403ACET7	48	512
AT32F403ACGT7	AT32F403ACGT7	48	1024
AT32F403ACCU7	AT32F403ACCU7	48	256
AT32F403ACEU7	AT32F403ACEU7	48	512
AT32F403ACGU7	AT32F403ACGU7	48	1024
AT32F403ARCT7	AT32F403ARCT7	64	256
AT32F403ARET7	AT32F403ARET7	64	512
AT32F403ARGT7	AT32F403ARGT7	64	1024
AT32F403AVCT7	AT32F403AVCT7	100	256
AT32F403AVET7	AT32F403AVET7	100	512
AT32F403AVGT7	AT32F403AVGT7	100	1024
AT32F407RCT7	AT32F407RCT7	64	256
AT32F407RET7	AT32F407RET7	64	512
AT32F407RGT7	AT32F407RGT7	64	1024
AT32F407VCT7	AT32F407VCT7	100	256
AT32F407VET7	AT32F407VET7	100	512
AT32F407VGT7	AT32F407VGT7	100	1024
AT32F407AVCT7	AT32F407AVCT7	100	256
AT32F407AVGT7	AT32F407AVGT7	100	1024

② 系列芯片头文件中（at32f403a_407.h），USE_STDPERIPH_DRIVER 宏定义用于区别是否使用 Keil RTE 功能，在未使用 Keil RTE 功能时开启这个宏定义可规避 Keil-MDK 的某些版本误开启 _RTE_的错误问题。

③ 配置头文件中（at32f403a_407_conf.h），定义了外设模块开启的宏定义，可用于控制外设模块的使用，关闭时只需屏蔽掉外设对应的_MODULE_ENABLED 宏定义即可，如下图所示：

图 26. 外设使能宏定义

```
#define CRM_MODULE_ENABLED
#define TMR_MODULE_ENABLED
#define RTC_MODULE_ENABLED
#define BPR_MODULE_ENABLED
#define GPIO_MODULE_ENABLED
#define I2C_MODULE_ENABLED
#define USART_MODULE_ENABLED
#define PWC_MODULE_ENABLED
#define CAN_MODULE_ENABLED
#define ADC_MODULE_ENABLED
#define DAC_MODULE_ENABLED
#define SPI_MODULE_ENABLED
#define DMA_MODULE_ENABLED
#define DEBUG_MODULE_ENABLED
#define FLASH_MODULE_ENABLED
#define CRC_MODULE_ENABLED
#define WWDT_MODULE_ENABLED
#define WDT_MODULE_ENABLED
#define EXINT_MODULE_ENABLED
#define SDIO_MODULE_ENABLED
#define XMC_MODULE_ENABLED
#define USB_MODULE_ENABLED
#define ACC_MODULE_ENABLED
#define MISC_MODULE_ENABLED
#define EMAC_MODULE_ENABLED
```

at32f403a_407_conf.h 同时也定义了外部高速时钟大小 HEXT_VALUE，更换外部高速晶振时须注意这里 HEXT_VALUE 同步修改。

④ 系统时钟配置文件（at32f403a_407_clock.c/.h），配置了默认的系统时钟频率及时钟路径。用户如有自定义需求时可自行修改倍频流程及系数，后续也可结合 ArteryTek 提供的时钟配置上位机来生成相应的时钟配置文件。

4.2 BSP 规范

BSP 按照以下章节所描述的规范进行编写。

4.2.1 外设缩写

表 2. 外设缩写对应表

外设缩写	外设
ADC	模拟/数字转换器
BPR	电池供电域
CAN	控制器局域网模块
CRC	CRC 计算单元
CRM	时钟和复位管理
DAC	数字/模拟转换器
DMA	直接存储器访问（DMA）控制器
DEBUG	调试
EXINT	外部中断/事件控制器
GPIO	通用功能输入输出

外设缩写	外设
IOMUX	复用功能输入输出
I2C	串行外设口
NVIC	嵌套的向量式中断控制器
PWC	电源控制
RTC	实时时钟
SPI	串行外设口
I2S	音频接口
SysTick	系统滴答
TMR	定时器
USART	通用同步异步收发器
WDT	看门狗
WWDT	窗口看门狗
XMC	外部存储控制器

4.2.2 命名规则

BSP 遵从以下命名规则

ip 表示任一外设缩写，例如：ADC，TMR，GPIO 等，小写含义相同，例如 adc,tmr,gpio…

- 源程序文件
  以**“at32fxxx_ip.c”**作为开头,例如 at32f403a_407_adc.c
- 头文件
  以**“at32fxxx_ip.h”**作为开头，例如：at32f403a_407_adc.h
- 常量
  被应用于一个文件的，定义于该文件中； 被应用于多个文件的，在对应头文件中定义。
  所有常量都由英文字母大写书写。
- 变量
  被应用于一个文件的，定义于该文件中； 被应用于多个文件的，在对应头文件中会用 extern 进行声明。
- 函数命名规则
  外设函数的命名以“**外设缩写_属性_动作**”或**外设缩写_动作**”为基本规则，常见的函数名如下：

外设复位函数	ip_reset，	例如 adc_reset
外设使能函数	ip_enable ，	例如 adc_enable
外设结构体反初始化函数	ip_default_para_init ，	例如 spi_default_para_init
外设初始化函数	ip_init，	例如 spi_init
外设中断开启函数	ip_interrupt_enable ，	例如 adc_interrupt_enable
外设标志位获取函数	ip_flag_get ，	例如 adc_flag_get
外设标志位清除函数	ip_flag_clear ，	例如 adc_flag_clear

4.2.3 编码规则

本章节描述了固件库函数的编码规则。

变量类型

```
typedef int32_t  INT32;
typedef int16_t  INT16;
typedef int8_t   INT8;
```

```
typedef uint32_t UINT32;
typedef uint16_t UINT16;
typedef uint8_t  UINT8;

typedef int32_t  s32;
typedef int16_t  s16;
typedef int8_t   s8;

typedef const int32_t sc32;   /*!< read only */
typedef const int16_t sc16;   /*!< read only */
typedef const int8_t  sc8;    /*!< read only */

typedef __IO int32_t  vs32;
typedef __IO int16_t  vs16;
typedef __IO int8_t   vs8;

typedef __I int32_t vsc32;    /*!< read only */
typedef __I int16_t vsc16;    /*!< read only */
typedef __I int8_t  vsc8;     /*!< read only */

typedef uint32_t u32;
typedef uint16_t u16;
typedef uint8_t  u8;

typedef const uint32_t uc32;  /*!< read only */
typedef const uint16_t uc16;  /*!< read only */
typedef const uint8_t  uc8;   /*!< read only */

typedef __IO uint32_t vu32;
typedef __IO uint16_t vu16;
typedef __IO uint8_t  vu8;

typedef __I uint32_t vuc32;   /*!< read only */
typedef __I uint16_t vuc16;   /*!< read only */
typedef __I uint8_t  vuc8;    /*!< read only */
```

4.2.3.1 标志位类型

```
typedef enum {RESET = 0, SET = !RESET} flag_status;
```

4.2.3.2 功能状态类型

```
typedef enum {FALSE = 0, TRUE = !FALSE} confirm_state;
```

4.2.3.3 错误标志位类型

```
typedef enum {ERROR = 0, SUCCESS = !ERROR} error_status;
```

4.2.3.4 外设类型

① 外设

在 at32fxxx_ip.h 定义外设基地址，例如 at32f403a_407.h 的定义如下

```
#define ADC1_BASE                          (APB2PERIPH_BASE + 0x2400)
#define ADC2_BASE                          (APB2PERIPH_BASE + 0x2800)
```

在 at32fxxx_ip.h 外设类型，例如 at32f403a_407_adc.h 的定义如下

```
#define ADC1                               ((adc_type *) ADC1_BASE)
#define ADC2                               ((adc_type *) ADC2_BASE)
```

② 外设寄存器和 bit 位

在 at32fxxx_ip.h 外设类型，例如 at32f403a_407_adc.h 的定义如下

```
/**
  * @brief type define adc register all
  */
typedef struct
{

  /**
    * @brief adc sts register, offset:0x00
    */
  union
  {
    __IO uint32_t sts;
    struct
    {
      __IO uint32_t vmor                 : 1; /* [0] */
      __IO uint32_t cce                  : 1; /* [1] */
      __IO uint32_t pcce                 : 1; /* [2] */
      __IO uint32_t pccs                 : 1; /* [3] */
      __IO uint32_t occs                 : 1; /* [4] */
      __IO uint32_t reserved1            : 27;/* [31:5] */
    } sts_bit;
  };
…
…
…
  /**
  * @brief adc odt register, offset:0x4C
  */
  union
  {
    __IO uint32_t odt;
    struct
    {
```

```
        __IO uint32_t odt                    : 16; /* [15:0] */
        __IO uint32_t adc2odt                : 16; /* [31:16] */
      } odt_bit;
    };

} adc_type;
```

③ 外设寄存器访问示例

寄存器读　　i = ADC1-> ctrl1;

寄存器写　　ADC1-> ctrl1 = i;

bit 5 按位域方式读　　i = ADC1-> ctrl1. cceien;

bit 5 按位域方式写 1　　ADC1-> ctrl1. cceien= TRUE;

bit 5 直接写 1　　ADC1-> ctrl1 |= 1<<5;

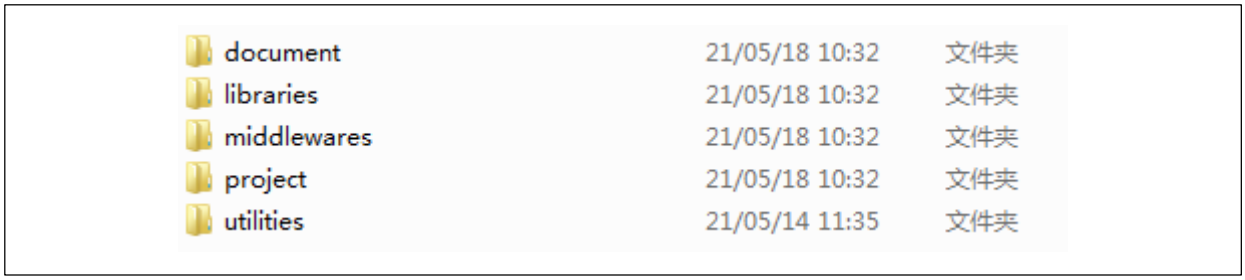
bit 5 直接写 0　　ADC1-> ctrl1&= ~(1<<5) ;

4.3 BSP 结构

4.3.1 BSP 文件夹结构

BSP(Board Support Package)中内容结构大致如下图所示：

图 27. BSP 内容结构



document：

- AT32Fxxx 固件库 BSP&Pack 应用指南.pdf：　　对应型号的 BSP/Pack 应用指南
- ReleaseNotes_AT32F403A_407_Firmware_Library.pdf：　进版记录

libraries：

- drivers：外设驱动

  src 文件夹　每个外设的底层驱动源文件：at32fxxx_ip.c

  inc 文件夹　每个外设的底层驱动头文件：at32fxxx_ip.h

- cmsis：内核相关文件

  cm4 文件夹　内核相关文件。包括 cortex-m4 库文件、系统初始化文件、启动文件等

  dsp 文件夹　dsp 库相关文件

middlewares：

第三方软件包或公用协议包。如 USB 协议层驱动、网络协议层驱动、操作系统源码等。

project：

examples：型号相关的示例 demo。

templates：模板工程。包括 Keil4 、keil5 、IAR6、 IAR7、 IAR8 及 eclipse_gcc

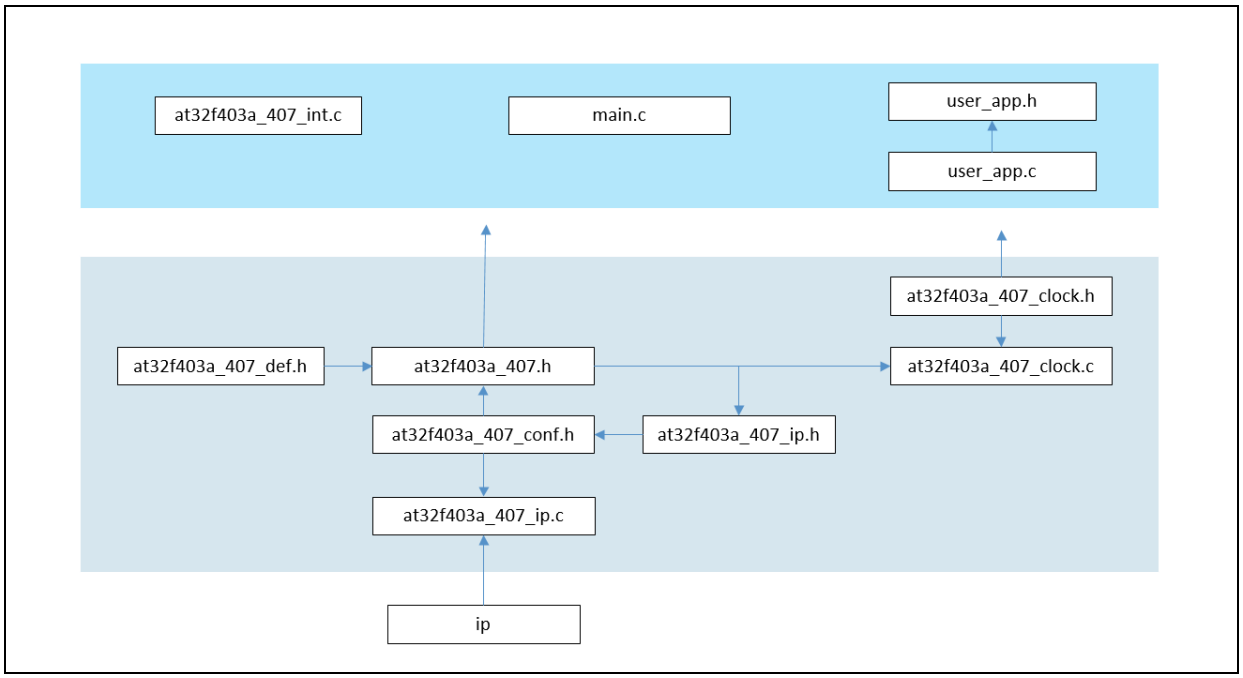
utilities：

各经典应用案例存放目录。

4.3.2 BSP 库函数文件描述

BSP 函数库的架构如下图

图 28. BSP 函数库的架构



BSP 函数库文件描述如下表

表 3. BSP 函数库文件描述

文件名	描述
at32f403a_407_conf.h	外设模块开启的宏定义，外部高速时钟 HEXT_VALUE 的宏定义
main.c	主函数
at32f403a_407_ip.c	外设驱动源文件，例如 at32f403a_407_adc.c
at32f403a_407_ip.h	外设驱动头文件，例如 at32f403a_407_adc.h
at32f403a_407.h	系列芯片头文件中（at32f403a_407.h），USE_STDPERIPH_DRIVER 宏定义用于区别是否使用 Keil RTE 功能，在未使用 Keil RTE 功能时开启这个宏定义可规避 Keil-MDK 的某些版本误开启_RTE_的错误问题
at32f403a_407_clock.c	时钟配置文件，设置了默认的时钟频率及时钟路径
at32f403a_407_clock.h	时钟配置头文件
at32f403a_407_int.c	中断函数源文件，默认编写了部分内核中断函数的代码流程。
at32f403a_407_int.h	中断函数头文件
at32f403a_407_misc.c	其他配置源文件，如 nvic 配置函数，systick 时钟源选择
at32f403a_407_misc.h	其他配置头文件
startup_at32f403a_407.s	启动文件

4.3.3 外设初始化和设置

本节以 GPIO 举例，描述了如何进行初始化和设置。

GPIO 做普通输入输出的初始化

Step 1: 定义 gpio_init_type 结构体，示例如下

```
gpio_init_type gpio_init_struct;
```

Step 2: 调用 crm_periph_clock_enable 函数，开启对应 GPIO 时钟

Step 3： 反初始化 gpio_init_struct 结构体，这样可以保证其他成员的值（多为 default 值）被正确填入。示例如下

```
gpio_default_para_init(&gpio_init_struct);
```

Step 4： 配置结构体成员，并将结构体参数通过 gpio_init 写入到 GPIO 寄存器

示例如下：

```
gpio_init_struct.gpio_pins = GPIO_PINS_2 | GPIO_PINS_3;
gpio_init_struct.gpio_mode = GPIO_MODE_OUTPUT;
gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
gpio_init_struct.gpio_drive_strength    = GPIO_DRIVE_STRENGTH_STRONGER;
gpio_init(GPIOA, &gpio_init_struct);
```

更多外设的初始化流程可参考 Reference Manual 外设的功能描述章节，以及 AT32Fxxx_Firmware_Library_V2.x.x.zip\project\at_start_fxxx\examples 中各外设的初始化流程和方法。

4.3.4 外设库函数格式

函数的描述按如下格式进行

表 4. 外设库函数格式

项目	描述
函数名	函数名称
函数原型	函数原形声明
功能描述	函数要实现的功能简要描述
输入参数 n	输入参数描述
输出参数 n	输出参数描述
返回值	函数的返回值
先决条件	调用函数前应满足的要求
被调用函数	其他被该函数调用的库函数

5 AT32F413 外设库函数概述

5.1 HICK 自动时钟校准（ACC）

ACC 寄存器结构 acc_type，定义于文件“at32f413_acc.h”如下：

```
/**
  * @brief type define acc register all
  */
typedef struct
{
      ……
} acc_type;
```

下表给出了 ACC 寄存器总览：

表 5.ACC 寄存器对应表

寄存器	描述
acc_sts	ACC 状态寄存器
acc_ctrl1	ACC 控制寄存器 1
acc_ctrl2	ACC 控制寄存器 2
acc_c1	ACC 比较值寄存器 1
acc_c2	ACC 比较值寄存器 2
acc_c3	ACC 比较值寄存器 3

下表给出了 ACC 库函数总览：

表 6.ACC 库函数总览

函数名	描述
acc_calibration_mode_enable	ACC 校准模式使能
acc_step_set	ACC 校准步长配置函数
acc_interrupt_enable	ACC 中断使能函数
acc_hicktrim_get	获取 ACC 精校验值
acc_hickcal_get	获取 ACC 粗校验值
acc_write_c1	写 ACC C1 寄存器值
acc_write_c2	写 ACC C2 寄存器值
acc_write_c3	写 ACC C3 寄存器值
acc_read_c1	读 ACC C1 寄存器值
acc_read_c2	读 ACC C2 寄存器值
acc_read_c3	读 ACC C3 寄存器值
acc_flag_get	ACC 中断标志位获取
acc_flag_clear	ACC 中断标志位清除

5.1.1 函数 acc_calibration_mode_enable

下表描述了函数 acc_calibration_mode_enable

表 7.函数 acc_calibration_mode_enable

项目	描述
函数名	acc_calibration_mode_enable
函数原型	void acc_calibration_mode_enable(uint16_t acc_trim, confirm_state new_state);
功能描述	ACC 校准模式使能
输入参数 1	acc_trim：校验模式选择，可选择： ACC_CAL_HICKCAL 或 ACC_CAL_HICKTRIM
输入参数 2	new_state：开启或关闭 ACC
输出参数	无
返回值	无
先决条件	无
被调用函数	无

acc_trim

校验模式选择

ACC_CAL_HICKCAL： 粗检验模式

ACC_CAL_HICKTRIM： 精校验模式

new_state

选择使能还是关闭 ACC

FALSE： 关闭中断

TRUE： 使能中断

示例

```
/* open acc calibration */
acc_calibration_mode_enable(ACC_CAL_HICKTRIM, TRUE);
```

函数 acc_step_set

下表描述了函数 acc_step_set

表 8.函数 acc_step_set

项目	描述
函数名	acc_step_set
函数原型	void acc_step_set(uint8_t step_value);
功能描述	ACC 校准步长配置
输入参数 1	step_value:校准步长设置
输出参数	无
返回值	无
先决条件	无
被调用函数	无

step_value

这 4 位定义了每次校准改变的值。

备注：为了获得更高的校准精度，建议将 step 设为 1。当 ENTRIM=0，仅校准 HICKCAL，若 step 改变 1，对应的 HICKCAL 也改变 1，HICK 频率改变 40KHz（设计值），为正相关关系。当 ENTRIM=1，仅校准 HICKTRIM，若 step 改变 1，对应的 HICKTRIM 也改变 1，HICK 频率改变 20KHz（设计值），为正相关关系。

示例

```
/* set acc step value */
acc_step_set(0x1);
```

5.1.2 函数 acc_interrupt_enable

下表描述了函数 acc_interrupt_enable

表 9.函数 acc_interrupt_enable

项目	描述
函数名	dma_ interrupt_enable
函数原型	void acc_interrupt_enable(uint16_t acc_int, confirm_state new_state);
功能描述	使能 acc 中断
输入参数 1	acc_int：中断源选择
输入参数 2	new_state：使能或关闭中断
输出参数	无
返回值	无
先决条件	无
被调用函数	无

acc_int

选择中断源

ACC_CALRDYIEN_INT： 校准完成中断

ACC_EIEN_INT： 参考信号丢失中断

new_state

选择中断是使能还是关闭

FALSE： 关闭中断

TRUE： 使能中断

示例

```
/* enable the acc reference signal lost interrupt */
acc_interrupt_enable(ACC_EIEN_INT, TRUE);
```

5.1.3 函数 acc_hicktrim_get

下表描述了函数 acc_hicktrim_get

表 10.函数 acc_hicktrim_get

项目	描述
函数名	acc_hicktrim_get
函数原型	uint8_t acc_hicktrim_get(void);
功能描述	获取 acc 精校验值
输入参数	无
输出参数	无
返回值	获取的 acc 精校验值
先决条件	无
被调用函数	无

示例

```
/* get trim value*/
uint8_t trim_value;
trim_value = acc_hicktrim_get();
```

5.1.4 函数 acc_hickcal_get

下表描述了函数 acc_hicktrim_get

表 11.函数 acc_hickcal_get

项目	描述
函数名	acc_hickcal_get
函数原型	uint8_t acc_hickcal_get(void);
功能描述	获取 acc 粗校验值
输入参数	无
输出参数	无
返回值	获取的 acc 粗校验值
先决条件	无
被调用函数	无

示例

```
/* get cal value*/
uint8_t cal_value;
cal_value = acc_hickcal_get ();
```

5.1.5 函数 acc_write_c1

下表描述了函数 acc_write_c1

表 12.函数 acc_write_c1

项目	描述
函数名	acc_write_c1
函数原型	void acc_write_c1(uint16_t acc_c1_value);
功能描述	写 ACC C1 寄存器值
输入参数	acc_c1_value：写入 C1 的值
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* update the c1 value */
acc_c2_value = 8000;
acc_write_c1(acc_c2_value - 10);
```

5.1.6 函数 acc_write_c2

下表描述了函数 acc_write_c2

表 13.函数 acc_write_c2

项目	描述
函数名	acc_write_c2
函数原型	void acc_write_c2(uint16_t acc_c2_value);
功能描述	写 ACC C2 寄存器值
输入参数	acc_c2_value：写入 C2 的值
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* update the c2 value */
acc_c2_value = 8000;
acc_write_c2(acc_c2_value - 10);
```

5.1.7 函数 acc_write_c3

下表描述了函数 acc_write_c3

表 14.函数 acc_write_c3

项目	描述
函数名	acc_write_c3
函数原型	void acc_write_c3(uint16_t acc_c3_value);
功能描述	写 ACC C3 寄存器值
输入参数	acc_c3_value：写入 C3 的值
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* update the c3 value */
acc_c2_value = 8000;
acc_write_c3(acc_c2_value - 10);
```

5.1.8 函数 acc_read_c1

下表描述了函数 acc_read_c1

表 15.函数 acc_read_c1

项目	描述
函数名	acc_read_c1
函数原型	uint16_t acc_read_c1(void);
功能描述	读 ACC C1 寄存器值
输入参数	无
输出参数	无
返回值	ACC C1 的值

项目	描述
先决条件	无
被调用函数	无

示例

```
/* get the c1 value */
uint16_t    acc_c1_value;
acc_c1_value = acc_read_c1();
```

5.1.9 函数 acc_read_c2

下表描述了函数 acc_read_c2

表 16.函数 acc_read_c2

项目	描述
函数名	acc_read_c2
函数原型	uint16_t acc_read_c2(void);
功能描述	读 ACC C2 寄存器值
输入参数	无
输出参数	无
返回值	ACC C2 的值
先决条件	无
被调用函数	无

示例

```
/* get the c2 value */
uint16_t    acc_c2_value;
acc_c2_value = acc_read_c2();
```

5.1.10 函数 acc_read_c3

下表描述了函数 acc_read_c3

表 17.函数 acc_read_c3

项目	描述
函数名	acc_read_c3
函数原型	uint16_t acc_read_c3(void);
功能描述	读 ACC C3 寄存器值
输入参数	无
输出参数	无
返回值	ACC C3 的值
先决条件	无
被调用函数	无

示例

```
/* get the c3 value */
uint16_t    acc_c3_value;
acc_c3_value = acc_read_c3();
```

5.1.11 函数 acc_flag_get

下表描述了函数 acc_flag_get

表 18.函数 acc_flag_get

项目	描述
函数名	acc_flag_get
函数原型	flag_status acc_flag_get(uint16_t acc_flag);
功能描述	获取 acc 标志位
输入参数 1	acc_flag：需要获取的标志位
输出参数	无
返回值	flag_status：标志位是否置起
先决条件	无
被调用函数	无

acc_flag

acc_flag 用于选择需要获取状态的标志，其可选参数罗列如下

ACC_RSLOST_FLAG：参考信号丢失中断

ACC_CALRDY_FLAG：校准完成中断

flag_status

RESET：相应标志位未置起

SET：　　相应标志位置起

示例

```
if(acc_flag_get(ACC_CALRDY_FLAG) != RESET)
{
  at32_led_toggle(LED2);
  /* clear acc calibration ready flag */
  acc_flag_clear(ACC_CALRDY_FLAG);
}
```

5.1.12 函数 acc_flag_clear

下表描述了函数 acc_flag_clear

表 19.函数 acc_flag_clear

项目	描述
函数名	acc_flag_clear
函数原型	void acc_flag_clear(uint16_t acc_flag);
功能描述	清除 acc 标志位
输入参数 1	acc_flag：需要清除的标志位
输出参数	无
返回值	无
先决条件	无
被调用函数	无

acc_flag

acc_flag 用于选择需要清除的标志，其可选参数罗列如下

ACC_RSLOST_FLAG：参考信号丢失中断

ACC_CALRDY_FLAG：校准完成中断

示例

```
if(acc_flag_get(ACC_CALRDY_FLAG) != RESET)
{
  at32_led_toggle(LED2);
  /* clear acc calibration ready flag */
  acc_flag_clear(ACC_CALRDY_FLAG);
}
```

5.2 模拟数字/转换器（ADC）

ADC 寄存器结构 adc_type，定义于文件“at32f413_adc.h”如下：

```
/**
  * @brief type define adc register all
  */
typedef struct
{
      ……
} adc_type;
```

下表给出了 ADC 寄存器总览：

表 20. ADC 寄存器对应表

寄存器	描述
sts	ADC 状态寄存器
ctrl1	ADC 控制寄存器 1
ctrl2	ADC 控制寄存器 2
spt1	ADC 采样时间寄存器 1
spt2	ADC 采样时间寄存器 2
pcdto1	ADC 抢占通道数据偏移寄存器 1
pcdto2	ADC 抢占通道数据偏移寄存器 2
pcdto3	ADC 抢占通道数据偏移寄存器 3
pcdto4	ADC 抢占通道数据偏移寄存器 4
vmhb	ADC 电压监测高边界寄存器
vmlb	ADC 电压监测低边界寄存器
osq1	ADC 普通序列寄存器 1
osq2	ADC 普通序列寄存器 2
osq3	ADC 普通序列寄存器 3
psq	ADC 抢占序列寄存器
pdt1	ADC 抢占数据寄存器 1
pdt2	ADC 抢占数据寄存器 2
pdt3	ADC 抢占数据寄存器 3
pdt4	ADC 抢占数据寄存器 4
odt	ADC 普通数据寄存器

下表给出了 ADC 库函数总览：

表 21. ADC 库函数总览

函数名	描述
adc_reset	复位 ADC 使其所有寄存器保持复位值
adc_enable	A/D 转换器使能
adc_combine_mode_select	主从组合模式选择
adc_base_default_para_init	为 adc_base_struct 指定初始默认值
adc_base_config	将 adc_base_struct 中指定的参数初始化到外设 ADC 的寄存器
adc_dma_mode_enable	普通通道转换数据的 DMA 传输使能
adc_interrupt_enable	被选择的 ADC 事件中断使能
adc_calibration_init	初始化校准
adc_calibration_init_status_get	初始化校准状态获取
adc_calibration_start	开始校准
adc_calibration_status_get	校准状态获取
adc_voltage_monitor_enable	普通/抢占通道的电压监测使能及单个通道的电压监测使能
adc_voltage_monitor_threshold_value_set	电压监测高低边界设定
adc_voltage_monitor_single_channel_select	单个通道电压监测功能下待监测通道选择
adc_ordinary_channel_set	普通通道设定，包括通道选择、转换序列编号及采样时间
adc_preempt_channel_length_set	抢占转换序列长度设定
adc_preempt_channel_set	抢占通道设定，包括通道选择、转换序列编号及采样时间
adc_ordinary_conversion_trigger_set	普通通道组转换的触发模式使能及触发事件选择
adc_preempt_conversion_trigger_set	抢占通道组转换的触发模式使能及触发事件选择
adc_preempt_offset_value_set	抢占通道转换数据偏移量设定
adc_ordinary_part_count_set	分割模式下每次触发转换的普通通道个数设定
adc_ordinary_part_mode_enable	普通通道上的分割模式使能
adc_preempt_part_mode_enable	抢占通道上的分割模式使能
adc_preempt_auto_mode_enable	普通通道组转换结束后的抢占组自动转换使能
adc_tempersensor_vintrv_enable	内部温度传感器及 V_{INTRV} 使能
adc_ordinary_software_trigger_enable	软件触发普通通道转换
adc_ordinary_software_trigger_status_get	获取软件触发的普通通道转换状态
adc_preempt_software_trigger_enable	软件触发抢占通道转换
adc_preempt_software_trigger_status_get	获取软件触发的抢占通道转换状态
adc_ordinary_conversion_data_get	获取非主从模式下普通通道转换数据
adc_combine_ordinary_conversion_data_get	获取主从组合模式下普通通道转换数据
adc_preempt_conversion_data_get	获取抢占通道转换数据
adc_flag_get	获取标志位状态
adc_flag_clear	清除已置位的标志位

5.2.1 函数 adc_reset

下表描述了函数 adc_reset

表 22. 函数 adc_reset

项目	描述
函数名	adc_reset
函数原型	void adc_reset(adc_type *adc_x)

项目	描述
功能描述	复位 ADC 使其所有寄存器保持复位值
输入参数	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset()

示例

```
/* deinitialize adc1 */
adc_reset(ADC1);
```

5.2.2 函数 adc_enable

下表描述了函数 adc_enable

表 23. 函数 adc_enable

项目	描述
函数名	adc_enable
函数原型	void adc_enable(adc_type *adc_x, confirm_state new_state)
功能描述	设定 A/D 转换器使能状态为关闭或开启
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	new_state：A/D 转换器的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable adc1 */
adc_enable(ADC1, TRUE);
```

注意：ADC 已处于使能状态时，再调用此 adc_enable 函数将会导致普通通道响应转换。

5.2.3 函数 adc_combine_mode_select

下表描述了函数 adc_combine_mode_select

表 24. 函数 adc_combine_mode_select

项目	描述
函数名	adc_combine_mode_select
函数原型	void adc_combine_mode_select(adc_combine_mode_type combine_mode)
功能描述	选择 ADC1 的主从组合模式
输入参数	combine_mode：ADC1 支持的主从模式 该参数可以选取 adc_combine_mode_type 内的任意一个枚举值.
输出参数	无

项目	描述
返回值	无
先决条件	无
被调用函数	无

combine_mode

combine_mode 用于选择主从模式，其可选参数罗列如下

ADC_INDEPENDENT_MODE：非主从模式

ADC_ORDINARY_SMLT_PREEMPT_SMLT_MODE：混合普通同时+抢占同时模式

ADC_ORDINARY_SMLT_PREEMPT_INTERLTRIG_MODE：混合普通同时+抢占交错触发模式

ADC_ORDINARY_SHORTSHIFT_PREEMPT_SMLT_MODE：混合抢占同时+普通短位移模式

ADC_ORDINARY_LONGSHIFT_PREEMPT_SMLT_MODE：混合抢占同时+普通长位移模式

ADC_PREEMPT_SMLT_ONLY_MODE：抢占同时模式

ADC_ORDINARY_SMLT_ONLY_MODE：普通同时模式

ADC_ORDINARY_SHORTSHIFT_ONLY_MODE：普通短位移模式

ADC_ORDINARY_LONGSHIFT_ONLY_MODE：普通长位移模式

ADC_PREEMPT_INTERLTRIG_ONLY_MODE：抢占交错触发模式

示例

```
/* select combine mode as independent mode */
adc_combine_mode_select(ADC_INDEPENDENT_MODE);
```

注意：adc_combine_mode_select 函数仅适用于 ADC1，其对 ADC2 无效。

5.2.4 函数 adc_base_default_para_init

下表描述了函数 adc_base_default_para_init

表 25. 函数 adc_base_default_para_init

项目	描述
函数名	adc_ base_default_para_init
函数原型	void adc_base_default_para_init(adc_base_config_type *adc_base_struct)
功能描述	为 adc_base_struct 指定初始默认值
输入参数	adc_base_struct：指向结构体 adc_base_config_type 的指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_base_struct 成员的初始默认值如下

sequence_mode：FALSE

repeat_mode：FALSE

data_align：ADC_RIGHT_ALIGNMENT

ordinary_channel_length：1

示例

```
/* initialize a adc_base_config_type structure */
adc_base_config_type adc_base_struct;
adc_base_default_para_init(&adc_base_struct);
```

5.2.5 函数 adc_base_config

下表描述了函数 adc_base_config

表 26. 函数 adc_base_config

项目	描述
函数名	adc_base_config
函数原型	void adc_base_config(adc_type *adc_x, adc_base_config_type *adc_base_struct);
功能描述	将 adc_base_struct 中指定的参数初始化到外设 ADC 的寄存器
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_base_struct：指向 adc_base_config_type 类型的结构体指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_base_config_type structure

adc_base_config_type 定义在 at32f413_adc.h 中

```
typedef struct
{
  confirm_state          sequence_mode;
  confirm_state          repeat_mode;
  adc_data_align_type    data_align;
  uint8_t                ordinary_channel_length;
} adc_base_config_type;
```

如下是对成员中各个参数的说明

sequence_mode

设置 ADC 工作的序列模式

FALSE： 转换选择的单一通道

TRUE： 转换设定的多个通道

repeat_mode

设置 ADC 工作的反复模式

FALSE： SQEN=0 时，每次触发转换单个通道，SQEN=1 时，每次触发转换一组通道

TRUE： SQEN =0 时，一次触发后将反复转换单个通道，SQEN=1 时，一次触发后将反复转换一组通道。直到 ADCEN 被清零。

data_align

设置 ADC 工作的数据对齐方式

ADC_RIGHT_ALIGNMENT： 右对齐

ADC_LEFT_ALIGNMENT： 左对齐

ordinary_channel_length

设置 ADC 工作的普通转换序列长度

示例

```
adc_base_config_type adc_base_struct;
adc_base_struct.sequence_mode = TRUE;
adc_base_struct.repeat_mode = FALSE;
adc_base_struct.data_align = ADC_RIGHT_ALIGNMENT;
```

```
adc_base_struct.ordinary_channel_length = 3;
adc_base_config(ADC1, &adc_base_struct);
```

5.2.6 函数 adc_dma_mode_enable

下表描述了函数 adc_dma_mode_enable

表 27. 函数 adc_dma_mode_enable

项目	描述
函数名	adc_dma_mode_enable
函数原型	void adc_dma_mode_enable(adc_type *adc_x, confirm_state new_state)
功能描述	普通通道转换数据的 DMA 传输使能
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	new_state：DMA 传输普通通道数据的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable dma transfer adc ordinary conversion data */
adc_dma_mode_enable(ADC1, TRUE);
```

注意：adc_dma_mode_enable 函数对 ADC2 无效，其仅适用于 ADC1。

5.2.7 函数 adc_interrupt_enable

下表描述了函数 adc_interrupt_enable

表 28. 函数 adc_interrupt_enable

项目	描述
函数名	adc_interrupt_enable
函数原型	void adc_interrupt_enable(adc_type *adc_x, uint32_t adc_int, confirm_state new_state)
功能描述	被选择的 ADC 事件中断使能
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_int：ADC 事件中断选择 该参数可以选取 ADC 支持的任意事件中断.
输入参数 3	new_state：ADC 事件中断的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_int

adc_int 用于选择需要设定状态的事件中断，其可选参数罗列如下
ADC_CCE_INT： 通道转换结束中断使能
ADC_VMOR_INT： 电压监测超出范围中断使能
ADC_PCCE_INT： 抢占通道组转换结束中断使能

示例

```
/* enable voltage monitoring out of range interrupt */
adc_interrupt_enable(ADC1, ADC_VMOR_INT, TRUE);
```

5.2.8 函数 adc_calibration_init

下表描述了函数 adc_calibration_init

表 29. 函数 adc_calibration_init

项目	描述
函数名	adc_calibration_init
函数原型	void adc_calibration_init(adc_type *adc_x)
功能描述	初始化校准
输入参数	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* initialize A/D calibration */
adc_calibration_init(ADC1);
```

5.2.9 函数 adc_calibration_init_status_get

下表描述了函数 adc_calibration_init_status_get

表 30. 函数 adc_calibration_init_status_get

项目	描述
函数名	adc_calibration_init_status_get
函数原型	flag_status adc_calibration_init_status_get(adc_type *adc_x)
功能描述	初始化校准状态获取
输入参数	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输出参数	无
返回值	flag_status：初始化校准的状态 该返回值可为罗列的其中之一：SET，RESET.
先决条件	无
被调用函数	无

示例

```
/* wait initialize A/D calibration success */
while(adc_calibration_init_status_get(ADC1));
```

5.2.10 函数 adc_calibration_start

下表描述了函数 adc_calibration_start

表 31. 函数 adc_calibration_start

项目	描述
函数名	adc_calibration_start
函数原型	void adc_calibration_start(adc_type *adc_x)
功能描述	开始校准
输入参数	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* start calibration process */
adc_calibration_start(ADC1);
```

5.2.11 函数 adc_calibration_status_get

下表描述了函数 adc_calibration_status_get

表 32. 函数 adc_calibration_status_get

项目	描述
函数名	adc_calibration_status_get
函数原型	flag_status adc_calibration_status_get(adc_type *adc_x)
功能描述	校准状态获取
输入参数	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输出参数	无
返回值	flag_status：校准的状态 该返回值可为罗列的其中之一：SET，RESET.
先决条件	无
被调用函数	无

示例

```
/* wait calibration success */
while(adc_calibration_status_get(ADC1));
```

5.2.12 函数 adc_voltage_monitor_enable

下表描述了函数 adc_voltage_monitor_enable

表 33. 函数 adc_voltage_monitor_enable

项目	描述
函数名	adc_voltage_monitor_enable

项目	描述
函数原型	void adc_voltage_monitor_enable(adc_type *adc_x, adc_voltage_monitoring_type adc_voltage_monitoring)
功能描述	普通/抢占通道的电压监测使能及单个通道的电压监测使能
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_voltage_monitoring：普通/抢占通道组及单个通道选择 该参数可以选取 adc_voltage_monitoring_type 内的任意一个枚举值.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_voltage_monitoring

adc_voltage_monitoring 用于设置电压监测作用范围为普通/抢占通道组的一个或多个通道，其可选参数罗列如下

ADC_VMONITOR_SINGLE_ORDINARY：电压监测作用于单个普通通道
ADC_VMONITOR_SINGLE_PREEMPT：电压监测作用于单个抢占通道
ADC_VMONITOR_SINGLE_ORDINARY_PREEMPT：电压监测作用于单个普通或抢占通道
ADC_VMONITOR_ALL_ORDINARY：电压监测作用于所有普通通道
ADC_VMONITOR_ALL_PREEMPT：电压监测作用于所有抢占通道
ADC_VMONITOR_ALL_ORDINARY_PREEMPT：电压监测作用于所有普通和抢占通道
ADC_VMONITOR_NONE：电压监测不作用于任何通道

示例

```
/* enable the voltage monitoring on all ordinary and preempt channels */
adc_voltage_monitor_enable(ADC1, ADC_VMONITOR_ALL_ORDINARY_PREEMPT);
```

5.2.13 函数 adc_voltage_monitor_threshold_value_set

下表描述了函数 adc_voltage_monitor_threshold_value_set

表 34. 函数 adc_voltage_monitor_threshold_value_set

项目	描述
函数名	adc_voltage_monitor_threshold_value_set
函数原型	void adc_voltage_monitor_threshold_value_set(adc_type *adc_x, uint16_t adc_high_threshold, uint16_t adc_low_threshold)
功能描述	电压监测高低边界设定
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_high_threshold：设定电压监测的高边界值 该参数可以被设定为 0x000~0xFFF 内的任意数值.
输入参数 3	adc_low_threshold：设定电压监测的低边界值 该参数可以被设定为 0x000~0xFFF 内的任意不大于 adc_high_threshold 的数值.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set voltage monitoring's high and low thresholds value */
adc_voltage_monitor_threshold_value_set(ADC1, 0xBBB, 0xAAA);
```

5.2.14 函数 adc_voltage_monitor_single_channel_select

下表描述了函数 adc_voltage_monitor_single_channel_select

表 35. 函数 adc_voltage_monitor_single_channel_select

项目	描述
函数名	adc_voltage_monitor_single_channel_select
函数原型	void adc_voltage_monitor_single_channel_select(adc_type *adc_x, adc_channel_select_type adc_channel)
功能描述	单个通道电压监测功能下待监测通道选择
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_channel：待监测通道选择 该参数详细描述见 *adc_channel*.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_channel

adc_channel 用于选择待监测通道，其可选参数罗列如下

ADC_CHANNEL_0： ADC 通道 0

ADC_CHANNEL_1： ADC 通道 1

……

ADC_CHANNEL_16： ADC 通道 16

ADC_CHANNEL_17： ADC 通道 17

示例

```
/* select the voltage monitoring's channel */
adc_voltage_monitor_single_channel_select(ADC1, ADC_CHANNEL_5);
```

5.2.15 函数 adc_ordinary_channel_set

下表描述了函数 adc_ordinary_channel_set

表 36. 函数 adc_ordinary_channel_set

项目	描述
函数名	adc_ordinary_channel_set
函数原型	void adc_ordinary_channel_set(adc_type *adc_x, adc_channel_select_type adc_channel, uint8_t adc_sequence, adc_sampletime_select_type adc_sampletime)
功能描述	普通通道设定，包括通道选择、转换序列编号及采样时间
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.

项目	描述
输入参数 2	adc_channel：待配置通道选择 该参数详细描述见 *adc_channel*.
输入参数 3	adc_sequence：通道转换序列设定 该参数可以被设定为 1~16 内的任意数值.
输入参数 4	adc_sampletime：通道采样时间设定 该参数详细描述见 *adc_sampletime*.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_sampletime

adc_sampletime 用于设定通道采样时间，其可选参数罗列如下

ADC_SAMPLETIME_1_5： 采样时间为 1.5 个 ADCCLK 周期

ADC_SAMPLETIME_7_5： 采样时间为 7.5 个 ADCCLK 周期

ADC_SAMPLETIME_13_5： 采样时间为 13.5 个 ADCCLK 周期

ADC_SAMPLETIME_28_5： 采样时间为 28.5 个 ADCCLK 周期

ADC_SAMPLETIME_41_5： 采样时间为 41.5 个 ADCCLK 周期

ADC_SAMPLETIME_55_5： 采样时间为 55.5 个 ADCCLK 周期

ADC_SAMPLETIME_71_5： 采样时间为 71.5 个 ADCCLK 周期

ADC_SAMPLETIME_239_5：采样时间为 239.5 个 ADCCLK 周期

示例

```
/* set ordinary channel's corresponding rank in the sequencer and sample time */
adc_ordinary_channel_set(ADC1, ADC_CHANNEL_4, 1, ADC_SAMPLETIME_239_5);
adc_ordinary_channel_set(ADC1, ADC_CHANNEL_5, 2, ADC_SAMPLETIME_239_5);
```

5.2.16 函数 adc_preempt_channel_length_set

下表描述了函数 adc_preempt_channel_length_set

表 37. 函数 adc_preempt_channel_length_set

项目	描述
函数名	adc_preempt_channel_length_set
函数原型	void adc_preempt_channel_length_set(adc_type *adc_x, uint8_t adc_channel_lenght)
功能描述	抢占转换序列长度设定
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_channel_lenght：抢占转换序列长度设定 该参数可以被设定为 0x1~0x4 内的任意数值.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set preempt channel lenghth */
```

```
adc_preempt_channel_length_set(ADC1, 3);
```

5.2.17 函数 adc_preempt_channel_set

下表描述了函数 adc_preempt_channel_set

表 38. 函数 adc_preempt_channel_set

项目	描述
函数名	adc_preempt_channel_set
函数原型	void adc_preempt_channel_set(adc_type *adc_x, adc_channel_select_type adc_channel, uint8_t adc_sequence, adc_sampletime_select_type adc_sampletime)
功能描述	抢占通道设定，包括通道选择、转换序列编号及采样时间
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_channel：待配置通道选择 该参数详细描述见 *adc_channel*.
输入参数 3	adc_sequence：通道转换序列设定 该参数可以被设定为 1~4 内的任意数值.
输入参数 4	adc_sampletime：通道采样时间设定 该参数详细描述见 *adc_sampletime*.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set ordinary channel's corresponding rank in the sequencer and sample time */
adc_preempt_channel_set(ADC1, ADC_CHANNEL_7, 1, ADC_SAMPLETIME_239_5);
adc_preempt_channel_set(ADC1, ADC_CHANNEL_8, 2, ADC_SAMPLETIME_239_5);
```

5.2.18 函数 adc_ordinary_conversion_trigger_set

下表描述了函数 adc_ordinary_conversion_trigger_set

表 39. 函数 adc_ordinary_conversion_trigger_set

项目	描述
函数名	adc_ordinary_conversion_trigger_set
函数原型	void adc_ordinary_conversion_trigger_set(adc_type *adc_x, adc_ordinary_trig_select_type adc_ordinary_trig, confirm_state new_state)
功能描述	普通通道组转换的触发模式使能及触发事件选择
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_ordinary_trig：普通通道组触发事件选择 该参数可以选取 adc_ordinary_trig_select_type 内的任意一个枚举值.
输入参数 3	new_state：触发模式的预设状态 该参数可以选取自其中之一：TRUE，FALSE.

项目	描述
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_ordinary_trig

adc_ordinary_trig 用于选择普通通道组转换的触发事件，其可选参数罗列如下

ADC1 &ADC2 的触发事件

ADC12_ORDINARY_TRIG_TMR1CH1： TMR1 的 CH1 事件

ADC12_ORDINARY_TRIG_TMR1CH2： TMR1 的 CH2 事件

ADC12_ORDINARY_TRIG_TMR1CH3： TMR1 的 CH3 事件

ADC12_ORDINARY_TRIG_TMR2CH2： TMR2 的 CH2 事件

ADC12_ORDINARY_TRIG_TMR3TRGOUT： TMR3 的 TRGOUT 事件

ADC12_ORDINARY_TRIG_TMR4CH4： TMR4 的 CH4 事件

ADC12_ORDINARY_TRIG_EXINT11_TMR8TRGOUT： EXINT 线 11/TMR8 的 TRGOUT 事件

ADC12_ORDINARY_TRIG_SOFTWARE： 软件触发事件

ADC12_ORDINARY_TRIG_TMR1TRGOUT： TMR1 的 TRGOUT 事件

ADC12_ORDINARY_TRIG_TMR8CH1： TMR8 的 CH1 事件

ADC12_ORDINARY_TRIG_TMR8CH2： TMR8 的 CH2 事件

示例

```
/* set ordinary external trigger event */
adc_ordinary_conversion_trigger_set(ADC1, ADC12_ORDINARY_TRIG_TMR1CH1, TRUE);
```

5.2.19 函数 adc_preempt_conversion_trigger_set

下表描述了函数 adc_preempt_conversion_trigger_set

表 40. 函数 adc_preempt_conversion_trigger_set

项目	描述
函数名	adc_preempt_conversion_trigger_set
函数原型	void adc_preempt_conversion_trigger_set(adc_type *adc_x, adc_preempt_trig_select_type adc_preempt_trig, confirm_state new_state)
功能描述	抢占通道组转换的触发模式使能及触发事件选择
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_ preempt _trig：抢占通道组触发事件选择 该参数可以选取 adc_preempt_trig_select_type 内的任意一个枚举值.
输入参数 3	new_state：触发模式的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_preempt _trig

adc_preempt _trig 用于选择抢占通道组转换的触发事件，其可选参数罗列如下

ADC1 &ADC2 的触发事件

ADC12_PREEMPT_TRIG_TMR1TRGOUT： TMR1 的 TRGOUT 事件
ADC12_PREEMPT_TRIG_TMR1CH4： TMR1 的 CH4 事件
ADC12_PREEMPT_TRIG_TMR2TRGOUT： TMR2 的 TRGOUT 事件
ADC12_PREEMPT_TRIG_TMR2CH1： TMR2 的 CH1 事件
ADC12_PREEMPT_TRIG_TMR3CH4： TMR3 的 CH4 事件
ADC12_PREEMPT_TRIG_TMR4TRGOUT： TMR4 的 TRGOUT 事件
ADC12_PREEMPT_TRIG_EXINT15_TMR8CH4： EXINT 线 15/TMR8 的 CH4 事件
ADC12_PREEMPT_TRIG_SOFTWARE： 软件触发事件
ADC12_PREEMPT_TRIG_TMR1CH1： TMR1 的 CH1 事件
ADC12_PREEMPT_TRIG_TMR8CH1： TMR8 的 CH1 事件
ADC12_PREEMPT_TRIG_TMR8TRGOUT： TMR8 的 TRGOUT 事件

示例

```
/* set preempt external trigger event */
adc_preempt_conversion_trigger_set(ADC1, ADC12_PREEMPT_TRIG_SOFTWARE, TRUE);
```

5.2.20 函数 adc_preempt_offset_value_set

下表描述了函数 adc_preempt_offset_value_set

表 41. 函数 adc_preempt_offset_value_set

项目	描述
函数名	adc_preempt_offset_value_set
函数原型	void adc_preempt_offset_value_set(adc_type *adc_x, adc_preempt_channel_type adc_preempt_channel, uint16_t adc_offset_value)
功能描述	抢占通道转换数据偏移量设定
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_preempt_channel：选择需要设定偏移量的通道 该参数详细描述见 *adc_preempt_channel*.
输入参数 3	adc_offset_value：设定通道偏移量值 该参数可以被设定为 0x000~0xFFF 内的任意数值.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

adc_preempt_channel

adc_preempt_channel 用于选择需要设定偏移量的通道，其可选参数罗列如下

ADC_PREEMPT_CHANNEL_1： 抢占通道 1
ADC_PREEMPT_CHANNEL_2： 抢占通道 2
ADC_PREEMPT_CHANNEL_3： 抢占通道 3
ADC_PREEMPT_CHANNEL_4： 抢占通道 4

示例

```
/* set preempt channel's conversion value offset */
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_1, 0x111);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_2, 0x222);
adc_preempt_offset_value_set(ADC1, ADC_PREEMPT_CHANNEL_3, 0x333);
```

5.2.21 函数 adc_ordinary_part_count_set

下表描述了函数 adc_ordinary_part_count_set

表 42. 函数 adc_ordinary_part_count_set

项目	描述
函数名	adc_ordinary_part_count_set
函数原型	void adc_ordinary_part_count_set(adc_type *adc_x, uint8_t adc_channel_count)
功能描述	分割模式下每次触发转换的普通通道个数设定
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_channel_count：分割模式下普通通道子组别个数设定 该参数可以被设定为 0x1~0x8 内的任意数值.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set partitioned mode channel count */
adc_ordinary_part_count_set(ADC1, 2);
```

注意：分割模式下，只有普通通道组的子组别个数可设定，抢占通道组的子组别个数固定为 1。

5.2.22 函数 adc_ordinary_part_mode_enable

下表描述了函数 adc_ordinary_part_mode_enable

表 43. 函数 adc_ordinary_part_mode_enable

项目	描述
函数名	adc_ordinary_part_mode_enable
函数原型	void adc_ordinary_part_mode_enable(adc_type *adc_x, confirm_state new_state)
功能描述	普通通道上的分割模式使能
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	new_state：普通通道分割模式的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable the partitioned mode on ordinary channel */
adc_ordinary_part_mode_enable(ADC1, TRUE);
```

5.2.23 函数 adc_preempt_part_mode_enable

下表描述了函数 adc_preempt_part_mode_enable

表 44. 函数 adc_preempt_part_mode_enable

项目	描述
函数名	adc_preempt_part_mode_enable
函数原型	void adc_preempt_part_mode_enable(adc_type *adc_x, confirm_state new_state)
功能描述	抢占通道上的分割模式使能
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	new_state：普通通道分割模式的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable the partitioned mode on preempt channel */
adc_preempt_part_mode_enable(ADC1, TRUE);
```

5.2.24 函数 adc_preempt_auto_mode_enable

下表描述了函数 adc_preempt_auto_mode_enable

表 45. 函数 adc_preempt_auto_mode_enable

项目	描述
函数名	adc_preempt_auto_mode_enable
函数原型	void adc_preempt_auto_mode_enable(adc_type *adc_x, confirm_state,new_state)
功能描述	普通通道组转换结束后的抢占组自动转换使能
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	new_state：抢占组自动转换的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable automatic preempt group conversion */
adc_preempt_auto_mode_enable(ADC1, TRUE);
```

5.2.25 函数 adc_tempersensor_vintrv_enable

下表描述了函数 adc_tempersensor_vintrv_enable

表 46. 函数 adc_tempersensor_vintrv_enable

项目	描述
函数名	adc_tempersensor_vintrv_enable

项目	描述
函数原型	void adc_tempersensor_vintrv_enable(confirm_state new_state)
功能描述	内部温度传感器及 V_{INTRV} 使能
输入参数	new_state：内部温度传感器及 V_{INTRV} 的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable the temperature sensor and vintrv channel */
adc_tempersensor_vintrv_enable(TRUE);
```

5.2.26 函数 adc_ordinary_software_trigger_enable

下表描述了函数 adc_ordinary_software_trigger_enable

表 47. 函数 adc_ordinary_software_trigger_enable

项目	描述
函数名	adc_ordinary_software_trigger_enable
函数原型	void adc_ordinary_software_trigger_enable(adc_type *adc_x, confirm_state new_state)
功能描述	软件触发普通通道转换
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	new_state：软件触发普通通道转换的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable ordinary software start conversion */
adc_ordinary_software_trigger_enable(ADC1, TRUE);
```

5.2.27 函数 adc_ordinary_software_trigger_status_get

下表描述了函数 adc_ordinary_software_trigger_status_get

表 48. 函数 adc_ordinary_software_trigger_status_get

项目	描述
函数名	adc_ordinary_software_trigger_status_get
函数原型	flag_status adc_ordinary_software_trigger_status_get(adc_type *adc_x)
功能描述	获取软件触发的普通通道转换状态
输入参数	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.

项目	描述
输出参数	无
返回值	flag_status：普通通道软件触发转换的状态 该返回值可为罗列的其中之一：SET，RESET.
先决条件	无
被调用函数	无

示例

```
/* wait ordinary software start conversion */
while(adc_ordinary_software_trigger_status_get(ADC1));
```

5.2.28 函数 adc_preempt_software_trigger_enable

下表描述了函数 adc_preempt_software_trigger_enable

表 49. 函数 adc_preempt_software_trigger_enable

项目	描述
函数名	adc_preempt_software_trigger_enable
函数原型	void adc_preempt_software_trigger_enable(adc_type *adc_x, confirm_state new_state)
功能描述	软件触发抢占通道转换
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	new_state：软件触发抢占通道转换的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable preempt software start conversion */
adc_preempt_software_trigger_enable(ADC1, TRUE);
```

5.2.29 函数 adc_preempt_software_trigger_status_get

下表描述了函数 adc_preempt_software_trigger_status_get

表 50. 函数 adc_preempt_software_trigger_status_get

项目	描述
函数名	adc_preempt_software_trigger_status_get
函数原型	flag_status adc_preempt_software_trigger_status_get(adc_type *adc_x)
功能描述	获取软件触发的抢占通道转换状态
输入参数	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输出参数	无
返回值	flag_status：抢占通道软件触发转换的状态 该返回值可为罗列的其中之一：SET，RESET.

项目	描述
先决条件	无
被调用函数	无

示例

```
/* wait preempt software start conversion */
while(adc_preempt_software_trigger_status_get(ADC1));
```

5.2.30 函数 adc_ordinary_conversion_data_get

下表描述了函数 adc_ordinary_conversion_data_get

表 51. 函数 adc_ordinary_conversion_data_get

项目	描述
函数名	adc_ordinary_conversion_data_get
函数原型	uint16_t adc_ordinary_conversion_data_get(adc_type *adc_x)
功能描述	获取非主从模式下普通通道转换数据
输入参数	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输出参数	无
返回值	16 位的普通通道转换数据.
先决条件	无
被调用函数	无

示例

```
uint16_t adc1_ordinary_index = 0;
adc1_ordinary_index = adc_ordinary_conversion_data_get(ADC1);
```

注意：只有配置 ADC 为独立模式，且各 ADC 仅配置单个通道时，才可使用此函数。

5.2.31 函数 adc_combine_ordinary_conversion_data_get

下表描述了函数 adc_combine_ordinary_conversion_data_get

表 52. 函数 adc_combine_ordinary_conversion_data_get

项目	描述
函数名	adc_combine_ordinary_conversion_data_get
函数原型	uint32_t adc_combine_ordinary_conversion_data_get(void)
功能描述	获取主从组合模式下普通通道转换数据
输入参数	无
输出参数	无
返回值	32 位的普通通道转换数据（高 16 为 ADC2 的数据，低 16 位为 ADC1 的数据）.
先决条件	无
被调用函数	无

示例

```
uint32_t common_ordinary_index = 0;
common_ordinary_index = adc_combine_ordinary_conversion_data_get();
```

注意：只有配置 ADC 为主从组合模式，且各 ADC 仅配置单个通道时，才可使用此函数。

5.2.32 函数 adc_preempt_conversion_data_get

下表描述了函数 adc_preempt_conversion_data_get

表 53. 函数 adc_preempt_conversion_data_get

项目	描述
函数名	adc_preempt_conversion_data_get
函数原型	uint16_t adc_preempt_conversion_data_get(adc_type *adc_x, adc_preempt_channel_type adc_preempt_channel)
功能描述	获取抢占通道转换数据
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_preempt_channel：抢占通道选择 该参数详细描述见 *adc_preempt_channel*.
输出参数	无
返回值	16 位的抢占通道转换数据.
先决条件	无
被调用函数	无

示例

```
uint16_t adc1_preempt_valuetab[3] = {0};
adc1_preempt_valuetab[0] = adc_preempt_conversion_data_get(ADC1, ADC_PREEMPT_CHANNEL_1);
adc1_preempt_valuetab[1] = adc_preempt_conversion_data_get(ADC1, ADC_PREEMPT_CHANNEL_2);
adc1_preempt_valuetab[2] = adc_preempt_conversion_data_get(ADC1, ADC_PREEMPT_CHANNEL_3);
```

5.2.33 函数 adc_flag_get

下表描述了函数 adc_flag_get

表 54. 函数 adc_flag_get

项目	描述
函数名	adc_flag_get
函数原型	flag_status adc_flag_get(adc_type *adc_x, uint8_t adc_flag)
功能描述	获取标志位状态
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_flag：需要获取状态的标志选择 该参数详细描述见 *adc_flag*
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为罗列的其中之一：SET，RESET.
先决条件	无
被调用函数	无

adc_flag

adc_flag 用于选择需要获取状态的标志，其可选参数罗列如下

ADC_VMOR_FLAG： 电压监测超出范围标志

ADC_CCE_FLAG： 通道转换结束标志

ADC_PCCE_FLAG： 抢占通道组转换结束标志
ADC_PCCS_FLAG： 抢占通道转换开始标志
ADC_OCCS_FLAG： 普通通道转换开始标志

示例

```
/* check if wakeup preempted channelsconversion end flag is set */
if(adc_flag_get(ADC1, ADC_PCCE_FLAG) != RESET)
```

5.2.34 函数 adc_flag_clear

下表描述了函数 adc_flag_clear

表 55. 函数 adc_flag_clear

项目	描述
函数名	adc_flag_clear
函数原型	void adc_flag_clear(adc_type *adc_x, uint32_t adc_flag)
功能描述	清除已置位的标志位
输入参数 1	adc_x：所选择的 ADC 外设 该参数可以选取自其中之一：ADC1，ADC2.
输入参数 2	adc_flag：待清除的标志选择 该参数详细描述见 *adc_flag*
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* preempted channelsconversion end flag clear */
adc_flag_clear(ADC1, ADC_PCCE_FLAG);
```

5.3 电池供电域（BPR）

BPR 寄存器结构 bpr_type，定义于文件“at32f413_bpr.h”如下：

```
/**
  * @brief type define bpr register all
  */
typedef struct
{


} bpr_type;
```

下表给出了 BPR 寄存器总览：

表 56. BPR 寄存器对应表

寄存器	描述
dt1	电池供电数据寄存器 1
dt2	电池供电数据寄存器 2

寄存器	描述
dt3	电池供电数据寄存器 3
dt4	电池供电数据寄存器 4
dt5	电池供电数据寄存器 5
dt6	电池供电数据寄存器 6
dt7	电池供电数据寄存器 7
dt8	电池供电数据寄存器 8
dt9	电池供电数据寄存器 9
dt10	电池供电数据寄存器 10
rtccal	RTC 校准寄存器
ctrl	电池供电控制寄存器
ctrlsts	电池供电控制/状态寄存器
dt11	电池供电数据寄存器 11
dt12	电池供电数据寄存器 12
dt13	电池供电数据寄存器 13
dt14	电池供电数据寄存器 14
dt15	电池供电数据寄存器 15
dt16	电池供电数据寄存器 16
dt17	电池供电数据寄存器 17
dt18	电池供电数据寄存器 18
dt19	电池供电数据寄存器 19
dt20	电池供电数据寄存器 20
dt21	电池供电数据寄存器 21
dt22	电池供电数据寄存器 22
dt23	电池供电数据寄存器 23
dt24	电池供电数据寄存器 24
dt25	电池供电数据寄存器 25
dt26	电池供电数据寄存器 26
dt27	电池供电数据寄存器 27
dt28	电池供电数据寄存器 28
dt29	电池供电数据寄存器 29
dt30	电池供电数据寄存器 30
dt31	电池供电数据寄存器 31
dt32	电池供电数据寄存器 32
dt33	电池供电数据寄存器 33
dt34	电池供电数据寄存器 34
dt35	电池供电数据寄存器 35
dt36	电池供电数据寄存器 36
dt37	电池供电数据寄存器 37
dt38	电池供电数据寄存器 38
dt39	电池供电数据寄存器 39
dt40	电池供电数据寄存器 40
dt41	电池供电数据寄存器 41
dt42	电池供电数据寄存器 42

下表给出了 BPR 库函数总览：

表 57. BPR 库函数总览

函数名	描述
bpr_reset	所有电池供电数据寄存器复位到默认值
bpr_flag_get	获取标志
bpr_flag_clear	清除标志
bpr_interrupt_enable	入侵检测中断使能
bpr_data_read	从电池供电数据寄存器读取数据
bpr_data_write	向电池供电数据寄存器读写数据
bpr_rtc_output_select	事件输出设置
bpr_rtc_clock_calibration_value_set	时钟校准设置
bpr_tamper_pin_enable	入侵检测使能
bpr_tamper_pin_active_level_set	入侵检测有效电平设置

5.3.1 函数 bpr_reset

下表描述了函数 bpr_reset

表 58. 函数 bpr_reset

项目	描述
函数名	bpr_reset
函数原型	void bpr_reset(void);
功能描述	所有电池供电数据寄存器复位到默认值
输入参数 1	无
输出参数	无
返回值	无
先决条件	无
被调用函数	void crm_battery_powered_domain_reset(confirm_state new_state);

示例

```
bpr_reset();
```

5.3.2 函数 bpr_flag_get

下表描述了函数 bpr_flag_get

表 59. 函数 bpr_flag_get

项目	描述
函数名	bpr_flag_get
函数原型	flag_status bpr_flag_get(uint32_t flag);
功能描述	获取标志位状态
输入参数 1	flag：需要获取状态的标志选择 该参数详细描述见 flag
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET、RESET

项目	描述
先决条件	无
被调用函数	无

flag

用于选择需要获取状态的标志，其可选参数罗列如下

BPR_TAMPER_INTERRUPT_FLAG： 入侵检测中断标志

BPR_TAMPER_EVENT_FLAG： 入侵检测事件标志

示例

```
bpr_flag_get(BPR_TAMPER_INTERRUPT_FLAG);
```

5.3.3 函数 bpr_flag_clear

下表描述了函数 bpr_flag_clear

表 60. 函数 bpr_flag_clear

项目	描述
函数名	bpr_flag_clear
函数原型	void bpr_flag_clear(uint32_t flag);
功能描述	清除标志位
输入参数 1	flag：待清除的标志选择 该参数详细描述见 flag
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flag

用于选择需要清除状态的标志，其可选参数罗列如下

BPR_TAMPER_INTERRUPT_FLAG： 入侵检测中断标志

BPR_TAMPER_EVENT_FLAG： 入侵检测事件标志

示例

```
bpr_flag_clear(BPR_TAMPER_INTERRUPT_FLAG);
```

5.3.4 函数 bpr_interrupt_enable

下表描述了函数 bpr_interrupt_enable

表 61. 函数 bpr_interrupt_enable

项目	描述
函数名	bpr_interrupt_enable
函数原型	void bpr_interrupt_enable(confirm_state new_state);
功能描述	入侵检测中断使能
输入参数 1	new_state：入侵检测中断使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无

项目	描述
被调用函数	无

示例

```
bpr_interrupt_enable(TRUE);
```

5.3.5 函数 bpr_data_read

下表描述了函数 bpr_data_read

表 62. 函数 bpr_data_read

项目	描述
函数名	bpr_data_read
函数原型	uint16_t bpr_data_read(bpr_data_type bpr_data);
功能描述	从电池供电数据寄存器读取数据
输入参数 1	bpr_data：数据寄存器 参阅章节：bpr_data 查阅更多该参数允许取值范围
输出参数	无
返回值	电池供电数据寄存器数据
先决条件	无
被调用函数	无

bpr_data

数据寄存器

BPR_DATA1：数据寄存器 1

BPR_DATA2：数据寄存器 2

BPR_DATA41：数据寄存器 41

BPR_DATA42：数据寄存器 42

示例

```
bpr_data_read(BPR_DATA1);
```

5.3.6 函数 bpr_data_write

下表描述了函数 bpr_data_write

表 63. 函数 bpr_data_write

项目	描述
函数名	bpr_data_write
函数原型	void bpr_data_write(bpr_data_type bpr_data, uint16_t data_value);
功能描述	向电池供电数据寄存器读写数据
输入参数 1	bpr_data：数据寄存器 参阅章节：bpr_data 查阅更多该参数允许取值范围
输入参数 2	data_value：16 位数据
输出参数	无
返回值	无
先决条件	无
被调用函数	无

bpr_data

数据寄存器

BPR_DATA1：数据寄存器 1

BPR_DATA2：数据寄存器 2

BPR_DATA41：数据寄存器 41

BPR_DATA42：数据寄存器 42

示例

```
bpr_data_write(BPR_DATA1, 0x5A5A);
```

5.3.7 函数 bpr_rtc_output_select

下表描述了函数 bpr_rtc_output_select

表 64. 函数 bpr_rtc_output_select

项目	描述
函数名	bpr_rtc_output_select
函数原型	void bpr_rtc_output_select(bpr_rtc_output_type output_source);
功能描述	事件输出设置
输入参数 1	output_source：输出事件 参阅章节：output_source 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

output_source

输出事件

BPR_RTC_OUTPUT_NONE：输出关闭

BPR_RTC_OUTPUT_CLOCK_CAL_BEFORE：校准前的时钟 64 分频输出

BPR_RTC_OUTPUT_ALARM：脉冲输出输出闹钟事件

BPR_RTC_OUTPUT_SECOND：脉冲输出输出秒事件

BPR_RTC_OUTPUT_CLOCK_CAL_AFTER：校准后的时钟 64 分频输出

BPR_RTC_OUTPUT_ALARM_TOGGLE：翻转输出闹钟事件

BPR_RTC_OUTPUT_SECOND_TOGGLE：翻转输出秒事件

示例

```
bpr_rtc_output_select(BPR_RTC_OUTPUT_ALARM);
```

5.3.8 函数 bpr_rtc_clock_calibration_value_set

下表描述了函数 bpr_rtc_clock_calibration_value_set

表 65. 函数 bpr_rtc_clock_calibration_value_set

项目	描述
函数名	bpr_rtc_clock_calibration_value_set
函数原型	void bpr_rtc_clock_calibration_value_set(uint8_t calibration_value);
功能描述	时钟校准设置

项目	描述
输入参数 1	value：校准值，范围 0~0x7F
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
bpr_rtc_clock_calibration_value_set(0x7F);
```

5.3.9 函数 bpr_tamper_pin_enable

下表描述了函数 bpr_tamper_pin_enable

表 66. 函数 bpr_tamper_pin_enable

项目	描述
函数名	bpr_tamper_pin_enable
函数原型	void bpr_tamper_pin_enable(confirm_state new_state);
功能描述	入侵检测使能
输入参数 1	new_state：中断使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
bpr_tamper_pin_enable(TRUE);
```

5.3.10 函数 bpr_tamper_pin_active_level_set

下表描述了函数 bpr_tamper_pin_active_level_set

表 67. 函数 bpr_tamper_pin_active_level_set

项目	描述
函数名	bpr_tamper_pin_active_level_set
函数原型	void bpr_tamper_pin_active_level_set(bpr_tamper_pin_active_level_type active_level);
功能描述	设置入侵检测有效电平
输入参数 1	active_level：入侵检测有效电平 参阅章节：active_level 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

active_level

入侵检测有效电平

BPR_TAMPER_PIN_ACTIVE_HIGH：高电平触发入侵检测

BPR_TAMPER_PIN_ACTIVE_LOW：低电平触发入侵检测

示例

```
bpr_tamper_pin_active_level_set(BPR_TAMPER_PIN_ACTIVE_HIGH);
```

5.4 控制器局域网模块（CAN）

CAN 寄存器结构 can_type，定义于文件“at32f413_can.h”如下：

```
/**
  * @brief type define can register all
  */
typedef struct
{
  ...
} can_type;
```

下表给出了 CAN 寄存器总览：

表 68. CAN 寄存器总览

寄存器	描述
mctrl	CAN 主控制寄存器
msts	CAN 主状态寄存器
tsts	CAN 发送状态寄存器
rf0	CAN 接收 FIFO 0 寄存器
fr1	CAN 接收 FIFO 1 寄存器
inten	CAN 中断使能寄存器
ests	CAN 错误状态寄存器
btmg	CAN 位时序寄存器
tmi0	发送邮箱 0 标识符寄存器
tmc0	发送邮箱 0 数据长度和时间戳寄存器
tmdtl0	发送邮箱 0 低字节数据寄存器
tmdth0	发送邮箱 0 高字节数据寄存器
tmi1	发送邮箱 1 标识符寄存器
tmc1	发送邮箱 1 数据长度和时间戳寄存器
tmdtl1	发送邮箱 1 低字节数据寄存器
tmdth1	发送邮箱 1 高字节数据寄存器
tmi2	发送邮箱 2 标识符寄存器
tmc2	发送邮箱 2 数据长度和时间戳寄存器
tmdtl2	发送邮箱 2 低字节数据寄存器
tmdth2	发送邮箱 2 高字节数据寄存器
rfi0	接收 FIFO0 邮箱标识符寄存器
rfc0	接收 FIFO0 邮箱数据长度和时间戳寄存器
rfdtl0	接收 FIFO0 邮箱低字节数据寄存器
rfdth0	接收 FIFO0 邮箱高字节数据寄存器
rfi1	接收 FIFO1 邮箱标识符寄存器
rfc1	接收 FIFO1 邮箱数据长度和时间戳寄存器
rfdtl1	接收 FIFO1 邮箱低字节数据寄存器

寄存器	描述
rfdth1	接收 FIFO1 邮箱高字节数据寄存器
fctrl	CAN 过滤器控制寄存器
fmcfg	CAN 过滤器模式配置寄存器
fscfg	CAN 过滤器位宽配置寄存器
frf	CAN 过滤器 FIFO 关联寄存器
facfg	CAN 过滤器激活控制寄存器
fb0f1	CAN 过滤器组 0 的过滤位寄存器 1
fb0f2	CAN 过滤器组 0 的过滤位寄存器 2
fb1f1	CAN 过滤器组 1 的过滤位寄存器 1
fb1f2	CAN 过滤器组 1 的过滤位寄存器 2
…	…
fb13f1	CAN 过滤器组 13 的过滤位寄存器 1
fb13f2	CAN 过滤器组 13 的过滤位寄存器 2

下表给出了 CAN 库函数总览：

表 69. CAN 库函数总览

函数名	描述
can_reset	将 CAN 所有寄存器值恢复到复位值
can_baudrate_default_para_init	给 CAN 波特率初始化结构体赋初值
can_baudrate_set	设置 CAN 波特率
can_default_para_init	给 CAN 初始化结构体赋初值
can_base_init	将 can_base_struct 中指定的参数初始化到 CAN 的相关寄存器
can_filter_default_para_init	给 CAN 过滤器初始化结构体赋初值
can_filter_init	将 can_filter_init_struct 中指定的参数初始化到 CAN 的相关寄存器
can_debug_transmission_prohibit	选择调试时禁止/不禁止收发报文
can_ttc_mode_enable	时间触发模式使能
can_message_transmit	发送一帧报文
can_transmit_status_get	获取发送状态
can_transmit_cancel	取消发送
can_message_receive	接收一帧报文
can_receive_fifo_release	释放接收 FIFO
can_receive_message_pending_get	获取 FIFO 中待读取的报文数目
can_operating_mode_set	CAN 工作模式设置
can_doze_mode_enter	进入睡眠模式
can_doze_mode_exit	退出睡眠模式
can_error_type_record_get	读取 CAN 错误类型
can_receive_error_counter_get	读取 CAN 接收错误计数
can_transmit_error_counter_get	读取 CAN 发送错误计数
can_interrupt_enable	使能选定的 CAN 中断
can_flag_get	读取选定的 CAN 标志
can_flag_clear	清除选定的 CAN 标志

5.4.1 函数 can_reset

下表描述了函数 can_reset

表 70. 函数 can_reset

项目	描述
函数名	can_reset
函数原型	void can_reset(can_type* can_x);
功能描述	将 can 寄存器值复位到默认值
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset();

示例

```
can_reset(CAN1);
```

5.4.2 函数 can_baudrate_default_para_init

下表描述了函数 can_baudrate_default_para_init

表 71. 函数 can_baudrate_default_para_init

项目	描述
函数名	can_baudrate_default_para_init
函数原型	void can_baudrate_default_para_init(can_baudrate_type* can_baudrate_struct);
功能描述	给 CAN 波特率初始化结构体赋初值
输入参数 1	can_baudrate_struct：指向 *can_baudrate_type* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 can_baudrate_type 类型的变量
被调用函数	无

示例

```
can_baudrate_type can_baudrate_struct;
can_baudrate_default_para_init(&can_baudrate_struct);
```

5.4.3 函数 can_baudrate_set

下表描述了函数 can_baudrate_set

表 72. 函数 can_baudrate_set

项目	描述
函数名	can_baudrate_set
函数原型	error_status can_baudrate_set(can_type* can_x, can_baudrate_type* can_baudrate_struct);
功能描述	设置 CAN 波特率

项目	描述
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	can_baudrate_struct：指向 *can_baudrate_type* 类型的指针
输出参数	无
返回值	status_index：波特率设置是否成功
先决条件	需要先定义一个 can_baudrate_type 类型的变量
被调用函数	无

can_baudrate_type 在 at32f413_can.h 中定义：

```
typedef struct
{
  uint16_t          baudrate_div;
  can_rsaw_type     rsaw_size;
  can_bts1_type     bts1_size;
  can_bts2_type     bts2_size;
} can_baudrate_type;
```

baudrate_div

CAN 时钟分频系数

取值范围：0x001~0x400

rsaw_size

重新同步同步跳跃宽度，即每个 bit 可以延长/缩短的时间上限

CAN_RSAW_1TQ：重同步跳跃宽度上限为 1 个时间单位

CAN_RSAW_2TQ：重同步跳跃宽度上限为 2 个时间单位

CAN_RSAW_3TQ：重同步跳跃宽度上限为 3 个时间单位

CAN_RSAW_4TQ：重同步跳跃宽度上限为 4 个时间单位

bts1_size

segment1 段时长

bts1_size 描述

CAN_BTS1_1TQ： 位时间段 1 时长为 1 个时间单位

……

CAN_BTS1_16TQ：位时间段 1 时长为 16 个时间单位

bts2_size

segment2 段时长

CAN_BTS2_1TQ： 位时间段 2 时长为 1 个时间单位

……

CAN_BTS2_8TQ： 位时间段 2 时长为 8 个时间单位

示例

```
/* can baudrate, set baudrate = pclk/(baudrate_div *(1 + bts1_size + bts2_size)) */
can_baudrate_struct.baudrate_div = 10;
can_baudrate_struct.rsaw_size = CAN_RSAW_3TQ;
can_baudrate_struct.bts1_size = CAN_BTS1_8TQ;
can_baudrate_struct.bts2_size = CAN_BTS2_3TQ;
can_baudrate_set(CAN1, &can_baudrate_struct);
```

5.4.4 函数 can_default_para_init

下表描述了函数 can_default_para_init

表 73. 函数 can_default_para_init

项目	描述
函数名	can_default_para_init
函数原型	void can_default_para_init(can_base_type* can_base_struct);
功能描述	给 CAN 初始化结构体赋初值
输入参数 1	can_base_struct：指向 *can_base_type* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 can_base_type 类型的变量
被调用函数	无

示例

```
can_base_type can_base_struct;
can_default_para_init (&can_base_struct);
```

5.4.5 函数 can_base_init

下表描述了函数 can_base_init

表 74. 函数 can_base_init

项目	描述
函数名	can_base_init
函数原型	error_status can_base_init(can_type* can_x, can_base_type* can_base_struct);
功能描述	将 can_base_struct 中指定的参数初始化到 CAN 的相关寄存器
输入参数 1	can_base_struct：指向 *can_base_type* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 can_base_type 类型的变量
被调用函数	无

can_base_type 在 at32f413_can.h 中定义：

```
typedef struct
{
  can_mode_type                mode_selection;
  confirm_state                ttc_enable;
  confirm_state                aebo_enable;
  confirm_state                aed_enable;
  confirm_state                prsf_enable;
  can_msg_discarding_rule_type mdrsel_selection;
  can_msg_sending_rule_type    mmssr_selection;
} can_base_type;
```

mode_selection

测试模式选择

CAN_MODE_COMMUNICATE： 通信模式
CAN_MODE_LOOPBACK： 环回模式
CAN_MODE_LISTENONLY： 只听模式
CAN_MODE_LISTENONLY_LOOPBACK： 环回+只听模式

ttc_enable

开启/关闭时间触发通信模式
FALSE： 关闭时间通信模式；
TRUE： 开启时间通信模式（接收/发送报文时，截取时间戳并存储在 CAN_RFCx 和 CAN_TMCx 寄存器）。

aebo_enable

自动退出离线状态模式使能
FALSE： 关闭自动退出离线模式；
TRUE： 开启自动退出离线模式。

aed_enable

自动退出睡眠模式使能
FALSE： 关闭自动退出睡眠模式；
TRUE： 开启自动退出睡眠模式。

prsf_enable

发送失败时禁止重传使能
FALSE： 发送失败时自动重传；
TRUE： 发送失败时禁止重传。

mdrsel_selection

接收溢出时报文丢弃规则选择
CAN_DISCARDING_FIRST_RECEIVED： 丢弃上一帧收到的报文；
CAN_DISCARDING_LAST_RECEIVED： 丢弃最新收到的报文。

mmssr_selection

多报文发送顺序规则选择。
CAN_SENDING_BY_ID： 标识符最小的最先被发送；
CAN_SENDING_BY_REQUEST： 最先请求的最先被发送。

示例

```
/* can base init */
can_base_struct.mode_selection = CAN_MODE_COMMUNICATE;
can_base_struct.ttc_enable = FALSE;
can_base_struct.aebo_enable = TRUE;
can_base_struct.aed_enable = TRUE;
can_base_struct.prsf_enable = FALSE;
can_base_struct.mdrsel_selection = CAN_DISCARDING_FIRST_RECEIVED;
can_base_struct.mmssr_selection = CAN_SENDING_BY_ID;
can_base_init(CAN1, &can_base_struct);
```

5.4.6 函数 can_filter_default_para_init

下表描述了函数 can_filter_default_para_init

表 75. 函数 can_filter_default_para_init

项目	描述
函数名	can_filter_default_para_init
函数原型	void can_filter_default_para_init(can_filter_init_type* can_filter_init_struct);
功能描述	给 CAN 过滤器初始化结构体赋初值
输入参数 1	can_filter_init_struct：指向 *can_filter_init_type* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 can_filter_init_type 类型的变量
被调用函数	无

示例

```
can_filter_init_type can_filter_init_struct;
can_filter_default_para_init(&can_filter_init_struct);
```

5.4.7 函数 can_filter_init

下表描述了函数 can_filter_init

表 76. 函数 can_filter_init

项目	描述
函数名	can_filter_init
函数原型	void can_filter_init(can_type* can_x, can_filter_init_type* can_filter_init_struct);
功能描述	将 can_base_struct 中指定的参数初始化到 CAN 的相关寄存器
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	can_filter_init_struct：指向 *can_filter_init_type* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 can_filter_init_type 类型的变量
被调用函数	无

can_filter_init_type 在 at32f413_can.h 中定义：

```
typedef struct
{
  confirm_state              filter_activate_enable;
  can_filter_mode_type       filter_mode;
  can_filter_fifo_type       filter_fifo;
  uint8_t                    filter_number;
  can_filter_bit_width_type  filter_bit;
  uint16_t                   filter_id_high;
  uint16_t                   filter_id_low;
  uint16_t                   filter_mask_high;
  uint16_t                   filter_mask_low;
} can_filter_init_type;
```

filter_activate_enable

开启/关闭过滤器组
FALSE: 关闭过滤器组
TRUE: 使能过滤器组

filter_mode

过滤器组关联 FIFO 选择
CAN_FILTER_MODE_ID_MASK： 掩码模式
CAN_FILTER_MODE_ID_LIST： 列表模式

filter_fifo

过滤器组关联 FIFO 选择
CAN_FILTER_FIFO0： 关联 FIFO0
CAN_FILTER_FIFO1： 关联 FIFO1

filter_number

过滤器组选择
取值范围：0~13

filter_bit

过滤器宽度选择
CAN_FILTER_16BIT： 过滤器宽度 16bit
CAN_FILTER_32BIT： 过滤器宽度 32bit

filter_id_high

filter_id_high 用于设定过滤器标识符 1 高 16 位（32bit 位宽，屏蔽/列表模式）或设定过滤器标识符 2（16bit 位宽，列表模式）或设定过滤器屏蔽标识符 1（16bit 位宽，屏蔽模式）。
取值范围：0x0000~0xFFFF

filter_id_low

filter_id_low 用于设定过滤器标识符 1 低 16 位（32bit 位宽，屏蔽/列表模式）或设定过滤器标识符 1（16bit 位宽，屏蔽/列表模式）。
取值范围：0x0000~0xFFFF

filter_mask_high

filter_mask_high 用于设定过滤器屏蔽标识符 1 高 16 位（32bit 位宽，屏蔽模式）或设定过滤器屏蔽标识符 2（16bit 位宽，屏蔽模式）或设定过滤器标识符 2 高 16 位（32bit 位宽，列表模式）或设定过滤器标识符 4（16bit 位宽，列表模式）。
取值范围：0x0000~0xFFFF

filter_mask_low

filter_mask_low 用于设定过滤器屏蔽标识符 1 低 16 位（32bit 位宽，屏蔽模式）或设定过滤器标识符 2（16bit 位宽，屏蔽模式）或设定过滤器标识符 2 低 16 位（32bit 位宽，列表模式）或设定过滤器标识符 3（16bit 位宽，列表模式）。
取值范围：0x0000~0xFFFF

示例

```
/* can filter init */
can_filter_init_struct.filter_activate_enable = TRUE;
can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_MASK;
can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0;
can_filter_init_struct.filter_number = 0;
can_filter_init_struct.filter_bit = CAN_FILTER_32BIT;
can_filter_init_struct.filter_id_high = 0;
can_filter_init_struct.filter_id_low = 0;
can_filter_init_struct.filter_mask_high = 0;
```

```
can_filter_init_struct.filter_mask_low = 0;
can_filter_init(CAN1, &can_filter_init_struct);
```

5.4.8 函数 can_debug_transmission_prohibit

下表描述了函数 can_debug_transmission_prohibit

表 77. 函数 can_debug_transmission_prohibit

项目	描述
函数名	can_debug_transmission_prohibit
函数原型	void can_debug_transmission_prohibit(can_type* can_x, confirm_state new_state);
功能描述	选择调试时禁止/不禁止收发报文
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* prohibit can trans when debug*/
can_debug_transmission_prohibit(CAN1, TRUE);
```

5.4.9 函数 can_ttc_mode_enable

下表描述了函数 can_ttc_mode_enable

表 78. 函数 can_ttc_mode_enable

项目	描述
函数名	can_ttc_mode_enable
函数原型	void can_ttc_mode_enable(can_type* can_x, confirm_state new_state);
功能描述	时间触发模式使能
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* can time trigger operation communication mode enable*/
can_ttc_mode_enable (CAN1, TRUE);
```

注意：函数can_base_init中的ttc_enable一项使能后，仅开启时间戳功能（在接收/发送报文时，截取时间戳并存储在CAN_RFCx和CAN_TMCx寄存器中）。而此处的can_ttc_mode_enable函数使能后，会开启时间戳功能，且开启时间戳发送功能（在发送报文时，将时间戳填入数据段的第7和8字节发送）。

5.4.10 函数 can_message_transmit

下表描述了函数 can_message_transmit

表 79. 函数 can_message_transmit

项目	描述
函数名	can_message_transmit
函数原型	uint8_t can_message_transmit(can_type* can_x, can_tx_message_type* tx_message_struct);
功能描述	发送一帧报文
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	tx_message_struct：待发送的报文，参考 *can_tx_message_type*
输出参数	无
返回值	transmit_mailbox：发送这帧报文选用的邮箱号
先决条件	在 tx_message_struct 填入待发送的报文
被调用函数	无

can_tx_message_type 在 at32f413_can.h 中定义：

```
typedef struct
{
  uint32_t                standard_id;
  uint32_t                extended_id;
  can_identifier_type     id_type;
  can_trans_frame_type    frame_type;
  uint8_t                 dlc;
  uint8_t                 data[8];
} can_tx_message_type;
```

standard_id

标准标识符（11bit 有效）

取值范围：0x000~0x7FF

extended_id

扩展标识符（29bit 有效）

取值范围：0x000~0x1FFFFFFF

id_type

标识符类型

CAN_ID_STANDARD： 标准标识符

CAN_ID_EXTENDED： 扩展标识符

frame_type

帧类型

CAN_TFT_DATA： 数据帧

CAN_TFT_REMOTE： 远程帧

dlc

数据长度（单位 byte）
取值范围：0~8
data[8]
待发送的数据
取值范围 0x00~0xFF
示例

```
/* can transmit data */
static void can_transmit_data(void)
{
  uint8_t transmit_mailbox;
  can_tx_message_type tx_message_struct;
  tx_message_struct.standard_id = 0x400;
  tx_message_struct.extended_id = 0;
  tx_message_struct.id_type = CAN_ID_STANDARD;
  tx_message_struct.frame_type = CAN_TFT_DATA;
  tx_message_struct.dlc = 8;
  tx_message_struct.data[0] = 0x11;
  tx_message_struct.data[1] = 0x22;
  tx_message_struct.data[2] = 0x33;
  tx_message_struct.data[3] = 0x44;
  tx_message_struct.data[4] = 0x55;
  tx_message_struct.data[5] = 0x66;
  tx_message_struct.data[6] = 0x77;
  tx_message_struct.data[7] = 0x88;
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct);
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL);
}
```

5.4.11 函数 can_transmit_status_get

下表描述了函数 can_transmit_status_get

表 80. 函数 can_transmit_status_get

项目	描述
函数名	can_transmit_status_get
函数原型	can_transmit_status_type can_transmit_status_get(can_type* can_x, can_tx_mailbox_num_type transmit_mailbox);
功能描述	获取发送状态
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	transmit_mailbox：发送这帧报文选用的邮箱号
输出参数	无
返回值	state_index：发送状态
先决条件	先发送一帧报文并获取发送邮箱号
被调用函数	无

示例

```
/* can transmit data */
static void can_transmit_data(void)
{
  uint8_t transmit_mailbox;
  can_tx_message_type tx_message_struct;
  tx_message_struct.standard_id = 0x400;
  tx_message_struct.extended_id = 0;
  tx_message_struct.id_type = CAN_ID_STANDARD;
  tx_message_struct.frame_type = CAN_TFT_DATA;
  tx_message_struct.dlc = 8;
  tx_message_struct.data[0] = 0x11;
  tx_message_struct.data[1] = 0x22;
  tx_message_struct.data[2] = 0x33;
  tx_message_struct.data[3] = 0x44;
  tx_message_struct.data[4] = 0x55;
  tx_message_struct.data[5] = 0x66;
  tx_message_struct.data[6] = 0x77;
  tx_message_struct.data[7] = 0x88;
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct);
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL);
}
```

5.4.12 函数 can_transmit_cancel

下表描述了函数 can_transmit_cancel

表 81. 函数 can_transmit_cancel

项目	描述
函数名	can_transmit_cancel
函数原型	void can_transmit_cancel(can_type* can_x, can_tx_mailbox_num_type transmit_mailbox);
功能描述	取消发送
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	transmit_mailbox：发送这帧报文选用的邮箱号
输出参数	无
返回值	无
先决条件	先发送一帧报文并获取发送邮箱号
被调用函数	无

示例

```
/* cancel a transmit request */
uint8_t transmit_mailbox;
transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct);
```

```
can_transmit_cancel(CAN1, (can_tx_mailbox_num_type)transmit_mailbox);
```

5.4.13 函数 can_message_receive

下表描述了函数 can_message_receive

表 82. 函数 can_message_receive

项目	描述
函数名	can_message_receive
函数原型	void can_message_receive(can_type* can_x, can_rx_fifo_num_type fifo_number, can_rx_message_type* rx_message_struct);
功能描述	接收一帧报文
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	fifo_number：使用的接收 FIFO 该参数可以选取自其中之一 ：CAN_RX_FIFO0，CAN_RX_FIFO1
输出参数	rx_message_struct：接收到的报文，参考 *can_rx_message_type*
返回值	无
先决条件	接收 FIFO 非空（FIFO 报文数目不为 0）
被调用函数	void can_receive_fifo_release(can_type* can_x, can_rx_fifo_num_type fifo_number);

can_rx_message_type 在 at32f413_can.h 中定义：

```
typedef struct
{
    uint32_t                standard_id;
    uint32_t                extended_id;
    can_identifier_type     id_type;
    can_trans_frame_type    frame_type;
    uint8_t                 dlc;
    uint8_t                 data[8];
    uint8_t                 filter_index;
} can_rx_message_type;
```

standard_id

标准标识符（11bit 有效）

取值范围：0x000~0x7FF

extended_id

扩展标识符（29bit 有效）

取值范围：0x000~0x1FFFFFFF

id_type

标识符类型

CAN_ID_STANDARD： 标准标识符

CAN_ID_EXTENDED： 扩展标识符

frame_type

帧类型

CAN_TFT_DATA： 数据帧

CAN_TFT_REMOTE： 远程帧

dlc

数据长度（单位 byte）
取值范围：0~8
data[8]
待发送的数据
取值范围：0x00~0xFF
filter_index
过滤器匹配序号（指示成功通过的过滤器的索引序号）
取值范围：0x00~0xFF
示例

```
/* can receive message */
can_rx_message_type rx_message_struct;
can_message_receive(CAN1, CAN_RX_FIFO0, &rx_message_struct);
```

5.4.14 函数 can_receive_fifo_release

下表描述了函数 can_receive_fifo_release

表 83. 函数 can_receive_fifo_release

项目	描述
函数名	can_receive_fifo_release
函数原型	void can_receive_fifo_release(can_type* can_x, can_rx_fifo_num_type fifo_number);
功能描述	释放接收 FIFO
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	fifo_number：使用的接收 FIFO 该参数可以选取自其中之一 ：CAN_RX_FIFO0，CAN_RX_FIFO1
输出参数	无
返回值	无
先决条件	已读取 FIFO 中的报文
被调用函数	无

示例

```
/* can receive message */
void can_message_receive(can_type* can_x, can_rx_fifo_num_type fifo_number, can_rx_message_type*
rx_message_struct)
{
  /* get the id type */
  rx_message_struct->id_type = (can_identifier_type)can_x->fifo_mailbox[fifo_number].rfi_bit.rfidi;
  …

  /* get the data field */
  rx_message_struct->data[0] = can_x->fifo_mailbox[fifo_number].rfdtl_bit.rfdt0;
  …
  rx_message_struct->data[7] = can_x->fifo_mailbox[fifo_number].rfdth_bit.rfdt7;

  /*释放 FIFO 前必须先读取 FIFO*/
```

```
  /* release the fifo */
  can_receive_fifo_release(can_x, fifo_number);
}
```

5.4.15 函数 can_receive_message_pending_get

下表描述了函数 can_receive_message_pending_get

表 84. 函数 can_receive_message_pending_get

项目	描述
函数名	can_receive_message_pending_get
函数原型	uint8_t can_receive_message_pending_get(can_type* can_x, can_rx_fifo_num_type fifo_number);
功能描述	获取 FIFO 中待读取的报文数目
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	fifo_number：使用的接收 FIFO 该参数可以选取自其中之一 ：CAN_RX_FIFO0，CAN_RX_FIFO1
输出参数	无
返回值	message_pending：FIFO 中待读取的报文数目
先决条件	无
被调用函数	无

示例

```
/* return the number of pending messages of */
can_receive_message_pending_get (CAN1, CAN_RX_FIFO0);
```

5.4.16 函数 can_operating_mode_set

下表描述了函数 can_operating_mode_set

表 85. 函数 can_operating_mode_set

项目	描述
函数名	can_operating_mode_set
函数原型	error_status can_operating_mode_set(can_type* can_x, can_operating_mode_type can_operating_mode);
功能描述	CAN 工作模式设置
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	*can_operating_mode*：CAN 工作模式选择
输出参数	无
返回值	status：设置是否成功
先决条件	无
被调用函数	无

can_operating_mode

CAN_OPERATINGMODE_FREEZE：　　冻结模式--用于 CAN 控制器初始化
CAN_OPERATINGMODE_DOZE：　　睡眠模式--CAN 时钟停止，节省电能
CAN_OPERATINGMODE_COMMUNICATE：通信模式--用于正常通信

示例

```
/* set the operation mode –enter freeze mode*/
can_operating_mode_set (CAN1, CAN_OPERATINGMODE_FREEZE);

/*进行 CAN 控制器初始化*/
…

/* set the operation mode –enter communicate mode*/
can_operating_mode_set (CAN1, CAN_OPERATINGMODE_COMMUNICATE);

/*开始正常通信：收/发报文*/
…
```

5.4.17 函数 can_doze_mode_enter

下表描述了函数 can_doze_mode_enter

表 86. 函数 can_doze_mode_enter

项目	描述
函数名	can_doze_mode_enter
函数原型	can_enter_doze_status_type can_doze_mode_enter(can_type* can_x);
功能描述	进入睡眠模式
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输出参数	无
返回值	*can_enter_doze_status*：进入睡眠模式是否成功
先决条件	无
被调用函数	无

can_enter_doze_status

进入睡眠模式是否成功

CAN_ENTER_DOZE_FAILED：　　进入睡眠模式失败
CAN_ENTER_DOZE_SUCCESSFUL：　进入睡眠模式成功

示例

```
/* can enter the low power mode */
can_enter_doze_status_type can_enter_doze_status;
can_enter_doze_status = can_doze_mode_enter(CAN1);
```

5.4.18 函数 can_doze_mode_exit

下表描述了函数 can_doze_mode_exit

表 87. 函数 can_doze_mode_exit

项目	描述
函数名	can_doze_mode_exit
函数原型	can_quit_doze_status_type can_doze_mode_exit(can_type* can_x);
功能描述	退出睡眠模式
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输出参数	无
返回值	*can_quit_doze_status*：退出睡眠模式是否成功
先决条件	无
被调用函数	无

can_quit_doze_status

退出睡眠模式是否成功

CAN_QUIT_DOZE_FAILED： 退出睡眠模式失败

CAN_QUIT_DOZE_SUCCESSFUL： 退出睡眠模式成功

示例

```
/* can exit the low power mode */
can_quit_doze_status_type can_quit_doze_status;
can_quit_doze_status = can_doze_mode_exit (CAN1);
```

5.4.19 函数 can_error_type_record_get

下表描述了函数 can_error_type_record_get

表 88. 函数 can_error_type_record_get

项目	描述
函数名	can_error_type_record_get
函数原型	can_error_record_type can_error_type_record_get(can_type* can_x);
功能描述	读取 CAN 错误类型
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输出参数	无
返回值	*can_error_record*：错误类型
先决条件	无
被调用函数	无

can_error_record

错误类型

CAN_ERRORRECORD_NOERR： 没有错误产生

CAN_ERRORRECORD_STUFFERR： 位填充错误

CAN_ERRORRECORD_FORMERR： 格式错误

CAN_ERRORRECORD_ACKERR： 应答错误

CAN_ERRORRECORD_BITRECESSIVEERR： 隐性位错误

CAN_ERRORRECORD_BITDOMINANTERR： 显性位错误

CAN_ERRORRECORD_CRCERR： CRC 校验错误

CAN_ERRORRECORD_SOFTWARESETERR： 软件设置错误

示例

```
/* get the error type record (etr) */
can_error_record_type can_error_record;
can_error_record = can_error_type_record_get (CAN1);
```

5.4.20 函数 can_receive_error_counter_get

下表描述了函数 can_receive_error_counter_get

表 89. 函数 can_receive_error_counter_get

项目	描述
函数名	can_receive_error_counter_get
函数原型	uint8_t can_receive_error_counter_get(can_type* can_x);
功能描述	读取 CAN 接收错误计数
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输出参数	无
返回值	receive_error_counter：接收错误计数 参数范围：0x00~0xFF
先决条件	无
被调用函数	无

示例

```
/* get the receive error counter (rec) */
uint8_t receive_error_counter;
receive_error_counter = can_receive_error_counter_get (CAN1);
```

5.4.21 函数 can_transmit_error_counter_get

下表描述了函数 can_transmit_error_counter_get

表 90. 函数 can_transmit_error_counter_get

项目	描述
函数名	can_transmit_error_counter_get
函数原型	uint8_t can_transmit_error_counter_get(can_type* can_x);
功能描述	读取 CAN 发送错误计数
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输出参数	无
返回值	transmit_error_counter：发送错误计数 参数范围：0x00~0xFF
先决条件	无
被调用函数	无

示例

```
/* get the transmit error counter (tec) */
uint8_t transmit_error_counter;
transmit_error_counter = can_transmit_error_counter_get (CAN1);
```

5.4.22 函数 can_interrupt_enable

下表描述了函数 can_interrupt_enable

表 91. 函数 can_interrupt_enable

项目	描述
函数名	can_interrupt_enable
函数原型	void can_interrupt_enable(can_type* can_x, uint32_t can_int, confirm_state new_state);
功能描述	使能选定的 CAN 中断
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	*can_int*：CAN 中断选择
输入参数 3	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

can_int

CAN 中断选择

CAN_TCIEN_INT： 发送邮箱发送完成中断使能
CAN_RF0MIEN_INT： 接收 FIFO 0 报文接收中断使能
CAN_RF0FIEN_INT： 接收 FIFO 0 满中断使能
CAN_RF0OIEN_INT： 接收 FIFO 0 溢出中断使能
CAN_RF1MIEN_INT： 接收 FIFO 1 报文接收中断使能
CAN_RF1FIEN_INT： 接收 FIFO 1 满中断使能
CAN_RF1OIEN_INT： 接收 FIFO 1 溢出中断使能
CAN_EAIEN_INT： 错误警告中断使能
CAN_EPIEN_INT： 错误被动中断使能
CAN_BOIEN_INT： 总线关闭中断使能
CAN_ETRIEN_INT： 错误类型记录中断使能
CAN_EOIEN_INT： 出现错误的中断使能
CAN_QDZIEN_INT： 退出睡眠模式的中断使能
CAN_EDZIEN_INT： 进入睡眠模式的中断使能

示例

```
/* can interrupt config */
nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00);/*CAN1 错误/状态变化中断*/
nvic_irq_enable(USBFS_L_CAN1_RX0_IRQn, 0x00, 0x00);/*CAN1 FIFO0 接收中断*/

/* FIFO 0 receive message interrupt enable */
```

```
can_interrupt_enable(CAN1, CAN_RF0MIEN_INT, TRUE);
/* error type record interrupt enable */
can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE);

/*此项为错误中断总开关，需要使能错误相关中断必须要使能此项*/
can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE);
```

5.4.23 函数 can_flag_get

下表描述了函数 can_flag_get

表 92. 函数 can_flag_get

项目	描述
函数名	can_flag_get
函数原型	flag_status can_flag_get(can_type* can_x, uint32_t can_flag);
功能描述	获取所选择的 CAN 标志
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	*can_flag*：需要获取状态的标志选择 该参数详细描述见 can_flag
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET，RESET
先决条件	无
被调用函数	无

can_flag

CAN 用于选择需要获取状态的标志，其可选参数罗列如下：

CAN_EAF_FLAG：　错误主动标志
CAN_EPF_FLAG：　错误被动标志
CAN_BOF_FLAG：　总线关闭标志
CAN_ETR_FLAG：　错误类型记录标志（错误类型非 0 标志）
CAN_EOIF_FLAG：　出现错误标志
CAN_TM0TCF_FLAG：　邮箱 0 发送完成标志
CAN_TM1TCF_FLAG：　邮箱 1 发送完成标志
CAN_TM2TCF_FLAG：　邮箱 2 发送完成标志
CAN_RF0MN_FLAG：　FIFO0 非空标志
CAN_RF0FF_FLAG：　FIFO0 满标志
CAN_RF0OF_FLAG：　FIFO0 溢出标志
CAN_RF1MN_FLAG：　FIFO1 非空标志
CAN_RF1FF_FLAG：　FIFO1 满标志
CAN_RF1OF_FLAG：　FIFO1 溢出标志
CAN_QDZIF_FLAG：　退出睡眠模式标志
CAN_EDZC_FLAG：　进入睡眠模式标志
CAN_TMEF_FLAG：　发送邮箱空标志（三个发送邮箱任一为空）

示例

```
/* get receive fifo 0 message num flag */
flag_status bit_status = RESET;
bit_status = can_flag_get (CAN1, CAN_RF0MN_FLAG);
```

5.4.24 函数 can_flag_clear

下表描述了函数 can_flag_clear

表 93. 函数 can_flag_clear

项目	描述
函数名	can_flag_clear
函数原型	void can_flag_clear(can_type* can_x, uint32_t can_flag);
功能描述	清除选定的 CAN 标志
输入参数 1	can_x：所选择的 CAN 外设 该参数可以选取自其中之一：CAN1，CAN2.
输入参数 2	*can_flag*：待清除的标志选择 该参数详细描述见 can_flag
输出参数	无
返回值	无
先决条件	无
被调用函数	无

can_flag：

CAN 用于选择需要清除的标志，其可选参数罗列如下：

CAN_EAF_FLAG： 错误主动标志
CAN_EPF_FLAG： 错误被动标志
CAN_BOF_FLAG： 总线关闭标志
CAN_ETR_FLAG： 错误类型记录标志（错误类型非 0 标志）
CAN_EOIF_FLAG： 出现错误标志
CAN_TM0TCF_FLAG： 邮箱 0 发送完成标志
CAN_TM1TCF_FLAG： 邮箱 1 发送完成标志
CAN_TM2TCF_FLAG： 邮箱 2 发送完成标志
CAN_RF0FF_FLAG： FIFO0 满标志
CAN_RF0OF_FLAG： FIFO0 溢出标志
CAN_RF1FF_FLAG： FIFO1 满标志
CAN_RF1OF_FLAG： FIFO1 溢出标志
CAN_QDZIF_FLAG： 退出睡眠模式标志
CAN_EDZC_FLAG： 进入睡眠模式标志
CAN_TMEF_FLAG： 发送邮箱空标志（三个发送邮箱任一为空）

注意：CAN_RF0MN_FLAG（FIFO0非空标志）和CAN_RF1MN_FLAG（FIFO1非空标志）是软件自定义的标志，因此不存在清除操作。

示例

```
/* clear receive fifo 0 overflow flag */
can_flag_clear (CAN1, CAN_RF1OF_FLAG);
```

5.5 CRC 计算单元（CRC）

CRC 寄存器结构 crc_type，定义于文件“at32f413_crc.h”如下：

```
/**
  * @brief type define crc register all
  */
typedef struct
{
      ...

} crc_type;
```

下表给出了 CRC 寄存器总览：

表 94. CRC 寄存器对应表

寄存器	描述
dt	数据寄存器
cdt	通用数据寄存器
ctrl	控制寄存器
idt	初始化寄存器
poly	生成多项式寄存器

下表给出了 CRC 库函数总览：

表 95. CRC 库函数总览

函数名	描述
crc_data_reset	数据寄存器复位
crc_one_word_calculate	输入一个 32-bit 数据与上一次计算结果进行 CRC 计算并返回计算结果
crc_block_calculate	依次写入一个数据块逐次进行 CRC 计算并返回计算结果
crc_data_get	返回当前 CRC 计算结果
crc_common_data_set	设置通用寄存器值
crc_common_date_get	返回通用寄存器值
crc_init_data_set	设置 CRC 初始化寄存器值
crc_reverse_input_data_set	设置 CRC 输入数据 bit 反转数据类型
crc_reverse_output_data_set	设置 CRC 输出数据反转类型
crc_poly_value_set	设置多项式参数值
crc_poly_value_get	获取当前多项式参数值
crc_poly_size_set	设置多项式有效宽度
crc_poly_size_get	获取当前多项式有效宽度

5.5.1 函数 crc_data_reset

下表描述了函数 crc_data_reset

表 96. 函数 crc_data_reset

项目	描述
函数名	crc_data_reset

项目	描述
函数原型	void crc_data_reset(void);
功能描述	数据寄存器复位，会将初始化寄存器的值刷新到数据寄存器作为初始值，默认复位值为 0xFFFFFFFF
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* reset crc data register */
crc_data_reset();
```

5.5.2 函数 crc_one_word_calculate

下表描述了函数 crc_one_word_calculate

表 97. 函数 crc_one_word_calculate

项目	描述
函数名	crc_one_word_calculate
函数原型	uint32_t crc_one_word_calculate(uint32_t data);
功能描述	输入一个 32-bit 数据与上一次计算结果进行 CRC 计算并返回计算结果
输入参数 1	data：输入计算的 32-bit 数据
输入参数 2	无
输出参数	无
返回值	uint32_t：返回 CRC 计算结果
先决条件	无
被调用函数	无

示例

```
/* calculate and return result */
uint32_t data = 0x12345678, result = 0;
result = crc_one_word_calculate (data);
```

5.5.3 函数 crc_block_calculate

下表描述了函数 crc_block_calculate

表 98. 函数 crc_block_calculate

项目	描述
函数名	crc_block_calculate
函数原型	uint32_t crc_block_calculate(uint32_t *pbuffer, uint32_t length);
功能描述	依次写入一个数据块逐次进行 CRC 计算并返回计算结果
输入参数 1	pbuffer：指针指向待进行 CRC 计算的数据块
输入参数 2	length：待计算数据块长度，长度以 32-bit 计算
输出参数	无

项目	描述
返回值	uint32_t：返回 CRC 计算结果
先决条件	无
被调用函数	无

示例

```
/* calculate and return result */
uint32_t pbuffer[2] = {0x12345678, 0x87654321};
uint32_t result = 0;
result = crc_block_calculate (pbuffer, 2);
```

5.5.4 函数 crc_data_get

下表描述了函数 crc_data_get

表 99. 函数 crc_data_get

项目	描述
函数名	crc_data_get
函数原型	uint32_t crc_data_get(void);
功能描述	返回当前 CRC 计算结果
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	uint32_t：返回 CRC 计算结果
先决条件	无
被调用函数	无

示例

```
/* get result */
uint32_t result = 0;
result = crc_data_get ();
```

5.5.5 函数 crc_common_data_set

下表描述了函数 crc_common_data_set

表 100. 函数 crc_common_data_set

项目	描述
函数名	crc_common_data_set
函数原型	void crc_common_data_set(uint8_t cdt_value);
功能描述	设置通用寄存器值
输入参数 1	cdt_value：8-bit 通用数据，可作为临时存储数据使用
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set common data */
crc_common_data_set (0x88);
```

5.5.6 函数 crc_common_data_get

下表描述了函数 crc_common_data_get

表 101. 函数 crc_common_data_get

项目	描述
函数名	crc_common_data_get
函数原型	uint8_t crc_common_data_get(void);
功能描述	返回通用寄存器值
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	uint8_t：返回之前设置的通用寄存器值
先决条件	无
被调用函数	无

示例

```
/* get common data */
uint8_t cdt_value = 0;
cdt_value = crc_common_data_get ();
```

5.5.7 函数 crc_init_data_set

下表描述了函数 crc_init_data_set

表 102. 函数 crc_init_data_set

项目	描述
函数名	crc_init_data_set
函数原型	void crc_init_data_set(uint32_t value);
功能描述	设置 CRC 初始化寄存器值
输入参数 1	value：CRC 初始化寄存器值
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

CRC 初始化寄存器值设定好后，每次进行 crc_data_reset 函数调用时会将 CRC 初始化寄存器的值刷新到 CRC 数据寄存器。

示例

```
/* set initial data */
uint32_t init_value = 0x11223344;
crc_init_data_set (init_value);
```

5.5.8 函数 crc_reverse_input_data_set

下表描述了函数 crc_reverse_input_data_set

表 103. 函数 crc_reverse_input_data_set

项目	描述
函数名	crc_reverse_input_data_set
函数原型	void crc_reverse_input_data_set(crc_reverse_input_type value);
功能描述	设置 CRC 输入数据 bit 反转数据类型
输入参数 1	value：输入数据 bit 反转类型 该参数详细描述见 value
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

指定输入数据 bit 反转数据类型

CRC_REVERSE_INPUT_NO_AFFECTE: 数据数据不进行 bit 反转

CRC_REVERSE_INPUT_BY_BYTE: 32-bit 数据按字节进行 bit 反转

CRC_REVERSE_INPUT_BY_HALFWORD: 32-bit 数据按半字进行 bit 反转

CRC_REVERSE_INPUT_BY_WORD: 32-bit 数据按字进行 bit 反转

示例

```
/* set input data reversing type */
crc_reverse_input_data_set(CRC_REVERSE_INPUT_BY_WORD);
```

5.5.9 函数 crc_reverse_output_data_set

下表描述了函数 crc_reverse_output_data_set

表 104. 函数 crc_reverse_output_data_set

项目	描述
函数名	crc_reverse_output_data_set
函数原型	void crc_reverse_output_data_set(crc_reverse_output_type value);
功能描述	设置 CRC 输出数据反转类型
输入参数 1	value：输出数据 bit 反转类型 该参数详细描述见 value
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

指定输出数据 bit 反转数据类型

CRC_REVERSE_OUTPUT_NO_AFFECTE: 输出数据不进行 bit 反转

CRC_REVERSE_OUTPUT_DATA: 32-bit 输出数据按字进行 bit 反转

示例

```
/* set output data reversing type */
crc_reverse_output_data_set (CRC_REVERSE_OUTPUT_DATA);
```

5.5.10 函数 crc_poly_value_set

下表描述了函数 crc_poly_value_set

表 105. 函数 crc_poly_value_set

项目	描述
函数名	crc_poly_value_set
函数原型	void crc_poly_value_set(uint32_t value);
功能描述	设置 CRC 多项式参数值
输入参数 1	value：多项式值
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set poly value */
crc_poly_value_set(0x12345671);
```

5.5.11 函数 crc_poly_value_get

下表描述了函数 crc_poly_value_get

表 106. 函数 crc_poly_value_get

项目	描述
函数名	crc_poly_value_get
函数原型	uint32_t crc_poly_value_get(void);
功能描述	获取当前 CRC 多项式值
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	uint32_t：当前多项式值
先决条件	无
被调用函数	无

示例

```
/* get poly value */
uint32_t poly = 0;
poly = crc_poly_value_get();
```

5.5.12 函数 crc_poly_size_set

下表描述了函数 crc_poly_size_set

表 107. 函数 crc_poly_size_set

项目	描述
函数名	crc_poly_size_set
函数原型	void crc_poly_size_set(crc_poly_size_type size);
功能描述	设置 CRC 多项式有效宽度
输入参数 1	size：多项式有效宽度
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

size

指定多项式有效宽度类型

CRC_POLY_SIZE_32B: 多项式有效宽度 32-bit

CRC_POLY_SIZE_16B: 多项式有效宽度 16-bit

CRC_POLY_SIZE_8B: 多项式有效宽度 8-bit

CRC_POLY_SIZE_7B: 多项式有效宽度 7-bit

示例

```
/* set poly size 32-bit */
crc_poly_size_set(CRC_POLY_SIZE_32B);
```

5.5.13 函数 crc_poly_size_get

下表描述了函数 crc_poly_size_get

表 108. 函数 crc_poly_size_get

项目	描述
函数名	crc_poly_size_get
函数原型	crc_poly_size_type crc_poly_size_get(void);
功能描述	返回 CRC 多项式有效宽度
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	crc_poly_size_type：多项式有效宽度类型
先决条件	无
被调用函数	无

crc_poly_size_type

多项式有效宽度类型

CRC_POLY_SIZE_32B: 多项式有效宽度 32-bit

CRC_POLY_SIZE_16B: 多项式有效宽度 16-bit

CRC_POLY_SIZE_8B: 多项式有效宽度 8-bit

CRC_POLY_SIZE_7B: 多项式有效宽度 7-bit

示例

```
/* get poly size */
crc_poly_size_type size;
size = crc_poly_size_get();
```

5.6 时钟和复位管理（CRM）

5.6.1 函数 crm_reset

下表描述了函数 crm_reset

表 109. 函数 crm_reset

项目	描述
函数名	crm_reset
函数原型	void crm_reset(void);
功能描述	将时钟复位管理模块的寄存器和控制状态复位
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

1. 该函数不改动寄存器 CRM_CTRL 的 HICKTRIM[5:0]位。
2. 改函数不重置寄存器 CRM_BPDC 和寄存器 CRM_CTRLSTS。

示例

```
/* reset crm */
crm_reset();
```

5.6.2 函数 crm_lext_bypass

下表描述了函数 crm_lext_bypass

表 110. 函数 crm_lext_bypass

项目	描述
函数名	crm_lext_bypass
函数原型	void crm_lext_bypass(confirm_state new_state);
功能描述	低速外部时钟旁路设置
输入参数 1	new_state：lext 旁路的新状态。使能旁路（TRUE），关闭旁路（FALSE）
输入参数 2	无
输出参数	无
返回值	无
先决条件	外部低速时钟在未使能的情况下进行设定
被调用函数	无

示例

```
/* enable lext bypass mode */
crm_lext_bypass(TRUE);
```

5.6.3 函数 crm_hext_bypass

下表描述了函数 crm_hext_bypass

表 111. 函数 crm_hext_bypass

项目	描述
函数名	crm_hext_bypass
函数原型	void crm_hext_bypass(confirm_state new_state);
功能描述	高速外部时钟旁路设置
输入参数 1	new_state：hext 旁路的新状态。使能旁路（TRUE），关闭旁路（FALSE）
输入参数 2	无
输出参数	无
返回值	无
先决条件	外部高速时钟在未使能的情况下进行设定
被调用函数	无

示例

```
/* enable hext bypass mode */
crm_hext_bypass(TRUE);
```

5.6.4 函数 crm_flag_get

下表描述了函数 crm_flag_get

表 112. 函数 crm_flag_get

项目	描述
函数名	crm_flag_get
函数原型	flag_status crm_flag_get(uint32_t flag);
功能描述	检查指定的 flag 标志位设置与否
输入参数 1	flag：指定的需要读取判断的 flag 标志
输入参数 2	无
输出参数	无
返回值	flag_status：flag 标志是否置起。置起（SET），未置起（RESET）
先决条件	无
被调用函数	无

flag

指定需要读取判断的 flag 标志

CRM_HICK_STABLE_FLAG： 内部高速时钟稳定标志
CRM_HEXT_STABLE_FLAG： 外部高速时钟稳定标志
CRM_PLL_STABLE_FLAG： PLL 时钟稳定标志
CRM_LEXT_STABLE_FLAG： 外部低速时钟稳定标志
CRM_LICK_STABLE_FLAG： 内部低速时钟稳定标志
CRM_NRST_RESET_FLAG： NRST 管脚复位标志
CRM_POR_RESET_FLAG： 上电/低电压复位标志
CRM_SW_RESET_FLAG： 软件复位标志标志
CRM_WDT_RESET_FLAG： 看门狗复位标志
CRM_WWDT_RESET_FLAG： 窗口看门狗复位标志
CRM_LOWPOWER_RESET_FLAG： 低功耗复位标志
CRM_LICK_READY_INT_FLAG： 低速内部时钟稳定中断标志
CRM_LEXT_READY_INT_FLAG： 低速外部时钟稳定中断标志
CRM_HICK_READY_INT_FLAG： 高速内部时钟稳定中断标志

CRM_HEXT_READY_INT_FLAG： 高速外部时钟稳定中断标志
CRM_PLL_READY_INT_FLAG： PLL 时钟稳定中断标志
CRM_CLOCK_FAILURE_INT_FLAG： 时钟失效中断标志

示例

```
/* wait till pll is ready */
while(crm_flag_get(CRM_PLL_STABLE_FLAG) != SET)
{
}
```

5.6.5 函数 crm_hext_stable_wait

下表描述了函数 crm_hext_stable_wait

表 113. 函数 crm_hext_stable_wait

项目	描述
函数名	crm_hext_stable_wait
函数原型	error_status crm_hext_stable_wait(void);
功能描述	等待外部高速时钟起振并稳定
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	error_status：返回起振和稳定状态。成功（SUCCESS），失败（ERROR）
先决条件	无
被调用函数	无

示例

```
/* wait till hext is ready */
while(crm_hext_stable_wait() == ERROR)
{
}
```

5.6.6 函数 crm_hick_clock_trimming_set

下表描述了函数 crm_hick_clock_trimming_set

表 114. 函数 crm_hick_clock_trimming_set

项目	描述
函数名	crm_hick_clock_trimming_set
函数原型	void crm_hick_clock_trimming_set(uint8_t trim_value);
功能描述	步进调整内部高速时钟校准值
输入参数 1	trim_value：校准补偿值。默认值为 0x20，设置范围为 0~0x3F
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set trimming value */
crm_hick_clock_trimming_set(0x1F);
```

5.6.7 函数 crm_hick_clock_calibration_set

下表描述了函数 crm_hick_clock_calibration_set

表 115. 函数 crm_hick_clock_calibration_set

项目	描述
函数名	crm_hick_clock_calibration_set
函数原型	void crm_hick_clock_calibration_set(uint8_t cali_value);
功能描述	调整内部高速时钟校准值
输入参数 1	cali_value：校准补偿值。默认值为出厂校准值，设置范围为 0~0xFF
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set trimming value */
crm_hick_clock_trimming_set(0x80);
```

5.6.8 函数 crm_periph_clock_enable

下表描述了函数 crm_periph_clock_enable

表 116. 函数 crm_periph_clock_enable

项目	描述
函数名	crm_periph_clock_enable
函数原型	void crm_periph_clock_enable(crm_periph_clock_type value, confirm_state new_state);
功能描述	外设时钟使能设置
输入参数 1	value：指定的片上外设时钟类型
输入参数 2	new_state：新的时钟设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

指定的外设，crm_periph_clock_type 在 at32f413_crm.h 中。此参数类型的命名规则为：CRM_外设名_PERIPH_CLOCK。

CRM_DMA1_PERIPH_CLOCK： 外设 dma1 的外设时钟定义

CRM_DMA2_PERIPH_CLOCK： 外设 dma2 的外设时钟定义

…

CRM_PWC_PERIPH_CLOCK： 外设 pwc 的外设时钟定义

示例

```
/* enable gpioa periph clock */
crm_periph_clock_enable(CRM_GPIOA_PERIPH_CLOCK, TRUE);
```

5.6.9 函数 crm_periph_reset

下表描述了函数 crm_periph_reset

表 117. 函数 crm_periph_reset

项目	描述
函数名	crm_periph_reset
函数原型	void crm_periph_reset(crm_periph_reset_type value, confirm_state new_state);
功能描述	外设复位设置
输入参数 1	value：指定的片上外设复位类型
输入参数 2	new_state：新的时钟设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

指定的外设，crm_periph_reset_type 在 at32f413_crm.h 中。此参数类型的命名规则为：CRM_外设名_PERIPH_RESET。

CRM_DMA1_PERIPH_RESET： 外设 dma1 的外设复位定义

CRM_DMA2_PERIPH_RESET： 外设 dma2 的外设复位定义

…

CRM_PWC_PERIPH_RESET： 外设 pwc 的外设复位定义

示例

```
/* reset gpioa periph */
crm_periph_reset(CRM_GPIOA_PERIPH_RESET, TRUE);
```

5.6.10 函数 crm_periph_sleep_mode_clock_enable

下表描述了函数 crm_periph_sleep_mode_clock_enable

表 118. 函数 crm_periph_sleep_mode_clock_enable

项目	描述
函数名	crm_periph_sleep_mode_clock_enable
函数原型	void crm_periph_sleep_mode_clock_enable(crm_periph_clock_sleepmd_type value, confirm_state new_state);
功能描述	外设在睡眠模式下时钟使能设置
输入参数 1	value：指定的片上外设睡眠模式下时钟类型
输入参数 2	new_state：新的时钟设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

指定的外设，crm_periph_clock_sleepmd_type 在 at32f413_crm.h 中。此参数类型的命名规则为：CRM_外设名_PERIPH_CLOCK_SLEEP_MODE。

CRM_SRAM_PERIPH_RESET： 外设 sram 在睡眠模式下的时钟定义

CRM_FLASH_PERIPH_RESET： 外设 flash 在睡眠模式下的时钟定义

示例

```
/* disable flash clock when entry sleep mode */
crm_periph_sleep_mode_clock_enable (CRM_FLASH_PERIPH_CLOCK_SLEEP_MODE, FALSE);
```

5.6.11 函数 crm_clock_source_enable

下表描述了函数 crm_clock_source_enable

表 119. 函数 crm_clock_source_enable

项目	描述
函数名	crm_clock_source_enable
函数原型	void crm_clock_source_enable(crm_clock_source_type source, confirm_state new_state);
功能描述	各时钟源使能设置
输入参数 1	source：指定的时钟源类型
输入参数 2	new_state：新的时钟设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

source

指定的时钟源。

CRM_CLOCK_SOURCE_HICK： 高速内部时钟源

CRM_CLOCK_SOURCE_HEXT： 高速外部时钟源

CRM_CLOCK_SOURCE_PLL： PLL 时钟源

CRM_CLOCK_SOURCE_LEXT： 低速外部时钟源

CRM_CLOCK_SOURCE_LICK： 低速内部时钟源

示例

```
/* enable hext */
crm_clock_source_enable (CRM_CLOCK_SOURCE_HEXT, FALSE);
```

5.6.12 函数 crm_flag_clear

下表描述了函数 crm_flag_clear

表 120. 函数 crm_flag_clear

项目	描述
函数名	crm_flag_clear
函数原型	void crm_flag_clear(uint32_t flag);
功能描述	清除指定标志位
输入参数 1	flag：指定的需要清除的 flag 标志

项目	描述
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flag

指定需要清除的 flag 标志

CRM_NRST_RESET_FLAG： NRST 管脚复位标志
CRM_POR_RESET_FLAG： 上电/低电压复位标志
CRM_SW_RESET_FLAG： 软件复位标志标志
CRM_WDT_RESET_FLAG： 看门狗复位标志
CRM_WWDT_RESET_FLAG： 窗口看门狗复位标志
CRM_LOWPOWER_RESET_FLAG： 低功耗复位标志
CRM_ALL_RESET_FLAG： 所有复位标志
CRM_LICK_READY_INT_FLAG： 低速内部时钟稳定中断标志
CRM_LEXT_READY_INT_FLAG： 低速外部时钟稳定中断标志
CRM_HICK_READY_INT_FLAG： 高速内部时钟稳定中断标志
CRM_HEXT_READY_INT_FLAG： 高速外部时钟稳定中断标志
CRM_PLL_READY_INT_FLAG： PLL 时钟稳定中断标志
CRM_CLOCK_FAILURE_INT_FLAG： 时钟失效中断标志

示例

```
/* clear clock failure detection flag */
crm_flag_clear(CRM_CLOCK_FAILURE_INT_FLAG);
```

5.6.13 函数 crm_rtc_clock_select

下表描述了函数 crm_rtc_clock_select

表 121. 函数 crm_rtc_clock_select

项目	描述
函数名	crm_rtc_clock_select
函数原型	void crm_rtc_clock_select(crm_rtc_clock_type value);
功能描述	rtc 时钟源选择
输入参数 1	value：需设置的 rtc 时钟源类型
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

指定需要设置的 rtc 时钟源

CRM_RTC_CLOCK_NOCLK： 无时钟源选作 rtc 时钟
CRM_RTC_CLOCK_LEXT： 外部低速时钟选作 rtc 时钟
CRM_RTC_CLOCK_LICK： 内部低速时钟选择 rtc 时钟
CRM_RTC_CLOCK_HEXT_DIV： 外部高速时钟 128 分频后选作 rtc 时钟

示例

```
/* config lext as rtc clock */
crm_rtc_clock_select (CRM_RTC_CLOCK_LEXT);
```

5.6.14 函数 crm_rtc_clock_enable

下表描述了函数 crm_rtc_clock_enable

表 122. 函数 crm_rtc_clock_enable

项目	描述
函数名	crm_rtc_clock_enable
函数原型	void crm_rtc_clock_enable(confirm_state new_state);
功能描述	rtc 时钟使能设置
输入参数 1	new_state：新的 rtc 时钟设置状态。开启（TRUE），关闭（FALSE）
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable rtc clock */
crm_rtc_clock_enable (TRUE);
```

5.6.15 函数 crm_ahb_div_set

下表描述了函数 crm_ahb_div_set

表 123. 函数 crm_ahb_div_set

项目	描述
函数名	crm_ahb_div_set
函数原型	void crm_ahb_div_set(crm_ahb_div_type value);
功能描述	SCLK 到 AHB 时钟的分频设置
输入参数 1	value：需设置的分频值
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

CRM_AHB_DIV_1： SCLK 时钟 1 分频作为 AHB 时钟
CRM_AHB_DIV_2： SCLK 时钟 2 分频作为 AHB 时钟
CRM_AHB_DIV_4： SCLK 时钟 4 分频作为 AHB 时钟
CRM_AHB_DIV_8： SCLK 时钟 8 分频作为 AHB 时钟
CRM_AHB_DIV_16： SCLK 时钟 16 分频作为 AHB 时钟
CRM_AHB_DIV_64： SCLK 时钟 64 分频作为 AHB 时钟
CRM_AHB_DIV_128： SCLK 时钟 128 分频作为 AHB 时钟

CRM_AHB_DIV_256： SCLK 时钟 256 分频作为 AHB 时钟
CRM_AHB_DIV_512： SCLK 时钟 512 分频作为 AHB 时钟

示例

```
/* config ahbclk */
crm_ahb_div_set(CRM_AHB_DIV_1);
```

5.6.16 函数 crm_apb1_div_set

下表描述了函数 crm_apb1_div_set

表 124. 函数 crm_apb1_div_set

项目	描述
函数名	crm_apb1_div_set
函数原型	void crm_apb1_div_set(crm_apb1_div_type value);
功能描述	AHB 时钟到 APB1 时钟的分频设置
输入参数 1	value：需设置的分频值
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

CRM_APB1_DIV_1： AHB 时钟 1 分频作为 APB1 时钟
CRM_APB1_DIV_2： AHB 时钟 2 分频作为 APB1 时钟
CRM_APB1_DIV_4： AHB 时钟 4 分频作为 APB1 时钟
CRM_APB1_DIV_8： AHB 时钟 8 分频作为 APB1 时钟
CRM_APB1_DIV_16： AHB 时钟 16 分频作为 APB1 时钟

示例

```
/* config apb1clk */
crm_apb1_div_set(CRM_APB1_DIV_2);
```

5.6.17 函数 crm_apb2_div_set

下表描述了函数 crm_apb2_div_set

表 125. 函数 crm_apb2_div_set

项目	描述
函数名	crm_apb2_div_set
函数原型	void crm_apb2_div_set(crm_apb2_div_type value);
功能描述	AHB 时钟到 APB2 时钟的分频设置
输入参数 1	value：需设置的分频值
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

CRM_APB2_DIV_1： AHB 时钟 1 分频作为 APB2 时钟
CRM_APB2_DIV_2： AHB 时钟 2 分频作为 APB2 时钟
CRM_APB2_DIV_4： AHB 时钟 4 分频作为 APB2 时钟
CRM_APB2_DIV_8： AHB 时钟 8 分频作为 APB2 时钟
CRM_APB2_DIV_16： AHB 时钟 16 分频作为 APB2 时钟

示例

```
/* config apb2clk */
crm_apb2_div_set(CRM_APB2_DIV_2);
```

5.6.18 函数 crm_adc_clock_div_set

下表描述了函数 crm_adc_clock_div_set

表 126. 函数 crm_adc_clock_div_set

项目	描述
函数名	crm_adc_clock_div_set
函数原型	void crm_adc_clock_div_set(crm_adc_div_type div_value);
功能描述	ADC 时钟分频设置
输入参数 1	div_value：需设置的分频值
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

div_value

CRM_ADC_DIV_2： APB 时钟 2 分频作为 ADC 时钟
CRM_ADC_DIV_4： APB 时钟 4 分频作为 ADC 时钟
CRM_ADC_DIV_6： APB 时钟 6 分频作为 ADC 时钟
CRM_ADC_DIV_8： APB 时钟 8 分频作为 ADC 时钟
CRM_ADC_DIV_12： APB 时钟 12 分频作为 ADC 时钟
CRM_ADC_DIV_16： APB 时钟 16 分频作为 ADC 时钟

示例

```
/* config adc div 4 */
crm_adc_clock_div_set (CRM_ADC_DIV_4);
```

5.6.19 函数 crm_usb_clock_div_set

下表描述了函数 crm_usb_clock_div_set

表 127. 函数 crm_usb_clock_div_set

项目	描述
函数名	crm_usb_clock_div_set
函数原型	void crm_usb_clock_div_set(crm_usb_div_type div_value);
功能描述	PLL 时钟到 USB 时钟的分频设置
输入参数 1	div_value：需设置的分频值

项目	描述
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

div_value

CRM_USB_DIV_1_5： PLL 时钟 1.5 倍分频作为 USB 时钟
CRM_USB_DIV_1： PLL 时钟 1 倍分频作为 USB 时钟
CRM_USB_DIV_2_5： PLL 时钟 2.5 倍分频作为 USB 时钟
CRM_USB_DIV_2： PLL 时钟 2 倍分频作为 USB 时钟
CRM_USB_DIV_3_5： PLL 时钟 3.5 倍分频作为 USB 时钟
CRM_USB_DIV_3： PLL 时钟 3 倍分频作为 USB 时钟
CRM_USB_DIV_4： PLL 时钟 4 倍分频作为 USB 时钟

示例

```
/* config usb div 2 */
crm_usb_clock_div_set (CRM_USB_DIV_2);
```

5.6.20 函数 crm_clock_failure_detection_enable

下表描述了函数 crm_clock_failure_detection_enable

表 128. 函数 crm_clock_failure_detection_enable

项目	描述
函数名	crm_clock_failure_detection_enable
函数原型	void crm_clock_failure_detection_enable(confirm_state new_state);
功能描述	时钟失效检测功能使能设置
输入参数 1	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable clock failure detection */
crm_clock_failure_detection_enable(TRUE);
```

5.6.21 函数 crm_battery_powered_domain_reset

下表描述了函数 crm_battery_powered_domain_reset

表 129. 函数 crm_battery_powered_domain_reset

项目	描述
函数名	crm_battery_powered_domain_reset
函数原型	void crm_battery_powered_domain_reset(confirm_state new_state);
功能描述	电池供电域的复位设置

项目	描述
输入参数 1	new_state：新的设置状态。复位（TRUE），不复位（FALSE）
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

在需要对电池供电域复位时，通常的操作流程是先 TRUE 设定对电池供电域进行复位，完成复位后再 FALSE 设定关闭电池供电域复位。

示例

```
/* reset battery powered domain */
crm_battery_powered_domain_reset (TRUE);
```

5.6.22 函数 crm_pll_config

下表描述了函数 crm_pll_config

表 130. 函数 crm_pll_config

项目	描述
函数名	crm_pll_config
函数原型	void crm_pll_config(crm_pll_clock_source_type clock_source, crm_pll_mult_type mult_value, crm_pll_output_range_type pll_range);
功能描述	设置 PLL 时钟源及倍频系数
输入参数 1	clock_source：PLL 倍频时钟源
输入参数 2	mult_value：倍频系数
输入参数 3	pll_range：按 pll 输出频率的范围是否大于 72MHz 来进行设定
输出参数	无
返回值	无
先决条件	在配置和使能 PLL 前应确保 pll 倍频时钟源已开启且稳定
被调用函数	无

clock_source

CRM_PLL_SOURCE_HICK：　选择内部高速时钟作为 PLL 时钟源
CRM_PLL_SOURCE_HEXT：　选择外部高速时钟作为 PLL 时钟源
CRM_PLL_SOURCE_HEXT_DIV：选择外部高速时钟分频后作为 PLL 时钟源

mult_value

CRM_PLL_MULT_2：　PLL 按输入时钟的 2 倍进行倍频
CRM_PLL_MULT_3：　PLL 按输入时钟的 3 倍进行倍频
…
CRM_PLL_MULT_63：　PLL 按输入时钟的 63 倍进行倍频
CRM_PLL_MULT_64：　PLL 按输入时钟的 64 倍进行倍频

pll_range

CRM_PLL_OUTPUT_RANGE_LE72MHZ：PLL 输出频率小于等于 72MHz 时所需设定
CRM_PLL_OUTPUT_RANGE_GT72MHZ：PLL 输出频率大于 72MHz 时所需设定

示例

```
/* config pll clock resource */
crm_pll_config(CRM_PLL_SOURCE_HEXT_DIV, CRM_PLL_MULT_60,
```

```
CRM_PLL_OUTPUT_RANGE_GT72MHZ);
```

5.6.23 函数 crm_sysclk_switch

下表描述了函数 crm_sysclk_switch

表 131. 函数 crm_sysclk_switch

项目	描述
函数名	crm_sysclk_switch
函数原型	void crm_sysclk_switch(crm_sclk_type value);
功能描述	用作系统时钟的时钟源切换
输入参数 1	value：将用作系统时钟的时钟源
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

CRM_SCLK_HICK： 选择内部高速时钟作为系统时钟

CRM_SCLK_HEXT： 选择外部高速时钟作为系统时钟

CRM_SCLK_PLL： 选择 PLL 时钟作为系统时钟

示例

```
/* select pll as system clock source */
crm_sysclk_switch(CRM_SCLK_PLL);
```

5.6.24 函数 crm_sysclk_switch_status_get

下表描述了函数 crm_sysclk_switch_status_get

表 132. 函数 crm_sysclk_switch_status_get

项目	描述
函数名	crm_sysclk_switch_status_get
函数原型	crm_sclk_type crm_sysclk_switch_status_get(void);
功能描述	返回用作系统时钟的时钟源
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	crm_sclk_type：返回用作系统时钟的时钟源
先决条件	无
被调用函数	无

示例

```
/* wait till pll is used as system clock source */
while(crm_sysclk_switch_status_get() != CRM_SCLK_PLL)
{
}
```

5.6.25 函数 crm_clocks_freq_get

下表描述了函数 crm_clocks_freq_get

表 133. 函数 crm_clocks_freq_get

项目	描述
函数名	crm_clocks_freq_get
函数原型	void crm_clocks_freq_get(crm_clocks_freq_type *clocks_struct);
功能描述	返回片上不同的时钟频率
输入参数 1	clocks_struct：指向结构体 crm_clocks_freq_type 的指针，包含了各个时钟的频率
输入参数 2	无
输出参数	无
返回值	crm_sclk_type：返回用作系统时钟的时钟源
先决条件	无
被调用函数	无

crm_clocks_freq_type

crm_clocks_freq_type 在 at32f413_crm.h 中

```
typedef struct
{
  uint32_t      sclk_freq;
  uint32_t      ahb_freq;
  uint32_t      apb2_freq;
  uint32_t      apb1_freq;
  uint32_t      adc_freq;
} crm_clocks_freq_type;
```

sclk_freq

该成员返回系统时钟频率，单位 Hz

ahb_freq

该成员返回 AHB 总线时钟频率，单位 Hz

apb2_freq

该成员返回 APB2 总线时钟频率，单位 Hz

apb1_freq

该成员返回 APB1 总线时钟频率，单位 Hz

adc_freq

该成员返回 ADC 时钟频率，单位 Hz

示例

```
/* get frequency */
crm_clocks_freq_type clocks_struct;
crm_clocks_freq_get(&clocks_struct);
```

5.6.26 函数 crm_clock_out_set

下表描述了函数 crm_clock_out_set

表 134. 函数 crm_clock_out_set

项目	描述
函数名	crm_clock_out_set

项目	描述
函数原型	void crm_clock_out_set(crm_clkout_select_type clkout);
功能描述	选择在 clkout 管脚上输出的时钟源
输入参数 1	clkout：clkout 管脚上输出的时钟源
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* config PA8 output pll/4 */
crm_clock_out_set(CRM_CLKOUT_PLL_DIV_4);
```

5.6.27 函数 crm_interrupt_enable

下表描述了函数 crm_interrupt_enable

表 135. 函数 crm_interrupt_enable

项目	描述
函数名	crm_interrupt_enable
函数原型	void crm_interrupt_enable(uint32_t crm_int, confirm_state new_state);
功能描述	指定的中断使能设置
输入参数 1	crm_int：指定的 crm 中断
输入参数 2	new_state：中断新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

crm_int

CRM_LICK_STABLE_INT： 低速内部时钟稳定中断
CRM_LEXT_STABLE_INT： 低速外部时钟稳定中断
CRM_HICK_STABLE_INT： 高速内部时钟稳定中断
CRM_HEXT_STABLE_INT： 高速外部时钟稳定中断
CRM_PLL_STABLE_INT： PLL 时钟稳定中断
CRM_CLOCK_FAILURE_INT： 时钟失效中断

示例

```
/* enable pll stable interrupt */
crm_interrupt_enable (CRM_PLL_STABLE_INT);
```

5.6.28 函数 crm_auto_step_mode_enable

下表描述了函数 crm_auto_step_mode_enable

表 136. 函数 crm_auto_step_mode_enable

项目	描述
函数名	crm_auto_step_mode_enable

项目	描述
函数原型	void crm_auto_step_mode_enable(confirm_state new_state);
功能描述	自动顺滑使能设置
输入参数 1	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable auto step mode */
crm_auto_step_mode_enable(TRUE);
```

5.6.29 函数 crm_usb_interrupt_remapping_set

下表描述了函数 crm_usb_interrupt_remapping_set

表 137. 函数 crm_usb_interrupt_remapping_set

项目	描述
函数名	crm_usb_interrupt_remapping_set
函数原型	void crm_usb_interrupt_remapping_set(crm_usb_int_map_type int_remap);
功能描述	USB 中断号重映射设置
输入参数 1	int_remap：USB 需要使用的中断号
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

int_remap

CRM_USB_INT19_INT20：USB 使用 19 和 20 号中断

CRM_USB_INT73_INT74：USB 使用 73 和 74 号中断

示例

```
/* config usb IRQ number with 73/74 */
crm_usb_interrupt_remapping_set (CRM_USB_INT73_INT74);
```

5.6.30 函数 crm_hick_sclk_frequency_select

下表描述了函数 crm_hick_sclk_frequency_select

表 138. 函数 crm_hick_sclk_frequency_select

项目	描述
函数名	crm_hick_sclk_frequency_select
函数原型	void crm_hick_sclk_frequency_select(crm_hick_sclk_frequency_type value);
功能描述	当内部高速时钟被选择作为系统时钟时，设置系统时钟频率为 8M 或 48M
输入参数 1	value：8M 或者 48M 内部高速时钟
输入参数 2	无

项目	描述
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

CRM_HICK_SCLK_8MHZ： 内部高速时钟 8MHz 作为系统时钟

CRM_HICK_SCLK_48MHZ： 内部高速时钟 48MHz 作为系统时钟

示例

```
/* config sysclk with hick 48mhz */
crm_hick_sclk_frequency_select (CRM_HICK_SCLK_48MHZ);
```

5.6.31 函数 crm_usb_clock_source_select

下表描述了函数 crm_usb_clock_source_select

表 139. 函数 crm_usb_clock_source_select

项目	描述
函数名	crm_usb_clock_source_select
函数原型	void crm_usb_clock_source_select(crm_usb_clock_source_type value);
功能描述	选择 PLL 或内部高速时钟（48M）作为 USB 时钟源
输入参数 1	value：PLL 或者内部高速时钟（48M）
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

CRM_USB_CLOCK_SOURCE_PLL： PLL 时钟作为 USB 时钟源

CRM_USB_CLOCK_SOURCE_HICK： 内部高速时钟作为 USB 时钟源

示例

```
/* select hick48 as usb clock */
crm_usb_clock_source_select (CRM_USB_CLOCK_SOURCE_HICK);
```

5.6.32 函数 crm_clkout_to_tmr10_enable

下表描述了函数 crm_clkout_to_tmr10_enable

表 140. 函数 crm_clkout_to_tmr10_enable

项目	描述
函数名	crm_clkout_to_tmr10_enable
函数原型	void crm_clkout_to_tmr10_enable(confirm_state new_state);
功能描述	clkout 内部连接到 tmr10 的通道 1 使能设置
输入参数 1	new_state：新的设置状态。使能（TRUE），失能（FALSE）
输入参数 2	无
输出参数	无

项目	描述
返回值	无
先决条件	无
被调用函数	无

示例

```
/* config clkout internal connect to tmr10 channel1 */
crm_clkout_to_tmr10_enable (TRUE);
```

5.6.33 函数 crm_clkout_div_set

下表描述了函数 crm_clkout_div_set

表 141. 函数 crm_clkout_div_set

项目	描述
函数名	crm_clkout_div_set
函数原型	void crm_clkout_div_set(crm_clkout_div_type clkout_div);
功能描述	clkout 输出时钟分频设置
输入参数 1	clkout_div：clkout 输出频率的分频值
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

value

CRM_CLKOUT_DIV_1： CLKOUT 将选择输出的内部时钟按 1 分频输出
CRM_CLKOUT_DIV_2： CLKOUT 将选择输出的内部时钟按 2 分频输出
CRM_CLKOUT_DIV_4： CLKOUT 将选择输出的内部时钟按 4 分频输出
CRM_CLKOUT_DIV_8： CLKOUT 将选择输出的内部时钟按 8 分频输出
CRM_CLKOUT_DIV_16： CLKOUT 将选择输出的内部时钟按 16 分频输出
CRM_CLKOUT_DIV_64： CLKOUT 将选择输出的内部时钟按 64 分频输出
CRM_CLKOUT_DIV_128： CLKOUT 将选择输出的内部时钟按 128 分频输出
CRM_CLKOUT_DIV_256： CLKOUT 将选择输出的内部时钟按 256 分频输出
CRM_CLKOUT_DIV_512： CLKOUT 将选择输出的内部时钟按 512 分频输出

示例

```
/* config clkout division */
crm_clkout_div_set(CRM_CLKOUT_DIV_1);
```

5.7 调试（DEBUG）

DEBUG 寄存器结构 debug_type，定义于文件“at32f413_debug.h”如下：

```
/**
  * @brief type define debug register all
  */
typedef struct
{
  ...
```

} debug_type;

下表给出了 DEBUG 寄存器总览：

表 142. DEBUG 寄存器对应表

寄存器	描述
idcode	设备 ID
ctrl	控制寄存器

下表给出了 DEBUG 库函数总览：

表 143. DEBUG 库函数总览

函数名	描述
debug_device_id_get	读取设备 idcode
debug_periph_mode_set	对应外设的 debug 模式设置

5.7.1 函数 debug_device_id_get

下表描述了函数 debug_device_id_get

表 144. 函数 debug_device_id_get

项目	描述
函数名	debug_device_id_get
函数原型	uint32_t debug_device_id_get(void);
功能描述	读取设备 idcode
输入参数 1	无
输入参数 2	无
输出参数	无
返回值	返回 32-bit idcode
先决条件	无
被调用函数	无

示例

```
/* get idcode */
uint32_t idcode = 0;
idcode = debug_device_id_get();
```

5.7.2 函数 debug_periph_mode_set

下表描述了函数 debug_periph_mode_set

表 145. 函数 debug_periph_mode_set

项目	描述
函数名	debug_periph_mode_set
函数原型	void debug_periph_mode_set(uint32_t periph_debug_mode, confirm_state new_state);
功能描述	指定外设或模式进行 debug 模式设置

项目	描述
输入参数 1	periph_debug_mode：指定外设或模式
输入参数 2	new_state：设置新状态，开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

periph_debug_mode

指定对应的外设或模式进行 DEBUG 设置

DEBUG_SLEEP：SLEEP 模式下 DEBUG 设置
DEBUG_DEEPSLEEP：DEEPSLEEP 模式下 DEBUG 设置
DEBUG_STANDBY：STANDBY 模式下 DEBUG 设置
DEBUG_WDT_PAUSE：看门狗是否计数的 DEBUG 设置
DEBUG_WWDT_PAUSE：窗口看门狗是否计数的 DEBUG 设置
DEBUG_TMR1_PAUSE：TMR1 是否计数的 DEBUG 设置
DEBUG_TMR2_PAUSE：TMR2 是否计数的 DEBUG 设置
DEBUG_TMR3_PAUSE：TMR3 是否计数的 DEBUG 设置
DEBUG_TMR4_PAUSE：TMR4 是否计数的 DEBUG 设置
DEBUG_TMR5_PAUSE：TMR5 是否计数的 DEBUG 设置
DEBUG_TMR8_PAUSE：TMR8 是否计数的 DEBUG 设置
DEBUG_TMR9_PAUSE：TMR9 是否计数的 DEBUG 设置
DEBUG_TMR10_PAUSE：TMR10 是否计数的 DEBUG 设置
DEBUG_TMR11_PAUSE：TMR11 是否计数的 DEBUG 设置
DEBUG_I2C1_SMBUS_TIMEOUT：I2C1 SMBUS TIMEOUT 是否计数的 DEBUG 设置
DEBUG_I2C2_SMBUS_TIMEOUT：I2C2 SMBUS TIMEOUT 是否计数的 DEBUG 设置
DEBUG_CAN1_PAUSE：CAN1 接收寄存器是否继续工作的 DEBUG 设置

示例

```
/* enable tmr1 debug mode */
debug_periph_mode_set(DEBUG_TMR1_PAUSE, TRUE);
```

5.8 DMA 控制器（DMA）

DMA 寄存器结构 dma_type，定义于文件“at32f413_dma.h”如下：

```
/**
  * @brief type define dma register
  */
typedef struct
{
      …

} dma_type;
```

DMA 通道寄存器结构 dma_channel_type，定义于文件“at32f413_dma.h”如下：

```
/**
  * @brief type define dma channel register all
  */
```

```
typedef struct
{
    …

} dma_channel_type;
```

下表给出了 DMA 寄存器总览：

表 146.DMA 寄存器对应表

寄存器	描述
dma_sts	DMA 状态寄存器
dma_clr	DMA 状态清除寄存器
dma_c1ctrl	DMA 通道 1 配置寄存器
dma_c1dtcnt	DMA 通道 1 数据传输量寄存器
dma_c1paddr	DMA 通道 1 外设地址寄存器
dma_c1maddr	DMA 通道 1 存储器地址寄存器
dma_c2ctrl	DMA 通道 2 配置寄存器
dma_c2dtcnt	DMA 通道 2 数据传输量寄存器
dma_c2paddr	DMA 通道 2 外设地址寄存器
dma_c2maddr	DMA 通道 2 存储器地址寄存器
dma_c3ctrl	DMA 通道 3 配置寄存器
dma_c3dtcnt	DMA 通道 3 数据传输量寄存器
dma_c3paddr	DMA 通道 3 外设地址寄存器
dma_c3maddr	DMA 通道 3 存储器地址寄存器
dma_c4ctrl	DMA 通道 4 配置寄存器
dma_c4dtcnt	DMA 通道 4 数据传输量寄存器
dma_c4paddr	DMA 通道 4 外设地址寄存器
dma_c4maddr	DMA 通道 4 存储器地址寄存器
dma_c5ctrl	DMA 通道 5 配置寄存器
dma_c5dtcnt	DMA 通道 5 数据传输量寄存器
dma_c5paddr	DMA 通道 5 外设地址寄存器
dma_c5maddr	DMA 通道 5 存储器地址寄存器
dma_c6ctrl	DMA 通道 6 配置寄存器
dma_c6dtcnt	DMA 通道 6 数据传输量寄存器
dma_c6paddr	DMA 通道 6 外设地址寄存器
dma_c6maddr	DMA 通道 6 存储器地址寄存器
dma_c7ctrl	DMA 通道 7 配置寄存器
dma_c7dtcnt	DMA 通道 7 数据传输量寄存器
dma_c7paddr	DMA 通道 7 外设地址寄存器
dma_c7maddr	DMA 通道 7 存储器地址寄存器
dma_src_sel0	通道来源寄存器 0
dma_src_sel1	通道来源寄存器 1

下表给出了 DMA 库函数总览：

表 147.DMA 库函数总览

函数名	描述
dma_default_para_init	将 dma_init_struct 中的参数初始化
dma_init	初始化指定的 DMA 通道
dma_reset	复位指定的 DMA 通道
dma_data_number_set	设置指定通道的数据传输量寄存器值
dma_data_number_get	获取指定通道的数据传输量寄存器值
dma_interrupt_enable	使能指定通道的相应中断
dma_channel_enable	使能指定通道
dma_flexible_config	配置弹性请求映射
dma_flag_get	获取通道相关标志位
dma_flag_clear	清除通道相关标志位

5.8.1 函数 dma_default_para_init

下表描述了函数 dma_default_para_init

表 148.函数 dma_default_para_init

项目	描述
函数名	dma_default_para_init
函数原型	void dma_default_para_init(dma_init_type* dma_init_struct);
功能描述	将 dma_init_struct 中的参数初始化
输入参数 1	dma_init_struct：指向 dma_init_type 类型结构体
输出参数	无
返回值	无
先决条件	无
被调用函数	无

结构体 dma_init_struct 成员默认值如下表所示：

表 149.dma_init_struct 默认值

成员	默认值
peripheral_base_addr	0x0
memory_base_addr	0x0
direction	DMA_DIR_PERIPHERAL_TO_MEMORY
buffer_size	0x0
peripheral_inc_enable	FALSE
memory_inc_enable	FALSE
peripheral_data_width	DMA_PERIPHERAL_DATA_WIDTH_BYTE
memory_data_width	DMA_MEMORY_DATA_WIDTH_BYTE
loop_mode_enable	FALSE
priority	DMA_PRIORITY_LOW

示例

```
/* dma init config with its default value */
dma_init_type dma_init_struct = {0};
dma_default_para_init(&dma_init_struct);
```

5.8.2 函数 dma_init

下表描述了函数 dma_init

表 150.函数 dma_init

项目	描述
函数名	dma_init
函数原型	void dma_init(dma_channel_type* dmax_channely, dma_init_type* dma_init_struct)
功能描述	初始化指定的 DMA 通道
输入参数 1	dmax_channely：DMAx_CHANNELy 指定 DMA 通道号, x=1 或 2，y=1...7
输入参数 2	dma_init_struct：指向 dma_init_type 类型结构体
输出参数	无
返回值	无
先决条件	无
被调用函数	无

dma_init_type structure

dma_init_type 在 at32f413_dma.h 中

```
typedef struct
{
  uint32_t                      peripheral_base_addr;
  uint32_t                      memory_base_addr;
  dma_dir_type                  direction;
  uint16_t                      buffer_size;
  confirm_state                 peripheral_inc_enable;
  confirm_state                 memory_inc_enable;
  dma_peripheral_data_size_type peripheral_data_width;
  dma_memory_data_size_type     memory_data_width;
  confirm_state                 loop_mode_enable;
  dma_priority_level_type       priority;
} dma_init_type;
```

peripheral_base_addr

设置 DMA 通道的外设地址

memory_base_addr

设置 DMA 通道存储器地址

direction

设置 DMA 通道传输方向类型

DMA_DIR_PERIPHERAL_TO_MEMORY： 方向为外设到存储器

DMA_DIR_MEMORY_TO_PERIPHERAL： 方向为存储器到外设

DMA_DIR_MEMORY_TO_MEMORY： 方向为存储器到存储器

buffer_size

设置 DMA 通道传输数据量

peripheral_inc_enable

设置 DMA 通道外设地址是否自动递增

FALSE： 外设地址不递增

TRUE： 外设地址递增

memory_inc_enable

设置 DMA 通道存储器地址是否自动递增

FALSE： 存储器地址不递增

TRUE： 存储器地址递增

peripheral_data_width

设置 DMA 通道外设数据宽度

DMA_PERIPHERAL_DATA_WIDTH_BYTE： 外设数据宽度为字节

DMA_PERIPHERAL_DATA_WIDTH_HALFWORD： 外设数据宽度为半字

DMA_PERIPHERAL_DATA_WIDTH_WORD： 外设数据宽度为字

memory_data_width

设置 DMA 通道存储器数据宽度

DMA_MEMORY_DATA_WIDTH_BYTE： 存储器数据宽度为字节

DMA_MEMORY_DATA_WIDTH_HALFWORD： 存储器数据宽度为半字

DMA_MEMORY_DATA_WIDTH_WORD： 存储器数据宽度为字

loop_mode_enable

设置 DMA 通道是否为循环模式

FALSE： DMA 通道为单次模式

TRUE： DMA 通道为循环模式

priority

设置 DMA 通道优先级

DMA_PRIORITY_LOW： DMA 通道优先级为低

DMA_PRIORITY_MEDIUM： DMA 通道优先级为中

DMA_PRIORITY_HIGH： DMA 通道优先级为高

DMA_PRIORITY_VERY_HIGH： DMA 通道优先级为非常高

示例

```
dma_init_type dma_init_struct = {0};
/* dma2 channel1 configuration */
dma_init_struct.buffer_size = BUFFER_SIZE;
dma_init_struct.direction = DMA_DIR_MEMORY_TO_PERIPHERAL;
dma_init_struct.memory_base_addr = (uint32_t)src_buffer;
dma_init_struct.memory_data_width = DMA_MEMORY_DATA_WIDTH_HALFWORD;
dma_init_struct.memory_inc_enable = TRUE;
dma_init_struct.peripheral_base_addr = (uint32_t)0x4001100C;
dma_init_struct.peripheral_data_width = DMA_PERIPHERAL_DATA_WIDTH_HALFWORD;
dma_init_struct.peripheral_inc_enable = FALSE;
dma_init_struct.priority = DMA_PRIORITY_MEDIUM;
dma_init_struct.loop_mode_enable = FALSE;
dma_init(DMA2_CHANNEL1, &dma_init_struct);
```

5.8.3 函数 dma_reset

下表描述了函数 dma_reset

表 151.函数 dma_reset

项目	描述
函数名	dma_reset
函数原型	void dma_reset(dma_channel_type* dmax_channely);
功能描述	复位指定的 DMA 通道
输入参数 1	dmax_channely：DMAx_CHANNELy 指定 DMA 通道号, x=1 或 2，y=1...7
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* reset dma2 channel1 */
dma_reset(DMA2_CHANNEL1);
```

5.8.4 函数 dma_data_number_set

下表描述了函数 dma_data_number_set

表 152.函数 dma_data_number_set

项目	描述
函数名	dma_ data_number_set
函数原型	void dma_data_number_set(dma_channel_type* dmax_channely, uint16_t data_number);
功能描述	设置指定通道的数据传输量寄存器值
输入参数 1	dmax_channely：DMAx_CHANNELy 指定 DMA 通道号, x=1 或 2，y=1...7
输入参数 2	data_number：数据传输量,最大 65535
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* set dma2 channel1 data count is 0x100*/
dma_data_number_set(DMA2_CHANNEL1, 0x100);
```

5.8.5 函数 dma_data_number_get

下表描述了函数 dma_data_number_get

表 153.函数 dma_data_number_get

项目	描述
函数名	dma_data_number_get
函数原型	uint16_t dma_data_number_get(dma_channel_type* dmax_channely);
功能描述	获取指定通道的数据传输量寄存器值
输入参数 1	dmax_channely：DMAx_CHANNELy 指定 DMA 通道号, x=1 或 2，y=1...7
输出参数	无
返回值	获取的指定通道数据传输量

项目	描述
先决条件	无
被调用函数	无

示例

```
/* get dma2 channel1 data count*/
uint16_t data_counter;
data_counter = dma_data_number_set(DMA2_CHANNEL1);
```

5.8.6 函数 dma_interrupt_enable

下表描述了函数 dma_interrupt_enable

表 154.函数 dma_interrupt_enable

项目	描述
函数名	dma_interrupt_enable
函数原型	void dma_interrupt_enable(dma_channel_type* dmax_channely, uint32_t dma_int, confirm_state new_state);
功能描述	使能指定通道的相应中断
输入参数 1	dmax_channely：DMAx_CHANNELy 指定 DMA 通道号, x=1 或 2，y=1...7
输入参数 2	dma_int：中断源选择
输入参数 3	new_state：使能或关闭中断
输出参数	无
返回值	无
先决条件	无
被调用函数	无

dma_int

选择 DMA 通道中断源

DMA_FDT_INT： 传输完成中断

DMA_HDT_INT： 传输半完成中断

DMA_DTERR_INT： 传输错误中断

new_state

选择 DMA 通道中断是使能还是关闭

FALSE： 关闭中断

TRUE： 使能中断

示例

```
/* enable dma2 channel1 transfer full data intterrupt */
dma_interrupt_enable(DMA2_CHANNEL1, DMA_FDT_INT, TRUE);
```

5.8.7 函数 dma_channel_enable

下表描述了函数 dma_interrupt_enable

表 155.函数 dma_channel_enable

项目	描述
函数名	dma_channel_enable

项目	描述
函数原型	void dma_channel_enable(dma_channel_type* dmax_channely, confirm_state new_state);
功能描述	使能指定通道
输入参数 1	dmax_channely：DMAx_CHANNELy 指定 DMA 通道号, x=1 或 2，y=1...7
输入参数 2	new_state：使能或关闭通道
输出参数	无
返回值	无
先决条件	无
被调用函数	无

new_state

选择 DMA 通道是使能还是关闭

FALSE： 关闭通道

TRUE： 使能通道

示例

```
/* enable dma channel */
dma_channel_enable(DMA2_CHANNEL1, TRUE);
```

5.8.8 函数 dma_flexible_config

下表描述了函数 dma_flexible_config

表 156.函数 dma_flexible_config

项目	描述
函数名	dma_flexible_config
函数原型	void dma_flexible_config(dma_type* dma_x, uint8_t flex_channelx, dma_flexible_request_type flexible_request);
功能描述	配置弹性请求映射
输入参数 1	dma_x：指定 DMAx, x=1 或 2
输入参数 2	flex_channelx：FLEX_CHANNELx 指定通道号，x=1…7
输入参数 3	flexible_request：弹性映射请求来源 ID 号
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flexible_request

弹性映射请求来源 ID 号如下表所示：

表 157.弹性映射请求来源 ID 号

请求来源ID号	请求来源	请求来源ID号	请求来源
0x01	DMA_FLEXIBLE_ADC1	0x0A	DMA_FLEXIBLE_SPI1_TX
0x09	DMA_FLEXIBLE_SPI1_RX	0x0C	DMA_FLEXIBLE_SPI2_TX
0x0B	DMA_FLEXIBLE_SPI2_RX	0x1A	DMA_FLEXIBLE_UART1_TX
0x19	DMA_FLEXIBLE_UART1_RX	0x1C	DMA_FLEXIBLE_UART2_TX

请求来源ID号	请求来源	请求来源ID号	请求来源
0x1B	DMA_FLEXIBLE_UART2_RX	0x1E	DMA_FLEXIBLE_UART3_TX
0x1D	DMA_FLEXIBLE_UART3_RX	0x20	DMA_FLEXIBLE_UART4_TX
0x1F	DMA_FLEXIBLE_UART4_RX	0x22	DMA_FLEXIBLE_UART5_TX
0x21	DMA_FLEXIBLE_UART5_RX	0x2A	DMA_FLEXIBLE_I2C1_TX
0x29	DMA_FLEXIBLE_I2C1_RX	0x2C	DMA_FLEXIBLE_I2C2_TX
0x2B	DMA_FLEXIBLE_I2C2_RX	0x36	DMA_FLEXIBLE_TMR1_HALL
0x31	DMA_FLEXIBLE_SDIO1	0x38	DMA_FLEXIBLE_TMR1_CH1
0x35	DMA_FLEXIBLE_TMR1_TRIG	0x3A	DMA_FLEXIBLE_TMR1_CH3
0x37	DMA_FLEXIBLE_TMR1_OVERFLOW	0x3D	DMA_FLEXIBLE_TMR2_TRIG
0x39	DMA_FLEXIBLE_TMR1_CH2	0x40	DMA_FLEXIBLE_TMR2_CH1
0x3B	DMA_FLEXIBLE_TMR1_CH4	0x42	DMA_FLEXIBLE_TMR2_CH3
0x3F	DMA_FLEXIBLE_TMR2_OVERFLOW	0x45	DMA_FLEXIBLE_TMR3_TRIG
0x41	DMA_FLEXIBLE_TMR2_CH2	0x48	DMA_FLEXIBLE_TMR3_CH1
0x43	DMA_FLEXIBLE_TMR2_CH4	0x4A	DMA_FLEXIBLE_TMR3_CH3
0x47	DMA_FLEXIBLE_TMR3_OVERFLOW	0x4D	DMA_FLEXIBLE_TMR4_TRIG
0x49	DMA_FLEXIBLE_TMR3_CH2	0x50	DMA_FLEXIBLE_TMR4_CH1
0x4B	DMA_FLEXIBLE_TMR3_CH4	0x52	DMA_FLEXIBLE_TMR4_CH3
0x4F	DMA_FLEXIBLE_TMR4_OVERFLOW	0x55	DMA_FLEXIBLE_TMR5_TRIG
0x51	DMA_FLEXIBLE_TMR4_CH2	0x58	DMA_FLEXIBLE_TMR5_CH1
0x53	DMA_FLEXIBLE_TMR4_CH4	0x5A	DMA_FLEXIBLE_TMR5_CH3
0x57	DMA_FLEXIBLE_TMR5_OVERFLOW	0x6D	DMA_FLEXIBLE_TMR8_TRIG
0x59	DMA_FLEXIBLE_TMR5_CH2	0x6F	DMA_FLEXIBLE_TMR8_OVERFLOW
0x5B	DMA_FLEXIBLE_TMR5_CH4	0x71	DMA_FLEXIBLE_TMR8_CH2
0x6E	DMA_FLEXIBLE_TMR8_HALL	0x73	DMA_FLEXIBLE_TMR8_CH4
0x70	DMA_FLEXIBLE_TMR8_CH1	0x72	DMA_FLEXIBLE_TMR8_CH3

示例

```
/* tmr2 flexible function enable */
dma_flexible_config(DMA2, FLEX_CHANNEL1, DMA_FLEXIBLE_TMR2_OVERFLOW);
```

5.8.9 函数 dma_flag_get

下表描述了函数 dma_flag_get

表 158.函数 dma_flag_get

项目	描述
函数名	dma_flag_get
函数原型	flag_status dma_flag_get(uint32_t dmax_flag);
功能描述	获取通道相关标志位
输入参数 1	*dmax_flag*：需要获取的标志位
输出参数	无
返回值	flag_status：标志位是否置起
先决条件	无

项目	描述
被调用函数	无

dmax_flag

dmax_flag 用于选择需要获取状态的标志，其可选参数罗列如下：

DMA1_GL1_FLAG： DMA1 通道 1 全局标志
DMA1_FDT1_FLAG： DMA1 通道 1 传输完成标志
DMA1_HDT1_FLAG： DMA1 通道 1 半传输完成标志
DMA1_DTERR1_FLAG： DMA1 通道 1 传输错误标志
DMA1_GL2_FLAG： DMA1 通道 2 全局标志
DMA1_FDT2_FLAG： DMA1 通道 2 传输完成标志
DMA1_HDT2_FLAG： DMA1 通道 2 半传输完成标志
DMA1_DTERR2_FLAG： DMA1 通道 2 传输错误标志
DMA1_GL3_FLAG： DMA1 通道 3 全局标志
DMA1_FDT3_FLAG： DMA1 通道 3 传输完成标志
DMA1_HDT3_FLAG： DMA1 通道 3 半传输完成标志
DMA1_DTERR3_FLAG： DMA1 通道 3 传输错误标志
DMA1_GL4_FLAG： DMA1 通道 4 全局标志
DMA1_FDT4_FLAG： DMA1 通道 4 传输完成标志
DMA1_HDT4_FLAG： DMA1 通道 4 半传输完成标志
DMA1_DTERR4_FLAG： DMA1 通道 4 传输错误标志
DMA1_GL5_FLAG： DMA1 通道 5 全局标志
DMA1_FDT5_FLAG： DMA1 通道 5 传输完成标志
DMA1_HDT5_FLAG： DMA1 通道 5 半传输完成标志
DMA1_DTERR5_FLAG： DMA1 通道 5 传输错误标志
DMA1_GL6_FLAG： DMA1 通道 6 全局标志
DMA1_FDT6_FLAG： DMA1 通道 6 传输完成标志
DMA1_HDT6_FLAG： DMA1 通道 6 半传输完成标志
DMA1_DTERR6_FLAG： DMA1 通道 6 传输错误标志
DMA1_GL7_FLAG： DMA1 通道 7 全局标志
DMA1_FDT7_FLAG： DMA1 通道 7 传输完成标志
DMA1_HDT7_FLAG： DMA1 通道 7 半传输完成标志
DMA1_DTERR7_FLAG： DMA1 通道 7 传输错误标志
DMA2_GL1_FLAG： DMA2 通道 1 全局标志
DMA2_FDT1_FLAG： DMA2 通道 1 传输完成标志
DMA2_HDT1_FLAG： DMA2 通道 1 半传输完成标志
DMA2_DTERR1_FLAG： DMA2 通道 1 传输错误标志
DMA2_GL2_FLAG： DMA2 通道 2 全局标志
DMA2_FDT2_FLAG： DMA2 通道 2 传输完成标志
DMA2_HDT2_FLAG： DMA2 通道 2 半传输完成标志
DMA2_DTERR2_FLAG： DMA2 通道 2 传输错误标志
DMA2_GL3_FLAG： DMA2 通道 3 全局标志
DMA2_FDT3_FLAG： DMA2 通道 3 传输完成标志
DMA2_HDT3_FLAG： DMA2 通道 3 半传输完成标志
DMA2_DTERR3_FLAG： DMA2 通道 3 传输错误标志
DMA2_GL4_FLAG： DMA2 通道 4 全局标志
DMA2_FDT4_FLAG： DMA2 通道 4 传输完成标志

DMA2_HDT4_FLAG： DMA2 通道 4 半传输完成标志
DMA2_DTERR4_FLAG： DMA2 通道 4 传输错误标志
DMA2_GL5_FLAG： DMA2 通道 5 全局标志
DMA2_FDT5_FLAG： DMA2 通道 5 传输完成标志
DMA2_HDT5_FLAG： DMA2 通道 5 半传输完成标志
DMA2_DTERR5_FLAG： DMA2 通道 5 传输错误标志
DMA2_GL6_FLAG： DMA2 通道 6 全局标志
DMA2_FDT6_FLAG： DMA2 通道 6 传输完成标志
DMA2_HDT6_FLAG： DMA2 通道 6 半传输完成标志
DMA2_DTERR6_FLAG： DMA2 通道 6 传输错误标志
DMA2_GL7_FLAG： DMA2 通道 7 全局标志
DMA2_FDT7_FLAG： DMA2 通道 7 传输完成标志
DMA2_HDT7_FLAG： DMA2 通道 7 半传输完成标志
DMA2_DTERR7_FLAG： DMA2 通道 7 传输错误标志

flag_status

RESET：相应标志位未置起
SET： 相应标志位置起

示例

```
if(dma_flag_get(DMA2_FDT1_FLAG) != RESET)
{
    /* turn led2/led3/led4 on */
    at32_led_on(LED2);
    at32_led_on(LED3);
    at32_led_on(LED4);
}
```

5.8.10 函数 dma_flag_clear

下表描述了函数 dma_flag_clear

表 159.函数 dma_flag_clear

项目	描述
函数名	dma_flag_clear
函数原型	void dma_flag_clear(uint32_t dmax_flag);
功能描述	清除通道相关标志位
输入参数 1	*dmax_flag*：需要清除的标志位
输出参数	无
返回值	无
先决条件	无
被调用函数	无

dmax_flag

dmax_flag 用于选择需要获取状态的标志，其可选参数罗列如下：

DMA1_GL1_FLAG： DMA1 通道 1 全局标志
DMA1_FDT1_FLAG： DMA1 通道 1 传输完成标志
DMA1_HDT1_FLAG： DMA1 通道 1 半传输完成标志

DMA1_DTERR1_FLAG： DMA1 通道 1 传输错误标志
DMA1_GL2_FLAG： DMA1 通道 2 全局标志
DMA1_FDT2_FLAG： DMA1 通道 2 传输完成标志
DMA1_HDT2_FLAG： DMA1 通道 2 半传输完成标志
DMA1_DTERR2_FLAG： DMA1 通道 2 传输错误标志
DMA1_GL3_FLAG： DMA1 通道 3 全局标志
DMA1_FDT3_FLAG： DMA1 通道 3 传输完成标志
DMA1_HDT3_FLAG： DMA1 通道 3 半传输完成标志
DMA1_DTERR3_FLAG： DMA1 通道 3 传输错误标志
DMA1_GL4_FLAG： DMA1 通道 4 全局标志
DMA1_FDT4_FLAG： DMA1 通道 4 传输完成标志
DMA1_HDT4_FLAG： DMA1 通道 4 半传输完成标志
DMA1_DTERR4_FLAG： DMA1 通道 4 传输错误标志
DMA1_GL5_FLAG： DMA1 通道 5 全局标志
DMA1_FDT5_FLAG： DMA1 通道 5 传输完成标志
DMA1_HDT5_FLAG： DMA1 通道 5 半传输完成标志
DMA1_DTERR5_FLAG： DMA1 通道 5 传输错误标志
DMA1_GL6_FLAG： DMA1 通道 6 全局标志
DMA1_FDT6_FLAG： DMA1 通道 6 传输完成标志
DMA1_HDT6_FLAG： DMA1 通道 6 半传输完成标志
DMA1_DTERR6_FLAG： DMA1 通道 6 传输错误标志
DMA1_GL7_FLAG： DMA1 通道 7 全局标志
DMA1_FDT7_FLAG： DMA1 通道 7 传输完成标志
DMA1_HDT7_FLAG： DMA1 通道 7 半传输完成标志
DMA1_DTERR7_FLAG： DMA1 通道 7 传输错误标志
DMA2_GL1_FLAG： DMA2 通道 1 全局标志
DMA2_FDT1_FLAG： DMA2 通道 1 传输完成标志
DMA2_HDT1_FLAG： DMA2 通道 1 半传输完成标志
DMA2_DTERR1_FLAG： DMA2 通道 1 传输错误标志
DMA2_GL2_FLAG： DMA2 通道 2 全局标志
DMA2_FDT2_FLAG： DMA2 通道 2 传输完成标志
DMA2_HDT2_FLAG： DMA2 通道 2 半传输完成标志
DMA2_DTERR2_FLAG： DMA2 通道 2 传输错误标志
DMA2_GL3_FLAG： DMA2 通道 3 全局标志
DMA2_FDT3_FLAG： DMA2 通道 3 传输完成标志
DMA2_HDT3_FLAG： DMA2 通道 3 半传输完成标志
DMA2_DTERR3_FLAG： DMA2 通道 3 传输错误标志
DMA2_GL4_FLAG： DMA2 通道 4 全局标志
DMA2_FDT4_FLAG： DMA2 通道 4 传输完成标志
DMA2_HDT4_FLAG： DMA2 通道 4 半传输完成标志
DMA2_DTERR4_FLAG： DMA2 通道 4 传输错误标志
DMA2_GL5_FLAG： DMA2 通道 5 全局标志
DMA2_FDT5_FLAG： DMA2 通道 5 传输完成标志
DMA2_HDT5_FLAG： DMA2 通道 5 半传输完成标志
DMA2_DTERR5_FLAG： DMA2 通道 5 传输错误标志
DMA2_GL6_FLAG： DMA2 通道 6 全局标志

DMA2_FDT6_FLAG： DMA2 通道 6 传输完成标志
DMA2_HDT6_FLAG： DMA2 通道 6 半传输完成标志
DMA2_DTERR6_FLAG： DMA2 通道 6 传输错误标志
DMA2_GL7_FLAG： DMA2 通道 7 全局标志
DMA2_FDT7_FLAG： DMA2 通道 7 传输完成标志
DMA2_HDT7_FLAG： DMA2 通道 7 半传输完成标志
DMA2_DTERR7_FLAG： DMA2 通道 7 传输错误标志

示例

```
if(dma_flag_get(DMA2_FDT1_FLAG) != RESET)
{
    /* turn led2/led3/led4 on */
    at32_led_on(LED2);
    at32_led_on(LED3);
    at32_led_on(LED4);
    dma_flag_clear(DMA2_FDT1_FLAG);
}
```

5.9 外部中断/事件控制器（EXINT）

EXINT 寄存器结构 exint_type，定义于文件“at32f413_exint.h”如下：

```
/**
  * @brief type define exint register all
  */
typedef struct
{
  ...
} exint_type;
```

下表给出了 EXINT 寄存器总览：

表 160. EXINT 寄存器总览

寄存器	描述
inten	中断使能寄存器
evten	事件使能寄存器
polcfg1	极性配置寄存器 1
polcfg2	极性配置寄存器 2
swtrg	软件触发寄存器
intsts	中断状态寄存器

下表给出了 EXINT 寄存器总览：

表 161. EXINT 库函数总览

函数名	描述
exint_reset	将 EXINT 所有寄存器值恢复到复位值
exint_default_para_init	给 EXINT 初始化结构体赋初值

exint_init	EXINT 初始化
exint_flag_clear	清除选定 EXINT 的中断标志位
exint_flag_get	读取选定 EXINT 的中断标志位
exint_software_interrupt_event_generate	软件中断事件产生
exint_interrupt_enable	使能选定的 EXINT 中断
exint_event_enable	使能选定的 EXINT 事件

5.9.1 函数 exint_reset

下表描述了函数 exint_reset

表 162. 函数 exint_reset

项目	描述
函数名	exint_reset
函数原型	void exint_reset(void);
功能描述	将 EXINT 所有寄存器值恢复到复位值
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset();

示例

```
exint_reset ();
```

5.9.2 函数 exint_default_para_init

下表描述了函数 exint_default_para_init

表 163. 函数 exint_default_para_init

项目	描述
函数名	exint_default_para_init
函数原型	void exint_default_para_init(exint_init_type *exint_struct);
功能描述	给 EXINT 初始化结构体赋初值
输入参数 1	exint_struct：指向 *exint_init_type* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 exint_init_type 类型的变量
被调用函数	无

示例

```
exint_init_type exint_init_struct;
exint_default_para_init(&exint_init_struct);
```

5.9.3 函数 exint_init

下表描述了函数 exint_init

表 164. 函数 exint_init

项目	描述
函数名	exint_init
函数原型	void exint_init(exint_init_type *exint_struct);
功能描述	EXINT 初始化
输入参数 1	*exint_init_type*：指向 exint_init_struct 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 exint_init_type 类型的变量
被调用函数	无

exint_init_type 在 at32f413_exint.h 中定义：

```
typedef struct
{
  exint_line_mode_type          line_mode;
  uint32_t                      line_select;
  exint_polarity_config_type    line_polarity;
  confirm_state                 line_enable;
} exint_init_type;
```

line_mode

选择事件模式或中断模式

EXINT_LINE_INTERRUPUT：　中断模式

EXINT_LINE_EVENT：　事件模式

line_select

line 选择

EXINT_LINE_NONE：不选择任何 line

EXINT_LINE_0：　选择 line0

EXINT_LINE_1：　选择 line1

…

EXINT_LINE_18：　选择 line18

line_polarity

触发沿选择

EXINT_TRIGGER_RISING_EDGE：　上升沿

EXINT_TRIGGER_FALLING_EDGE：　下降沿

EXINT_TRIGGER_BOTH_EDGE：　上升沿/下降沿均选择

line_enable

使能/关闭选定 line。

FALSE：　关闭选定 line；

TRUE：　使能选定 line。

示例

```
exint_init_type exint_init_struct;
exint_default_para_init(&exint_init_struct);
exint_init_struct.line_enable = TRUE;
exint_init_struct.line_mode = EXINT_LINE_INTERRUPUT;
```

```
exint_init_struct.line_select = EXINT_LINE_0;
exint_init_struct.line_polarity = EXINT_TRIGGER_RISING_EDGE;
exint_init(&exint_init_struct);
```

5.9.4 函数 exint_flag_clear

下表描述了函数 exint_flag_clear

表 165. 函数 exint_flag_clear

项目	描述
函数名	exint_flag_clear
函数原型	void exint_flag_clear(uint32_t exint_line);
功能描述	清除选定 EXINT 的中断标志位
输入参数	exint_line：line 选择 取值范围：参考前文的 *line_select* 值
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
exint_flag_clear(EXINT_LINE_0);
```

5.9.5 函数 exint_flag_get

下表描述了函数 exint_flag_get

表 166. 函数 exint_flag_get

项目	描述
函数名	exint_flag_get
函数原型	flag_status exint_flag_get(uint32_t exint_line);
功能描述	获取选定 EXINT 的中断标志位
输入参数	exint_line：line 选择 该参数详细描述见 *line_select*
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET，RESET.
先决条件	无
被调用函数	无

示例

```
flag_status status = RESET;
status = exint_flag_get(EXINT_LINE_0);
```

5.9.6 函数 exint_software_interrupt_event_generate

下表描述了函数 exint_software_interrupt_event_generate

表 167. 函数 exint_software_interrupt_event_generate

项目	描述
函数名	exint_software_interrupt_event_generate
函数原型	void exint_software_interrupt_event_generate(uint32_t exint_line);
功能描述	软件中断事件产生
输入参数	exint_line：line 选择 取值范围：参考前文的 *line_select*
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
exint_software_interrupt_event_generate (EXINT_LINE_0);
```

5.9.7 函数 exint_interrupt_enable

下表描述了函数 exint_interrupt_enable

表 168. 函数 exint_interrupt_enable

项目	描述
函数名	exint_interrupt_enable
函数原型	void exint_interrupt_enable(uint32_t exint_line, confirm_state new_state);
功能描述	使能选定的 EXINT 中断
输入参数 1	exint_line：line 选择 取值范围：参考前文的 *line_select*
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
exint_interrupt_enable (EXINT_LINE_0);
```

5.9.8 函数 exint_event_enable

下表描述了函数 exint_event_enable

表 169. 函数 exint_event_enable

项目	描述
函数名	exint_event_enable
函数原型	void exint_event_enable(uint32_t exint_line, confirm_state new_state);
功能描述	使能选定的 EXINT 事件
输入参数 1	exint_line：line 选择

项目	描述
	取值范围：参考前文的 *line_select*
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
exint_event_enable (EXINT_LINE_0);
```

5.10 闪存控制器（FLASH）

FLASH 寄存器结构 flash_type，定义于文件“at32f413_flash.h”如下：

```
/**
  * @brief type define flash register all
  */
typedef struct
{
      …
} flash_type;
```

下表给出了 FLASH 寄存器总览：

表 170. FLASH 寄存器对应表

寄存器	描述
flash_psr	闪存性能选择寄存器器
flash_unlock	闪存解锁寄存器
flash_usd_unlock	闪存用户系统数据解锁寄存器
flash_sts	闪存状态寄存器
flash_ctrl	闪存控制寄存器
flash_addr	闪存地址寄存器
flash_usd	用户系统数据寄存器
flash_epps	擦除编程保护状态寄存器
flash_unlock3	闪存解锁寄存器 3
flash_select	闪存选择寄存器
flash_sts3	闪存状态寄存器 3
flash_ctrl3	闪存控制寄存器 3
flash_addr3	闪存地址寄存器 3
flash_da	闪存解密地址寄存器
slib_sts0	闪存安全库区状态寄存器 0
slib_sts1	闪存安全库区状态寄存器 1
slib_pwd_clr	闪存安全库区密码清除寄存器
slib_misc_sts	闪存安全库区额外状态寄存器

寄存器	描述
slib_set_pwd	闪存安全库区密码设定寄存器
slib_set_range	闪存安全库区地址设定寄存器
slib_unlock	闪存安全库区解锁寄存器
flash_crc_ctrl	闪存 CRC 校验控制寄存器
flash_crc_chkr	闪存 CRC 校验结果寄存器

下表给出了 FLASH 库函数总览：

表 171. FLASH 库函数总览

函数名	描述
flash_flag_get	获取标志状态
flash_flag_clear	清除已置位的标志
flash_operation_status_get	操作状态获取（内部闪存块）
flash_spim_operation_status_get	外部闪存操作状态获取
flash_operation_wait_for	等待操作完成（内部闪存块）
flash_spim_operation_wait_for	等待外部闪存操作完成
flash_unlock	闪存解锁（内部闪存块）
flash_spim_unlock	外部闪存解锁
flash_lock	闪存锁定（内部闪存块）
flash_spim_lock	外部闪存锁定
flash_sector_erase	扇区擦除
flash_internal_all_erase	内部闪存擦除
flash_spim_all_erase	外部闪存擦除
flash_user_system_data_erase	用户系统数据区擦除
flash_word_program	闪存按字编程
flash_halfword_program	闪存按半字编程
flash_byte_program	闪存按字节编程
flash_user_system_data_program	用户系统数据区编程
flash_epp_set	擦除编程保护设置
flash_epp_status_get	擦除编程保护状态获取
flash_fap_enable	访问保护设置
flash_fap_status_get	访问保护状态获取
flash_ssb_set	系统配置字节设置
flash_ssb_status_get	系统配置字节状态获取
flash_interrupt_enable	闪存中断配置
flash_spim_model_select	外部闪存类型选择
flash_spim_encryption_range_set	外部闪存加密范围设置
flash_slib_enable	安全库区使能
flash_slib_disable	安全库区失能
flash_slib_remaining_count_get	安全库区剩余可使用次数
flash_slib_state_get	安全库区状态获取
flash_slib_start_sector_get	安全库区开始扇区获取
flash_slib_datastart_sector_get	安全库区数据区开始扇区获取
flash_slib_end_sector_get	安全库区结束扇区获取

函数名	描述
flash_crc_calibrate	闪存 CRC 校验

5.10.1 函数 flash_flag_get

下表描述了函数 flash_flag_get

表 172. 函数 flash_flag_get

项目	描述
函数名	flash_flag_get
函数原型	flag_status flash_flag_get(uint32_t flash_flag);
功能描述	获取标志位状态
输入参数	flash_flag：需要获取状态的标志选择
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET，RESET.
先决条件	无
被调用函数	无

flash_flag

可获取的状态标志

FLASH_OBF_FLAG：闪存操作忙标志（内部闪存块）

FLASH_ODF_FLAG：闪存操作完成标志（内部闪存块）

FLASH_PRGMERR_FLAG：闪存编程错误标志（内部闪存块）

FLASH_EPPERR_FLAG：闪存擦写错误标志（内部闪存块）

FLASH_SPIM_OBF_FLAG：外部闪存操作忙标志

FLASH_SPIM_ODF_FLAG：外部闪存操作完成标志

FLASH_SPIM_PRGMERR_FLAG：外部闪存编程错误标志

FLASH_SPIM_EPPERR_FLAG：外部闪存擦写错误标志

FLASH_USDERR_FLAG：用户系统数据区错误标志

示例

```
flag_status status;
status = flash_flag_get (FLASH_ODF_FLAG);
```

5.10.2 函数 flash_flag_clear

下表描述了函数 flash_flag_clear

表 173. 函数 flash_flag_clear

项目	描述
函数名	flash_flag_clear
函数原型	void flash_flag_clear(uint32_t flash_flag);
功能描述	清除标志位
输入参数	flash_flag：待清除的标志选择
输出参数	无
返回值	无
先决条件	无

项目	描述
被调用函数	无

flash_flag

可清除的状态标志

FLASH_ODF_FLAG：　　闪存操作完成标志（内部闪存块）

FLASH_PRGMERR_FLAG：　　闪存编程错误标志（内部闪存块）

FLASH_EPPERR_FLAG：　　闪存擦写错误标志（内部闪存块）

FLASH_SPIM_ODF_FLAG：　　外部闪存操作完成标志

FLASH_SPIM_PRGMERR_FLAG：　　外部闪存编程错误标志

FLASH_SPIM_EPPERR_FLAG：　　外部闪存擦写错误标志

示例

```
flash_flag_clear(FLASH_ODF_FLAG);
```

5.10.3 函数 flash_operation_status_get

下表描述了函数 flash_operation_status_get

表 174. 函数 flash_operation_status_get

项目	描述
函数名	flash_operation_status_get
函数原型	flash_status_type flash_operation_status_get(void);
功能描述	获取操作状态
输入参数	无
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

flash_status_type

FLASH_OPERATE_BUSY：　　闪存操作忙

FLASH_PROGRAM_ERROR：　　闪存编程错误

FLASH_EPP_ERROR：　　闪存擦写错误

FLASH_OPERATE_DONE：　　闪存操作完成

FLASH_OPERATE_TIMEOUT：　　闪存操作超时

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
/* check for the flash status */
status = flash_operation_status_get();
```

5.10.4 函数 flash_spim_operation_status_get

下表描述了函数 flash_spim_operation_status_get

表 175. 函数 flash_spim_operation_status_get

项目	描述
函数名	flash_spim_operation_status_get
函数原型	flash_status_type flash_spim_operation_status_get (void);

项目	描述
功能描述	获取外部闪存操作状态
输入参数	无
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
/* check for the flash status */
status = flash_spim_operation_status_get();
```

5.10.5 函数 flash_operation_wait_for

下表描述了函数 flash_operation_wait_for

表 176. 函数 flash_operation_wait_for

项目	描述
函数名	flash_operation_wait_for
函数原型	flash_status_type flash_operation_wait_for(uint32_t time_out);
功能描述	等待闪存操作
输入参数	time_out：等待的超时退出时间 该参数在 flash.h 头文件中定义了部分常用的超时时间，详细描述见 *flash_time_out*
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

flash_time_out

ERASE_TIMEOUT： 擦除超时
PROGRAMMING_TIMEOUT： 编程超时
SPIM_ERASE_TIMEOUT： 外部闪存擦除超时
SPIM_PROGRAMMING_TIMEOUT： 外部闪存编程超时
OPERATION_TIMEOUT： 一般操作超时

示例

```
/* wait for operation to be completed */
status = flash_operation_wait_for(PROGRAMMING_TIMEOUT);
```

5.10.6 函数 flash_spim_operation_wait_for

下表描述了函数 flash_spim_operation_wait_for

表 177. 函数 flash_spim_operation_wait_for

项目	描述
函数名	flash_spim_operation_wait_for
函数原型	flash_status_type flash_spim_operation_wait_for(uint32_t time_out);
功能描述	等待外部闪存操作

项目	描述
输入参数	time_out：等待的超时退出时间 该参数在 flash.h 头文件中定义了部分常用的超时时间，详细描述见 *flash_time_out*
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

示例

```
/* wait for operation to be completed */
status = flash_spim_operation_wait_for(PROGRAMMING_TIMEOUT);
```

5.10.7 函数 flash_unlock

下表描述了函数 flash_unlock

表 178. 函数 flash_unlock

项目	描述
函数名	flash_unlock
函数原型	void flash_unlock(void);
功能描述	解锁内部闪存控制寄存器
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
flash_unlock();
```

5.10.8 函数 flash_spim_unlock

下表描述了函数 flash_spim_unlock

表 179. 函数 flash_spim_unlock

项目	描述
函数名	flash_spim_unlock
函数原型	void flash_spim_unlock(void);
功能描述	解锁外部闪存控制寄存器
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
flash_spim_unlock();
```

5.10.9 函数 flash_lock

下表描述了函数 flash_lock

表 180. 函数 flash_lock

项目	描述
函数名	flash_lock
函数原型	void flash_lock(void);
功能描述	锁定内部闪存控制寄存器
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
flash_lock();
```

5.10.10 函数 flash_spim_lock

下表描述了函数 flash_spim_lock

表 181. 函数 flash_spim_lock

项目	描述
函数名	flash_spim_lock
函数原型	void flash_spim_lock(void);
功能描述	锁定外部闪存控制寄存器
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
flash_spim_lock();
```

5.10.11 函数 flash_sector_erase

下表描述了函数 flash_sector_erase

表 182. 函数 flash_sector_erase

项目	描述
函数名	flash_sector_erase
函数原型	flash_status_type flash_sector_erase(uint32_t sector_address);
功能描述	擦除指定地址所在扇区的闪存数据
输入参数	sector_address：需要擦除的扇区所在地址，通常输入扇区的起始地址
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无

项目	描述
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
flash_unlock();
status = flash_sector_erase(0x08001000);
```

5.10.12 函数 flash_internal_all_erase

下表描述了函数 flash_internal_all_erase

表 183. 函数 flash_internal_all_erase

项目	描述
函数名	flash_internal_all_erase
函数原型	flash_status_type flash_internal_all_erase(void);
功能描述	擦除内部闪存数据
输入参数	无
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
flash_unlock();
status = flash_internal_all_erase();
```

5.10.13 函数 flash_spim_all_erase

下表描述了函数 flash_spim_all_erase

表 184. 函数 flash_spim_all_erase

项目	描述
函数名	flash_spim_all_erase
函数原型	flash_status_type flash_spim_all_erase(void);
功能描述	擦除外部闪存数据
输入参数	无
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
flash_spim_unlock();
status = flash_spim_all_erase();
```

5.10.14 函数 flash_user_system_data_erase

下表描述了函数 flash_user_system_data_erase

表 185. 函数 flash_user_system_data_erase

项目	描述
函数名	flash_user_system_data_erase
函数原型	flash_status_type flash_user_system_data_erase(void);
功能描述	擦除用户系统数据区数据
输入参数	无
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

注意：该函数会保持执行前闪存访问保护（FAP）的状态，仅擦除用户系统数据区除了 FAP 之外的其余数据。

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
flash_unlock();
status = flash_user_system_data_erase();
```

5.10.15 函数 flash_word_program

下表描述了函数 flash_word_program

表 186. 函数 flash_word_program

项目	描述
函数名	flash_word_program
函数原型	flash_status_type flash_word_program(uint32_t address, uint32_t data);
功能描述	编程一个字的数据到指定的地址
输入参数 1	address：编程的地址，需字对齐
输入参数 2	data：编程的数据
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	该地址的闪存数据必须全是 0xFF 才允许编程
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
uint32_t i;
flash_unlock();
status = flash_sector_erase(0x08001000);
if(status = FLASH_OPERATE_DONE)
{
  /* program 256 words */
  for(I = 0; I < 256; i++)
  {
```

```
    status = flash_word_program(0x08001000 + i*4, i);
  }
}
```

5.10.16 函数 flash_halfword_program

下表描述了函数 flash_halfword_program

表 187. 函数 flash_halfword_program

项目	描述
函数名	flash_halfword_program
函数原型	flash_status_type flash_halfword_program(uint32_t address, uint16_t data);
功能描述	编程一个半字的数据到指定的地址
输入参数 1	address：编程的地址，需半字对齐
输入参数 2	data：编程的数据
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	该地址的闪存数据必须全是 0xFF 才允许编程
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
uint32_t i;
flash_unlock();
status = flash_sector_erase(0x08001000);
if(status = FLASH_OPERATE_DONE)
{
  /* program 256 halfwords */
  for(I = 0; I < 256; i++)
  {
    status = flash_halfword_program(0x08001000 + i*2, (uint16_t)i);
  }
}
```

5.10.17 函数 flash_byte_program

下表描述了函数 flash_byte_program

表 188. 函数 flash_byte_program

项目	描述
函数名	flash_byte_program
函数原型	flash_status_type flash_byte_program(uint32_t address, uint8_t data);
功能描述	编程一个字节的数据到指定的地址
输入参数 1	address：编程的地址
输入参数 2	data：编程的数据
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*

项目	描述
先决条件	该地址的闪存数据必须是 0xFF 才允许编程
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
uint32_t i;
flash_unlock();
status = flash_sector_erase(0x08001000);
if(status = FLASH_OPERATE_DONE)
{
  /* program 256 bytes */
  for(I = 0; I < 256; i++)
  {
    status = flash_byte_program(0x08001000 + i*2, (uint8_t)i);
  }
}
```

5.10.18 函数 flash_user_system_data_program

下表描述了函数 flash_user_system_data_program

表 189. 函数 flash_user_system_data_program

项目	描述
函数名	flash_user_system_data_program
函数原型	flash_status_type flash_user_system_data_program (uint32_t address, uint8_t data);
功能描述	编程一个字节的数据到指定的用户系统数据区地址
输入参数 1	address：编程的地址
输入参数 2	data：编程的数据
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	该地址的用户系统数据及其反码数据必须都是 0xFF 才允许编程
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
flash_unlock();
status = flash_user_system_data_erase();
if(status = FLASH_OPERATE_DONE)
{
  /* program user system data */
  status = flash_user_system_data_program(0x1FFFF804, 0x55);
}
```

5.10.19 函数 flash_epp_set

下表描述了函数 flash_epp_set

表 190. 函数 flash_epp_set

项目	描述
函数名	flash_epp_set
函数原型	flash_status_type flash_epp_set(uint32_t *sector_bits);
功能描述	配置擦除编程保护
输入参数	*sector_bits：擦除编程保护扇区地址范围的指针，每一位保护 4KB 范围的扇区，最后一位保护剩余的扇区，位置 1 表示开启对应范围扇区的保护
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
uint32_t epp_val;
flash_unlock();
status = flash_user_system_data_erase();
if(status = FLASH_OPERATE_DONE)
{
  epp_val = 0x00000001;
  /* program epp */
  status = flash_epp_set(&epp_val);
}
```

5.10.20 函数 flash_epp_status_get

下表描述了函数 flash_epp_status_get

表 191. 函数 flash_epp_status_get

项目	描述
函数名	flash_epp_status_get
函数原型	void flash_epp_status_get(uint32_t *sector_bits);
功能描述	获取擦除编程保护状态
输入参数	无
输出参数	*sector_bits：擦除编程保护扇区地址范围的指针，每一位保护 4KB 范围的扇区，最后一位保护剩余的扇区，位置 1 表示开启对应范围扇区的保护
返回值	无
先决条件	无
被调用函数	无

示例

```
uint32_t epp_val;
/* get epp status */
flash_epp_status_get(&epp_val);
```

5.10.21 函数 flash_fap_enable

下表描述了函数 flash_fap_enable

表 192. 函数 flash_fap_enable

项目	描述
函数名	flash_fap_enable
函数原型	flash_status_type flash_fap_enable(confirm_state new_state);
功能描述	配置访问保护
输入参数	new_state：配置访问保护状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

注意：该函数将擦除所有用户系统数据区数据，如果调用之前有编程其他用户系统数据，调用后需重新编程。

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
flash_unlock();
status = flash_fap_enable(TRUE);
```

5.10.22 函数 flash_fap_status_get

下表描述了函数 flash_fap_status_get

表 193. 函数 flash_fap_status_get

项目	描述
函数名	flash_fap_status_get
函数原型	flag_status flash_fap_status_get(void);
功能描述	获取访问保护状态
输入参数	无
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET，RESET.
先决条件	无
被调用函数	无

示例

```
flag_status status;
status = flash_fap_status_get();
```

5.10.23 函数 flash_ssb_set

下表描述了函数 flash_ssb_set

表 194. 函数 flash_ssb_set

项目	描述
函数名	flash_ssb_set
函数原型	flash_status_type flash_ssb_set(uint8_t usd_ssb);
功能描述	配置系统配置字节

项目	描述
输入参数	usd_ssb：系统配置字节值，是其各组数据组合后的值，必须包括其所有组数据。 各组定义见 *ssb_data_define*
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

ssb_data_define

type 1:

USD_WDT_ATO_DISABLE：　　看门狗自动启动失能

USD_WDT_ATO_ENABLE：　　看门狗自动启动使能

type 2:

USD_DEPSLP_NO_RST：　　进入深度睡眠时不产生复位

USD_DEPSLP_RST：　　进入深度睡眠时产生复位

type 3:

USD_STDBY_NO_RST：　　进入待机模式时不产生复位

USD_STDBY_RST：　　进入待机模式时产生复位

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
flash_unlock();
status = flash_user_system_data_erase();
if(status = FLASH_OPERATE_DONE)
{
  status = flash_ssb_set(USD_WDT_ATO_DISABLE | USD_DEPSLP_NO_RST | USD_STDBY_RST);
}
```

5.10.24 函数 flash_ssb_status_get

下表描述了函数 flash_ssb_status_get

表 195. 函数 flash_ssb_status_get

项目	描述
函数名	flash_ssb_status_get
函数原型	uint8_t flash_ssb_status_get(void);
功能描述	获取系统配置字节状态
输入参数	无
输出参数	无
返回值	系统配置字节值，该值对应 bit 含义可参考 *ssb_data_define*
先决条件	无
被调用函数	无

示例

```
uint8_t ssb_val;
ssb_val = flash_ssb_status_get();
```

5.10.25 函数 flash_interrupt_enable

下表描述了函数 flash_interrupt_enable

表 196. 函数 flash_interrupt_enable

项目	描述
函数名	flash_interrupt_enable
函数原型	void flash_interrupt_enable(uint32_t flash_int, confirm_state new_state);
功能描述	配置闪存中断
输入参数 1	flash_int：闪存中断类型，可以是任意类型组合，详细类型描述可见 *flash_interrupt_type*
输入参数 2	new_state：配置中断状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flash_interrupt_type

FLASH_ERR_INT： 闪存错误中断

FLASH_ODF_INT： 闪存操作完成中断

FLASH_SPIM_ERR_INT： 外部闪存错误中断

FLASH_SPIM_ODF_INT： 外部闪存操作完成中断

示例

```
flash_interrupt_enable(FLASH_ERR_INT | FLASH_ODF_INT, TRUE);
```

5.10.26 函数 flash_spim_model_select

下表描述了函数 flash_spim_model_select

表 197. 函数 flash_spim_model_select

项目	描述
函数名	flash_spim_model_select
函数原型	void flash_spim_model_select(flash_spim_model_type mode);
功能描述	选择外部闪存类型
输入参数	mode：外部闪存类型，类型种类见 *flash_spim_mode_type*
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flash_spim_mode_type

FLASH_SPIM_MODEL1：外部闪存类型 1

FLASH_SPIM_MODEL2：外部闪存类型 2

示例

```
flash_spim_model_select(FLASH_SPIM_MODEL1);
```

5.10.27 函数 flash_spim_encryption_range_set

下表描述了函数 flash_spim_encryption_range_set

表 198. 函数 flash_spim_encryption_range_set

项目	描述
函数名	flash_spim_encryption_range_set
函数原型	void flash_spim_encryption_range_set(uint32_t decode_address);
功能描述	配置外部闪存数据加密范围
输入参数	decode_address：加密范围地址，需按字对齐，该地址之前的数据为密文存储
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
flash_spim_encryption_range_set(0x08401000);
```

5.10.28 函数 flash_slib_enable

下表描述了函数 flash_slib_enable

表 199. 函数 flash_slib_enable

项目	描述
函数名	flash_slib_enable
函数原型	flash_status_type flash_slib_enable(uint32_t pwd, uint16_t start_sector, uint16_t data_start_sector, uint16_t end_sector);
功能描述	使能安全库区功能，并配置其地址范围
输入参数 1	pwd：安全库区密码，安全库区数据为密文存储，加密计算跟其相关联，解除时需输入正确的密码
输入参数 2	start_sector：安全库区开始扇区号码
输入参数 3	data_start_sector：安全库区数据区开始扇区号码
输入参数 4	end_sector：安全库区结束扇区号码
输出参数	无
返回值	操作状态，该参数详细描述见 *flash_status_type*
先决条件	无
被调用函数	无

示例

```
flash_status_type status = FLASH_OPERATE_DONE;
status = flash_slib_enable(0x12345678, 0x04 , 0x05, 0x06);
```

5.10.29 函数 flash_slib_disable

下表描述了函数 flash_slib_disable

表 200. 函数 flash_slib_disable

项目	描述
函数名	flash_slib_disable

项目	描述
函数原型	error_status flash_slib_disable(uint32_t pwd);
功能描述	失能安全库区功能
输入参数	pwd：安全库区密码，需输入正确的密码，否则必须复位后才能再次输入
输出参数	无
返回值	错误状态 该值为其中之一：ERROE，SUCCESS.
先决条件	无
被调用函数	无

注意：调用该函数成功后会执行内部闪存全擦除操作。

示例

```
error_status status;
status = flash_slib_disable(0x12345678);
```

5.10.30 函数 flash_slib_remaining_count_get

下表描述了函数 flash_slib_remaining_count_get

表 201. 函数 flash_slib_remaining_count_get

项目	描述
函数名	flash_slib_remaining_count_get
函数原型	uint32_t flash_slib_remaining_count_get(void);
功能描述	获取安全库区功能剩余可使用次数
输入参数	无
输出参数	无
返回值	安全库区功能剩余可使用次数.
先决条件	无
被调用函数	无

示例

```
uint32_t num;
num = flash_slib_remaining_count_get();
```

5.10.31 函数 flash_slib_state_get

下表描述了函数 flash_slib_state_get

表 202. 函数 flash_slib_state_get

项目	描述
函数名	flash_slib_state_get
函数原型	flag_status flash_slib_state_get(void);
功能描述	获取安全库区功能状态
输入参数	无
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET，RESET.
先决条件	无

项目	描述
被调用函数	无

示例

```
flag_status status;
status = flash_slib_state_get();
```

5.10.32 函数 flash_slib_start_sector_get

下表描述了函数 flash_slib_start_sector_get

表 203. 函数 flash_slib_start_sector_get

项目	描述
函数名	flash_slib_start_sector_get
函数原型	uint16_t flash_slib_start_sector_get(void);
功能描述	获取安全库区起始扇区号码
输入参数	无
输出参数	无
返回值	安全库区起始扇区号码
先决条件	无
被调用函数	无

示例

```
uint16_t num;
num = flash_slib_start_sector_get();
```

5.10.33 函数 flash_slib_datastart_sector_get

下表描述了函数 flash_slib_datastart_sector_get

表 204. 函数 flash_slib_datastart_sector_get

项目	描述
函数名	flash_slib_datastart_sector_get
函数原型	uint16_t flash_slib_datastart_sector_get(void);
功能描述	获取安全库区数据区起始扇区号码
输入参数	无
输出参数	无
返回值	安全库区数据区起始扇区号码
先决条件	无
被调用函数	无

示例

```
uint16_t num;
num = flash_slib_datastart_sector_get();
```

5.10.34 函数 flash_slib_end_sector_get

下表描述了函数 flash_slib_end_sector_get

表 205. 函数 flash_slib_end_sector_get

项目	描述
函数名	flash_slib_end_sector_get
函数原型	uint16_t flash_slib_end_sector_get(void);
功能描述	获取安全库区结束扇区号码
输入参数	无
输出参数	无
返回值	安全库区结束扇区号码
先决条件	无
被调用函数	无

示例

```
uint16_t num;
num = flash_slib_end_sector_get();
```

5.10.35 函数 flash_crc_calibrate

下表描述了函数 flash_crc_calibrate

表 206. 函数 flash_crc_calibrate

项目	描述
函数名	flash_crc_calibrate
函数原型	uint32_t flash_crc_calibrate(uint32_t start_sector, uint32_t sector_cnt);
功能描述	内部闪存指定范围扇区的 CRC 计算
输入参数 1	start_sector：CRC 计算开始扇区号码
输入参数 2	sector_cnt：CRC 计算扇区个数
输出参数	无
返回值	计算的 CRC 值
先决条件	无
被调用函数	无

注意：计算的扇区不能既包括安全库区又包括普通区域，必须只能单一区域。

示例

```
uint32_t crc_val;
crc_val = flash_crc_calibrate(0, 10);
```

5.11 通用和复用功能输出输出（GPIO/IOMUX）

GPIO 和 IOMUX 寄存器结构 gpio_type 和 iomux_type，定义于文件“at32f413_gpio.h”如下：

```
/**
  * @brief type define gpio register all
  */
typedef struct
{


} gpio_type;
```

```
/**
  * @brief type define iomux register all
  */
typedef struct
{


} iomux_type;
```

下表给出了 GPIO 寄存器总览：

表 207. GPIO 寄存器对应表

寄存器	描述
cfglr	GPIO 配置低寄存器
cfghr	GPIO 配置高寄存器
idt	GPIO 输入数据寄存器
odt	GPIO 输出数据寄存器
scr	GPIO 设置/清除寄存器
clr	GPIO 清除寄存器
wpr	GPIO 写保护寄存器

下表给出了 IOMUX 寄存器总览：

表 208. IOMUX 寄存器对应表

寄存器	描述
evtout	事件输出控制寄存器
remap	IO 复用重映射寄存器
exintc1	复用外部中断配置寄存器 1
exintc2	复用外部中断配置寄存器 2
exintc3	复用外部中断配置寄存器 3
exintc4	复用外部中断配置寄存器 4
remap2	IO 复用重映射寄存器 2
remap3	IO 复用重映射寄存器 3
remap4	IO 复用重映射寄存器 4
remap5	IO 复用重映射寄存器 5
remap6	IO 复用重映射寄存器 6
remap7	IO 复用重映射寄存器 7

下表给出了 GPIO 和 IOMUX 库函数总览：

表 209. GPIO 和 IOMUX 库函数总览

函数名	描述
gpio_reset	GPIO 由 CRM 复位寄存器复位
gpio_iomux_reset	IOMUX 由 CRM 复位寄存器复位
gpio_init	初始化 GPIO 外设
gpio_default_para_init	初始化 GPIO 默认参数

gpio_input_data_bit_read	读取指定的 GPIO 输入端口的引脚
gpio_input_data_read	读取指定的 GPIO 输入端口
gpio_output_data_bit_read	读取指定的 GPIO 输出端口的引脚
gpio_output_data_read	读取指定的 GPIO 输出端口
gpio_bits_set	置位 GPIO 引脚
gpio_bits_reset	复位 GPIO 引脚
gpio_bits_write	写 GPIO 引脚值
gpio_port_write	写 GPIO 端口值
gpio_pin_wp_config	配置 GPIO 引脚写保护
gpio_event_output_config	配置 GPIO 事件输出功能
gpio_event_output_enable	启用或禁用 GPIO 事件输出功能
gpio_pin_remap_config	配置引脚 IOMUX 功能
gpio_exint_line_config	配置 GPIO 外部中断线

5.11.1 函数 gpio_reset

下表描述了函数 gpio_reset

表 210. 函数 gpio_reset

项目	描述
函数名	gpio_reset
函数原型	void gpio_reset(gpio_type *gpio_x);
功能描述	GPIO 由 CRM 复位寄存器复位
输入参数	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset();

示例

```
gpio_reset(GPIOA);
```

5.11.2 函数 gpio_iomux_reset

下表描述了函数 gpio_iomux_reset

表 211. 函数 gpio_iomux_reset

项目	描述
函数名	gpio_iomux_reset
函数原型	void gpio_iomux_reset ();
功能描述	IOMUX 由 CRM 复位寄存器复位
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset();

示例

```
gpio_iomux_reset();
```

5.11.3 函数 gpio_init

下表描述了函数 gpio_init

表 212. 函数 gpio_init

项目	描述
函数名	gpio_init
函数原型	void gpio_init(gpio_type *gpio_x, gpio_init_type *gpio_init_struct);
功能描述	初始化 GPIO 外设
输入参数 1	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输入参数 2	gpio_init_struct：指向结构体 gpio_init_type 的指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

gpio_init_type structure

gpio_init_type 在 at32f413_gpio.h 中

```
typedef struct
{
  uint32_t                gpio_pins;
  gpio_output_type        gpio_out_type;
  gpio_pull_type          gpio_pull;
  gpio_mode_type          gpio_mode;
  gpio_drive_type         gpio_drive_strength;
} gpio_init_type;
```

gpio_pins

选择需要配置的 GPIO 引脚

GPIO_PINS_0： GPIO 引脚 0
GPIO_PINS_1： GPIO 引脚 1
GPIO_PINS_2： GPIO 引脚 2
GPIO_PINS_3： GPIO 引脚 3
GPIO_PINS_4： GPIO 引脚 4
GPIO_PINS_5： GPIO 引脚 5
GPIO_PINS_6： GPIO 引脚 6
GPIO_PINS_7： GPIO 引脚 7
GPIO_PINS_8： GPIO 引脚 8
GPIO_PINS_9： GPIO 引脚 9
GPIO_PINS_10： GPIO 引脚 10
GPIO_PINS_11： GPIO 引脚 11
GPIO_PINS_12： GPIO 引脚 12
GPIO_PINS_13： GPIO 引脚 13
GPIO_PINS_14： GPIO 引脚 14

GPIO_PINS_15： GPIO 引脚 15

gpio_out_type

设置 GPIO 输出类型

GPIO_OUTPUT_PUSH_PULL： GPIO 推挽输出模式

GPIO_OUTPUT_OPEN_DRAIN： GPIO 开漏输出模式

gpio_pull

设置 GPIO 上下拉模式

GPIO_PULL_NONE： GPIO 无上下拉

GPIO_PULL_UP： GPIO 上拉模式

GPIO_PULL_DOWN： GPIO 下拉模式

gpio_mode

设置 GPIO 模式

GPIO_MODE_INPUT： 配置 GPIO 为输入模式

GPIO_MODE_OUTPUT： 配置 GPIO 为输出模式

GPIO_MODE_MUX： 配置 GPIO 为复用模式

GPIO_MODE_ANALOG： 配置 GPIO 为模拟模式

gpio_drive_strength

设置 GPIO 驱动能力

GPIO_DRIVE_STRENGTH_STRONGER：较大电流推动/吸入能力

GPIO_DRIVE_STRENGTH_MODERATE：适中电流推动/吸入能力

示例

```
gpio_init_type gpio_init_struct;
gpio_init_struct.gpio_pins = GPIO_PINS_0;
gpio_init_struct.gpio_mode = GPIO_MODE_MUX;
gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL;
gpio_init_struct.gpio_pull = GPIO_PULL_NONE;
gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER;
gpio_init(GPIOA, &gpio_init_struct);
```

5.11.4 函数 gpio_default_para_init

下表描述了函数 gpio_default_para_init

表 213. 函数 gpio_default_para_init

项目	描述
函数名	gpio_default_para_init
函数原型	void gpio_default_para_init(gpio_init_type *gpio_init_struct);
功能描述	初始化 GPIO 默认参数
输入参数	gpio_init_struct：指向结构体 gpio_init_type 的指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

下表描述了 gpio_init_struct 各个成员的默认值

表 214. gpio_init_struct 默认值

成员	默认值
gpio_pins	GPIO_PINS_ALL
gpio_mode	GPIO_MODE_INPUT
gpio_out_type	GPIO_OUTPUT_PUSH_PULL
gpio_pull	GPIO_PULL_NONE
gpio_drive_strength	GPIO_DRIVE_STRENGTH_STRONGER

示例

```
gpio_init_type gpio_init_struct;
gpio_default_para_init(&gpio_init_struct);
```

5.11.5 函数 gpio_input_data_bit_read

下表描述了函数 gpio_input_data_bit_read

表 215. 函数 gpio_input_data_bit_read

项目	描述
函数名	gpio_input_data_bit_read
函数原型	flag_status gpio_input_data_bit_read(gpio_type *gpio_x, uint16_t pins);
功能描述	读取指定的 GPIO 输入端口的引脚
输入参数 1	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输入参数 2	pins：将要配置的 GPIO 引脚，参考 *gpio_pins* 查看取值范围
输出参数	无
返回值	读取的 GPIO 输入引脚状态
先决条件	无
被调用函数	无

示例

```
gpio_input_data_bit_read(GPIOA, GPIO_PINS_0);
```

5.11.6 函数 gpio_input_data_read

下表描述了函数 gpio_input_data_read

表 216. 函数 gpio_input_data_read

项目	描述
函数名	gpio_input_data_read
函数原型	uint16_t gpio_input_data_read(gpio_type *gpio_x);
功能描述	读取指定的 GPIO 输入端口
输入参数	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输出参数	无
返回值	读取的 GPIO 输入端口状态
先决条件	无
被调用函数	无

示例

```
gpio_input_data_read(GPIOA);
```

5.11.7 函数 gpio_output_data_bit_read

下表描述了函数 gpio_output_data_bit_read

表 217. 函数 gpio_output_data_bit_read

项目	描述
函数名	gpio_output_data_bit_read
函数原型	uint16_t gpio_output_data_bit_read(gpio_type *gpio_x);
功能描述	读取指定的 GPIO 输出端口的引脚
输入参数 1	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输入参数 2	pins：将要配置的 GPIO 引脚，参考 *gpio_pins* 查看取值范围
输出参数	无
返回值	读取的 GPIO 输出引脚状态
先决条件	无
被调用函数	无

示例

```
gpio_output_data_bit_read(GPIOA, GPIO_PINS_0);
```

5.11.8 函数 gpio_output_data_read

下表描述了函数 gpio_output_data_read

表 218. 函数 gpio_output_data_read

项目	描述
函数名	gpio_output_data_read
函数原型	uint16_t gpio_output_data_read(gpio_type *gpio_x);
功能描述	读取指定的 GPIO 输出端口
输入参数	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输出参数	无
返回值	读取的 GPIO 输出端口状态
先决条件	无
被调用函数	无

示例

```
gpio_output_data_read(GPIOA);
```

5.11.9 函数 gpio_bits_set

下表描述了函数 gpio_bits_set

表 219. 函数 gpio_bits_set

项目	描述
函数名	gpio_bits_set
函数原型	void gpio_bits_set(gpio_type *gpio_x, uint16_t pins);

项目	描述
功能描述	置位 GPIO 引脚
输入参数 1	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输入参数 2	pins：将要配置的 GPIO 引脚，参考 *gpio_pins* 查看取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
gpio_bits_set(GPIOA, GPIO_PINS_0);
```

5.11.10 函数 gpio_bits_reset

下表描述了函数 gpio_bits_reset

表 220. 函数 gpio_bits_reset

项目	描述
函数名	gpio_bits_reset
函数原型	void gpio_bits_reset(gpio_type *gpio_x, uint16_t pins);
功能描述	复位 GPIO 引脚
输入参数 1	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输入参数 2	pins：将要配置的 GPIO 引脚，参考 *gpio_pins* 查看取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
gpio_bits_reset(GPIOA, GPIO_PINS_0);
```

5.11.11 函数 gpio_bits_write

下表描述了函数 gpio_bits_write

表 221. 函数 gpio_bits_write

项目	描述
函数名	gpio_bits_write
函数原型	void gpio_bits_write(gpio_type *gpio_x, uint16_t pins, confirm_state bit_state);
功能描述	写 GPIO 引脚值
输入参数 1	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输入参数 2	pins：将要配置的 GPIO 引脚，参考 *gpio_pins* 查看取值范围
输入参数 3	bit_state：将要写入的 GPIO 引脚值，可选择 1（TRUE）或 0（FALSE）
输出参数	无
返回值	无

项目	描述
先决条件	无
被调用函数	无

示例

```
gpio_bits_write(GPIOA, GPIO_PINS_0, TRUE);
```

5.11.12 函数 gpio_port_write

下表描述了函数 gpio_port_write

表 222. 函数 gpio_port_write

项目	描述
函数名	gpio_port_write
函数原型	void gpio_port_write(gpio_type *gpio_x, uint16_t port_value);
功能描述	写 GPIO 端口值
输入参数 1	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输入参数 2	port_value：将要写入的端口值，可取 0x0000~0xFFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
gpio_port_write(GPIOA, 0xFFFF);
```

5.11.13 函数 gpio_pin_wp_config

下表描述了函数 gpio_pin_wp_config

表 223. 函数 gpio_pin_wp_config

项目	描述
函数名	gpio_pin_wp_config
函数原型	void gpio_pin_wp_config(gpio_type *gpio_x, uint16_t pins);
功能描述	配置 GPIO 引脚写保护
输入参数 1	gpio_x：所选择的 GPIO 外设，该参数可以选取自其中之一： GPIOA, GPIOB, GPIOC, GPIOD, GPIOF
输入参数 2	pins：将要配置的 GPIO 引脚，参考 *gpio_pins* 查看取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
gpio_pin_wp_config(GPIOA, GPIO_PINS_0);
```

5.11.14 函数 gpio_event_output_config

下表描述了函数 gpio_event_output_config

表 224. 函数 gpio_event_output_config

项目	描述
函数名	gpio_event_output_config
函数原型	void gpio_event_output_config(gpio_port_source_type gpio_port_source, gpio_pins_source_type gpio_pin_source);
功能描述	配置 GPIO 事件输出功能
输入参数 1	gpio_port_source：将要配置的 GPIO 端口
输入参数 2	gpio_pin_source：将要配置的 GPIO 引脚
输出参数	无
返回值	无
先决条件	无
被调用函数	无

gpio_port_source

设置 GPIO 端口

GPIO_PORT_SOURCE_GPIOA：选择 GPIO 端口 A
GPIO_PORT_SOURCE_GPIOB：选择 GPIO 端口 B
GPIO_PORT_SOURCE_GPIOC：选择 GPIO 端口 C
GPIO_PORT_SOURCE_GPIOD：选择 GPIO 端口 D
GPIO_PORT_SOURCE_GPIOF：选择 GPIO 端口 F

gpio_pin_source

设置 GPIO 引脚

GPIO_PINS_SOURCE0： GPIO 引脚 0
GPIO_PINS_SOURCE1： GPIO 引脚 1
GPIO_PINS_SOURCE2： GPIO 引脚 2
GPIO_PINS_SOURCE3： GPIO 引脚 3
GPIO_PINS_SOURCE4： GPIO 引脚 4
GPIO_PINS_SOURCE5： GPIO 引脚 5
GPIO_PINS_SOURCE6： GPIO 引脚 6
GPIO_PINS_SOURCE7： GPIO 引脚 7
GPIO_PINS_SOURCE8： GPIO 引脚 8
GPIO_PINS_SOURCE9： GPIO 引脚 9
GPIO_PINS_SOURCE10： GPIO 引脚 10
GPIO_PINS_SOURCE11： GPIO 引脚 11
GPIO_PINS_SOURCE12： GPIO 引脚 12
GPIO_PINS_SOURCE13： GPIO 引脚 13
GPIO_PINS_SOURCE14： GPIO 引脚 14
GPIO_PINS_SOURCE15： GPIO 引脚 15

示例

```
gpio_event_output_config(GPIO_PORT_SOURCE_GPIOA, GPIO_PINS_SOURCE0);
```

5.11.15 函数 gpio_event_output_enable

下表描述了函数 gpio_event_output_enable

表 225. 函数 gpio_event_output_enable

项目	描述
函数名	gpio_event_output_enable
函数原型	void gpio_event_output_enable(confirm_state new_state);
功能描述	启用或禁用 GPIO 事件输出功能
输入参数	new_state：将要配置的 GPIO 事件输出状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
gpio_event_output_enable(TRUE);
```

5.11.16 函数 gpio_pin_remap_config

下表描述了函数 gpio_pin_remap_config

表 226. 函数 gpio_pin_remap_config

项目	描述
函数名	gpio_pin_remap_config
函数原型	void gpio_pin_remap_config(uint32_t gpio_remap, confirm_state new_state);
功能描述	配置引脚 IOMUX 功能
输入参数 1	gpio_remap：将要配置的 IOMUX 外设选项
输入参数 2	new_state：将要配置的 IOMUX 功能状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

gpio_remap

选择要配置的 IOMUX 外设，由于参数选择很多，这里就不一一列举，详情可在参考手册进行查看

SPI1_MUX_01：spi1_cs/i2s1_ws(pa15), spi1_sck/i2s1_ck(pb3), spi1_miso(pb4), spi1_mosi/i2s1_sd(pb5), i2s1_mck(pb0)

I2C1_MUX：i2c1_scl(pb8), i2c1_sda(pb9)

…

SWJTAG_GMUX_100：SWJ 接口全部禁用 (jtag-dp + sw-dp)

PD01_GMUX：pd0/pd1 映射到 osc_in/osc_out

示例

```
gpio_pin_remap_config(SPI1_MUX_01, TRUE);
```

5.11.17 函数 gpio_exint_line_config

下表描述了函数 gpio_exint_line_config

表 227. 函数 gpio_exint_line_config

项目	描述
函数名	gpio_exint_line_config
函数原型	void gpio_exint_line_config(gpio_port_source_type gpio_port_source, gpio_pins_source_type gpio_pin_source);
功能描述	配置 GPIO 外部中断线
输入参数 1	gpio_port_source：将要配置的 GPIO 端口
输入参数 2	gpio_pin_source：将要配置的 GPIO 引脚
输出参数	无
返回值	无
先决条件	无
被调用函数	无

gpio_port_source

设置 GPIO 端口，参考 *gpio_port_source* 查看取值范围

gpio_pin_source

设置 GPIO 引脚，参考 *gpio_pin_source* 查看取值范围

示例

```
gpio_exint_line_config(GPIO_PORT_SOURCE_GPIOA, GPIO_PINS_SOURCE0);
```

5.12 I2C 接口（I2C）

I2C 寄存器结构 i2c_type，定义于文件“at32f413_i2c.h”如下：

```
/**
  * @brief type define i2c register all
  */
typedef struct
{


} i2c_type;
```

下表给出了 I2C 寄存器总览：

表 228. I2C 寄存器对应表

寄存器	描述
ctrl1	I2C 控制寄存器 1
ctrl2	I2C 控制寄存器 2
oaddr1	I2C 本机地址寄存器 1
oaddr2	I2C 本机地址寄存器 2
dt	I2C 数据寄存器
sts1	I2C 状态寄存器 1
sts2	I2C 状态寄存器 2

寄存器	描述
clkctrl	I2C 时钟控制寄存器
tmrise	I2C 上升时间寄存器

下表给出了 I2C 库函数总览：

表 229. I2C 库函数总览

函数名	描述
i2c_reset	I2C 外设复位
i2c_software_reset	I2C 外设复位
i2c_init	设置 I2C 总线速度
i2c_own_address1_set	设置本机地址 1
i2c_own_address2_set	设置本机地址 2
i2c_own_address2_enable	本机地址 2 使能
i2c_smbus_enable	Smbus 模式使能
i2c_enable	I2C 外设使能
i2c_fast_mode_duty_set	设置快速模式占空比
i2c_clock_stretch_enable	时钟延展使能
i2c_ack_enable	ACK 响应使能
i2c_master_receive_ack_set	设置主机接收模式应答控制
i2c_pec_position_set	在 smbus 模式并且在主机接收模式下，用于设置 PEC 的位置
i2c_general_call_enable	广播地址使能
i2c_arp_mode_enable	SMBus ARP 地址使能
i2c_smbus_mode_set	SMBus 设备模式选择
i2c_smbus_alert_set	SMBus 提醒引脚电平设置
i2c_pec_transmit_enable	PEC 传输使能
i2c_pec_calculate_enable	PEC 计算使能
i2c_pec_value_get	获取当前 PEC 值
i2c_dma_end_transfer_set	DMA 传输结束指示
i2c_dma_enable	DMA 传输使能
i2c_interrupt_enable	I2C 中断使能
i2c_start_generate	产生起始条件
i2c_stop_generate	产生停止条件
i2c_7bit_address_send	发送 7 位从机地址
i2c_data_send	发送数据
i2c_data_receive	接收数据
i2c_flag_get	获取标志
i2c_flag_clear	清除标志

表 230. I2C 应用层库函数总览

函数名	描述
i2c_config	I2C 应用初始化
i2c_lowlevel_init	I2C 底层初始化
i2c_wait_end	I2C 等待数据传输结束
i2c_wait_flag	I2C 等待标志
i2c_master_transmit	I2C 主机发送数据（轮询模式）

函数名	描述
i2c_master_receive	I2C 主机接收数据（轮询模式）
i2c_slave_transmit	I2C 从机发送数据（轮询模式）
i2c_slave_receive	I2C 从机接收数据（轮询模式）
i2c_master_transmit_int	I2C 主机发送数据（中断模式）
i2c_master_receive_int	I2C 主机接收数据（中断模式）
i2c_slave_transmit_int	I2C 从机发送数据（中断模式）
i2c_slave_receive_int	I2C 从机接收数据（中断模式）
i2c_master_transmit_dma	I2C 主机发送数据（DMA 模式）
i2c_master_receive_dma	I2C 主机接收数据（DMA 模式）
i2c_slave_transmit_dma	I2C 从机发送数据（DMA 模式）
i2c_slave_receive_dma	I2C 从机接收数据（DMA 模式）
i2c_memory_write	I2C 写数据到 EEPROM（轮询模式）
i2c_memory_write_int	I2C 写数据到 EEPROM（中断模式）
i2c_memory_write_dma	I2C 写数据到 EEPROM（DMA 模式）
i2c_memory_read	I2C 从 EEPROM 读数据（轮询模式）
i2c_memory_read_int	I2C 从 EEPROM 读数据（中断模式）
i2c_memory_read_dma	I2C 从 EEPROM 读数据（DMA 模式）
i2c_evt_irq_handler	I2C 事件中断函数
i2c_err_irq_handler	I2C 错误中断函数
i2c_dma_tx_irq_handler	I2C DMA 发送中断函数
i2c_dma_rx_irq_handler	I2C DMA 接收中断函数

5.12.1 函数 i2c_reset

下表描述了函数 i2c_reset

表 231. 函数 i2c_reset

项目	描述
函数名	i2c_reset
函数原型	void i2c_reset(i2c_type *i2c_x)
功能描述	通过 CRM（时钟和复位管理）复位 I2C 外设，把 I2C 所有寄存器复位成初始值
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输出参数	无
返回值	无
先决条件	无
被调用函数	void crm_periph_reset(crm_periph_reset_type value, confirm_state new_state);

示例

```
i2c_reset(I2C1);
```

5.12.2 函数 i2c_software_reset

下表描述了函数 i2c_software_reset

表 232. 函数 i2c_software_reset

项目	描述
函数名	i2c_software_reset
函数原型	void i2c_software_reset(i2c_type *i2c_x, confirm_state new_state);
功能描述	通过 I2C 外设的内部软复位，复位 I2C 外设，实际效果和 i2c_reset(i2c_type *i2c_x)一样
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：软件复位状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_software_reset(I2C1, TRUE);
i2c_software_reset(I2C1, FALSE);
```

5.12.3 函数 i2c_init

下表描述了函数 i2c_init

表 233. 函数 i2c_init

项目	描述
函数名	i2c_init
函数原型	void i2c_init(i2c_type *i2c_x, i2c_fsmode_duty_cycle_type duty, uint32_t speed);
功能描述	设置 I2C 总线速度，以及快速模式下的占空比
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	duty：快速模式下 SCL 总线占空比 参阅章节：duty 查阅更多该参数允许取值范围
输入参数 3	speed：总线速度，单位 Hz
输出参数	无
返回值	无
先决条件	无
被调用函数	无

duty

快速模式（总线速度≥400kHz）下 SCL 总线占空比

I2C_FSMODE_DUTY_2_1： 快速模式下 SCL 总线占空比为 2：1

I2C_FSMODE_DUTY_16_9：快速模式下 SCL 总线占空比为 16：9

示例

```
i2c_init(I2C1, I2C_FSMODE_DUTY_2_1, 100000);
```

5.12.4 函数 i2c_own_address1_set

下表描述了函数 i2c_own_address1_set

表 234. 函数 i2c_own_address1_set

项目	描述
函数名	i2c_own_address1_set
函数原型	void i2c_own_address1_set(i2c_type *i2c_x, i2c_address_mode_type mode, uint16_t address);
功能描述	设置本机地址 1
输入参数 1	mode：本机地址 1 地址模式 参阅章节：mode 查阅更多该参数允许取值范围
输入参数 2	address：本机地址 1
输出参数	无
返回值	无
先决条件	无
被调用函数	无

mode

本机地址 1 地址模式

I2C_ADDRESS_MODE_7BIT： 7 位地址模式

I2C_ADDRESS_MODE_10BIT： 10 位地址模式

示例

```
i2c_own_address1_set(I2C1, I2C_ADDRESS_MODE_7BIT, 0xA0);
```

5.12.5 函数 i2c_own_address2_set

下表描述了函数 i2c_own_address2_set

表 235. 函数 i2c_own_address2_set

项目	描述
函数名	i2c_own_address2_set
函数原型	void i2c_own_address2_set(i2c_type *i2c_x, uint8_t address);
功能描述	设置本机地址 2，只有在本机地址 2 使能后，此地址才有效。需要注意的是该地址只支持 7 位地址，不支持 10 位地址
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	address：本机地址 2
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_own_address2_set(I2C1, 0xB0);
```

5.12.6 函数 i2c_own_address2_enable

下表描述了函数 i2c_own_address2_enable

表 236. 函数 i2c_own_address2_enable

项目	描述
函数名	i2c_own_address2_enable

项目	描述
函数原型	void i2c_own_address2_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	本机地址 2 使能，只有在本机地址 2 使能了之后本机地址 2 才有效，需要和 i2c_own_address2_set 配合使用
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：地址 2 使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_own_address2_enable(I2C1, TRUE);
```

5.12.7 函数 i2c_smbus_enable

下表描述了函数 i2c_smbus_enable

表 237. 函数 i2c_smbus_enable

项目	描述
函数名	i2c_smbus_enable
函数原型	void i2c_smbus_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	SMBus 模式使能，上电复位后默认是 I2C 模式
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：SMBus 模式使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_smbus_enable(I2C1, TRUE);
```

5.12.8 函数 i2c_enable

下表描述了函数 i2c_enable

表 238. 函数 i2c_enable

项目	描述
函数名	i2c_enable
函数原型	void i2c_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	I2C 外设使能
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2

项目	描述
输入参数 2	new_state：I2C 外设使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_enable(I2C1, TRUE);
```

5.12.9 函数 i2c_fast_mode_duty_set

下表描述了函数 i2c_fast_mode_duty_set

表 239. 函数 i2c_fast_mode_duty_set

项目	描述
函数名	i2c_fast_mode_duty_set
函数原型	void i2c_fast_mode_duty_set(i2c_type *i2c_x, i2c_fsmode_duty_cycle_type duty);
功能描述	设置快速模式下的 SCL 低电平与高电平宽度比值，该函数功能和初始化函数 void i2c_init(i2c_type *i2c_x, i2c_fsmode_duty_cycle_type duty, uint32_t speed)里的参数 duty 功能一样
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	duty：快速模式下 SCL 总线占空比 参阅章节：duty 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

duty

快速模式（总线速度≥400kHz）下 SCL 总线占空比

I2C_FSMODE_DUTY_2_1： 快速模式下 SCL 总线占空比为 2：1

I2C_FSMODE_DUTY_16_9：快速模式下 SCL 总线占空比为 16：9

示例

```
i2c_fast_mode_duty_set(I2C1, I2C_FSMODE_DUTY_2_1);
```

5.12.10 函数 i2c_clock_stretch_enable

下表描述了函数 i2c_clock_stretch_enable

表 240. 函数 i2c_clock_stretch_enable

项目	描述
函数名	i2c_clock_stretch_enable
函数原型	void i2c_clock_stretch_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	时钟延展模式使能，该函数用于从机，对主机无效，在大多数应用场景下，建议开启时钟延展，因为这样可以避免从机可能由于处理速度太慢导致数据来不及接收或

项目	描述
	发送而丢失数据，使用时需注意从机使用此功能的前提是主机要支持时钟延展，例如一些主机是用 IO 模拟的，那么一般是不支持这个特性的
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：时钟延展模式使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_clock_stretch_enable(I2C1, TRUE);
```

5.12.11 函数 i2c_ack_enable

下表描述了函数 i2c_ack_enable

表 241. 函数 i2c_ack_enable

项目	描述
函数名	i2c_ack_enable
函数原型	void i2c_ack_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	设置 ACK 和 NACK 的响应，该函数用于主机和从机控制每一个字节的 ACK 或 NACK，关于 I2C 通讯协议上的 ACK 响应可以去看 I2C 协议或者是 AT32 参考手册
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：ACK 响应状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_ack_enable(I2C1, TRUE);
```

5.12.12 函数 i2c_master_receive_ack_set

下表描述了函数 i2c_master_receive_ack_set

表 242. 函数 i2c_master_receive_ack_set

项目	描述
函数名	i2c_master_receive_ack_set
函数原型	void i2c_master_receive_ack_set(i2c_type *i2c_x, i2c_master_ack_type pos)
功能描述	主机接收模式应答控制，在主机接收模式下，用于设置函数 void i2c_ack_enable(i2c_type *i2c_x, confirm_state new_state)的生效位置。该函数的作用主要是为了在主机接收模式下接收两个字节时，能够正确的回复 NACK

项目	描述
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	pos：ACKEN 生效位置 参阅章节：pos 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

pos

ACKEN 生效位置

I2C_MASTER_ACK_CURRENT： ACKEN 位效果作用于当前传字节

I2C_MASTER_ACK_NEXT： ACKEN 位效果作用于第二个传输字节

示例

```
i2c_master_receive_ack_set(I2C1, TRUE);
```

5.12.13 函数 i2c_pec_position_set

下表描述了函数 i2c_pec_position_set

表 243. 函数 i2c_pec_position_set

项目	描述
函数名	i2c_pec_position_set
函数原型	void i2c_pec_position_set(i2c_type *i2c_x, i2c_pec_position_type pos);
功能描述	在 SMBus 模式并且在主机接收模式下，用于设置 PEC 的位置，该函数的作用主要是为了在主机接收模式下接收两个字节时，能够正确的接收 PEC 并回复 NACK
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	pos：PEC 位置 参阅章节：pos 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

pos

ACKEN 生效位置

I2C_PEC_POSITION_CURRENT： 当前字节是 PEC

I2C_PEC_POSITION_NEXT： 下一个字节是 PEC

示例

```
i2c_pec_position_set(I2C1, I2C_PEC_POSITION_CURRENT);
```

5.12.14 函数 i2c_general_call_enable

下表描述了函数 i2c_general_call_enable

表 244. 函数 i2c_general_call_enable

项目	描述
函数名	i2c_general_call_enable
函数原型	void i2c_general_call_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	广播地址使能，使能了后会响应广播地址 0x00
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：广播地址使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_general_call_enable(I2C1, TRUE);
```

5.12.15 函数 i2c_arp_mode_enable

下表描述了函数 i2c_arp_mode_enable

表 245. 函数 i2c_arp_mode_enable

项目	描述
函数名	i2c_arp_mode_enable
函数原型	void i2c_arp_mode_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	SMBus ARP 地址使能，使能了后 如果是 SMBus 主机：响应主机地址 0001000x 如果是 SMBus 设备：响应设备默认地址 0001100x 有关 ARP 协议的使用请参考 SMBUS 协议
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：ARP 地址使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_arp_mode_enable(I2C1, TRUE);
```

5.12.16 函数 i2c_smbus_mode_set

下表描述了函数 i2c_smbus_mode_set

表 246. 函数 i2c_smbus_mode_set

项目	描述
函数名	i2c_smbus_mode_set

项目	描述
函数原型	void i2c_smbus_mode_set(i2c_type *i2c_x, i2c_smbus_mode_set_type mode);
功能描述	SMBus 设备模式选择，可以选择 SMBus 主机或者 SMBus 设备
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	mode：SMBus 设备模式 参阅章节：mode 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

mode

SMBus 设备模式

I2C_SMBUS_MODE_DEVICE： SMBus 设备

I2C_SMBUS_MODE_HOST： SMBus 主机

示例

```
i2c_smbus_mode_set(I2C1, I2C_SMBUS_MODE_HOST);
```

5.12.17 函数 i2c_smbus_alert_set

下表描述了函数 i2c_smbus_alert_set

表 247. 函数 i2c_smbus_alert_set

项目	描述
函数名	i2c_smbus_alert_set
函数原型	void i2c_smbus_alert_set(i2c_type *i2c_x, i2c_smbus_alert_set_type level);
功能描述	SMBus 提醒引脚电平设置，可以将提醒引脚设置成高电平或低电平
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	level：SMBus 提醒引脚电平 参阅章节：level 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

level

SMBus 提醒引脚电平

I2C_SMBUS_ALERT_LOW：SMBus 提醒引脚输出低电平

I2C_SMBUS_ALERT_HIGH：SMBus 提醒引脚输出高电平

示例

```
i2c_smbus_alert_set(I2C1, I2C_SMBUS_ALERT_LOW);
```

5.12.18 函数 i2c_pec_transmit_enable

下表描述了函数 i2c_pec_transmit_enable

表 248. 函数 i2c_pec_transmit_enable

项目	描述
函数名	i2c_pec_transmit_enable
函数原型	void i2c_pec_transmit_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	PEC 传输使能，发送/接收 PEC，当调用此函数后，PEC 将会被立即发送或接收
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：PEC 传输使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_pec_transmit_enable(I2C1, TRUE);
```

5.12.19 函数 i2c_pec_calculate_enable

下表描述了函数 i2c_pec_calculate_enable

表 249. 函数 i2c_pec_calculate_enable

项目	描述
函数名	i2c_pec_calculate_enable
函数原型	void i2c_pec_calculate_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	使能 PEC 计算
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：PEC 计算使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_pec_calculate_enable(I2C1, TRUE);
```

5.12.20 函数 i2c_pec_value_get

下表描述了函数 i2c_pec_value_get

表 250. 函数 i2c_pec_value_get

项目	描述
函数名	i2c_pec_value_get
函数原型	uint8_t i2c_pec_value_get(i2c_type *i2c_x);
功能描述	获取当前 PEC 值
输入参数 1	i2c_x：所选择的 I2C 外设

项目	描述
	该参数可以选取自其中之一：I2C1，I2C2
输出参数	uint8_t：当前 PEC 值
返回值	无
先决条件	无
被调用函数	无

示例

```
Pec_value = i2c_pec_value_get(I2C1);
```

5.12.21 函数 i2c_dma_end_transfer_set

下表描述了函数 i2c_dma_end_transfer_set

表 251. 函数 i2c_dma_end_transfer_set

项目	描述
函数名	i2c_dma_end_transfer_set
函数原型	void i2c_dma_end_transfer_set(i2c_type *i2c_x, confirm_state new_state);
功能描述	DMA 传输结束指示，指示当前传输是否是最后一笔数据
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：是否是最后一笔数据 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_dma_end_transfer_set(I2C1, TRUE);
```

5.12.22 函数 i2c_dma_enable

下表描述了函数 i2c_dma_enable

表 252. 函数 i2c_dma_enable

项目	描述
函数名	i2c_dma_enable
函数原型	void i2c_dma_enable(i2c_type *i2c_x, confirm_state new_state);
功能描述	DMA 传输使能
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	new_state：DMA 使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_dma_enable(I2C1, TRUE);
```

5.12.23 函数 i2c_interrupt_enable

下表描述了函数 i2c_interrupt_enable

表 253. 函数 i2c_interrupt_enable

项目	描述
函数名	i2c_interrupt_enable
函数原型	void i2c_interrupt_enable(i2c_type *i2c_x, uint16_t source, confirm_state new_state)
功能描述	I2C 中断使能
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	source：中断源 参阅章节：source 查阅更多该参数允许取值范围
输入参数 3	new_state：中断使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

source

中断源

I2C_DATA_INT： 数据中断

I2C_EV_INT： 事件中断

I2C_ERR_INT： 错误中断

示例

```
i2c_interrupt_enable(I2C1, I2C_DATA_INT, TRUE);
```

5.12.24 函数 i2c_start_generate

下表描述了函数 i2c_start_generate

表 254. 函数 i2c_start_generate

项目	描述
函数名	i2c_start_generate
函数原型	void i2c_start_generate(i2c_type *i2c_x);
功能描述	产生起始条件（主机使用）
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_start_generate(I2C1);
```

5.12.25 函数 i2c_stop_generate

下表描述了函数 i2c_stop_generate

表 255. 函数 i2c_stop_generate

项目	描述
函数名	i2c_stop_generate
函数原型	void i2c_stop_generate(i2c_type *i2c_x);
功能描述	产生停止条件
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_stop_generate(I2C1);
```

5.12.26 函数 i2c_7bit_address_send

下表描述了函数 i2c_7bit_address_send

表 256. 函数 i2c_7bit_address_send

项目	描述
函数名	i2c_7bit_address_send
函数原型	void i2c_7bit_address_send(i2c_type *i2c_x, uint8_t address, i2c_direction_type direction);
功能描述	发送 7 位从机地址（主机使用）
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	address：从机地址
输入参数 3	direction：数据传输方向 参阅章节：direction 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

direction

数据传输方向

I2C_DIRECTION_TRANSMIT： 主机发送

I2C_DIRECTION_RECEIVE： 主机接收

示例

```
i2c_7bit_address_send(I2C1, 0xB0, I2C_DIRECTION_TRANSMIT);
```

5.12.27 函数 i2c_data_send

下表描述了函数 i2c_data_send

表 257. 函数 i2c_data_send

项目	描述
函数名	i2c_data_send
函数原型	void i2c_data_send(i2c_type *i2c_x, uint8_t data);
功能描述	发送数据
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	data：传输数据
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
i2c_data_send(I2C1, 0x55);
```

5.12.28 函数 i2c_data_receive

下表描述了函数 i2c_data_receive

表 258. 函数 i2c_data_receive

项目	描述
函数名	i2c_data_receive
函数原型	uint8_t i2c_data_receive(i2c_type *i2c_x);
功能描述	接收数据
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输出参数	uint8_t：接收数据
返回值	无
先决条件	无
被调用函数	无

示例

```
data_value = i2c_data_receive(I2C1);
```

5.12.29 函数 i2c_flag_get

下表描述了函数 i2c_flag_get

表 259. 函数 i2c_flag_get

项目	描述
函数名	i2c_flag_get
函数原型	flag_status i2c_flag_get(i2c_type *i2c_x, uint32_t flag);
功能描述	获取标志位状态

项目	描述
输入参数 1	i2c_x：所选择的 I2C 外设 该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	flag：需要获取状态的标志选择 该参数详细描述见 flag
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET、RESET
先决条件	无
被调用函数	无

flag

用于选择需要获取状态的标志，其可选参数罗列如下

I2C_STARTF_FLAG：起始条件产生完成标志
I2C_ADDR7F_FLAG：0~7 位地址匹配标志
I2C_TDC_FLAG：数据传输完成标志
I2C_ADDRHF_FLAG：主机 9~8 位地址头匹配标志
I2C_STOPF_FLAG：停止条件产生完成标志
I2C_RDBF_FLAG：接收数据缓冲器满标志
I2C_TDBE_FLAG：发送缓冲器空标志
I2C_BUSERR_FLAG：总线错误标志
I2C_ARLOST_FLAG：仲裁丢失标志
I2C_ACKFAIL_FLAG：应答失败标志
I2C_OUF_FLAG：溢出标志
I2C_PECERR_FLAG：PEC 接收错误标志
I2C_TMOUT_FLAG：SMBus 超时标志
I2C_ALERTF_FLAG：SMBus 提醒标志
I2C_TRMODE_FLAG：传输模式
I2C_BUSYF_FLAG：总线忙标志
I2C_DIRF_FLAG：传输方向标志
I2C_GCADDRF_FLAG：广播地址接收标志
I2C_DEVADDRF_FLAG：SMBus 设备地址接收标志
I2C_HOSTADDRF_FLAG：SMBus 主机地址接收标志
I2C_ADDR2_FLAG：接收到地址 2 标志

示例

```
i2c_flag_get(I2C1, I2C_STARTF_FLAG);
```

5.12.30 函数 i2c_flag_clear

下表描述了函数 i2c_flag_clear

表 260. 函数 i2c_flag_clear

项目	描述
函数名	i2c_flag_clear
函数原型	void i2c_flag_clear(i2c_type *i2c_x, uint32_t flag);
功能描述	清除标志位
输入参数 1	i2c_x：所选择的 I2C 外设

项目	描述
	该参数可以选取自其中之一：I2C1，I2C2
输入参数 2	flag：待清除的标志选择 该参数详细描述见 flag
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flag

用于选择需要清除状态的标志，其可选参数罗列如下

I2C_BUSERR_FLAG： 总线错误标志
I2C_ARLOST_FLAG： 仲裁丢失标志
I2C_ACKFAIL_FLAG： 应答失败标志
I2C_OUF_FLAG： 溢出标志
I2C_PECERR_FLAG： PEC 接收错误标志
I2C_TMOUT_FLAG： SMBus 超时标志
I2C_ALERTF_FLAG： SMBus 提醒标志
I2C_ADDR7F_FLAG： 0~7 位地址匹配标志
I2C_STOPF_FLAG： 停止条件产生完成标志

示例

```
i2c_flag_clear(I2C1, I2C_ACKFAIL_FLAG);
```

5.12.31 函数 i2c_config

下表描述了函数 i2c_config

表 261. 函数 i2c_config

项目	描述
函数名	i2c_config
函数原型	void i2c_config(i2c_handle_type* hi2c);
功能描述	I2C 初始化函数，用于初始化 I2C，函数内部调用 i2c_lowlevel_init()函数，实现 I2C 外设、GPIO、DMA、中断等初始化
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输出参数	无
返回值	无
先决条件	无
被调用函数	void i2c_lowlevel_init(i2c_handle_type* hi2c);

i2c_handle_type* hi2c

i2c_handle_type 在 i2c_application.h 中

```
typedef struct
{
    i2c_type                *i2cx;
    uint8_t                 *pbuff;
    __IO uint16_t           pcount;
```

```
    __IO uint32_t               mode;
    __IO uint32_t               timeout;
    __IO uint32_t               status;
    __IO i2c_status_type        error_code;
    dma_channel_type            *dma_tx_channel;
    dma_channel_type            *dma_rx_channel;
    dma_init_type               dma_init_struct;
}i2c_handle_type;
```

i2cx

所选择的 I2C 外设，该参数可以选取自其中之一：I2C1，I2C2

pbuff

发送/接收数据的数组

pcount

发送/接收数据的个数

mode

I2C 通讯模式，内部的状态机使用，用户无需关心

timeout

通讯超时时间

status

传输状态，内部的状态机使用，用户无需关心

error_code

枚举 i2c_status_type 类型错误代码，当通讯发生错误后，此变量记录错误代码

I2C_OK：没有错误,通讯正常

I2C_ERR_STEP_1：步骤 1 错误

I2C_ERR_STEP_2：步骤 2 错误

I2C_ERR_STEP_3：步骤 3 错误

I2C_ERR_STEP_4：步骤 4 错误

I2C_ERR_STEP_5：步骤 5 错误

I2C_ERR_STEP_6：步骤 6 错误

I2C_ERR_STEP_7：步骤 7 错误

I2C_ERR_STEP_8：步骤 8 错误

I2C_ERR_STEP_9：步骤 9 错误

I2C_ERR_STEP_10：步骤 10 错误

I2C_ERR_STEP_11：步骤 11 错误

I2C_ERR_STEP_12：步骤 12 错误

I2C_ERR_START：START 条件发送错误

I2C_ERR_ADDR10：10 位地址头（bit9~8）发送错误

I2C_ERR_ADDR：地址发送错误

I2C_ERR_STOP：STOP 条件发送错误

I2C_ERR_ACKFAIL：应答错误

I2C_ERR_TIMEOUT：超时错误

I2C_ERR_INTERRUPT：有错误事件发生，并进入了错误中断

dma_tx_channel

I2C 发送 DMA 通道

dma_rx_channel

I2C 接收 DMA 通道

dma_init_struct

DMA 初始化结构体

示例

```
i2c_handle_type hi2c;
hi2c.i2cx = I2C1;
i2c_config(&hi2c);
```

5.12.32 函数 i2c_lowlevel_init

下表描述了函数 i2c_lowlevel_init

表 262. 函数 i2c_lowlevel_init

项目	描述
函数名	i2c_lowlevel_init
函数原型	void i2c_lowlevel_init(i2c_handle_type* hi2c);
功能描述	I2C 底层初始化回调函数，在函数 i2c_config 内部调用，用于实现初始化 I2C 外设、GPIO、DMA、中断等初始化，需要用户在函数内部实现 I2C 初始化过程
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
void i2c_lowlevel_init(i2c_handle_type* hi2c)
{
  if(hi2c->i2cx == I2C1)
  {
    实现 I2C1 的初始化
  }
  else if(hi2c->i2cx == I2C2)
  {
    实现 I2C2 的初始化
  }
}
```

5.12.33 函数 i2c_wait_end

下表描述了函数 i2c_wait_end

表 263. 函数 i2c_wait_end

项目	描述
函数名	i2c_wait_end
函数原型	i2c_status_type i2c_wait_end(i2c_handle_type* hi2c, uint32_t timeout);
功能描述	等待通讯结束，该函数用于 DMA 以及中断传输模式，因为这两种传输模式函数是非阻塞的，所以可以使用这个函数来等待传输结束

项目	描述
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
if (i2c_master_transmit_dma(&hi2c, 0xB0, tx_buf, 8, 0xFFFFFFFF) != I2C_OK)
{
  error_handler(i2c_status);
}

/* 等待通讯结束 */
if(i2c_wait_end(&hi2c, 0xFFFFFFFF) != I2C_OK)
{
  error_handler(i2c_status);
}
```

5.12.34 函数 i2c_wait_flag

下表描述了函数 i2c_wait_flag

表 264. 函数 i2c_wait_flag

项目	描述
函数名	i2c_wait_flag
函数原型	i2c_status_type i2c_wait_flag(i2c_handle_type* hi2c, uint32_t flag, uint32_t event_check, uint32_t timeout)
功能描述	等待标志置起或者复位 只有等待 BUSYF 标志是等待标志复位，其余标志均是等待标志置起
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 参阅章节：0 查阅更多该参数允许取值范围
输入参数 2	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 3	event_check：等待标志的同时检测该事件是否发生 参阅章节：event_check 查阅更多该参数允许取值范围
输入参数 4	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

flag

需要等待的标志

I2C_STARTF_FLAG：　起始条件产生完成标志
I2C_ADDR7F_FLAG：　0~7 位地址匹配标志
I2C_TDC_FLAG：　数据传输完成标志
I2C_ADDRHF_FLAG：　主机 9~8 位地址头匹配标志
I2C_STOPF_FLAG：　停止条件产生完成标志
I2C_RDBF_FLAG：　接收数据缓冲器满标志
I2C_TDBE_FLAG：　发送缓冲器空标志
I2C_BUSERR_FLAG：　总线错误标志
I2C_ARLOST_FLAG：　仲裁丢失标志
I2C_ACKFAIL_FLAG：　应答失败标志
I2C_OUF_FLAG：　溢出标志
I2C_PECERR_FLAG：　PEC 接收错误标志
I2C_TMOUT_FLAG：　SMBus 超时标志
I2C_ALERTF_FLAG：　SMBus 提醒标志
I2C_TRMODE_FLAG：　传输模式
I2C_BUSYF_FLAG：　总线忙标志
I2C_DIRF_FLAG：　传输方向标志
I2C_GCADDRF_FLAG：　广播地址接收标志
I2C_DEVADDRF_FLAG：　SMBus 设备地址接收标志
I2C_HOSTADDRF_FLAG：　SMBus 主机地址接收标志
I2C_ADDR2_FLAG：　接收到地址 2 标志

event_check

等待标志的同时检测该事件是否发生

I2C_EVENT_CHECK_NONE：　不检查事件
I2C_EVENT_CHECK_ACKFAIL：　检查 ACKFAIL 事件
I2C_EVENT_CHECK_STOP：　检查 STOP 事件

示例

```
i2c_wait_flag(&hi2c, I2C_BUSYF_FLAG, I2C_EVENT_CHECK_NONE, 0xFFFFFFFF);
```

5.12.35　函数 i2c_master_transmit

下表描述了函数 i2c_master_transmit

表 265. 函数 i2c_master_transmit

项目	描述
函数名	i2c_master_transmit
函数原型	i2c_status_type i2c_master_transmit(i2c_handle_type* hi2c, uint16_t address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	主机发送数据（轮询方式），该函数是阻塞方式，执行完成后，I2C 也传输完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	address：从机地址
输入参数 3	pdata：待发送数据的数组地址
输入参数 4	size：数据发送个数
输入参数 5	timeout：等待超时时间
输出参数	无

项目	描述
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_master_transmit(&hi2c, 0xB0, tx_buf, 8, 0xFFFFFFFF);
```

5.12.36 函数 i2c_master_receive

下表描述了函数 i2c_master_receive

表 266. 函数 i2c_master_receive

项目	描述
函数名	i2c_master_receive
函数原型	i2c_status_type i2c_master_receive(i2c_handle_type* hi2c, uint16_t address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	主机接收数据（轮询方式），该函数是阻塞方式，执行完成后，I2C 也传输完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	address：从机地址
输入参数 3	pdata：接收数据的数组地址
输入参数 4	size：数据接收个数
输入参数 5	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_master_receive(&hi2c, 0xB0, rx_buf, 8, 0xFFFFFFFF);
```

5.12.37 函数 i2c_slave_transmit

下表描述了函数 i2c_slave_transmit

表 267. 函数 i2c_slave_transmit

项目	描述
函数名	i2c_slave_transmit
函数原型	i2c_status_type i2c_slave_transmit(i2c_handle_type* hi2c, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	从机发送数据（轮询方式），该函数是阻塞方式，执行完成后，I2C 也传输完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	pdata：待发送数据的数组地址
输入参数 3	size：数据发送个数

项目	描述
输入参数 4	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节：0 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_slave_transmit(&hi2c, tx_buf, 8, 0xFFFFFFFF);
```

5.12.38 函数 i2c_slave_receive

下表描述了函数 i2c_slave_receive

表 268. 函数 i2c_slave_receive

项目	描述
函数名	i2c_slave_receive
函数原型	i2c_status_type i2c_slave_receive(i2c_handle_type* hi2c, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	从机接收数据（轮询方式），该函数是阻塞方式，执行完成后，I2C 也传输完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	pdata：接收数据的数组地址
输入参数 3	size：数据接收个数
输入参数 4	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_slave_receive(&hi2c, rx_buf, 8, 0xFFFFFFFF);
```

5.12.39 函数 i2c_master_transmit_int

下表描述了函数 i2c_master_transmit_int

表 269. 函数 i2c_master_transmit_int

项目	描述
函数名	i2c_master_transmit_int
函数原型	i2c_status_type i2c_master_transmit_int(i2c_handle_type* hi2c, uint16_t address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	主机发送数据（中断方式）该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*

项目	描述
输入参数 2	address：从机地址
输入参数 3	pdata：待发送数据的数组地址
输入参数 4	size：数据发送个数
输入参数 5	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_master_transmit_int(&hi2c, 0xB0, tx_buf, 8, 0xFFFFFFFF);
```

5.12.40 函数 i2c_master_receive_int

下表描述了函数 i2c_master_receive_int

表 270. 函数 i2c_master_receive_int

项目	描述
函数名	i2c_master_receive_int
函数原型	i2c_status_type i2c_master_receive_int(i2c_handle_type* hi2c, uint16_t address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	主机接收数据（中断方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	address：从机地址
输入参数 3	pdata：接收数据的数组地址
输入参数 4	size：数据接收个数
输入参数 5	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_master_receive_int(&hi2c, 0xB0, rx_buf, 8, 0xFFFFFFFF);
```

5.12.41 函数 i2c_slave_transmit_int

下表描述了函数 i2c_slave_transmit_int

表 271. 函数 i2c_slave_transmit_int

项目	描述
函数名	i2c_slave_transmit_int
函数原型	i2c_status_type i2c_slave_transmit_int(i2c_handle_type* hi2c, uint8_t* pdata,

项目	描述
	uint16_t size, uint32_t timeout);
功能描述	从机发送数据（中断方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	pdata：待发送数据的数组地址
输入参数 3	size：数据发送个数
输入参数 4	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_slave_transmit_int(&hi2c, tx_buf, 8, 0xFFFFFFFF);
```

5.12.42 函数 i2c_slave_receive_int

下表描述了函数 i2c_slave_receive_int

表 272. 函数 i2c_slave_receive_int

项目	描述
函数名	i2c_slave_receive_int
函数原型	i2c_status_type i2c_slave_receive_int(i2c_handle_type* hi2c, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	从机接收数据（中断方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	pdata：接收数据的数组地址
输入参数 3	size：数据接收个数
输入参数 4	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_slave_receive_int(&hi2c, rx_buf, 8, 0xFFFFFFFF);
```

5.12.43 函数 i2c_master_transmit_dma

下表描述了函数 i2c_master_transmit_dma

表 273. 函数 i2c_master_transmit_dma

项目	描述
函数名	i2c_master_transmit_dma
函数原型	i2c_status_type i2c_master_transmit_dma(i2c_handle_type* hi2c, uint16_t address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	主机发送数据（DMA 方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	address：从机地址
输入参数 3	pdata：待发送数据的数组地址
输入参数 4	size：数据发送个数
输入参数 5	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_master_transmit_dma(&hi2c, 0xB0, tx_buf, 8, 0xFFFFFFFF);
```

5.12.44 函数 i2c_master_receive_dma

下表描述了函数 i2c_master_receive_dma

表 274. 函数 i2c_master_receive_dma

项目	描述
函数名	i2c_master_receive_dma
函数原型	i2c_status_type i2c_master_receive_dma(i2c_handle_type* hi2c, uint16_t address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	主机接收数据（DMA 方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	address：从机地址
输入参数 3	pdata：接收数据的数组地址
输入参数 4	size：数据接收个数
输入参数 5	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_master_receive_dma(&hi2c, 0xB0, rx_buf, 8, 0xFFFFFFFF);
```

5.12.45 函数 i2c_slave_transmit_dma

下表描述了函数 i2c_slave_transmit_dma

表 275. 函数 i2c_slave_transmit_dma

项目	描述
函数名	i2c_slave_transmit_dma
函数原型	i2c_status_type i2c_slave_transmit_dma(i2c_handle_type* hi2c, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	从机发送数据（DMA 方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	pdata：待发送数据的数组地址
输入参数 3	size：数据发送个数
输入参数 4	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_slave_transmit_dma(&hi2c, tx_buf, 8, 0xFFFFFFFF);
```

5.12.46 函数 i2c_slave_receive_dma

下表描述了函数 i2c_slave_receive_dma

表 276. 函数 i2c_slave_receive_dma

项目	描述
函数名	i2c_slave_receive_dma
函数原型	i2c_status_type i2c_slave_receive_dma(i2c_handle_type* hi2c, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	从机接收数据（DMA 方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	pdata：接收数据的数组地址
输入参数 3	size：数据接收个数
输入参数 4	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

示例

```
i2c_slave_receive_dma(&hi2c, rx_buf, 8, 0xFFFFFFFF);
```

5.12.47 函数 i2c_memory_write

下表描述了函数 i2c_memory_write

表 277. 函数 i2c_memory_write

项目	描述
函数名	i2c_memory_write
函数原型	i2c_status_type i2c_memory_write(i2c_handle_type* hi2c, i2c_mem_address_width_type mem_address_width, uint16_t address, uint16_t mem_address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	向 EEPROM 写数据（轮询方式），该函数是阻塞方式，执行完成后，I2C 也传输完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	mem_address_width：EEPROM 存储地址宽度 参阅章节：mem_address_width 查阅更多该参数允许取值范围
输入参数 3	address：EEPROM 地址
输入参数 4	mem_address：EEPROM 数据存储地址
输入参数 5	pdata：待发送数据的数组地址
输入参数 6	size：数据发送个数
输入参数 7	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

mem_address_width

EEPROM 存储地址宽度

I2C_MEM_ADDR_WIDIH_8： 8 位地址宽度

I2C_MEM_ADDR_WIDIH_16： 16 位地址宽度

示例

```
i2c_memory_write(&hi2c, 0xA0, 0x05, tx_buf, 8, 0xFFFFFFFF);
```

5.12.48 函数 i2c_memory_write_int

下表描述了函数 i2c_memory_write_int

表 278. 函数 i2c_memory_write_int

项目	描述
函数名	i2c_memory_write_int
函数原型	i2c_status_type i2c_memory_write_int(i2c_handle_type* hi2c, i2c_mem_address_width_type mem_address_width, uint16_t address, uint16_t mem_address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	向 EEPROM 写数据（中断方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成

项目	描述
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	mem_address_width：EEPROM 存储地址宽度 参阅章节：mem_address_width 查阅更多该参数允许取值范围
输入参数 3	address：EEPROM 地址
输入参数 4	mem_address：EEPROM 数据存储地址
输入参数 5	pdata：待发送数据的数组地址
输入参数 6	size：数据发送个数
输入参数 7	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

mem_address_width

EEPROM 存储地址宽度

I2C_MEM_ADDR_WIDIH_8： 8 位地址宽度

I2C_MEM_ADDR_WIDIH_16： 16 位地址宽度

示例

```
i2c_memory_write_int(&hi2c, 0xA0, 0x05, tx_buf, 8, 0xFFFFFFFF);
```

5.12.49 函数 i2c_memory_write_dma

下表描述了函数 i2c_memory_write_dma

表 279. 函数 i2c_memory_write_dma

项目	描述
函数名	i2c_memory_write_dma
函数原型	i2c_status_type i2c_memory_write_dma(i2c_handle_type* hi2c, i2c_mem_address_width_type mem_address_width, uint16_t address, uint16_t mem_address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	向 EEPROM 写数据（DMA 方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	mem_address_width：EEPROM 存储地址宽度 参阅章节：mem_address_width 查阅更多该参数允许取值范围
输入参数 3	address：EEPROM 地址
输入参数 4	mem_address：EEPROM 数据存储地址
输入参数 5	pdata：待发送数据的数组地址
输入参数 6	size：数据发送个数
输入参数 7	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围

项目	描述
先决条件	无
被调用函数	无

mem_address_width

EEPROM 存储地址宽度

I2C_MEM_ADDR_WIDIH_8： 8 位地址宽度

I2C_MEM_ADDR_WIDIH_16： 16 位地址宽度

示例

```
i2c_memory_write_dma(&hi2c, 0xA0, 0x05, tx_buf, 8, 0xFFFFFFFF);
```

5.12.50 函数 i2c_memory_read

下表描述了函数 i2c_memory_read

表 280. 函数 i2c_memory_read

项目	描述
函数名	i2c_memory_read
函数原型	i2c_status_type i2c_memory_read(i2c_handle_type* hi2c, i2c_mem_address_width_type mem_address_width, uint16_t address, uint16_t mem_address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	从 EEPROM 读数据（轮询方式），该函数是阻塞方式，执行完成后，I2C 也传输完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	mem_address_width：EEPROM 存储地址宽度 参阅章节：mem_address_width 查阅更多该参数允许取值范围
输入参数 3	address：EEPROM 地址
输入参数 4	mem_address：EEPROM 数据存储地址
输入参数 5	pdata：读取数据的数组地址
输入参数 6	size：数据读取个数
输入参数 7	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

mem_address_width

EEPROM 存储地址宽度

I2C_MEM_ADDR_WIDIH_8： 8 位地址宽度

I2C_MEM_ADDR_WIDIH_16： 16 位地址宽度

示例

```
i2c_memory_read(&hi2c, 0xA0, 0x05, rx_buf, 8, 0xFFFFFFFF);
```

5.12.51 函数 i2c_memory_read_int

下表描述了函数 i2c_memory_read_int

表 281. 函数 i2c_memory_read_int

项目	描述
函数名	i2c_memory_read_int
函数原型	i2c_status_type i2c_memory_read_int(i2c_handle_type* hi2c, i2c_mem_address_width_type mem_address_width, uint16_t address, uint16_t mem_address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	从 EEPROM 读数据（中断方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	mem_address_width：EEPROM 存储地址宽度 参阅章节：mem_address_width 查阅更多该参数允许取值范围
输入参数 3	address：EEPROM 地址
输入参数 4	mem_address：EEPROM 数据存储地址
输入参数 5	pdata：读取数据的数组地址
输入参数 6	size：数据读取个数
输入参数 7	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

mem_address_width

EEPROM 存储地址宽度

I2C_MEM_ADDR_WIDIH_8： 8 位地址宽度

I2C_MEM_ADDR_WIDIH_16： 16 位地址宽度

示例

```
i2c_memory_read_int(&hi2c, 0xA0, 0x05, rx_buf, 8, 0xFFFFFFFF);
```

5.12.52 函数 i2c_memory_read_dma

下表描述了函数 i2c_memory_read_dma

表 282. 函数 i2c_memory_read_dma

项目	描述
函数名	i2c_memory_read_dma
函数原型	i2c_status_type i2c_memory_read_dma(i2c_handle_type* hi2c, i2c_mem_address_width_type mem_address_width, uint16_t address, uint16_t mem_address, uint8_t* pdata, uint16_t size, uint32_t timeout);
功能描述	从 EEPROM 读数据（DMA 方式），该函数是非阻塞方式，该函数执行完成后 I2C 通讯还未结束，可以通过调用函数 i2c_wait_end()等待通讯完成
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输入参数 2	mem_address_width：EEPROM 存储地址宽度 参阅章节：mem_address_width 查阅更多该参数允许取值范围
输入参数 3	address：EEPROM 地址

项目	描述
输入参数 4	mem_address：EEPROM 数据存储地址
输入参数 5	pdata：读取数据的数组地址
输入参数 6	size：数据读取个数
输入参数 7	timeout：等待超时时间
输出参数	无
返回值	i2c_status_type：错误代码 参阅章节 5.12.31 查阅更多该参数允许取值范围
先决条件	无
被调用函数	无

mem_address_width

EEPROM 存储地址宽度

I2C_MEM_ADDR_WIDIH_8： 8 位地址宽度

I2C_MEM_ADDR_WIDIH_16： 16 位地址宽度

示例

```
i2c_memory_read_dma(&hi2c, 0xA0, 0x05, rx_buf, 8, 0xFFFFFFFF);
```

5.12.53 函数 i2c_evt_irq_handler

下表描述了函数 i2c_evt_irq_handler

表 283. 函数 i2c_evt_irq_handler

项目	描述
函数名	i2c_evt_irq_handler
函数原型	void i2c_evt_irq_handler(i2c_handle_type* hi2c);
功能描述	事件中断函数，用于处理 I2C 事件中断
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
void I2C1_EVT_IRQHandler(void)
{
  i2c_evt_irq_handler(&hi2c);
}
```

5.12.54 函数 i2c_err_irq_handler

下表描述了函数 i2c_err_irq_handler

表 284. 函数 i2c_err_irq_handler

项目	描述
函数名	i2c_err_irq_handler
函数原型	void i2c_err_irq_handler(i2c_handle_type* hi2c);

项目	描述
功能描述	错误中断函数，用于处理 I2C 错误中断
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
void I2C1_ERR_IRQHandler(void)
{
  i2c_err_irq_handler(&hi2c);
}
```

5.12.55 函数 i2c_dma_tx_irq_handler

下表描述了函数 i2c_dma_tx_irq_handler

表 285. 函数 i2c_dma_tx_irq_handler

项目	描述
函数名	i2c_dma_tx_irq_handler
函数原型	void i2c_dma_tx_irq_handler(i2c_handle_type* hi2c);
功能描述	DMA 发送中断函数，用于处理 DMA 发送中断
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
void DMA1_Channel6_IRQHandler(void)
{
  i2c_dma_rx_irq_handler(&hi2c);
}
```

5.12.56 函数 i2c_dma_rx_irq_handler

下表描述了函数 i2c_dma_rx_irq_handler

表 286. 函数 i2c_dma_rx_irq_handler

项目	描述
函数名	i2c_dma_rx_irq_handler
函数原型	void i2c_dma_rx_irq_handler(i2c_handle_type* hi2c);
功能描述	DMA 接收中断函数，用于处理 DMA 接收中断
输入参数 1	hi2c：指向 i2c_handle_type 类型的结构体 该参数详细描述见 *i2c_handle_type*

项目	描述
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
void DMA1_Channel7_IRQHandler(void)
{
  i2c_dma_tx_irq_handler(&hi2c);
}
```

5.13 嵌套的向量式中断控制器（NVIC）

NVIC 寄存器结构 NVIC_Type，定义于文件“core_cm4.h”如下：

```
/**
  * @brief Structure type to access the Nested Vectored Interrupt Controller (NVIC).
  */
typedef struct
{
      ……
} NVIC_Type;
```

下表给出了 NVIC 寄存器总览：

表 287. PWC 寄存器对应表

寄存器	描述
iser	中断使能设置寄存器
icer	中断使能清除寄存器
ispr	中断挂起设置寄存器
icpr	中断挂起清除寄存器
iabr	中断激活位寄存器
ip	中断优先级寄存器
stir	软件触发中断寄存器

下表给出了 NVIC 库函数总览：

表 288. PWC 库函数总览

函数名	描述
nvic_system_reset	系统软件复位命令
nvic_irq_enable	NVIC 中断使能及优先级配置
nvic_irq_disable	NVIC 中断失能
nvic_priority_group_config	NVIC 中断优先级分组配置
nvic_vector_table_set	NVIC 中断向量表基地址及偏移地址设定
nvic_lowpower_mode_config	NVIC 低功耗模式相关配置

5.13.1 函数 nvic_system_reset

下表描述了函数 nvic_system_reset

表 289. 函数 nvic_system_reset

项目	描述
函数名	nvic_system_reset
函数原型	void nvic_system_reset(void)
功能描述	系统软件复位命令
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	NVIC_SystemReset()

示例

```
/* system reset */
nvic_system_reset();
```

5.13.2 函数 nvic_irq_enable

下表描述了函数 nvic_irq_enable

表 290. 函数 nvic_irq_enable

项目	描述
函数名	nvic_irq_enable
函数原型	void nvic_irq_enable(IRQn_Type irqn, uint32_t preempt_priority, uint32_t sub_priority)
功能描述	NVIC 中断使能及优先级配置
输入参数 1	irqn：中断向量选择 该参数详细描述见 *irqn*.
输入参数 2	preempt_priority：抢占优先级设定 该参数的数值不可超过 NVIC_PRIORITY_GROUP_x 定义的最大抢占优先级
输入参数 3	sub_priority：响应优先级设定 该参数的数值不可超过 NVIC_PRIORITY_GROUP_x 定义的最大响应优先级
输出参数	无
返回值	无
先决条件	无
被调用函数	NVIC_SetPriority() NVIC_EnableIRQ()

irqn

irqn 用于选择需要操作的中断向量，其可选参数罗列如下

WWDT_IRQn： 窗口定时器中断

PVM_IRQn： 连到 EXINT 的电源电压检测（PVM）中断

……

USBFS_MAPL_IRQn： USBFS 重映射低优先级中断

DMA2_Channel6_7_IRQn： DMA2 通道 6 和 DMA2 通道 7 全局中断

示例

```
/* enable nvic irq */
nvic_irq_enable(ADC1_2_IRQn, 0, 0);
```

5.13.3 函数 nvic_irq_disable

下表描述了函数 nvic_irq_disable

表 291. 函数 nvic_irq_disable

项目	描述
函数名	nvic_irq_disable
函数原型	void nvic_irq_disable(IRQn_Type irqn)
功能描述	NVIC 中断失能
输入参数	irqn：中断向量选择 该参数详细描述见 *irqn*.
输出参数	无
返回值	无
先决条件	无
被调用函数	NVIC_DisableIRQ()

示例

```
/* disable nvic irq */
nvic_irq_disable(ADC1_2_IRQn);
```

5.13.4 函数 nvic_priority_group_config

下表描述了函数 nvic_priority_group_config

表 292. 函数 nvic_priority_group_config

项目	描述
函数名	nvic_priority_group_config
函数原型	void nvic_priority_group_config(nvic_priority_group_type priority_group)
功能描述	NVIC 中断优先级分组配置
输入参数	priority_group：中断优先级分组选择 该参数可以选取 nvic_priority_group_type 内的任意一个枚举值.
输出参数	无
返回值	无
先决条件	无
被调用函数	NVIC_SetPriorityGrouping()

priority_group

priority_group 用于选择中断优先级分组，其可选参数罗列如下

NVIC_PRIORITY_GROUP_0：优先级组 0（0 位用于抢占优先级，4 位用于响应优先级）

NVIC_PRIORITY_GROUP_1：优先级组 1（1 位用于抢占优先级，3 位用于响应优先级）

NVIC_PRIORITY_GROUP_2：优先级组 2（2 位用于抢占优先级，2 位用于响应优先级）

NVIC_PRIORITY_GROUP_3：优先级组 3（3 位用于抢占优先级，1 位用于响应优先级）

NVIC_PRIORITY_GROUP_4：优先级组 4（4 位用于抢占优先级，0 位用于响应优先级）

示例

```
/* config nvic priority group */
nvic_priority_group_config(NVIC_PRIORITY_GROUP_4);
```

5.13.5 函数 nvic_vector_table_set

下表描述了函数 nvic_vector_table_set

表 293. 函数 nvic_vector_table_set

项目	描述
函数名	nvic_vector_table_set
函数原型	void nvic_vector_table_set(uint32_t base, uint32_t offset)
功能描述	NVIC 中断向量表基地址及偏移地址设定
输入参数 1	base：中断向量表基地址 该参数可选择设定基地址位于 RAM 或是 FLASH.
输入参数 2	offset：中断向量表偏移地址 该参数决定实际中断向量表起始地址，必须要设定为 0x200 的倍数
输出参数	无
返回值	无
先决条件	无
被调用函数	无

base

base 用于选择中断向量表的基地址，其可选参数罗列如下

NVIC_VECTTAB_RAM： 中断向量表基地址位于 RAM

NVIC_VECTTAB_FLASH： 中断向量表基地址位于 FLASH

示例

```
/* config vector table offset */
nvic_vector_table_set(NVIC_VECTTAB_FLASH, 0x4000);
```

5.13.6 函数 nvic_lowpower_mode_config

下表描述了函数 nvic_lowpower_mode_config

表 294. 函数 nvic_lowpower_mode_config

项目	描述
函数名	nvic_lowpower_mode_config
函数原型	void nvic_lowpower_mode_config(nvic_lowpower_mode_type lp_mode, confirm_state new_state)
功能描述	NVIC 低功耗模式相关配置
输入参数 1	lp_mode：选择需要配置的低功耗模式 该参数可以选取 nvic_lowpower_mode_type 内的任意一个枚举值.
输入参数 2	new_state：电池供电区域的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

lp_mode

lp_mode 用于选择需要配置的低功耗模式，其可选参数罗列如下

NVIC_LP_SEVONPEND： 当中断挂起时发送唤醒事件（此项通常与 WFE 组合使用）

NVIC_LP_SLEEPDEEP： 深度睡眠模式控制位（控制内核时钟的开关状态）

NVIC_LP_SLEEPONEXIT： 系统从最低优先级中断退出时，立即进入睡眠模式

示例

```
/* enable sleep-on-exit feature */
nvic_lowpower_mode_config(NVIC_LP_SLEEPONEXIT, TRUE);
```

5.14 电源控制（PWC）

PWC 寄存器结构 pwc_type，定义于文件“at32f413_pwc.h”如下：

```
/**
  * @brief type define pwc register all
  */
typedef struct
{
      ……
} pwc_type;
```

下表给出了 PWC 寄存器总览：

表 295. PWC 寄存器对应表

寄存器	描述
ctrl	电源控制寄存器
ctrlsts	电源控制及状态寄存器

下表给出了 PWC 库函数总览：

表 296. PWC 库函数总览

函数名	描述
pwc_reset	复位 PWC 使其所有寄存器保持复位值
pwc_battery_powered_domain_access	电池供电区域的写入使能
pwc_pvm_level_select	电压监测器的监测电压临界值选择
pwc_power_voltage_monitor_enable	电压监测器的电压监测使能
pwc_wakeup_pin_enable	待机唤醒管脚使能
pwc_flag_clear	清除已置位的标志位
pwc_flag_get	获取标志位状态
pwc_sleep_mode_enter	进入睡眠模式
pwc_deep_sleep_mode_enter	进入深度睡眠模式
pwc_standby_mode_enter	进入待机模式

5.14.1 函数 pwc_reset

下表描述了函数 pwc_reset

表 297. 函数 pwc_reset

项目	描述
函数名	pwc_reset
函数原型	void pwc_reset(void)
功能描述	复位 PWC 使其所有寄存器保持复位值
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset()

示例

```
/* deinitialize pwc */
pwc_reset();
```

5.14.2 函数 pwc_battery_powered_domain_access

下表描述了函数 pwc_battery_powered_domain_access

表 298. 函数 pwc_battery_powered_domain_access

项目	描述
函数名	pwc_battery_powered_domain_access
函数原型	void pwc_battery_powered_domain_access(confirm_state new_state)
功能描述	电池供电区域的写入使能
输入参数	new_state：电池供电区域的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable the battery-powered domain write operations */
pwc_battery_powered_domain_access(TRUE);
```

注意：只有通过此函数进行电池供电区域的写使能后，才能操作电池供电区域，比如 RTC。

5.14.3 函数 pwc_pvm_level_select

下表描述了函数 pwc_pvm_level_select

表 299. 函数 pwc_pvm_level_select

项目	描述
函数名	pwc_pvm_level_select
函数原型	void pwc_pvm_level_select(pwc_pvm_voltage_type pvm_voltage)
功能描述	电压监测器的监测电压临界值选择
输入参数	pvm_voltage：监测电压临界值选择 该参数可以选取 pwc_pvm_voltage_type 内的任意一个枚举值.
输出参数	无

项目	描述
返回值	无
先决条件	无
被调用函数	无

pvm_voltage

pvm_voltage 用于设置电压监测器的监测电压临界值，其可选参数罗列如下

PWC_PVM_VOLTAGE_2V3：监测电压临界值为 2.3V

PWC_PVM_VOLTAGE_2V4：监测电压临界值为 2.4V

PWC_PVM_VOLTAGE_2V5：监测电压临界值为 2.5V

PWC_PVM_VOLTAGE_2V6：监测电压临界值为 2.6V

PWC_PVM_VOLTAGE_2V7：监测电压临界值为 2.7V

PWC_PVM_VOLTAGE_2V8：监测电压临界值为 2.8V

PWC_PVM_VOLTAGE_2V9：监测电压临界值为 2.9V

示例

```
/* set the threshold voltage to 2.9v */
pwc_pvm_level_select(PWC_PVM_VOLTAGE_2V9);
```

5.14.4 函数 pwc_power_voltage_monitor_enable

下表描述了函数 pwc_power_voltage_monitor_enable

表 300. 函数 pwc_power_voltage_monitor_enable

项目	描述
函数名	pwc_power_voltage_monitor_enable
函数原型	void pwc_power_voltage_monitor_enable(confirm_state new_state)
功能描述	电压监测器的电压监测使能
输入参数	new_state：电压监测的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable power voltage monitor */
pwc_power_voltage_monitor_enable(TRUE);
```

5.14.5 函数 pwc_wakeup_pin_enable

下表描述了函数 pwc_wakeup_pin_enable

表 301. 函数 pwc_wakeup_pin_enable

项目	描述
函数名	pwc_wakeup_pin_enable
函数原型	void pwc_wakeup_pin_enable(uint32_t pin_num, confirm_state new_state)
功能描述	待机唤醒管脚使能
输入参数 1	pin_num：需要配置的待机唤醒管脚

项目	描述
	该参数可以选取任意具备待机唤醒功能的管脚.
输入参数 2	new_state：待机唤醒管脚的预设状态 该参数可以选取自其中之一：TRUE，FALSE.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

pin_num

pin_num 用于选择需要配置的待机唤醒管脚，其可选参数罗列如下

PWC_WAKEUP_PIN_1：待机唤醒管脚 1（对应 GPIO 为 PA0）

示例

```
/* enable wakeup pin - pa0 */
pwc_wakeup_pin_enable(PWC_WAKEUP_PIN_1, TRUE);
```

5.14.6 函数 pwc_flag_clear

下表描述了函数 pwc_flag_clear

表 302. 函数 pwc_flag_clear

项目	描述
函数名	pwc_flag_clear
函数原型	void pwc_flag_clear(uint32_t pwc_flag)
功能描述	清除已置位的标志位
输入参数	pwc_flag：待清除的标志选择 该参数详细描述见 *pwc_flag*
输出参数	无
返回值	无
先决条件	无
被调用函数	无

pwc_flag

pwc_flag 用于选择需要被清除的标志，其可选参数罗列如下

PWC_WAKEUP_FLAG：　　待机唤醒事件标志

PWC_STANDBY_FLAG：　　进入待机模式标志

PWC_PVM_OUTPUT_FLAG：　　电源电压检测输出标志（此参数不支持软件清除）

示例

```
/* wakeup event flag clear */
pwc_flag_clear(PWC_WAKEUP_FLAG);
```

5.14.7 函数 pwc_flag_get

下表描述了函数 pwc_flag_get

表 303. 函数 pwc_flag_get

项目	描述
函数名	pwc_flag_get

项目	描述
函数原型	flag_status pwc_flag_get(uint32_t pwc_flag)
功能描述	获取标志位状态
输入参数	pwc_flag：需要获取状态的标志选择 该参数详细描述见 *pwc_flag*
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为罗列的其中之一：SET，RESET.
先决条件	无
被调用函数	无

示例

```
/* check if wakeup event flag is set */
if(pwc_flag_get(PWC_WAKEUP_FLAG) != RESET)
```

5.14.8 函数 pwc_sleep_mode_enter

下表描述了函数 pwc_sleep_mode_enter

表 304. 函数 pwc_sleep_mode_enter

项目	描述
函数名	pwc_sleep_mode_enter
函数原型	void pwc_sleep_mode_enter(pwc_sleep_enter_type pwc_sleep_enter)
功能描述	进入睡眠模式
输入参数	pwc_sleep_enter：睡眠模式进入方式选择 该参数可以选取 pwc_sleep_enter_type 内的任意一个枚举值.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

pwc_sleep_enter

pwc_sleep_enter 用于选择睡眠模式进入的方式，其可选参数罗列如下

PWC_SLEEP_ENTER_WFI： 通过 WFI 命令进入睡眠模式

PWC_SLEEP_ENTER_WFE： 通过 WFE 命令进入睡眠模式

示例

```
/* enter sleep mode */
pwc_sleep_mode_enter(PWC_SLEEP_ENTER_WFI);
```

5.14.9 函数 pwc_deep_sleep_mode_enter

下表描述了函数 pwc_deep_sleep_mode_enter

表 305. 函数 pwc_deep_sleep_mode_enter

项目	描述
函数名	pwc_deep_sleep_mode_enter
函数原型	void pwc_deep_sleep_mode_enter(pwc_deep_sleep_enter_type pwc_deep_sleep_enter)

项目	描述
功能描述	进入深度睡眠模式
输入参数	pwc_deep_sleep_enter：深度睡眠模式进入方式选择 该参数可以选取 pwc_deep_sleep_enter_type 内的任意一个枚举值.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

pwc_deep_sleep_enter

pwc_deep_sleep_enter 用于选择深度睡眠模式进入的方式，其可选参数罗列如下

PWC_DEEP_SLEEP_ENTER_WFI： 通过 WFI 命令进入深度睡眠模式

PWC_DEEP_SLEEP_ENTER_WFE： 通过 WFE 命令进入深度睡眠模式

示例

```
/* enter deep sleep mode */
pwc_deep_sleep_mode_enter(PWC_DEEP_SLEEP_ENTER_WFI);
```

5.14.10 函数 pwc_standby_mode_enter

下表描述了函数 pwc_standby_mode_enter

表 306. 函数 pwc_standby_mode_enter

项目	描述
函数名	pwc_standby_mode_enter
函数原型	void pwc_standby_mode_enter(void)
功能描述	进入待机模式
输入参数	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enter standby mode */
pwc_standby_mode_enter();
```

5.15 实时时钟（RTC）

RTC 寄存器结构 rtc_type，定义于文件“at32f413_rtc.h”如下：

```
/**
  * @brief type define rtc register all
  */
typedef struct
{


} rtc_type;
```

下表给出了 RTC 寄存器总览：

表 307. RTC 寄存器对应表

寄存器	描述
ctrlh	RTC 控制寄存器高位
ctrll	RTC 控制寄存器低位
divh	RTC 分频系数寄存器高位
divl	RTC 分频系数寄存器低位
divcnth	RTC 分频计数寄存器高位
divcntl	RTC 分频计数寄存器低位
cnth	RTC 计数值寄存器高位
cntl	RTC 计数值寄存器低位
tah	RTC 闹钟寄存器高位
tal	RTC 闹钟寄存器低位

下表给出了 RTC 库函数总览：

表 308. RTC 库函数总览

函数名	描述
rtc_counter_set	RTC 计数值设置
rtc_counter_get	RTC 计数值获取
rtc_divider_set	RTC 分频器设置
rtc_divider_get	RTC 分频值获取
rtc_alarm_set	RTC 闹钟设置
rtc_interrupt_enable	RTC 中断使能
rtc_flag_get	RTC 标志获取
rtc_flag_clear	RTC 标志清除
rtc_wait_config_finish	RTC 等待配置完成
rtc_wait_update_finish	RTC 等待时间更新完成

5.15.1 函数 rtc_counter_set

下表描述了函数 rtc_counter_set

表 309. 函数 rtc_counter_set

项目	描述
函数名	rtc_counter_set
函数原型	void rtc_counter_set(uint32_t counter_value);
功能描述	计数值设置
输入参数 1	counter_value：RTC 计数值，范围（0~0xFFFFFFFF）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
rtc_counter_set(0x00000008);
```

5.15.2 函数 rtc_counter_get

下表描述了函数 rtc_counter_get

表 310. 函数 rtc_counter_get

项目	描述
函数名	rtc_counter_get
函数原型	uint32_t rtc_counter_get(void);
功能描述	计数值获取
输入参数 1	无
输出参数	无
返回值	uint32_t：当前计数值，计数值通常情况下 1 秒加 1
先决条件	无
被调用函数	无

示例

```
value = rtc_counter_get();
```

5.15.3 函数 rtc_divider_set

下表描述了函数 rtc_divider_set

表 311. 函数 rtc_divider_set

项目	描述
函数名	rtc_divider_set
函数原型	void rtc_divider_set(uint32_t div_value);
功能描述	分频器设置
输入参数 1	div_value：RTC 分频值，范围（0~0x000FFFFF）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
rtc_divider_set(32767);
```

5.15.4 函数 rtc_divider_get

下表描述了函数 rtc_divider_get

表 312. 函数 rtc_divider_get

项目	描述
函数名	rtc_divider_get
函数原型	uint32_t rtc_divider_get(void);
功能描述	分频值获取
输入参数 1	无
输出参数	无
返回值	uint32_t：当前分频值

项目	描述
先决条件	无
被调用函数	无

示例

```
value = rtc_divider_get();
```

5.15.5 函数 rtc_alarm_set

下表描述了函数 rtc_alarm_set

表 313. 函数 rtc_alarm_set

项目	描述
函数名	rtc_alarm_set
函数原型	void rtc_alarm_set(uint32_t alarm_value);
功能描述	闹钟设置
输入参数 1	alarm_value：RTC 闹钟值，范围（0~0xFFFFFFFF），当 RTC 计数值等于闹钟值时，将会发生闹钟事件
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
rtc_alarm_set(0x00000006);
```

5.15.6 函数 rtc_interrupt_enable

下表描述了函数 rtc_interrupt_enable

表 314. 函数 rtc_interrupt_enable

项目	描述
函数名	rtc_interrupt_enable
函数原型	void rtc_interrupt_enable(uint16_t source, confirm_state new_state);
功能描述	中断使能
输入参数 1	source：中断源 参阅章节：source 查阅更多该参数允许取值范围
输入参数 2	new_state：中断使能状态 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

source

中断源

RTC_TS_INT： 秒钟中断

RTC_TA_INT： 闹钟中断

RTC_OVF_INT： 计数器溢出中断

示例

```
rtc_interrupt_enable(RTC_TS_INT, TRUE);
```

5.15.7 函数 rtc_flag_get

下表描述了函数 rtc_flag_get

表 315. 函数 rtc_flag_get

项目	描述
函数名	rtc_flag_get
函数原型	flag_status rtc_flag_get(uint16_t flag);
功能描述	获取标志位状态
输入参数 1	flag：需要获取状态的标志选择 该参数详细描述见 flag
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET、RESET
先决条件	无
被调用函数	无

flag

用于选择需要获取状态的标志，其可选参数罗列如下

RTC_TS_FLAG： 秒钟标志

RTC_TA_FLAG： 闹钟标志

RTC_OVF_FLAG： 计数值溢出标志

RTC_UPDF_FLAG： 时间更新标志

RTC_CFGF_FLAG： RTC 寄存器配置完成标志

示例

```
rtc_flag_get(RTC_TS_FLAG);
```

5.15.8 函数 rtc_flag_clear

下表描述了函数 rtc_flag_clear

表 316. 函数 rtc_flag_clear

项目	描述
函数名	rtc_flag_clear
函数原型	void rtc_flag_clear(uint16_t flag);
功能描述	清除标志位
输入参数 1	flag：待清除的标志选择 该参数详细描述见 flag
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flag

用于选择需要清除状态的标志，其可选参数罗列如下

RTC_TS_FLAG： 秒钟标志
RTC_TA_FLAG： 闹钟标志
RTC_OVF_FLAG： 计数值溢出标志
RTC_UPDF_FLAG： 时间更新标志

示例

```
rtc_flag_clear(RTC_TS_FLAG);
```

5.15.9 函数 rtc_wait_config_finish

下表描述了函数 rtc_wait_config_finish

表 317. 函数 rtc_wait_config_finish

项目	描述
函数名	rtc_wait_config_finish
函数原型	void rtc_wait_config_finish(void);
功能描述	RTC 等待配置完成
输入参数 1	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
rtc_wait_config_finish();
```

5.15.10 函数 rtc_wait_update_finish

下表描述了函数 rtc_wait_update_finish

表 318. 函数 rtc_wait_update_finish

项目	描述
函数名	rtc_wait_update_finish
函数原型	void rtc_wait_update_finish(void);
功能描述	RTC 等待时间更新完成
输入参数 1	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
rtc_wait_update_finish();
```

5.16 SDIO 接口（SDIO）

SDIO 寄存器结构 crm_type，定义于文件“at32f413_sdio.h”如下：

```
/**
  * @brief type define sdio register all
  */
```

```
typedef struct
{
    ...

} sdio_type;
```

下表给出了 SDIO 寄存器总览：

表 319. SDIO 寄存器对应表

寄存器	描述
pwrctrl	电源控制寄存器
clkctrl	时钟控制寄存器
arg	参数寄存器
cmd	命名寄存器
rspcmd	命令响应寄存器
rsp1	响应寄存器 1
rsp2	响应寄存器 2
rsp3	响应寄存器 3
rsp4	响应寄存器 4
dttmr	数据定时器寄存器
dtlen	数据长度寄存器
dtctrl	数据控制寄存器
dtcntr	数据计算器寄存器
sts	状态寄存器
intclr	清除中断寄存器
inten	中断屏蔽寄存器
bufcntr	BUF 计数器寄存器
buf	数据 BUF 寄存器

下表给出了 SDIO 库函数总览：

表 320. SDIO 库函数总览

函数名	描述
sdio_reset	将 SDIO 外设的寄存器和控制状态复位
sdio_power_set	设置控制器的电源状态
sdio_power_status_get	获取控制器的电源状态
sdio_clock_config	时钟参数配置
sdio_bus_width_config	总线宽度配置
sdio_clock_bypass	时钟旁路模式使能设置
sdio_power_saving_mode_enable	控制器省电模式使能设置
sdio_flow_control_enable	流控模式使能设置
sdio_clock_enable	时钟使能设置
sdio_dma_enable	dma 使能设置
sdio_interrupt_enable	中断使能设置
sdio_flag_get	读取判断指定的标志是否置起
sdio_flag_clear	清除指定的标志位
sdio_command_config	命令参数配置

sdio_command_state_machine_enable	命令状态机使能设置
sdio_command_response_get	返回收到的命令响应所对应的命令号
sdio_response_get	返回卡的命令响应
sdio_data_config	数据参数配置
sdio_data_state_machine_enable	数据状态机使能设置
sdio_data_counter_get	返回待传输的数据字节数
sdio_data_read	从接收 fifo 读取一个 word 数据
sdio_buffer_counter_get	返回将要写入 BUF 或将要从 BUF 读出数据字的数目
sdio_data_write	写一个 word 数据到发送 fifo
sdio_read_wait_mode_set	读等待模式设置
sdio_read_wait_start	读等待开始设置
sdio_read_wait_stop	读等待停止设置
sdio_io_function_enable	IO 功能模式使能设置
sdio_io_suspend_command_set	IO 功能模式下的挂起命令使能设置

5.16.1 函数 sdio_reset

下表描述了函数 sdio_reset

表 321. 函数 sdio_reset

项目	描述
函数名	sdio_reset
函数原型	void sdio_reset(sdio_type *sdio_x);
功能描述	将 SDIO 外设的寄存器和控制状态复位
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* reset sdio */
sdio_reset(SDIO1);
```

5.16.2 函数 sdio_power_set

下表描述了函数 sdio_power_set

表 322. 函数 sdio_power_set

项目	描述
函数名	sdio_power_set
函数原型	void sdio_power_set(sdio_type *sdio_x, sdio_power_state_type power_state);
功能描述	设置控制器的电源状态
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	power_state：控制器电源状态
输出参数	无

项目	描述
返回值	无
先决条件	无
被调用函数	无

power_state

控制器电源状态

SDIO_POWER_ON： 控制器电源开

SDIO_POWER_OFF： 控制器电源关

示例

```
/* sdio power on */
sdio_power_set(SDIO1, SDIO_POWER_ON);
```

5.16.3 函数 sdio_power_status_get

下表描述了函数 sdio_power_status_get

表 323. 函数 sdio_power_status_get

项目	描述
函数名	sdio_power_status_get
函数原型	sdio_power_state_type sdio_power_status_get(sdio_type *sdio_x);
功能描述	获取控制器的电源状态
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	无
输出参数	无
返回值	sdio_power_state_type：控制器电源状态
先决条件	无
被调用函数	无

示例

```
/* check power status */
if(sdio_power_status_get(SDIO1) == SDIO_POWER_OFF)
{
  return SD_REQ_NOT_APPLICABLE;
}
```

5.16.4 函数 sdio_clock_config

下表描述了函数 sdio_clock_config

表 324. 函数 sdio_clock_config

项目	描述
函数名	sdio_clock_config
函数原型	void sdio_clock_config(sdio_type *sdio_x, uint16_t clk_div, sdio_edge_phase_type clk_edg);
功能描述	时钟参数配置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	clk_div：时钟分频设置，范围 0~0x3FF

项目	描述
输入参数 2	clk_edg：时钟边沿设置
输出参数	无
返回值	无
先决条件	无
被调用函数	无

clk_edg

指定时钟边沿

SDIO_CLOCK_EDGE_RISING： 时钟上升沿

SDIO_CLOCK_EDGE_FALLING： 时钟下降沿

示例

```
/* config sdio clock divide and edge phase */
sdio_clock_config(SDIO1, 0x2, SDIO_CLOCK_EDGE_FALLING);
```

5.16.5 函数 sdio_bus_width_config

下表描述了函数 sdio_bus_width_config

表 325. 函数 sdio_bus_width_config

项目	描述
函数名	sdio_bus_width_config
函数原型	void sdio_bus_width_config(sdio_type *sdio_x, sdio_bus_width_type width);
功能描述	总线宽度配置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	width：指定设置的总线宽度
输出参数	无
返回值	无
先决条件	无
被调用函数	无

width

数据总线宽度定义

SDIO_BUS_WIDTH_D1：1-bit 数据总线宽度

SDIO_BUS_WIDTH_D4：4-bit 数据总线宽度

SDIO_BUS_WIDTH_D8：8-bit 数据总线宽度

示例

```
/* config sdio bus width */
sdio_bus_width_config(SDIOx, SDIO_BUS_WIDTH_D1);
```

5.16.6 函数 sdio_clock_bypass

下表描述了函数 sdio_clock_bypass

表 326. 函数 sdio_clock_bypass

项目	描述
函数名	sdio_clock_bypass
函数原型	void sdio_clock_bypass(sdio_type *sdio_x, confirm_state new_state);

项目	描述
功能描述	时钟旁路模式使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。旁路开启（TRUE），旁路关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* disable clock bypass */
sdio_clock_bypass(SDIO1, FALSE);
```

5.16.7 函数 sdio_power_saving_mode_enable

下表描述了函数 sdio_power_saving_mode_enable

表 327. 函数 sdio_power_saving_mode_enable

项目	描述
函数名	sdio_power_saving_mode_enable
函数原型	void sdio_power_saving_mode_enable(sdio_type *sdio_x, confirm_state new_state);
功能描述	控制器省电模式使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* disable power saving mode */
sdio_power_saving_mode_enable(SDIO1, FALSE);
```

5.16.8 函数 sdio_flow_control_enable

下表描述了函数 sdio_flow_control_enable

表 328. 函数 sdio_flow_control_enable

项目	描述
函数名	sdio_flow_control_enable
函数原型	void sdio_flow_control_enable(sdio_type *sdio_x, confirm_state new_state);
功能描述	流控模式使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无

项目	描述
被调用函数	无

示例

```
/* disable flow control */
sdio_flow_control_enable(SDIO1, FALSE);
```

5.16.9 函数 sdio_clock_enable

下表描述了函数 sdio_clock_enable

表 329. 函数 sdio_clock_enable

项目	描述
函数名	sdio_clock_enable
函数原型	void sdio_clock_enable(sdio_type *sdio_x, confirm_state new_state);
功能描述	时钟使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable to output sdio_ck */
sdio_clock_enable(SDIO1, TRUE);
```

5.16.10 函数 sdio_dma_enable

下表描述了函数 sdio_dma_enable

表 330. 函数 sdio_dma_enable

项目	描述
函数名	sdio_dma_enable
函数原型	void sdio_dma_enable(sdio_type *sdio_x, confirm_state new_state);
功能描述	dma 使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable sdio dma */
sdio_dma_enable(SDIO1, TRUE);
```

5.16.11 函数 sdio_interrupt_enable

下表描述了函数 sdio_interrupt_enable

表 331. 函数 crm_flag_clear

项目	描述
函数名	sdio_interrupt_enable
函数原型	void sdio_interrupt_enable(sdio_type *sdio_x, uint32_t int_opt, confirm_state new_state);
功能描述	中断使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	int_opt：指定的中断类型
输入参数 3	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

int_opt

指定的中断类型

SDIO_CMDFAIL_INT： 命令 CRC 检测失败中断
SDIO_DTFAIL_INT： 数据块 CRC 检测失败中断
SDIO_CMDTIMEOUT_INT： 命令超时中断
SDIO_DTTIMEOUT_INT： 数据超时中断
SDIO_TXERRU_INT： 发送 BUF 下溢错误中断
SDIO_RXERRO_INT： 接收 BUF 上溢错误中断
SDIO_CMDRSPCMPL_INT： 接收到命令响应中断
SDIO_CMDCMPL_INT： 命令发送完成中断
SDIO_DTCMP_INT： 数据传输完成中断
SDIO_SBITERR_INT： 起始位错误中断
SDIO_DTBLKCMPL_INT： 数据块传输完成中断
SDIO_DOCMD_INT： 正在传输命令中断
SDIO_DOTX_INT： 正在传输数据中断
SDIO_DORX_INT： 正在接收数据中断
SDIO_TXBUFH_INT： 发送 BUF 半空中断
SDIO_RXBUFH_INT： 接收 BUF 半空中断
SDIO_TXBUFF_INT： 发送 BUF 满中断
SDIO_RXBUFF_INT： 接收 BUF 满中断
SDIO_TXBUFE_INT： 发送 BUF 空中断
SDIO_RXBUFE_INT： 接收 BUF 空中断
SDIO_TXBUF_INT： 发送 BUF 中的数据有效中断
SDIO_RXBUF_INT： 接收 BUF 中的数据有效中断
SDIO_SDIOIF_INT： SD I/O 模式接收中断

示例

```
/* disable interrupt */
sdio_interrupt_enable(SDIO1, (SDIO_DTFAIL_INT   | SDIO_DTTIMEOUT_INT | \
                              SDIO_DTCMP_INT | SDIO_TXBUFH_INT | SDIO_RXBUFH_INT      | \
```

```
                                SDIO_TXERRU_INT| SDIO_RXERRO_INT | SDIO_SBITERR_INT), FALSE);
```

5.16.12 函数 sdio_flag_get

下表描述了函数 sdio_flag_get

表 332. 函数 sdio_flag_get

项目	描述
函数名	sdio_flag_get
函数原型	flag_status sdio_flag_get(sdio_type *sdio_x, uint32_t flag);
功能描述	读取判断指定的标志是否置起
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	flag：指定的中断类型
输出参数	无
返回值	flag_status：返回标志位的状态，置起（SET）未置起（RESET）
先决条件	无
被调用函数	无

flag

指定的 flag 标志

SDIO_CMDFAIL_FLAG： 命令 CRC 检测失败标志
SDIO_DTFAIL_FLAG： 数据块 CRC 检测失败标志
SDIO_CMDTIMEOUT_FLAG： 命令超时标志
SDIO_DTTIMEOUT_FLAG： 数据超时标志
SDIO_TXERRU_FLAG： 发送 BUF 下溢错误标志
SDIO_RXERRO_FLAG： 接收 BUF 上溢错误标志
SDIO_CMDRSPCMPL_FLAG： 接收到命令响应标志
SDIO_CMDCMPL_FLAG： 命令发送完成标志
SDIO_DTCMP_FLAG： 数据传输完成标志
SDIO_SBITERR_FLAG： 起始位错误标志
SDIO_DTBLKCMPL_FLAG： 数据块传输完成标志
SDIO_DOCMD_FLAG： 正在传输命令标志
SDIO_DOTX_FLAG： 正在传输数据标志
SDIO_DORX_FLAG： 正在接收数据标志
SDIO_TXBUFH_FLAG： 发送 BUF 半空标志
SDIO_RXBUFH_FLAG： 接收 BUF 半空标志
SDIO_TXBUFF_FLAG： 发送 BUF 满标志
SDIO_RXBUFF_FLAG： 接收 BUF 满标志
SDIO_TXBUFE_FLAG： 发送 BUF 空标志
SDIO_RXBUFE_FLAG： 接收 BUF 空标志
SDIO_TXBUF_FLAG： 发送 BUF 中的数据有效标志
SDIO_RXBUF_FLAG： 接收 BUF 中的数据有效标志
SDIO_SDIOIF_FLAG： SD I/O 模式接收标志

示例

```
/* check dttimeout flag */
if(sdio_flag_get(SDIOx, SDIO_DTTIMEOUT_FLAG) != RESET)
{
```

```
}
```

5.16.13 函数 sdio_flag_clear

下表描述了函数 sdio_flag_clear

表 333. 函数 sdio_flag_clear

项目	描述
函数名	sdio_flag_clear
函数原型	void sdio_flag_clear(sdio_type *sdio_x, uint32_t flag);
功能描述	清除指定的标志位
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	flag：指定的中断类型
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flag

指定的 flag 标志

SDIO_CMDFAIL_FLAG： 命令 CRC 检测失败标志

SDIO_DTFAIL_FLAG： 数据块 CRC 检测失败标志

SDIO_CMDTIMEOUT_FLAG： 命令超时标志

SDIO_DTTIMEOUT_FLAG： 数据超时标志

SDIO_TXERRU_FLAG： 发送 BUF 下溢错误标志

SDIO_RXERRO_FLAG： 接收 BUF 上溢错误标志

SDIO_CMDRSPCMPL_FLAG： 接收到命令响应标志

SDIO_CMDCMPL_FLAG： 命令发送完成标志

SDIO_DTCMP_FLAG： 数据传输完成标志

SDIO_SBITERR_FLAG： 起始位错误标志

SDIO_DTBLKCMPL_FLAG： 数据块传输完成标志

SDIO_SDIOIF_FLAG： SD I/O 模式接收标志

示例

```
/* clear flags */
#define SDIO_STATIC_FLAGS                    ((uint32_t)0x000005FF)
sdio_flag_clear(SDIO1, SDIO_STATIC_FLAGS);
```

5.16.14 函数 sdio_command_config

下表描述了函数 sdio_command_config

表 334. 函数 sdio_command_config

项目	描述
函数名	sdio_command_config
函数原型	void sdio_command_config(sdio_type *sdio_x, sdio_command_struct_type *command_struct);
功能描述	命令参数配置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1

项目	描述
输入参数 2	command_struct：指向结构体 sdio_command_struct_type 的指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

command_struct

sdio_command_struct_type 结构体定义在 at32f413_sdio.h 文件中，内容如下：

```
typedef struct
{
  uint32_t                argument;
  uint8_t                 cmd_index;
  sdio_reponse_type       rsp_type;
  sdio_wait_type          wait_type;
} sdio_command_struct_type;
```

argument

命令参数作为发送给卡的命令信息的一部分，由各命令类型来决定

cmd_index

发送的命令编号

rsp_type

响应类型参数，由各命令类型来决定，可选的类型如下

SDIO_RESPONSE_NO：　无响应

SDIO_RESPONSE_SHORT：短响应

SDIO_RESPONSE_LONG：　长响应

wait_type

等待类型参数，由各命令类型来决定，可选的类型如下

SDIO_WAIT_FOR_NO：　无等待

SDIO_WAIT_FOR_INT：　等待中断请求

SDIO_WAIT_FOR_PEND：　等待传输结束

示例

```
/* send cmd16, set block length */
sdio_command_struct_type sdio_command_init_struct;
sdio_command_init_struct.argument = (uint32_t)8;
sdio_command_init_struct.cmd_index = SD_CMD_SET_BLOCKLEN;
sdio_command_init_struct.rsp_type = SDIO_RESPONSE_SHORT;
sdio_command_init_struct.wait_type = SDIO_WAIT_FOR_NO;
/* sdio command config */
sdio_command_config(SDIOx, &sdio_command_init_struct);
```

5.16.15 函数 sdio_command_state_machine_enable

下表描述了函数 sdio_command_state_machine_enable

表 335. 函数 sdio_command_state_machine_enable

项目	描述
函数名	sdio_command_state_machine_enable

项目	描述
函数原型	void sdio_command_state_machine_enable(sdio_type *sdio_x, confirm_state new_state);
功能描述	命令状态机使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable ccsm */
sdio_command_state_machine_enable(SDIO1, TRUE);
```

5.16.16 函数 sdio_command_response_get

下表描述了函数 sdio_command_response_get

表 336. 函数 sdio_command_response_get

项目	描述
函数名	sdio_command_response_get
函数原型	uint8_t sdio_command_response_get(sdio_type *sdio_x);
功能描述	返回收到的命令响应所对应的命令号
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	无
输出参数	无
返回值	uint8_t：返回收到的命令响应所对应的命令号
先决条件	无
被调用函数	无

示例

```
/* get response of command index */
uint8_t rsp_cmd = 0;
rsp_cmd = sdio_command_response_get(SDIO1);
```

5.16.17 函数 sdio_response_get

下表描述了函数 sdio_response_get

表 337. 函数 sdio_response_get

项目	描述
函数名	sdio_response_get
函数原型	uint32_t sdio_response_get(sdio_type *sdio_x, sdio_rsp_index_type reg_index);
功能描述	返回卡的命令响应
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	reg_index：响应寄存器编号，有编号 1/2/3/4 响应寄存器，获取对应寄存器的响应值

项目	描述
输出参数	无
返回值	无
先决条件	无
被调用函数	无

reg_div

响应寄存器编号类型

SDIO_RSP1_INDEX：响应寄存器编号 1

SDIO_RSP2_INDEX：响应寄存器编号 2

SDIO_RSP3_INDEX：响应寄存器编号 3

SDIO_RSP4_INDEX：响应寄存器编号 4

示例

```
/* get response register1 */
response = sdio_response_get(SDIO1, SDIO_RSP1_INDEX);
```

5.16.18 函数 sdio_data_config

下表描述了函数 sdio_data_config

表 338. 函数 sdio_data_config

项目	描述
函数名	sdio_data_config
函数原型	void sdio_data_config(sdio_type *sdio_x, sdio_data_struct_type *data_struct);
功能描述	数据参数配置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	data_struct：指向结构体 sdio_data_struct_type 的指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

data_struct

sdio_data_struct_type 结构体定义在 at32f413_sdio.h 文件中，内容如下：

```
typedef struct
{
  uint32_t                    timeout;
  uint32_t                    data_length;
  sdio_block_size_type        block_size;
  sdio_transfer_mode_type     transfer_mode;
  sdio_transfer_direction_type transfer_direction;
} sdio_data_struct_type;
```

timeout

进行数据传输的超时周期值，由总线时钟作为计数基准

data_length

要传输的数据长度字节数目

block_size

块 size 大小，有如下类型

SDIO_DATA_BLOCK_SIZE_1B： 块大小为 1-bit
SDIO_DATA_BLOCK_SIZE_2B： 块大小为 2-bit
SDIO_DATA_BLOCK_SIZE_4B： 块大小为 4-bit
SDIO_DATA_BLOCK_SIZE_8B： 块大小为 8-bit
SDIO_DATA_BLOCK_SIZE_16B： 块大小为 16-bit
SDIO_DATA_BLOCK_SIZE_32B： 块大小为 32-bit
SDIO_DATA_BLOCK_SIZE_64B： 块大小为 64-bit
SDIO_DATA_BLOCK_SIZE_128B： 块大小为 128-bit
SDIO_DATA_BLOCK_SIZE_256B： 块大小为 256-bit
SDIO_DATA_BLOCK_SIZE_512B： 块大小为 512-bit
SDIO_DATA_BLOCK_SIZE_1024B： 块大小为 1024-bit
SDIO_DATA_BLOCK_SIZE_2048B： 块大小为 2048-bit
SDIO_DATA_BLOCK_SIZE_4096B： 块大小为 4096-bit
SDIO_DATA_BLOCK_SIZE_8192B： 块大小为 8192-bit
SDIO_DATA_BLOCK_SIZE_16384B： 块大小为 16384-bit

transfer_mode

数据传输模式类型

SDIO_DATA_BLOCK_TRANSFER： 数据按块模式传输
SDIO_DATA_STREAM_TRANSFER： 数据按流模式传输

transfer_direction

数据传输方向类型

SDIO_DATA_TRANSFER_TO_CARD： 数据由控制器到卡
SDIO_DATA_TRANSFER_TO_CONTROLLER： 数据由卡到控制器

示例

```
sdio_data_struct_type sdio_data_init_struct;
sdio_data_init_struct.block_size = SDIO_DATA_BLOCK_SIZE_512B;
sdio_data_init_struct.data_length = 8 ;
sdio_data_init_struct.timeout = SD_DATATIMEOUT ;
sdio_data_init_struct.transfer_direction = SDIO_DATA_TRANSFER_TO_CARD;
sdio_data_init_struct.transfer_mode = SDIO_DATA_BLOCK_TRANSFER;
/* config sdio data */
sdio_data_config(SDIO1, &sdio_data_init_struct);
```

5.16.19 函数 sdio_data_state_machine_enable

下表描述了函数 sdio_data_state_machine_enable

表 339. 函数 sdio_data_state_machine_enable

项目	描述
函数名	sdio_data_state_machine_enable
函数原型	void sdio_data_state_machine_enable(sdio_type *sdio_x, confirm_state new_state);
功能描述	数据状态机使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无

项目	描述
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable dcsm */
sdio_data_state_machine_enable(SDIO1, TRUE);
```

5.16.20 函数 sdio_data_counter_get

下表描述了函数 sdio_data_counter_get

表 340. 函数 sdio_data_counter_get

项目	描述
函数名	sdio_data_counter_get
函数原型	uint32_t sdio_data_counter_get(sdio_type *sdio_x);
功能描述	返回待传输的数据字节数
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	无
输出参数	无
返回值	uint32_t：返回待传输的数据字节数
先决条件	无
被调用函数	无

示例

```
/* get data counter */
uint32_t count = 0;
count = sdio_data_counter_get (SDIO1);
```

5.16.21 函数 sdio_data_read

下表描述了函数 sdio_data_read

表 341. 函数 sdio_data_read

项目	描述
函数名	sdio_data_read
函数原型	uint32_t sdio_data_read(sdio_type *sdio_x);
功能描述	从接收 fifo 读取一个 word 数据
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	无
输入参数 3	无
输出参数	无
返回值	uint32_t：读取的一个 word 数据
先决条件	无
被调用函数	无

示例

```
/* read data */
```

```
uint32_t data = 0;
data = sdio_data_read(SDIO1);
```

5.16.22 函数 sdio_buffer_counter_get

下表描述了函数 sdio_buffer_counter_get

表 342. 函数 sdio_buffer_counter_get

项目	描述
函数名	sdio_buffer_counter_get
函数原型	uint32_t sdio_buffer_counter_get(sdio_type *sdio_x);
功能描述	返回将要写入 BUF 或将要从 BUF 读出数据字的数目
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* get buffer count */
uint32_t count = 0;
count = sdio_buffer_counter_get(SDIO1);
```

5.16.23 函数 sdio_data_write

下表描述了函数 sdio_data_write

表 343. 函数 sdio_data_write

项目	描述
函数名	sdio_data_write
函数原型	void sdio_data_write(sdio_type *sdio_x, uint32_t data);
功能描述	写一个 word 数据到发送 fifo
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* write data */
uin32_t data = 0x11223344;
sdio_data_write(SDIO1, data);
```

5.16.24 函数 sdio_read_wait_mode_set

下表描述了函数 sdio_read_wait_mode_set

表 344. 函数 sdio_read_wait_mode_set

项目	描述
函数名	sdio_read_wait_mode_set
函数原型	void sdio_read_wait_mode_set(sdio_type *sdio_x, sdio_read_wait_mode_type mode);
功能描述	读等待模式设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	mode：读等待模式
输出参数	无
返回值	无
先决条件	无
被调用函数	无

mode

SDIO_READ_WAIT_CONTROLLED_BY_D2：读等待由 DATA Line2 控制

SDIO_READ_WAIT_CONTROLLED_BY_CK：读等待由时钟线控制

示例

```
/* config read wait mode */
sdio_read_wait_mode_set(SDIO1, SDIO_READ_WAIT_CONTROLLED_BY_D2);
```

5.16.25 函数 sdio_read_wait_start

下表描述了函数 sdio_read_wait_start

表 345. 函数 sdio_read_wait_start

项目	描述
函数名	sdio_read_wait_start
函数原型	void sdio_read_wait_start(sdio_type *sdio_x, confirm_state new_state);
功能描述	读等待开始设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

此函数调用开启表示开始读等待，关闭表示无动作。

示例

```
/* start read wait mode */
sdio_read_wait_start (SDIO1, TRUE);
```

5.16.26 函数 sdio_read_wait_stop

下表描述了函数 sdio_read_wait_stop

表 346. 函数 sdio_read_wait_stop

项目	描述
函数名	sdio_read_wait_stop

项目	描述
函数原型	void sdio_read_wait_stop(sdio_type *sdio_x, confirm_state new_state);
功能描述	读等待停止设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

此函数调用开启表示关闭读等待，关闭表示持续读等待。

示例

```
/* stop read wait mode */
sdio_read_wait_stop (SDIO1, TRUE);
```

5.16.27 函数 sdio_io_function_enable

下表描述了函数 sdio_io_function_enable

表 347. 函数 sdio_io_function_enable

项目	描述
函数名	sdio_io_function_enable
函数原型	void sdio_io_function_enable(sdio_type *sdio_x, confirm_state new_state);
功能描述	IO 功能模式使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable sdio IO mode */
sdio_io_function_enable (SDIO1, TRUE);
```

5.16.28 函数 sdio_io_suspend_command_set

下表描述了函数 sdio_io_suspend_command_set

表 348. 函数 sdio_io_suspend_command_set

项目	描述
函数名	sdio_io_suspend_command_set
函数原型	void sdio_io_suspend_command_set(sdio_type *sdio_x, confirm_state new_state);
功能描述	IO 功能模式下的挂起命令使能设置
输入参数 1	sdio_x：指定的 SDIO 外设，如：SDIO1
输入参数 2	new_state：新的设置状态。开启（TRUE），关闭（FALSE）
输出参数	无

项目	描述
返回值	无
先决条件	无
被调用函数	无

示例

```
/* send suspend command */
sdio_io_suspend_command_set (SDIO1, TRUE);
```

5.17 串行外设口（SPI）/音频接口（I²S）

SPI 寄存器结构 spi_type，定义于文件“at32f413_spi.h”如下：

```
/**
  * @brief type define spi register all
  */
typedef struct
{
  ...
} spi_type;
```

下表给出了 SPI 寄存器总览：

表 349. SPI 寄存器总览

寄存器	描述
ctrl1	SPI 控制寄存器 1
ctrl2	SPI 控制寄存器 2
sts	SPI 状态寄存器
dt	SPI 数据寄存器
cpoly	SPI CRC 多项式寄存器
rcrc	SPI RxCRC 寄存器
tcrc	SPI TxCRC 寄存器
i2sctrl	SPI_I2S 配置寄存器
i2sclkp	SPI_I2S 预分频寄存器

下表给出了 SPI 库函数总览：

表 350. SPI 库函数总览

函数名	描述
spi_i2s_reset	将 SPI/I²S 所有寄存器值恢复到复位值
spi_default_para_init	给 SPI 初始化结构体赋初值
spi_init	SPI 初始化
spi_crc_next_transmit	下一笔数据传输 CRC 命令
spi_crc_polynomial_set	SPI CRC 多项式设置
spi_crc_polynomial_get	获取 SPI CRC 多项式
spi_crc_enable	SPI CRC 使能
spi_crc_value_get	获取 SPI 接收/发送 CRC 结果
spi_hardware_cs_output_enable	硬件 CS 输出使能

spi_software_cs_internal_level_set	设置软件 CS 内部电平
spi_frame_bit_num_set	设置帧位个数
spi_half_duplex_direction_set	设置单线双向半双工模式的传输方向
spi_enable	SPI 使能
i2s_default_para_init	给 I²S 初始化结构体赋初值
i2s_init	I²S 初始化
i2s_enable	I²S 使能
spi_i2s_interrupt_enable	使能选定的 SPI/I²S 中断
spi_i2s_dma_transmitter_enable	SPI/I²S DMA 发送使能
spi_i2s_dma_receiver_enable	SPI/I²S DMA 接收使能
spi_i2s_data_transmit	SPI/I²S 发送一笔数据
spi_i2s_data_receive	SPI/I²S 接收一笔数据
spi_i2s_flag_get	读取选定的 SPI/I²S 标志
spi_i2s_flag_clear	清除选定的 SPI/I²S 标志

5.17.1 函数 spi_i2s_reset

下表描述了函数 spi_i2s_reset

表 351. 函数 spi_i2s_reset

项目	描述
函数名	spi_i2s_reset
函数原型	void spi_i2s_reset(spi_type *spi_x);
功能描述	将 SPI/I²S 所有寄存器值恢复到复位值
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset();

示例

```
spi_i2s_reset (SPI1);
```

5.17.2 函数 spi_default_para_init

下表描述了函数 spi_default_para_init

表 352. 函数 spi_default_para_init

项目	描述
函数名	spi_default_para_init
函数原型	void spi_default_para_init(spi_init_type* spi_init_struct);
功能描述	给 SPI 初始化结构体赋初值
输入参数 1	spi_init_struct：指向 *spi_init_type* 类型的指针
输出参数	无
返回值	无

项目	描述
先决条件	需要先定义一个 spi_init_type 类型的变量
被调用函数	无

示例

```
spi_init_type spi_init_struct;
spi_default_para_init (&spi_init_struct);
```

5.17.3 函数 spi_init

下表描述了函数 spi_init

表 353. 函数 spi_init

项目	描述
函数名	spi_init
函数原型	void spi_init(spi_type* spi_x, spi_init_type* spi_init_struct);
功能描述	SPI 初始化
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	spi_init_struct：指向 *spi_init_type* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 spi_init_type 类型的变量
被调用函数	无

spi_init_type 在 at32f413_spi.h 中定义：

```
typedef struct
{
  spi_transmission_mode_type      transmission_mode;
  spi_master_slave_mode_type      master_slave_mode;
  spi_mclk_freq_div_type          mclk_freq_division;
  spi_first_bit_type              first_bit_transmission;
  spi_frame_bit_num_type          frame_bit_num;
  spi_clock_polarity_type         clock_polarity;
  spi_clock_phase_type            clock_phase;
  spi_cs_mode_type                cs_mode_selection;
} spi_init_type;
```

spi_transmission_mode

SPI 传输模式

SPI_TRANSMIT_FULL_DUPLEX： 双线单向全双工模式
SPI_TRANSMIT_SIMPLEX_RX： 双线单向只收模式
SPI_TRANSMIT_HALF_DUPLEX_RX： 单线双向只收模式
SPI_TRANSMIT_HALF_DUPLEX_TX： 单线双向只发模式

master_slave_mode

主从模式选择

SPI_MODE_SLAVE： 从机模式

SPI_MODE_MASTER： 主机模式

mclk_freq_division

分频系数选择

SPI_MCLK_DIV_2： 2 分频

SPI_MCLK_DIV_4： 4 分频

SPI_MCLK_DIV_8： 8 分频

SPI_MCLK_DIV_16： 16 分频

SPI_MCLK_DIV_32： 32 分频

SPI_MCLK_DIV_64： 64 分频

SPI_MCLK_DIV_128： 128 分频

SPI_MCLK_DIV_256： 256 分频

SPI_MCLK_DIV_512： 512 分频

SPI_MCLK_DIV_1024：1024 分频

first_bit_transmission

SPI 先发送高位/低位

SPI_FIRST_BIT_MSB： 先发送高位

SPI_FIRST_BIT_LSB： 先发送低位

frame_bit_num

设置帧位个数

SPI_FRAME_8BIT： 一帧包含 8bit 数据

SPI_FRAME_16BIT： 一帧包含 16bit 数据

clock_polarity

时钟极性

SPI_CLOCK_POLARITY_LOW： 空闲时，时钟输出低电平

SPI_CLOCK_POLARITY_HIGH： 空闲时，时钟输出高电平

clock_phase

时钟相位

SPI_CLOCK_PHASE_1EDGE：SPI 第一个时钟沿进行数据采样

SPI_CLOCK_PHASE_2EDGE：SPI 第二个时钟沿进行数据采样

cs_mode_selection

时钟相位

SPI_CS_HARDWARE_MODE： 硬件 CS 模式

SPI_CS_SOFTWARE_MODE： 软件 CS 模式

示例

```
spi_init_type spi_init_struct;
spi_default_para_init(&spi_init_struct);
spi_init_struct.transmission_mode = SPI_TRANSMIT_FULL_DUPLEX;
spi_init_struct.master_slave_mode = SPI_MODE_MASTER;
spi_init_struct.mclk_freq_division = SPI_MCLK_DIV_8;
spi_init_struct.first_bit_transmission = SPI_FIRST_BIT_MSB;
spi_init_struct.frame_bit_num = SPI_FRAME_16BIT;
spi_init_struct.clock_polarity = SPI_CLOCK_POLARITY_LOW;
spi_init_struct.clock_phase = SPI_CLOCK_PHASE_2EDGE;
spi_init_struct.cs_mode_selection = SPI_CS_SOFTWARE_MODE;
spi_init(SPI1, &spi_init_struct);
```

5.17.4 函数 spi_crc_next_transmit

下表描述了函数 spi_crc_next_transmit

表 354. 函数 spi_crc_next_transmit

项目	描述
函数名	spi_crc_next_transmit
函数原型	void spi_crc_next_transmit(spi_type* spi_x);
功能描述	下一笔数据传输 CRC 命令
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
spi_crc_next_transmit (SPI1);
```

5.17.5 函数 spi_crc_polynomial_set

下表描述了函数 spi_crc_polynomial_set

表 355. 函数 spi_crc_polynomial_set

项目	描述
函数名	spi_crc_polynomial_set
函数原型	void spi_crc_polynomial_set(spi_type* spi_x, uint16_t crc_poly);
功能描述	SPI CRC 多项式设置
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	crc_poly：CRC 多项式 取值范围：0x0000~0xFFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/*set spi crc polynomial value */
spi_crc_polynomial_set (SPI1, 0x07);
```

5.17.6 函数 spi_crc_polynomial_get

下表描述了函数 spi_crc_polynomial_get

表 356. 函数 spi_crc_polynomial_get

项目	描述
函数名	spi_crc_polynomial_get
函数原型	uint16_t spi_crc_polynomial_get(spi_type* spi_x);
功能描述	获取 SPI CRC 多项式
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输出参数	无
返回值	CRC 多项式 取值范围：0x0000~0xFFFF
先决条件	无
被调用函数	无

示例

```
/*get spi crc polynomial value */
uint16_t crc_poly;
crc_poly = spi_crc_polynomial_get (SPI1);
```

5.17.7 函数 spi_crc_enable

下表描述了函数 spi_crc_enable

表 357. 函数 spi_crc_enable

项目	描述
函数名	spi_crc_enable
函数原型	void spi_crc_enable(spi_type* spi_x, confirm_state new_state);
功能描述	SPI CRC 使能
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* spi crc enable */
spi_crc_enable (SPI1, TRUE);
```

5.17.8 函数 spi_crc_value_get

下表描述了函数 spi_crc_value_get

表 358. 函数 spi_crc_value_get

项目	描述
函数名	spi_crc_value_get
函数原型	uint16_t spi_crc_value_get(spi_type* spi_x, spi_crc_direction_type crc_direction);
功能描述	获取 SPI 接收/发送 CRC 结果
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	*crc_direction*：接收/发送 CRC 选择
输出参数	无
返回值	无
先决条件	无
被调用函数	无

crc_direction

接收/发送 CRC 选择

SPI_CRC_RX： 选择接收 CRC

SPI_CRC_TX： 选择发送 CRC

示例

```
/* get spi rx & tx crc enable */
uint16_t spi_rx_crc, spi_tx_crc;
spi_rx_crc = spi_crc_value_get (SPI1, SPI_CRC_RX);
spi_tx_crc = spi_crc_value_get (SPI1, SPI_CRC_TX);
```

5.17.9 函数 spi_hardware_cs_output_enable

下表描述了函数 spi_hardware_cs_output_enable

表 359. 函数 spi_hardware_cs_output_enable

项目	描述
函数名	spi_hardware_cs_output_enable
函数原型	void spi_hardware_cs_output_enable(spi_type* spi_x, confirm_state new_state);
功能描述	硬件 CS 输出使能
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	SPI 必须为主机模式时，此项设置才有效
被调用函数	无

示例

```
/* enable the hardware cs output */
spi_hardware_cs_output_enable (SPI1, TRUE);
```

5.17.10 函数 spi_software_cs_internal_level_set

下表描述了函数 spi_software_cs_internal_level_set

表 360. 函数 spi_software_cs_internal_level_set

项目	描述
函数名	spi_software_cs_internal_level_set
函数原型	void spi_software_cs_internal_level_set(spi_type* spi_x, spi_software_cs_level_type level);
功能描述	设置软件 CS 内部电平
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	*level*：设置软件 CS 内部电平
输出参数	无
返回值	无
先决条件	1、 仅在软件 CS 模式下有效； 2、 主机模式下，level 值必须为“SPI_SWCS_INTERNAL_LEVEL_HIGHT”。
被调用函数	无

level

设置软件 CS 内部电平

SPI_SWCS_INTERNAL_LEVEL_LOW： 设置软件 CS 内部电平为低电平

SPI_SWCS_INTERNAL_LEVEL_HIGHT： 设置软件 CS 内部电平为高电平

示例

```
/* set the internal level high */
spi_software_cs_internal_level_set (SPI1, SPI_SWCS_INTERNAL_LEVEL_HIGHT);
```

5.17.11 函数 spi_frame_bit_num_set

下表描述了函数 spi_frame_bit_num_set

表 361. 函数 spi_frame_bit_num_set

项目	描述
函数名	spi_frame_bit_num_set
函数原型	void spi_frame_bit_num_set(spi_type* spi_x, spi_frame_bit_num_type bit_num);
功能描述	设置 SPI 帧位个数
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	*bit_num*：设置帧位个数
输出参数	无
返回值	无
先决条件	无
被调用函数	无

bit_num

设置帧位个数

SPI_FRAME_8BIT： 一帧包含 8bit 数据
SPI_FRAME_16BIT： 一帧包含 16bit 数据
示例

```
/* set the data frame bit num as 8 */
spi_frame_bit_num_set (SPI1, SPI_FRAME_8BIT);
```

5.17.12 函数 spi_half_duplex_direction_set

下表描述了函数 spi_half_duplex_direction_set

表 362. 函数 spi_half_duplex_direction_set

项目	描述
函数名	spi_half_duplex_direction_set
函数原型	void spi_half_duplex_direction_set(spi_type* spi_x, spi_half_duplex_direction_type direction);
功能描述	设置单线双向半双工模式的传输方向
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	*direction*：传输方向
输出参数	无
返回值	无
先决条件	仅在单线双向半双工模式下有效
被调用函数	无

direction
传输方向
SPI_HALF_DUPLEX_DIRECTION_RX：接收
SPI_HALF_DUPLEX_DIRECTION_TX：发送
示例

```
/* set the data transmission direction as transmit */
spi_half_duplex_direction_set (SPI1, SPI_HALF_DUPLEX_DIRECTION_TX);
```

5.17.13 函数 spi_enable

下表描述了函数 spi_enable

表 363. 函数 spi_enable

项目	描述
函数名	spi_enable
函数原型	void spi_enable(spi_type* spi_x, confirm_state new_state);
功能描述	SPI 使能
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无

项目	描述
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable spi */
spi_enable (SPI1, TRUE);
```

5.17.14 函数 i2s_default_para_init

下表描述了函数 i2s_default_para_init

表 364. 函数 i2s_default_para_init

项目	描述
函数名	i2s_default_para_init
函数原型	void i2s_default_para_init(i2s_init_type* i2s_init_struct);
功能描述	给 I²S 初始化结构体赋初值
输入参数 1	i2s_init_struct：指向 *spi_i2s_flag* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 i2s_init_type 类型的变量
被调用函数	无

示例

```
i2s_init_type i2s_init_struct;
i2s_default_para_init (&i2s_init_struct);
```

5.17.15 函数 i2s_init

下表描述了函数 i2s_init

表 365. 函数 i2s_init

项目	描述
函数名	i2s_init
函数原型	void i2s_init(spi_type* spi_x, i2s_init_type* i2s_init_struct);
功能描述	I²S 初始化
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	i2s_init_struct：指向 *spi_i2s_flag* 类型的指针
输出参数	无
返回值	无
先决条件	需要先定义一个 i2s_init_type 类型的变量
被调用函数	无

i2s_init_type 在 at32f413_spi.h 中定义：

```
typedef struct
{
  i2s_operation_mode_type          operation_mode;
  i2s_audio_protocol_type          audio_protocol;
  i2s_audio_sampling_freq_type     audio_sampling_freq;
  i2s_data_channel_format_type     data_channel_format;
  i2s_clock_polarity_type          clock_polarity;
  confirm_state                    mclk_output_enable;
} i2s_init_type;
```

operation_mode

I²S 传输模式

I2S_MODE_SLAVE_TX： I2S 从机发送模式
I2S_MODE_SLAVE_RX： I2S 从机接收模式
I2S_MODE_MASTER_TX： I2S 主机发送模式
I2S_MODE_MASTER_RX： I2S 主机接收模式

audio_protocol

I²S 音频协议标准

I2S_AUDIO_PROTOCOL_PHILLIPS： 飞利浦标准
I2S_AUDIO_PROTOCOL_MSB： 高字节对齐标准（左对齐）
I2S_AUDIO_PROTOCOL_LSB： 低字节对齐标准（右对齐）
I2S_AUDIO_PROTOCOL_PCM_SHORT： PCM 短帧同步标准
I2S_AUDIO_PROTOCOL_PCM_LONG： PCM 长帧同步标准

audio_sampling_freq

I²S 音频采样率选择

I2S_AUDIO_FREQUENCY_DEFAULT：保持复位值（采样率会随 SCLK 变化而变化）
I2S_AUDIO_FREQUENCY_8K： I2S 采样率 8K
I2S_AUDIO_FREQUENCY_11_025K： I2S 采样率 11.025K
I2S_AUDIO_FREQUENCY_16K： I2S 采样率 16K
I2S_AUDIO_FREQUENCY_22_05K： I2S 采样率 22.05K
I2S_AUDIO_FREQUENCY_32K： I2S 采样率 32K
I2S_AUDIO_FREQUENCY_44_1K： I2S 采样率 44.1K
I2S_AUDIO_FREQUENCY_48K： I2S 采样率 48K
I2S_AUDIO_FREQUENCY_96K： I2S 采样率 96K
I2S_AUDIO_FREQUENCY_192K： I2S 采样率 192K

data_channel_format

I²S 数据/声道位数格式

I2S_DATA_16BIT_CHANNEL_16BIT：数据位数 16bit，声道位数 16bit
I2S_DATA_16BIT_CHANNEL_32BIT：数据位数 16bit，声道位数 32bit
I2S_DATA_24BIT_CHANNEL_32BIT：数据位数 24bit，声道位数 32bit
I2S_DATA_32BIT_CHANNEL_32BIT：数据位数 32bit，声道位数 32bit

clock_polarity

I²S 时钟极性

I2S_CLOCK_POLARITY_LOW： 空闲时，时钟输出低电平
I2S_CLOCK_POLARITY_HIGH： 空闲时，时钟输出高电平

mclk_output_enable

mclk 主时钟输出使能

取值范围：FALSE，TRUE。

示例

```
i2s_init_type i2s_init_struct;
i2s_default_para_init(&i2s_init_struct);
i2s_init_struct.audio_protocol = I2S_AUDIO_PROTOCOL_PHILLIPS;
i2s_init_struct.data_channel_format = I2S_DATA_16BIT_CHANNEL_32BIT;
i2s_init_struct.mclk_output_enable = FALSE;
i2s_init_struct.audio_sampling_freq = I2S_AUDIO_FREQUENCY_48K;
i2s_init_struct.clock_polarity = I2S_CLOCK_POLARITY_LOW;
i2s_init_struct.operation_mode = I2S_MODE_MASTER_TX;
i2s_init(SPI2, &i2s_init_struct);
```

5.17.16 函数 i2s_enable

下表描述了函数 i2s_enable

表 366. 函数 i2s_enable

项目	描述
函数名	i2s_enable
函数原型	void i2s_enable(spi_type* spi_x, confirm_state new_state);
功能描述	I²S 使能
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable i2s*/
i2s_enable (SPI1, TRUE);
```

5.17.17 函数 spi_i2s_interrupt_enable

下表描述了函数 spi_i2s_interrupt_enable

表 367. 函数 spi_i2s_interrupt_enable

项目	描述
函数名	spi_i2s_interrupt_enable
函数原型	void spi_i2s_interrupt_enable(spi_type* spi_x, uint32_t spi_i2s_int, confirm_state new_state);
功能描述	使能选定的 SPI/I²S 中断
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.

项目	描述
输入参数 2	*spi_i2s_int*：SPI 中断选择
输入参数 3	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

spi_i2s_int

SPI/I²S 中断选择

SPI_I2S_ERROR_INT： SPI/I²S 错误中断（包含 CRC 校验错误，上溢错误，下溢错误，模式错误）。

SPI_I2S_RDBF_INT： 接收数据缓冲器满中断

SPI_I2S_TDBE_INT： 发送数据缓冲器空中断

示例

```
/* enable the specified spi/i2s interrupts */
spi_i2s_interrupt_enable (SPI1, SPI_I2S_ERROR_INT);
spi_i2s_interrupt_enable (SPI1, SPI_I2S_RDBF_INT);
spi_i2s_interrupt_enable (SPI1, SPI_I2S_TDBE_INT);
```

5.17.18 函数 spi_i2s_dma_transmitter_enable

下表描述了函数 spi_i2s_dma_transmitter_enable

表 368. 函数 spi_i2s_dma_transmitter_enable

项目	描述
函数名	spi_i2s_dma_transmitter_enable
函数原型	void spi_i2s_dma_transmitter_enable(spi_type* spi_x, confirm_state new_state);
功能描述	SPI/I²S DMA 发送使能
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable spi transmitter dma */
spi_i2s_dma_transmitter_enable (SPI1, TRUE);
```

5.17.19 函数 spi_i2s_dma_receiver_enable

下表描述了函数 spi_i2s_dma_receiver_enable

表 369. 函数 spi_i2s_dma_receiver_enable

项目	描述
函数名	spi_i2s_dma_receiver_enable
函数原型	void spi_i2s_dma_receiver_enable(spi_type* spi_x, confirm_state new_state);
功能描述	SPI/I²S DMA 接收使能
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	new_state：使能或关闭 该参数可以选取自其中之一 ：FALSE，TRUE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable spi dma transmitter */
spi_i2s_dma_transmitter_enable (SPI1, TRUE);
```

5.17.20 函数 spi_i2s_data_transmit

下表描述了函数 spi_i2s_data_transmit

表 370. 函数 spi_i2s_data_transmit

项目	描述
函数名	spi_i2s_data_transmit
函数原型	void spi_i2s_data_transmit(spi_type* spi_x, uint16_t tx_data);
功能描述	SPI/I²S 发送一笔数据
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	tx_data：待发送的数据 取值范围（帧位个数为 8bit 时）：0x00~0xFF 取值范围（帧位个数为 16bit 时）：0x0000~0xFFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* spi data transmit */
uint16_t tx_data = 0x6666;
spi_i2s_data_transmit (SPI1, tx_data);
```

5.17.21 函数 spi_i2s_data_receive

下表描述了函数 spi_i2s_data_receive

表 371. 函数 spi_i2s_data_receive

项目	描述
函数名	spi_i2s_data_receive
函数原型	uint16_t spi_i2s_data_receive(spi_type* spi_x);
功能描述	SPI/I²S 接收一笔数据
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输出参数	rx_data：接收到的数据 参数范围（帧位个数为 8bit 时）：0x00~0xFF 参数范围（帧位个数为 16bit 时）：0x0000~0xFFFF
返回值	无
先决条件	无
被调用函数	无

示例

```
/* spi data receive */
uint16_t rx_data = 0;
rx_data = spi_i2s_data_receive (SPI1);
```

5.17.22 函数 spi_i2s_flag_get

下表描述了函数 spi_i2s_flag_get

表 372. 函数 spi_i2s_flag_get

项目	描述
函数名	spi_i2s_flag_get
函数原型	flag_status spi_i2s_flag_get(spi_type* spi_x, uint32_t spi_i2s_flag);
功能描述	读取选定的 SPI/I²S 标志
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	spi_i2s_flag：需要获取状态的标志选择 该参数详细描述见 spi_i2s_flag
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET，RESET.
先决条件	无
被调用函数	无

spi_i2s_flag

SPI/I²S 用于选择需要获取状态的标志，其可选参数罗列如下：

SPI_I2S_RDBF_FLAG： SPI/I²S 接收数据缓冲器满标志

SPI_I2S_TDBE_FLAG： SPI/I²S 发送数据缓冲器空标志

I2S_ACS_FLAG： I2S 音频通道状态标志（指示左/右声道）

I2S_TUERR_FLAG： I2S 发送器欠载错误标志

SPI_CCERR_FLAG： SPI CRC 校验错误标志

SPI_MMERR_FLAG： SPI 主模式错误标志

SPI_I2S_ROERR_FLAG： SPI/I²S 接收器溢出错误标志
SPI_I2S_BF_FLAG： SPI/I²S 通信忙标志

示例

```
/* get receive data buffer full flag */
flag_status status;
status = spi_i2s_flag_get(SPI1, SPI_I2S_RDBF_FLAG);
```

5.17.23 函数 spi_i2s_flag_clear

下表描述了函数 spi_i2s_flag_clear

表 373. 函数 spi_i2s_flag_clear

项目	描述
函数名	spi_i2s_flag_clear
函数原型	void spi_i2s_flag_clear(spi_type* spi_x, uint32_t spi_i2s_flag)
功能描述	清除选定的 SPI/I²S 标志
输入参数 1	spi_x：所选择的 SPI 外设 该参数可以选取自其中之一：SPI1，SPI2.
输入参数 2	*spi_i2s_flag*：待清除的标志选择 该参数详细描述见 spi_i2s_flag
输出参数	无
返回值	无
先决条件	无
被调用函数	无

spi_i2s_flag：

SPI/I²S 用于选择需要清除的标志，其可选参数罗列如下：
SPI_I2S_RDBF_FLAG： SPI/I²S 接收数据缓冲器满标志
I2S_TUERR_FLAG： I2S 发送器欠载错误标志
SPI_CCERR_FLAG： SPI CRC 校验错误标志
SPI_MMERR_FLAG： SPI 主模式错误标志
SPI_I2S_ROERR_FLAG： SPI/I²S 接收器溢出错误标志

注意：SPI_I2S_TDBE_FLAG（SPI/I2S 发送数据缓冲器空标志）、I2S_ACS_FLAG（I2S 音频通道状态标志）和 SPI_I2S_BF_FLAG（SPI/I2S 通信忙标志）全由硬件置位和清除，用以指示通信状态，无需软件清除。

示例

```
/* clear receive data buffer full flag */
spi_i2s_flag_clear (SPI1, SPI_I2S_RDBF_FLAG);
```

5.18 系统滴答（SysTick）

SysTick 寄存器结构 SysTick_Type，定义于文件“core_cm4.h”如下：

```
typedef struct
{
  ...
```

} SysTick_Type;

下表给出了 SysTick 寄存器总览：

表 374. SysTick 寄存器对应表

寄存器	描述
ctrl	控制状态寄存器
load	重载值寄存器
val	当前计数值寄存器
calib	校准寄存器

下表给出了 SysTick 库函数总览：

表 375. SysTick 库函数总览

函数名	描述
systick_clock_source_config	系统滴答时钟源配置
SysTick_Config	系统滴答计数重载值设置及中断使能

5.18.1 函数 systick_clock_source_config

下表描述了函数 systick_clock_source_config

表 376. 函数 systick_clock_source_config

项目	描述
函数名	systick_clock_source_config
函数原型	void systick_clock_source_config(systick_clock_source_type source);
功能描述	系统滴答时钟源配置
输入参数 1	source：配置的 systick 时钟源
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

source

SYSTICK_CLOCK_SOURCE_AHBCLK_DIV8： AHB 总线时钟 8 分频作为 SysTick 时钟

SYSTICK_CLOCK_SOURCE_AHBCLK_NODIV： AHB 总线时钟不分频作为 SysTick 时钟

示例

```
/* config systick clock source */
systick_clock_source_config(SYSTICK_CLOCK_SOURCE_AHBCLK_NODIV);
```

5.18.2 函数 SysTick_Config

下表描述了函数 SysTick_Config

表 377. 函数 SysTick_Config

项目	描述
函数名	SysTick_Config

项目	描述
函数原型	uint32_t SysTick_Config(uint32_t ticks);
功能描述	系统滴答计数重载值设置及中断使能
输入参数 1	ticks：系统滴答计数中断重载值
输入参数 2	
输出参数	无
返回值	返回函数设置状态，成功（0），失败（1）
先决条件	无
被调用函数	无

示例

```
/* config systick reload value and enable interrupt */
SysTick_Config(1000);
```

5.19 定时器（TMR）

TMR 寄存器结构 tmr_type，定义于文件“at32f413_tmr.h”如下：

```
/**
  * @brief type define tmr register all
  */
typedef struct
{


} tmr_type;
```

下表给出了 TMR 寄存器总览：

表 378. TMR 寄存器对应表

寄存器	描述
ctrl1	TMR 控制寄存器 1
ctrl2	TMR 控制寄存器 2
stctrl	TMR 次定时器控制寄存器
iden	TMR DMA/中断使能寄存器
ists	TMR 中断状态寄存器
swevt	TMR 软件事件寄存器
cm1	TMR 通道模式寄存器 1
cm2	TMR 通道模式寄存器 2
cctrl	TMR 通道控制寄存器
cval	TMR 计数值
div	TMR 预分频器
pr	TMR 周期寄存器
rpr	TMR 重复周期寄存器
c1dt	TMR 通道 1 数据寄存器
c2dt	TMR 通道 2 数据寄存器
c3dt	TMR 通道 3 数据寄存器

寄存器	描述
c4dt	TMR 通道 4 数据寄存器
brk	TMR 刹车寄存器
dmactrl	TMR DMA 控制寄存器
dmadt	TMR DMA 数据寄存器

下表给出了 TMR 库函数总览：

表 379. TMR 库函数总览

函数名	描述
tmr_reset	TMR 由 CRM 复位寄存器复位
tmr_counter_enable	启用或禁用 TMR 计数器
tmr_output_default_para_init	初始化 TMR 输出默认参数
tmr_input_default_para_init	初始化 TMR 输入默认参数
tmr_brkdt_default_para_init	初始化 TMR brkdt 默认参数
tmr_base_init	初始化 TMR 周期、分频
tmr_clock_source_div_set	设置 TMR 时钟源分频系数
tmr_cnt_dir_set	设置 TMR 计数器计数方向
tmr_repetition_counter_set	设置重复周期寄存器（rpr）的值
tmr_counter_value_set	设置 TMR 计数器值
tmr_counter_value_get	获取 TMR 计数器值
tmr_div_value_set	设置 TMR 分频器值
tmr_div_value_get	获取 TMR 分频器值
tmr_output_channel_config	配置 TMR 输出通道
tmr_output_channel_mode_select	选择 TMR 输出通道模式
tmr_period_value_set	设置 TMR 周期值
tmr_period_value_get	获取 TMR 周期值
tmr_channel_value_set	设置 TMR 通道值
tmr_channel_value_get	获取 TMR 通道值
tmr_period_buffer_enable	启用或禁用 TMR 周期缓冲区
tmr_output_channel_buffer_enable	启用或禁用 TMR 输出通道缓冲区
tmr_output_channel_immediately_set	设置 TMR 输出通道立即使能
tmr_output_channel_switch_set	设置 TMR 输出通道开关
tmr_one_cycle_mode_enable	启用或禁用 TMR 单周期模式
tmr_32_bit_function_enable	启用或禁用 TMR 32 位功能（plus 模式）
tmr_overflow_request_source_set	选择 TMR 溢出事件源
tmr_overflow_event_disable	启用或禁用 TMR 溢出事件产生
tmr_input_channel_init	初始化 TMR 输入通道
tmr_channel_enable	启用或禁用 TMR 通道
tmr_input_channel_filter_set	设置 TMR 输入通道过滤器
tmr_pwm_input_config	配置 TMR pwm 输入
tmr_channel1_input_select	选择 TMR 通道 1 输入
tmr_input_channel_divider_set	设置 TMR 输入通道分频器
tmr_primary_mode_select	选择 TMR 主模式
tmr_sub_mode_select	选择 TMR 次定时器模式

tmr_channel_dma_select	选择 TMR 通道的 DMA 请求源
tmr_hall_select	选择 TMR hall 模式
tmr_channel_buffer_enable	启用或禁用 TMR 通道缓冲区
tmr_trigger_input_select	选择 TMR 次定时器触发输入
tmr_sub_sync_mode_set	设置 TMR 次定时器同步模式
tmr_dma_request_enable	启用或禁用 TMR DMA 请求
tmr_interrupt_enable	启用或禁用 TMR 中断
tmr_flag_get	获取 TMR 标记
tmr_flag_clear	清除 TMR 标记
tmr_event_sw_trigger	软件触发 TMR 事件
tmr_output_enable	启用或禁用 TMR 输出使能
tmr_internal_clock_set	设置 TMR 内部时钟
tmr_output_channel_polarity_set	设置 TMR 输出通道极性
tmr_external_clock_config	配置 TMR 外部时钟
tmr_external_clock_mode1_config	配置 TMR 外部时钟模式 1
tmr_external_clock_mode2_config	配置 TMR 外部时钟模式 2
tmr_encoder_mode_config	配置 TMR 编码器模式
tmr_force_output_set	设置 TMR 强制输出
tmr_dma_control_config	配置 TMR DMA 控制
tmr_brkdt_config	配置 TMR 刹车模式和死区时间

5.19.1 函数 tmr_reset

下表描述了函数 tmr_reset

表 380. 函数 tmr_reset

项目	描述
函数名	tmr_reset
函数原型	void tmr_reset(tmr_type *tmr_x);
功能描述	TMR 由 CRM 复位寄存器复位
输入参数	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset();

示例

```
tmr_reset(TMR1);
```

5.19.2 函数 tmr_counter_enable

下表描述了函数 tmr_counter_enable

表 381. 函数 tmr_counter_enable

项目	描述
函数名	tmr_counter_enable

项目	描述
函数原型	void tmr_counter_enable(tmr_type *tmr_x, confirm_state new_state);
功能描述	启用或禁用 TMR 计数器
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	new_state：将要配置的计数器状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_counter_enable(TMR1, TRUE);
```

5.19.3 函数 tmr_output_default_para_init

下表描述了函数 tmr_output_default_para_init

表 382. 函数 tmr_output_default_para_init

项目	描述
函数名	tmr_output_default_para_init
函数原型	void tmr_output_default_para_init(tmr_output_config_type *tmr_output_struct);
功能描述	初始化 tmr 输出默认参数
输入参数	tmr_output_struct：指向结构体 tmr_output_config_type 的待初始化指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

下表描述了 tmr_output_struct 各个成员的默认值

表 383. tmr_output_struct 默认值

成员	默认值
oc_mode	TMR_OUTPUT_CONTROL_OFF
oc_idle_state	FALSE
occ_idle_state	FALSE
oc_polarity	TMR_OUTPUT_ACTIVE_HIGH
occ_polarity	TMR_OUTPUT_ACTIVE_HIGH
oc_output_state	FALSE
occ_output_state	FALSE

示例

```
tmr_output_config_type tmr_output_struct;
tmr_output_default_para_init(&tmr_output_struct);
```

5.19.4 函数 tmr_input_default_para_init

下表描述了函数 tmr_input_default_para_init

表 384. 函数 tmr_input_default_para_init

项目	描述
函数名	tmr_input_default_para_init
函数原型	void tmr_input_default_para_init(tmr_input_config_type *tmr_input_struct);
功能描述	初始化 TMR 输入默认参数
输入参数	tmr_input_struct：指向结构体 tmr_input_config_type 的待初始化指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

下表描述了 tmr_input_struct 各个成员的默认值

表 385. tmr_input_struct 默认值

成员	默认值
input_channel_select	TMR_SELECT_CHANNEL_1
input_polarity_select	TMR_INPUT_RISING_EDGE
input_mapped_select	TMR_CC_CHANNEL_MAPPED_DIRECT
input_filter_value	0x0

示例

```
tmr_input_config_type tmr_input_struct;
tmr_input_default_para_init(&tmr_input_struct);
```

5.19.5 函数 tmr_brkdt_default_para_init

下表描述了函数 tmr_brkdt_default_para_init

表 386. 函数 tmr_brkdt_default_para_init

项目	描述
函数名	tmr_brkdt_default_para_init
函数原型	void tmr_brkdt_default_para_init(tmr_brkdt_config_type *tmr_brkdt_struct);
功能描述	初始化 TMR brkdt 默认参数
输入参数	tmr_brkdt_struct：指向结构体 tmr_brkdt_config_type 的待初始化指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

下表描述了 tmr_brkdt_struct 各个成员的默认值

表 387. tmr_brkdt_struct 默认值

成员	默认值
deadtime	0x0
brk_polarity	TMR_BRK_INPUT_ACTIVE_LOW
wp_level	TMR_WP_OFF
auto_output_enable	FALSE
fcsoen_state	FALSE
fcsodis_state	FALSE

成员	默认值
brk_enable	FALSE

示例

```
tmr_brkdt_config_type tmr_brkdt_struct;
tmr_brkdt_default_para_init(&tmr_brkdt_struct);
```

5.19.6 函数 tmr_base_init

下表描述了函数 tmr_base_init

表 388. 函数 tmr_base_init

项目	描述
函数名	tmr_base_init
函数原型	void tmr_base_init(tmr_type* tmr_x, uint32_t tmr_pr, uint32_t tmr_div);
功能描述	初始化 TMR 周期、分频
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_pr：定时器周期值，16 位定时器可取 0x0000~0xFFFF，32 位定时器可取 0x0000_0000~0xFFFF_FFFF
输入参数 3	tmr_div：定时器分频值，0x0000~0xFFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_base_init(TMR1, 0xFFFF, 0xFFFF);
```

5.19.7 函数 tmr_clock_source_div_set

下表描述了函数 tmr_clock_source_div_set

表 389. 函数 tmr_clock_source_div_set

项目	描述
函数名	tmr_clock_source_div_set
函数原型	void tmr_clock_source_div_set(tmr_type *tmr_x, tmr_clock_division_type tmr_clock_div);
功能描述	设置 TMR 时钟源分频系数
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_clock_div：定时器时钟源分频系数
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_clock_div

设置 TMR 时钟源分频系数

TMR_CLOCK_DIV1：定时器时钟源分频系数为 1
TMR_CLOCK_DIV2：定时器时钟源分频系数为 2
TMR_CLOCK_DIV4：定时器时钟源分频系数为 4

示例

```
tmr_clock_source_div_set(TMR1, TMR_CLOCK_DIV4);
```

5.19.8 函数 tmr_cnt_dir_set

下表描述了函数 tmr_cnt_dir_set

表 390. 函数 tmr_cnt_dir_set

项目	描述
函数名	tmr_cnt_dir_set
函数原型	void tmr_cnt_dir_set(tmr_type *tmr_x, tmr_count_mode_type tmr_cnt_dir);
功能描述	设置 TMR 计数器计数方向
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_cnt_dir：定时器计数方向
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_cnt_dir
设置定时器计数方向
TMR_COUNT_UP： 定时器计数器向上计数
TMR_COUNT_DOWN： 定时器计数器向下计数
TMR_COUNT_TWO_WAY_1： 定时器计数器中央双向对齐计数模式 1
TMR_COUNT_TWO_WAY_2： 定时器计数器中央双向对齐计数模式 2
TMR_COUNT_TWO_WAY_3： 定时器计数器中央双向对齐计数模式 3

示例

```
tmr_cnt_dir_set(TMR1, TMR_COUNT_UP);
```

5.19.9 函数 tmr_repetition_counter_set

下表描述了函数 tmr_repetition_counter_set

表 391. 函数 tmr_repetition_counter_set

项目	描述
函数名	tmr_repetition_counter_set
函数原型	void tmr_repetition_counter_set(tmr_type *tmr_x, uint8_t tmr_rpr_value);
功能描述	设置重复周期寄存器（rpr）的值
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR8
输入参数 2	tmr_rpr_value：定时器重复周期值，可取 0x00~0xFF
输出参数	无
返回值	无

项目	描述
先决条件	无
被调用函数	无

示例

```
tmr_repetition_counter_set(TMR1, 0x10);
```

5.19.10 函数 tmr_counter_value_set

下表描述了函数 tmr_counter_value_set

表 392. 函数 tmr_counter_value_set

项目	描述
函数名	tmr_counter_value_set
函数原型	void tmr_counter_value_set(tmr_type *tmr_x, uint32_t tmr_cnt_value);
功能描述	设置 TMR 计数器值
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_cnt_value：定时器计数器值，16 位定时器可取 0x0000~0xFFFF，32 位定时器可取 0x0000_0000~0xFFFF_FFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_counter_value_set(TMR1, 0xFFFF);
```

5.19.11 函数 tmr_counter_value_get

下表描述了函数 tmr_counter_value_get

表 393. 函数 tmr_counter_value_get

项目	描述
函数名	tmr_counter_value_get
函数原型	uint32_t tmr_counter_value_get(tmr_type *tmr_x);
功能描述	获取 TMR 计数器值
输入参数	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输出参数	无
返回值	定时器计数器值
先决条件	无
被调用函数	无

示例

```
uint32_t counter_value;
counter_value = tmr_counter_value_get(TMR1);
```

5.19.12 函数 tmr_div_value_set

下表描述了函数 tmr_div_value_set

表 394. 函数 tmr_div_value_set

项目	描述
函数名	tmr_div_value_set
函数原型	void tmr_div_value_set(tmr_type *tmr_x, uint32_t tmr_div_value);
功能描述	设置 TMR 分频器值
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_div_value：定时器分频器值，16 位定时器可取 0x0000~0xFFFF，32 位定时器可取 0x0000_0000~0xFFFF_FFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_div_value_set(TMR1, 0xFFFF);
```

5.19.13 函数 tmr_div_value_get

下表描述了函数 tmr_div_value_get

表 395. 函数 tmr_div_value_get

项目	描述
函数名	tmr_div_value_get
函数原型	uint32_t tmr_div_value_get(tmr_type *tmr_x);
功能描述	获取 TMR 分频器值
输入参数	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输出参数	无
返回值	定时器分频器值
先决条件	无
被调用函数	无

示例

```
uint32_t div_value;
div_value = tmr_div_value_get(TMR1);
```

5.19.14 函数 tmr_output_channel_config

下表描述了函数 tmr_output_channel_config

表 396. 函数 tmr_output_channel_config

项目	描述
函数名	tmr_output_channel_config
函数原型	void tmr_output_channel_config(tmr_type *tmr_x, tmr_channel_select_type

项目	描述
	tmr_channel, tmr_output_config_type *tmr_output_struct);
功能描述	配置 TMR 输出通道
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	tmr_output_struct：指向结构体 tmr_output_config_type 的指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

tmr_output_config_type structure

tmr_output_config_type 在 at32f413_tmr.h 中

```
typedef struct
{
  tmr_output_control_mode_type      oc_mode;
  confirm_state                     oc_idle_state;
  confirm_state                     occ_idle_state;
  tmr_output_polarity_type          oc_polarity;
  tmr_output_polarity_type          occ_polarity;
  confirm_state                     oc_output_state;
  confirm_state                     occ_output_state;
} tmr_output_config_type;
```

oc_mode

配置输出通道模式，即对通道原始信号（CxORAW）进行配置

TMR_OUTPUT_CONTROL_OFF：　　断开通道输出（CxOUT）与 CxORAW 的连接

TMR_OUTPUT_CONTROL_HIGH：　　设置 CxORAW 为高

TMR_OUTPUT_CONTROL_LOW：　　设置 CxORAW 为低

TMR_OUTPUT_CONTROL_SWITCH：　　切换 CxORAW 的电平

TMR_OUTPUT_CONTROL_FORCE_LOW：　　固定 CxORAW 为低

TMR_OUTPUT_CONTROL_FORCE_HIGH：　　固定 CxORAW 为高

TMR_OUTPUT_CONTROL_PWM_MODE_A：　　PWM 模式 A

TMR_OUTPUT_CONTROL_PWM_MODE_B：　　PWM 模式 B

oc_idle_state

配置输出通道空闲状态

FALSE：　输出通道空闲状态为 0

TRUE：　输出通道空闲状态为 1

occ_idle_state

配置互补输出通道空闲状态

FALSE：　互补输出通道空闲状态为 0

TRUE： 互补输出通道空闲状态为 1

oc_polarity

配置输出通道极性

TMR_OUTPUT_ACTIVE_HIGH： 输出通道极性高

TMR_OUTPUT_ACTIVE_LOW： 输出通道极性低

occ_polarity

配置互补输出通道极性

TMR_OUTPUT_ACTIVE_HIGH： 互补输出通道极性高

TMR_OUTPUT_ACTIVE_LOW： 互补输出通道极性低

oc_output_state

配置输出通道状态

FALSE： 输出通道关闭

TRUE： 输出通道开启

occ_output_state

配置互补输出通道状态

FALSE： 互补输出通道关闭

TRUE： 互补输出通道开启

示例

```
tmr_output_config_type tmr_output_struct;
tmr_output_struct.oc_mode = TMR_OUTPUT_CONTROL_OFF;
tmr_output_struct.oc_output_state = TRUE;
tmr_output_struct.oc_polarity = TMR_OUTPUT_ACTIVE_HIGH;
tmr_output_struct.oc_idle_state = TRUE;
tmr_output_struct.occ_output_state = TRUE;
tmr_output_struct.occ_polarity = TMR_OUTPUT_ACTIVE_HIGH;
tmr_output_struct.occ_idle_state = TRUE;
tmr_output_channel_config(TMR1, TMR_SELECT_CHANNEL_1, &tmr_output_struct);
```

5.19.15 函数 tmr_output_channel_mode_select

下表描述了函数 tmr_output_channel_mode_select

表 397. 函数 tmr_output_channel_mode_select

项目	描述
函数名	tmr_output_channel_mode_select
函数原型	void tmr_output_channel_mode_select(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, tmr_output_control_mode_type oc_mode);
功能描述	选择 TMR 输出通道模式
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	oc_mode：输出模式
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

oc_mode

配置输出通道模式，即对通道原始信号（CxORAW）进行配置

TMR_OUTPUT_CONTROL_OFF：　断开通道输出（CxOUT）与 CxORAW 的连接

TMR_OUTPUT_CONTROL_HIGH：　设置 CxORAW 为高

TMR_OUTPUT_CONTROL_LOW：　设置 CxORAW 为低

TMR_OUTPUT_CONTROL_SWITCH：　切换 CxORAW 的电平

TMR_OUTPUT_CONTROL_FORCE_LOW：　固定 CxORAW 为低

TMR_OUTPUT_CONTROL_FORCE_HIGH：　固定 CxORAW 为高

TMR_OUTPUT_CONTROL_PWM_MODE_A：　PWM 模式 A

TMR_OUTPUT_CONTROL_PWM_MODE_B：　PWM 模式 B

示例

```
tmr_output_channel_mode_select(TMR1, TMR_SELECT_CHANNEL_1, TMR_OUTPUT_CONTROL_SWITCH);
```

5.19.16　函数 tmr_period_value_set

下表描述了函数 tmr_period_value_set

表 398. 函数 tmr_period_value_set

项目	描述
函数名	tmr_period_value_set
函数原型	void tmr_period_value_set(tmr_type *tmr_x, uint32_t tmr_pr_value);
功能描述	设置 TMR 周期值
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_pr_value：定时器周期值，16 位定时器可取 0x0000~0xFFFF，32 位定时器可取 0x0000_0000~0xFFFF_FFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_period_value_set(TMR1, 0xFFFF);
```

5.19.17　函数 tmr_period_value_get

下表描述了函数 tmr_period_value_get

表 399. 函数 tmr_period_value_get

项目	描述
函数名	tmr_period_value_get
函数原型	uint32_t tmr_period_value_get(tmr_type *tmr_x);

项目	描述
功能描述	获取 TMR 周期值
输入参数	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输出参数	无
返回值	定时器周期值
先决条件	无
被调用函数	无

示例

```
uint32_t pr_value;
pr_value = tmr_period_value_get(TMR1);
```

5.19.18 函数 tmr_channel_value_set

下表描述了函数 tmr_channel_value_set

表 400. 函数 tmr_channel_value_set

项目	描述
函数名	tmr_channel_value_set
函数原型	void tmr_channel_value_set(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, uint32_t tmr_channel_value);
功能描述	设置 TMR 通道值
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	tmr_channel_value：定时器通道值，16 位定时器可取 0x0000~0xFFFF，32 位定时器可取 0x0000_0000~0xFFFF_FFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

示例

```
tmr_channel_value_set(TMR1, TMR_SELECT_CHANNEL_1, 0xFFFF);
```

5.19.19 函数 tmr_channel_value_get

下表描述了函数 tmr_channel_value_get

表 401. 函数 tmr_channel_value_get

项目	描述
函数名	tmr_channel_value_get
函数原型	uint32_t tmr_channel_value_get(tmr_type *tmr_x, tmr_channel_select_type tmr_channel);
功能描述	获取 TMR 通道值
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输出参数	定时器通道值
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

示例

```
uint32_t ch_value;
ch_value = tmr_channel_value_get(TMR1, TMR_SELECT_CHANNEL_1);
```

5.19.20 函数 tmr_period_buffer_enable

下表描述了函数 tmr_period_buffer_enable

表 402. 函数 tmr_period_buffer_enable

项目	描述
函数名	tmr_period_buffer_enable
函数原型	void tmr_period_buffer_enable(tmr_type *tmr_x, confirm_state new_state);
功能描述	启用或禁用 TMR 周期缓冲区
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	new_state：将要配置的周期缓冲区状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_period_buffer_enable(TMR1, TRUE);
```

5.19.21 函数 tmr_output_channel_buffer_enable

下表描述了函数 tmr_output_channel_buffer_enable

表 403. 函数 tmr_output_channel_buffer_enable

项目	描述
函数名	tmr_output_channel_buffer_enable
函数原型	void tmr_output_channel_buffer_enable(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, confirm_state new_state);
功能描述	启用或禁用 TMR 输出通道缓冲区
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	new_state：将要配置的输出通道缓冲区状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

示例

```
tmr_output_channel_buffer_enable(TMR1, TMR_SELECT_CHANNEL_1, TRUE);
```

5.19.22 函数 tmr_output_channel_immediately_set

下表描述了函数 tmr_output_channel_immediately_set

表 404. 函数 tmr_output_channel_immediately_set

项目	描述
函数名	tmr_output_channel_immediately_set
函数原型	void tmr_output_channel_immediately_set(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, confirm_state new_state);
功能描述	设置 TMR 输出通道立即使能
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	new_state：将要配置的输出通道立即使能状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2
TMR_SELECT_CHANNEL_3：选择定时器通道 3
TMR_SELECT_CHANNEL_4：选择定时器通道 4

示例

```
tmr_output_channel_immediately_set(TMR1, TMR_SELECT_CHANNEL_1, TRUE);
```

5.19.23 函数 tmr_output_channel_switch_set

下表描述了函数 tmr_output_channel_switch_set

表 405. 函数 tmr_output_channel_switch_set

项目	描述
函数名	tmr_output_channel_switch_set
函数原型	void tmr_output_channel_switch_set(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, confirm_state new_state);
功能描述	设置 TMR 输出通道开关
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	new_state：将要配置的输出通道开关状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1
TMR_SELECT_CHANNEL_2：选择定时器通道 2
TMR_SELECT_CHANNEL_3：选择定时器通道 3
TMR_SELECT_CHANNEL_4：选择定时器通道 4

示例

```
tmr_output_channel_switch_set(TMR1, TMR_SELECT_CHANNEL_1, TRUE);
```

5.19.24 函数 tmr_one_cycle_mode_enable

下表描述了函数 tmr_one_cycle_mode_enable

表 406. 函数 tmr_one_cycle_mode_enable

项目	描述
函数名	tmr_one_cycle_mode_enable
函数原型	void tmr_one_cycle_mode_enable(tmr_type *tmr_x, confirm_state new_state);
功能描述	启用或禁用 TMR 单周期模式
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	new_state：将要配置的单周期模式状态，可选择启用（TRUE）或禁用

项目	描述
	（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_one_cycle_mode_enable(TMR1, TRUE);
```

5.19.25 函数 tmr_32_bit_function_enable

下表描述了函数 tmr_32_bit_function_enable

表 407. 函数 tmr_32_bit_function_enable

项目	描述
函数名	tmr_32_bit_function_enable
函数原型	void tmr_32_bit_function_enable(tmr_type *tmr_x, confirm_state new_state);
功能描述	启用或禁用 TMR 32 位功能（plus 模式）
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR2, TMR5
输入参数 2	new_state：将要配置的 32 位模式状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_32_bit_function_enable(TMR2, TRUE);
```

5.19.26 函数 tmr_overflow_request_source_set

下表描述了函数 tmr_overflow_request_source_set

表 408. 函数 tmr_overflow_request_source_set

项目	描述
函数名	tmr_overflow_request_source_set
函数原型	void tmr_overflow_request_source_set(tmr_type *tmr_x, confirm_state new_state);
功能描述	选择 TMR 溢出事件源
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	new_state：将要配置的溢出事件源
输出参数	无
返回值	无
先决条件	无
被调用函数	无

new_state

将要配置的溢出事件源

FALSE： 来源于计数器溢出、设置 OVFSWTR 位或次定时器控制器产生的溢出事件

TRUE： 只能来源于计数器溢出

示例

```
tmr_overflow_request_source_set(TMR1, TRUE);
```

5.19.27 函数 tmr_overflow_event_disable

下表描述了函数 tmr_overflow_event_disable

表 409. 函数 tmr_overflow_event_disable

项目	描述
函数名	tmr_overflow_event_disable
函数原型	void tmr_overflow_event_disable(tmr_type *tmr_x, confirm_state new_state);
功能描述	启用或禁用 TMR 溢出事件产生
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	new_state：将要配置的溢出事件产生状态
输出参数	无
返回值	无
先决条件	无
被调用函数	无

new_state

将要配置的溢出事件产生状态

FALSE：允许溢出事件产生，溢出事件可以由下列事件产生：

- 计数器溢出
- 将 OVFSWTR 位置 1
- 通过次定时器控制器产生的溢出事件

TRUE：禁止溢出事件产生

示例

```
tmr_overflow_event_disable(TMR1, TRUE);
```

5.19.28 函数 tmr_input_channel_init

下表描述了函数 tmr_input_channel_init

表 410. 函数 tmr_input_channel_init

项目	描述
函数名	tmr_input_channel_init
函数原型	void tmr_input_channel_init(tmr_type *tmr_x, tmr_input_config_type *input_struct, tmr_channel_input_divider_type divider_factor);
功能描述	初始化 TMR 输入通道
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	input_struct：指向结构体 tmr_input_config_type 的指针
输入参数 3	divider_factor：输入通道分频系数

项目	描述
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_input_config_type structure

tmr_input_config_type 在 at32f413_tmr.h 中

```
typedef struct
{
  tmr_channel_select_type          input_channel_select;
  tmr_input_polarity_type          input_polarity_select;
  tmr_input_direction_mapped_type  input_mapped_select;
  uint8_t                          input_filter_value;
} tmr_input_config_type;
```

input_channel_select

选择 TMR 输入通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

input_polarity_select

选择输入通道极性

TMR_INPUT_RISING_EDGE：　输入通道的有效边沿为上升沿

TMR_INPUT_FALLING_EDGE：　输入通道的有效边沿为下降沿

TMR_INPUT_BOTH_EDGE：　输入通道的有效边沿为上升沿和下降沿

input_mapped_select

选择输入通道映射

TMR_CC_CHANNEL_MAPPED_DIRECT：选择 TMR 输入通道 1，2，3 和 4 对应地与 C1IRAW，C2IRAW，C3IRAW 和 C4IRAW 相连

TMR_CC_CHANNEL_MAPPED_INDIRECT：选择 TMR 输入通道 1，2，3 和 4 对应地与 C2IRAW，C1IRAW，C4IRAW 和 C3IRAW 相连

TMR_CC_CHANNEL_MAPPED_STI：选择 TMR 输入通道映射在 STI 上

input_filter_value

配置输入通道滤波值，可取 0x00~0x0F

divider_factor

输入通道分频系数

TMR_CHANNEL_INPUT_DIV_1：输入通道分频系数为 1

TMR_CHANNEL_INPUT_DIV_2：输入通道分频系数为 2

TMR_CHANNEL_INPUT_DIV_4：输入通道分频系数为 4

TMR_CHANNEL_INPUT_DIV_8：输入通道分频系数为 8

示例

```
tmr_input_config_type tmr_input_config_struct;
tmr_input_config_struct.input_channel_select = TMR_SELECT_CHANNEL_2;
tmr_input_config_struct.input_mapped_select = TMR_CC_CHANNEL_MAPPED_DIRECT;
tmr_input_config_struct.input_polarity_select = TMR_INPUT_RISING_EDGE;
tmr_input_config_struct.input_filter_value = 0x00;
```

```
tmr_input_channel_init(TMR1, &tmr_input_config_struct, TMR_CHANNEL_INPUT_DIV_1);
```

5.19.29 函数 tmr_channel_enable

下表描述了函数 tmr_channel_enable

表 411. 函数 tmr_channel_enable

项目	描述
函数名	tmr_channel_enable
函数原型	void tmr_channel_enable(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, confirm_state new_state);
功能描述	启用或禁用 TMR 通道
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	new_state：将要配置的通道状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1： 选择定时器通道 1
TMR_SELECT_CHANNEL_1C： 选择定时器互补通道 1
TMR_SELECT_CHANNEL_2： 选择定时器通道 2
TMR_SELECT_CHANNEL_2C： 选择定时器互补通道 2
TMR_SELECT_CHANNEL_3： 选择定时器通道 3
TMR_SELECT_CHANNEL_3C： 选择定时器互补通道 3
TMR_SELECT_CHANNEL_4： 选择定时器通道 4

示例

```
tmr_channel_enable(TMR1, TMR_SELECT_CHANNEL_1, TRUE);
```

5.19.30 函数 tmr_input_channel_filter_set

下表描述了函数 tmr_input_channel_filter_set

表 412. 函数 tmr_input_channel_filter_set

项目	描述
函数名	tmr_input_channel_filter_set
函数原型	void tmr_input_channel_filter_set(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, uint16_t filter_value);
功能描述	设置 TMR 输入通道过滤器
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	filter_value：配置输入通道滤波值，可取 0x00~0x0F

项目	描述
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

示例

```
tmr_input_channel_filter_set(TMR1, TMR_SELECT_CHANNEL_1, 0x0F);
```

5.19.31 函数 tmr_pwm_input_config

下表描述了函数 tmr_pwm_input_config

表 413. 函数 tmr_pwm_input_config

项目	描述
函数名	tmr_pwm_input_config
函数原型	void tmr_pwm_input_config(tmr_type *tmr_x, tmr_input_config_type *input_struct, tmr_channel_input_divider_type divider_factor);
功能描述	配置 TMR pwm 输入
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	input_struct：指向结构体 tmr_input_config_type 的指针
输入参数 3	divider_factor：输入通道分频系数
输出参数	无
返回值	无
先决条件	无
被调用函数	无

input_struct

指向结构体 tmr_input_config_type 的指针，参考 *tmr_input_config_type* 查看取值范围

divider_factor

输入通道分频系数

TMR_CHANNEL_INPUT_DIV_1：输入通道分频系数为 1

TMR_CHANNEL_INPUT_DIV_2：输入通道分频系数为 2

TMR_CHANNEL_INPUT_DIV_4：输入通道分频系数为 4

TMR_CHANNEL_INPUT_DIV_8：输入通道分频系数为 8

示例

```
tmr_input_config_type tmr_ic_init_structure;
tmr_ic_init_structure.input_filter_value = 0;
tmr_ic_init_structure.input_channel_select = TMR_SELECT_CHANNEL_2;
tmr_ic_init_structure.input_mapped_select = TMR_CC_CHANNEL_MAPPED_DIRECT;
tmr_ic_init_structure.input_polarity_select = TMR_INPUT_RISING_EDGE;
```

```
tmr_pwm_input_config(TMR1, &tmr_ic_init_structure, TMR_CHANNEL_INPUT_DIV_1);
```

5.19.32 函数 tmr_channel1_input_select

下表描述了函数 tmr_channel1_input_select

表 414. 函数 tmr_channel1_input_select

项目	描述
函数名	tmr_channel1_input_select
函数原型	void tmr_channel1_input_select(tmr_type *tmr_x, tmr_channel1_input_connected_type ch1_connect);
功能描述	选择 TMR 通道 1 输入
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8
输入参数 2	ch1_connect：通道 1 输入选择
输出参数	无
返回值	无
先决条件	无
被调用函数	无

ch1_connect

将要配置的通道 1 输入连接

TMR_CHANEL1_CONNECTED_C1IRAW：将 CH1 管脚连到 C1IRAW 输入

TMR_CHANEL1_2_3_CONNECTED_C1IRAW_XOR：将 CH1、CH2 和 CH3 管脚异或结果连到 C1IRAW 输入

示例

```
tmr_channel1_input_select(TMR1, TMR_CHANEL1_2_3_CONNECTED_C1IRAW_XOR);
```

5.19.33 函数 tmr_input_channel_divider_set

下表描述了函数 tmr_input_channel_divider_set

表 415. 函数 tmr_input_channel_divider_set

项目	描述
函数名	tmr_input_channel_divider_set
函数原型	void tmr_input_channel_divider_set(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, tmr_channel_input_divider_type divider_factor);
功能描述	设置 TMR 输入通道分频器
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	divider_factor：输入通道分频系数
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

divider_factor

输入通道分频系数

TMR_CHANNEL_INPUT_DIV_1：输入通道分频系数为 1

TMR_CHANNEL_INPUT_DIV_2：输入通道分频系数为 2

TMR_CHANNEL_INPUT_DIV_4：输入通道分频系数为 4

TMR_CHANNEL_INPUT_DIV_8：输入通道分频系数为 8

示例

```
tmr_input_channel_divider_set(TMR1, TMR_SELECT_CHANNEL_1, TMR_CHANNEL_INPUT_DIV_2);
```

5.19.34 函数 tmr_primary_mode_select

下表描述了函数 tmr_primary_mode_select

表 416. 函数 tmr_primary_mode_select

项目	描述
函数名	tmr_primary_mode_select
函数原型	void tmr_primary_mode_select(tmr_type *tmr_x, tmr_primary_select_type primary_mode);
功能描述	选择 TMR 主模式
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一：TMR1, TMR2, TMR3, TMR4, TMR5, TMR8
输入参数 2	primary_mode：将要配置的主模式
输出参数	无
返回值	无
先决条件	无
被调用函数	无

primary_mode

将要配置的主模式，即主定时器输出信号选择

TMR_PRIMARY_SEL_RESET：　　主模式的输出信号选择复位

TMR_PRIMARY_SEL_ENABLE：　　主模式的输出信号选择使能

TMR_PRIMARY_SEL_OVERFLOW：　　主模式的输出信号选择溢出

TMR_PRIMARY_SEL_COMPARE：　　主模式的输出信号选择比较脉冲

TMR_PRIMARY_SEL_C1ORAW：　　主模式的输出信号选择 C1ORAW 信号

TMR_PRIMARY_SEL_C2ORAW：　　主模式的输出信号选择 C2ORAW 信号

TMR_PRIMARY_SEL_C3ORAW：　　主模式的输出信号选择 C3ORAW 信号

TMR_PRIMARY_SEL_C4ORAW：　　主模式的输出信号选择 C4ORAW 信号

示例

```
tmr_primary_mode_select(TMR1, TMR_PRIMARY_SEL_RESET);
```

5.19.35 函数 tmr_sub_mode_select

下表描述了函数 tmr_sub_mode_select

表 417. 函数 tmr_sub_mode_select

项目	描述
函数名	tmr_sub_mode_select
函数原型	void tmr_sub_mode_select(tmr_type *tmr_x, tmr_sub_mode_select_type sub_mode);
功能描述	选择 TMR 次定时器模式
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9
输入参数 2	sub_mode：将要配置的次定时器模式
输出参数	无
返回值	无
先决条件	无
被调用函数	无

primary_mode

选择要配置的次定时器模式

TMR_SUB_MODE_DIABLE：关闭次定时器模式
TMR_SUB_ENCODER_MODE_A：选择编码器模式 A
TMR_SUB_ENCODER_MODE_B：选择编码器模式 B
TMR_SUB_ENCODER_MODE_C：选择编码器模式 C
TMR_SUB_RESET_MODE：选择复位模式
TMR_SUB_HANG_MODE：选择挂起模式
TMR_SUB_TRIGGER_MODE：选择触发模式
TMR_SUB_EXTERNAL_CLOCK_MODE_A：选择外部时钟模式 A

示例

```
tmr_sub_mode_select(TMR1, TMR_SUB_HANG_MODE);
```

5.19.36 函数 tmr_channel_dma_select

下表描述了函数 tmr_channel_dma_select

表 418. 函数 tmr_channel_dma_select

项目	描述
函数名	tmr_channel_dma_select
函数原型	void tmr_channel_dma_select(tmr_type *tmr_x, tmr_dma_request_source_type cc_dma_select);
功能描述	选择 TMR 通道的 DMA 请求源
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9
输入参数 2	cc_dma_select：将要选择的 TMR 通道的 DMA 请求源
输出参数	无
返回值	无
先决条件	无

项目	描述
被调用函数	无

cc_dma_select

选择 TMR 通道的 DMA 请求源

TMR_DMA_REQUEST_BY_CHANNEL： 当发生通道事件（CxIF = 1）时产生 DMA 请求

TMR_DMA_REQUEST_BY_OVERFLOW： 当发生溢出事件（OVFIF = 1）时产生 DMA 请求

示例

```
tmr_channel_dma_select(TMR1, TMR_DMA_REQUEST_BY_OVERFLOW);
```

5.19.37 函数 tmr_hall_select

下表描述了函数 tmr_hall_select

表 419. 函数 tmr_hall_select

项目	描述
函数名	tmr_hall_select
函数原型	void tmr_hall_select(tmr_type *tmr_x, confirm_state new_state);
功能描述	选择 TMR hall 模式
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR8
输入参数 2	new_state：将要选择的 TMR hall 模式状态
输出参数	无
返回值	无
先决条件	无
被调用函数	无

new_state

选择 TMR hall 模式状态，用于通道控制位刷新选择

FALSE： 通过设置 HALL 位刷新控制位

TRUE： 通过设置 HALL 位或 TRGIN 的上升沿刷新控制位

示例

```
tmr_hall_select(TMR1, TRUE);
```

5.19.38 函数 tmr_channel_buffer_enable

下表描述了函数 tmr_channel_buffer_enable

表 420. 函数 tmr_channel_buffer_enable

项目	描述
函数名	tmr_channel_buffer_enable
函数原型	void tmr_channel_buffer_enable(tmr_type *tmr_x, confirm_state new_state);
功能描述	启用或禁用 TMR 通道缓冲区
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR8
输入参数 2	new_state：将要配置的 TMR 通道缓冲区状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无

项目	描述
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_channel_buffer_enable(TMR1, TRUE);
```

5.19.39 函数 tmr_trigger_input_select

下表描述了函数 tmr_trigger_input_select

表 421. 函数 tmr_trigger_input_select

项目	描述
函数名	tmr_trigger_input_select
函数原型	void tmr_trigger_input_select(tmr_type *tmr_x, sub_tmr_input_sel_type trigger_select);
功能描述	选择 TMR 次定时器触发输入
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一：TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9
输入参数 2	trigger_select：将要配置的 TMR 次定时器触发输入
输出参数	无
返回值	无
先决条件	无
被调用函数	无

trigger_select

选择 TMR 次定时器触发输入

TMR_SUB_INPUT_SEL_IS0： 选择内部输入 0
TMR_SUB_INPUT_SEL_IS1： 选择内部输入 1
TMR_SUB_INPUT_SEL_IS2： 选择内部输入 2
TMR_SUB_INPUT_SEL_IS3： 选择内部输入 3
TMR_SUB_INPUT_SEL_C1INC： 选择 C1IRAW 的输入检测器
TMR_SUB_INPUT_SEL_C1DF1： 选择滤波输入通道 1
TMR_SUB_INPUT_SEL_C2DF2： 选择滤波输入通道 2
TMR_SUB_INPUT_SEL_EXTIN： 选择外部输入通道 EXT

示例

```
tmr_trigger_input_select(TMR1, TMR_SUB_INPUT_SEL_IS0);
```

5.19.40 函数 tmr_sub_sync_mode_set

下表描述了函数 tmr_sub_sync_mode_set

表 422. 函数 tmr_sub_sync_mode_set

项目	描述
函数名	tmr_sub_sync_mode_set
函数原型	void tmr_sub_sync_mode_set(tmr_type *tmr_x, confirm_state new_state);
功能描述	设置 TMR 次定时器同步模式

项目	描述
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9
输入参数 2	new_state：将要配置的 TMR 次定时器同步模式状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_sub_sync_mode_set(TMR1, TRUE);
```

5.19.41 函数 tmr_dma_request_enable

下表描述了函数 tmr_dma_request_enable

表 423. 函数 tmr_dma_request_enable

项目	描述
函数名	tmr_dma_request_enable
函数原型	void tmr_dma_request_enable(tmr_type *tmr_x, tmr_dma_request_type dma_request, confirm_state new_state);
功能描述	启用或禁用 TMR DMA 请求
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	dma_request：将要配置的 DMA 请求
输入参数 3	new_state：将要配置的 DMA 请求状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

dma_request

设置 DMA 请求

TMR_OVERFLOW_DMA_REQUEST： 溢出事件的 DMA 请求
TMR_C1_DMA_REQUEST： 通道 1 的 DMA 请求
TMR_C2_DMA_REQUEST： 通道 2 的 DMA 请求
TMR_C3_DMA_REQUEST： 通道 3 的 DMA 请求
TMR_C4_DMA_REQUEST： 通道 4 的 DMA 请求
TMR_HALL_DMA_REQUEST： HALL 事件的 DMA 请求
TMR_TRIGGER_DMA_REQUEST： 触发事件的 DMA 请求

示例

```
tmr_dma_request_enable(TMR1, TMR_OVERFLOW_DMA_REQUEST, TRUE);
```

5.19.42 函数 tmr_interrupt_enable

下表描述了函数 tmr_interrupt_enable

表 424. 函数 tmr_interrupt_enable

项目	描述
函数名	tmr_interrupt_enable
函数原型	void tmr_interrupt_enable(tmr_type *tmr_x, uint32_t tmr_interrupt, confirm_state new_state);
功能描述	启用或禁用 TMR 中断
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_interrupt：将要配置的 TMR 中断
输入参数 3	new_state：将要配置的 TMR 中断状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_interrupt

设置 TMR 中断

TMR_OVF_INT： 溢出事件中断
TMR_C1_INT： 通道 1 事件中断
TMR_C2_INT： 通道 2 事件中断
TMR_C3_INT： 通道 3 事件中断
TMR_C4_INT： 通道 4 事件中断
TMR_HALL_INT： HALL 事件中断
TMR_TRIGGER_INT： 触发事件中断
TMR_BRK_INT： 刹车事件中断

示例

```
tmr_interrupt_enable(TMR1, TMR_OVF_INT, TRUE);
```

5.19.43 函数 tmr_flag_get

下表描述了函数 tmr_flag_get

表 425. 函数 tmr_flag_get

项目	描述
函数名	tmr_flag_get
函数原型	flag_status tmr_flag_get(tmr_type *tmr_x, uint32_t tmr_flag);
功能描述	获取标志位状态
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_flag：需要获取状态的标志选择 该参数详细描述见 tmr_flag
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET，RESET
先决条件	无
被调用函数	无

tmr_flag

用于选择需要获取状态的标志，其可选参数罗列如下

TMR_OVF_FLAG：　　溢出中断标记
TMR_C1_FLAG：　　通道 1 中断标记
TMR_C2_FLAG：　　通道 2 中断标记
TMR_C3_FLAG：　　通道 3 中断标记
TMR_C4_FLAG：　　通道 4 中断标记
TMR_HALL_FLAG：　　HALL 中断标记
TMR_TRIGGER_FLAG：　　触发中断标记
TMR_BRK_FLAG：　　刹车中断标记
TMR_C1_RECAPTURE_FLAG：　　通道 1 再捕获标记
TMR_C2_RECAPTURE_FLAG：　　通道 2 再捕获标记
TMR_C3_RECAPTURE_FLAG：　　通道 3 再捕获标记
TMR_C4_RECAPTURE_FLAG：　　通道 4 再捕获标记

示例

```
if(tmr_flag_get(TMR1, TMR_OVF_FLAG) != RESET)
```

5.19.44　函数 tmr_flag_clear

下表描述了函数 tmr_flag_clear

表 426. 函数 tmr_flag_clear

项目	描述
函数名	tmr_flag_clear
函数原型	void tmr_flag_clear(tmr_type *tmr_x, uint32_t tmr_flag);
功能描述	清除标志位
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_flag：待清除的标志选择 该参数详细描述见***错误!未找到引用源。***
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_flag_clear(TMR1, TMR_OVF_FLAG);
```

5.19.45　函数 tmr_event_sw_trigger

下表描述了函数 tmr_event_sw_trigger

表 427. 函数 tmr_event_sw_trigger

项目	描述
函数名	tmr_event_sw_trigger
函数原型	void tmr_event_sw_trigger(tmr_type *tmr_x, tmr_event_trigger_type tmr_event);
功能描述	软件触发 TMR 事件
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11

项目	描述
输入参数 2	tmr_event：将要通过软件触发的 TMR 事件
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_event

设置软件触发的 TMR 事件

TMR_OVERFLOW_SWTRIG： 软件触发溢出事件

TMR_C1_SWTRIG： 软件触发通道 1 事件

TMR_C2_SWTRIG： 软件触发通道 2 事件

TMR_C3_SWTRIG： 软件触发通道 3 事件

TMR_C4_SWTRIG： 软件触发通道 4 事件

TMR_HALL_SWTRIG： 软件触发 HALL 事件

TMR_TRIGGER_SWTRIG： 软件触发触发事件

TMR_BRK_SWTRIG： 软件触发刹车事件

示例

```
tmr_event_sw_trigger(TMR1, TMR_OVERFLOW_SWTRIG);
```

5.19.46 函数 tmr_output_enable

下表描述了函数 tmr_output_enable

表 428. 函数 tmr_output_enable

项目	描述
函数名	tmr_output_enable
函数原型	void tmr_output_enable(tmr_type *tmr_x, confirm_state new_state);
功能描述	启用或禁用 TMR 输出使能
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR8
输入参数 2	new_state：将要配置的 TMR 输出状态，可选择启用（TRUE）或禁用（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_output_enable(TMR1, TRUE);
```

5.19.47 函数 tmr_internal_clock_set

下表描述了函数 tmr_internal_clock_set

表 429. 函数 tmr_internal_clock_set

项目	描述
函数名	tmr_internal_clock_set
函数原型	void tmr_internal_clock_set(tmr_type *tmr_x);

项目	描述
功能描述	设置 TMR 内部时钟
输入参数	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
tmr_internal_clock_set(TMR1);
```

5.19.48 函数 tmr_output_channel_polarity_set

下表描述了函数 tmr_output_channel_polarity_set

表 430. 函数 tmr_output_channel_polarity_set

项目	描述
函数名	tmr_output_channel_polarity_set
函数原型	void tmr_output_channel_polarity_set(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, tmr_polarity_active_type oc_polarity);
功能描述	设置 TMR 输出通道极性
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	oc_polarity：将要配置的输出通道极性
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1： 选择定时器通道 1

TMR_SELECT_CHANNEL_1C： 选择定时器互补通道 1

TMR_SELECT_CHANNEL_2： 选择定时器通道 2

TMR_SELECT_CHANNEL_2C： 选择定时器互补通道 2

TMR_SELECT_CHANNEL_3： 选择定时器通道 3

TMR_SELECT_CHANNEL_3C： 选择定时器互补通道 3

TMR_SELECT_CHANNEL_4： 选择定时器通道 4

oc_polarity

设置 TMR 通道极性

TMR_POLARITY_ACTIVE_HIGH：输出通道极性高

TMR_POLARITY_ACTIVE_LOW： 输出通道极性低

示例

```
tmr_output_channel_polarity_set(TMR1, TMR_SELECT_CHANNEL_1, TMR_POLARITY_ACTIVE_HIGH);
```

5.19.49 函数 tmr_external_clock_config

下表描述了函数 tmr_external_clock_config

表 431. 函数 tmr_external_clock_config

项目	描述
函数名	tmr_external_clock_config
函数原型	void tmr_external_clock_config(tmr_type *tmr_x, tmr_external_signal_divider_type es_divide, tmr_external_signal_polarity_type es_polarity, uint16_t es_filter);
功能描述	配置 TMR 外部时钟
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8
输入参数 2	es_divide：外部信号分频系数
输入参数 3	es_polarity：外部信号极性
输入参数 4	es_filter：外部信号滤波值，可取 0x00~0x0F
输出参数	无
返回值	无
先决条件	无
被调用函数	无

es_divide

设置 TMR 外部信号分频系数

TMR_ES_FREQUENCY_DIV_1：外部信号分频系数为 1

TMR_ES_FREQUENCY_DIV_2：外部信号分频系数为 2

TMR_ES_FREQUENCY_DIV_4：外部信号分频系数为 4

TMR_ES_FREQUENCY_DIV_8：外部信号分频系数为 8

es_polarity

设置 TMR 外部信号极性

TMR_ES_POLARITY_NON_INVERTED： 外部信号极性为高电平或上升沿

TMR_ES_POLARITY_INVERTED： 外部信号极性为低电平或下降沿

示例

```
tmr_external_clock_config(TMR1, TMR_ES_FREQUENCY_DIV_1, TMR_ES_POLARITY_INVERTED, 0x0F);
```

5.19.50 函数 tmr_external_clock_mode1_config

下表描述了函数 tmr_external_clock_mode1_config

表 432. 函数 tmr_external_clock_mode1_config

项目	描述
函数名	tmr_external_clock_mode1_config
函数原型	void tmr_external_clock_mode1_config(tmr_type *tmr_x, tmr_external_signal_divider_type es_divide, tmr_external_signal_polarity_type es_polarity, uint16_t es_filter);
功能描述	配置 TMR 外部时钟模式 1（对应参考手册中的外部时钟模式 A）
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8
输入参数 2	es_divide：外部信号分频系数

项目	描述
输入参数 3	es_polarity：外部信号极性
输入参数 4	es_filter：外部信号滤波值，可取 0x00~0x0F
输出参数	无
返回值	无
先决条件	无
被调用函数	无

es_divide

设置 TMR 外部信号分频系数，参考 *es_divide* 查看取值范围

es_polarity

设置 TMR 外部信号极性，参考 *es_polarity* 查看取值范围

示例

```
tmr_external_clock_mode1_config(TMR1, TMR_ES_FREQUENCY_DIV_1, TMR_ES_POLARITY_INVERTED,
0x0F);
```

5.19.51 函数 tmr_external_clock_mode2_config

下表描述了函数 tmr_external_clock_mode2_config

表 433. 函数 tmr_external_clock_mode2_config

项目	描述
函数名	tmr_external_clock_mode2_config
函数原型	void tmr_external_clock_mode2_config(tmr_type *tmr_x, tmr_external_signal_divider_type es_divide, tmr_external_signal_polarity_type es_polarity, uint16_t es_filter);
功能描述	配置 TMR 外部时钟模式 2（对应参考手册中的外部时钟模式 B）
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一：TMR1, TMR2, TMR3, TMR4, TMR5, TMR8
输入参数 2	es_divide：外部信号分频系数
输入参数 3	es_polarity：外部信号极性
输入参数 4	es_filter：外部信号滤波值，可取 0x00~0x0F
输出参数	无
返回值	无
先决条件	无
被调用函数	无

es_divide

设置 TMR 外部信号分频系数，参考 *es_divide* 查看取值范围

es_polarity

设置 TMR 外部信号极性，参考 *es_polarity* 查看取值范围

示例

```
tmr_external_clock_mode2_config(TMR1, TMR_ES_FREQUENCY_DIV_1, TMR_ES_POLARITY_INVERTED,
0x0F);
```

5.19.52 函数 tmr_encoder_mode_config

下表描述了函数 tmr_encoder_mode_config

表 434. 函数 tmr_encoder_mode_config

项目	描述
函数名	tmr_encoder_mode_config
函数原型	void tmr_encoder_mode_config(tmr_type *tmr_x, tmr_encoder_mode_type encoder_mode, tmr_input_polarity_type ic1_polarity, tmr_input_polarity_type ic2_polarity);
功能描述	配置 TMR 编码器模式
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一：TMR1, TMR2, TMR3, TMR4, TMR5, TMR8
输入参数 2	encoder_mode：将要配置的编码器模式
输入参数 3	ic1_polarity：输入通道 1 极性
输入参数 4	Ic2_polarity：输入通道 2 极性
输出参数	无
返回值	无
先决条件	无
被调用函数	无

encoder_mode

设置 TMR 编码器模式

TMR_ENCODER_MODE_A：编码器模式 A

TMR_ENCODER_MODE_B：编码器模式 B

TMR_ENCODER_MODE_C：编码器模式 C

ic1_polarity

设置 TMR 输入通道 1 极性

TMR_INPUT_RISING_EDGE： 输入通道的有效边沿为上升沿

TMR_INPUT_FALLING_EDGE： 输入通道的有效边沿为下降沿

TMR_INPUT_BOTH_EDGE： 输入通道的有效边沿为上升沿和下降沿

ic2_polarity

设置 TMR 输入通道 2 极性

TMR_INPUT_RISING_EDGE： 输入通道的有效边沿为上升沿

TMR_INPUT_FALLING_EDGE： 输入通道的有效边沿为下降沿

TMR_INPUT_BOTH_EDGE： 输入通道的有效边沿为上升沿和下降沿

示例

```
tmr_encoder_mode_config(TMR1, TMR_ENCODER_MODE_A, TMR_INPUT_RISING_EDGE,
TMR_INPUT_RISING_EDGE);
```

5.19.53 函数 tmr_force_output_set

下表描述了函数 tmr_force_output_set

表 435. 函数 tmr_force_output_set

项目	描述
函数名	tmr_force_output_set
函数原型	void tmr_force_output_set(tmr_type *tmr_x, tmr_channel_select_type tmr_channel, tmr_force_output_type force_output);
功能描述	设置 TMR 强制输出
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一：

项目	描述
	TMR1, TMR2, TMR3, TMR4, TMR5, TMR8, TMR9, TMR10, TMR11
输入参数 2	tmr_channel：定时器通道
输入参数 3	force_output：强制输出电平
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_channel

设置 TMR 通道

TMR_SELECT_CHANNEL_1：选择定时器通道 1

TMR_SELECT_CHANNEL_2：选择定时器通道 2

TMR_SELECT_CHANNEL_3：选择定时器通道 3

TMR_SELECT_CHANNEL_4：选择定时器通道 4

force_output

输出通道的强制输出电平

TMR_FORCE_OUTPUT_HIGH： 强制 CxORAW 为高

TMR_FORCE_OUTPUT_LOW： 强制 CxORAW 为低

示例

```
tmr_force_output_set(TMR1, TMR_SELECT_CHANNEL_1, TMR_FORCE_OUTPUT_HIGH);
```

5.19.54 函数 tmr_dma_control_config

下表描述了函数 tmr_dma_control_config

表 436. 函数 tmr_dma_control_config

项目	描述
函数名	tmr_dma_control_config
函数原型	void tmr_dma_control_config(tmr_type *tmr_x, tmr_dma_transfer_length_type dma_length, tmr_dma_address_type dma_base_address);
功能描述	配置 TMR DMA 控制
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR2, TMR3, TMR4, TMR5, TMR8
输入参数 2	dma_length：DMA 传输字节数
输入参数 3	dma_base_address：DMA 传输偏移地址
输出参数	无
返回值	无
先决条件	无
被调用函数	无

dma_length

设置 DMA 传输字节数，共 18 个可选参数

TMR_DMA_TRANSFER_1BYTE： 1 个字节

TMR_DMA_TRANSFER_2BYTES： 2 个字节

TMR_DMA_TRANSFER_3BYTES： 3 个字节

…

TMR_DMA_TRANSFER_17BYTES： 17 个字节

TMR_DMA_TRANSFER_18BYTES： 18 个字节

dma_base_address

设置 DMA 传输偏移地址，从 TMR 控制寄存器 1 开始偏移，可选参数如下

TMR_CTRL1_ADDRESS
TMR_CTRL2_ADDRESS
TMR_STCTRL_ADDRESS
TMR_IDEN_ADDRESS
TMR_ISTS_ADDRESS
TMR_SWEVT_ADDRESS
TMR_CM1_ADDRESS
TMR_CM2_ADDRESS
TMR_CCTRL_ADDRESS
TMR_CVAL_ADDRESS
TMR_DIV_ADDRESS
TMR_PR_ADDRESS
TMR_RPR_ADDRESS
TMR_C1DT_ADDRESS
TMR_C2DT_ADDRESS
TMR_C3DT_ADDRESS
TMR_C4DT_ADDRESS
TMR_BRK_ADDRESS
TMR_DMACTRL_ADDRESS

示例

```
tmr_dma_control_config(TMR1, TMR_DMA_TRANSFER_8BYTES, TMR_CTRL1_ADDRESS);
```

5.19.55 函数 tmr_brkdt_config

下表描述了函数 tmr_brkdt_config

表 437. 函数 tmr_brkdt_config

项目	描述
函数名	tmr_brkdt_config
函数原型	void tmr_brkdt_config(tmr_type *tmr_x, tmr_brkdt_config_type *brkdt_struct);
功能描述	配置 TMR 刹车模式和死区时间
输入参数 1	tmr_x：所选择的 TMR 外设，该参数可以选取自其中之一： TMR1, TMR8
输入参数 2	brkdt_struct：指向结构体 tmr_brkdt_config_type 的指针
输出参数	无
返回值	无
先决条件	无
被调用函数	无

tmr_brkdt_config_type structure

tmr_brkdt_config_type 在 at32f413_tmr.h 中

```
typedef struct
{
  uint8_t                       deadtime;
```

```
  tmr_brk_polarity_type      brk_polarity;
  tmr_wp_level_type          wp_level;
  confirm_state              auto_output_enable;
  confirm_state              fcsoen_state;
  confirm_state              fcsodis_state;
  confirm_state              brk_enable;
} tmr_brkdt_config_type;
```

deadtime

设置死区时间，可取 0x00~0xFF

brk_polarity

选择刹车输入极性

TMR_BRK_INPUT_ACTIVE_LOW：刹车输入极性为低电平

TMR_BRK_INPUT_ACTIVE_HIGH：刹车输入极性为高电平

wp_level

设置写保护等级

TMR_WP_OFF：关闭写保护

TMR_WP_LEVEL_3：3 级写保护，以下 bit 位受写保护：

- TMRx_BRK：DTC、BRKEN、BRKV 和 AOEN
- TMRx_CTRL2：CxIOS 和 CxCIOS

TMR_WP_LEVEL_2：2 级写保护，除 3 级写保护的内容外，以下 bit 位也受写保护：

- TMRx_CCTRL：CxP 和 CxCP
- TMRx_BRK：FCSODIS 和 FCSOEN

TMR_WP_LEVEL_1：1 级写保护，除 2 级写保护的内容外，以下 bit 位也受写保护：

- TMRx_CMx：CxOCTRL 和 CxOBEN

auto_output_enable

自动输出使能，可选择启用（TRUE）或禁用（FALSE）

fcsoen_state

总输出开时的冻结状态，用于配置具有互补输出的通道，在定时器不工作且输出使能开启（OEN=1）时的通道状态

FALSE： 关闭 CxOUT/CxCOUT 输出

TRUE： 开启 CxOUT/CxCOUT 输出，输出为无效电平

fcsodis_state

总输出关时的冻结状态，用于配置具有互补输出的通道，在定时器不工作且且输出使能关闭（OEN=0）时的通道状态

FALSE： 关闭 CxOUT/CxCOUT 输出

TRUE： 开启 CxOUT/CxCOUT 输出，输出为空闲电平

brk_enable

刹车使能，可选择启用（TRUE）或禁用（FALSE）

示例

```
tmr_brkdt_config_type tmr_brkdt_config_struct;
tmr_brkdt_config_struct.brk_enable = TRUE;
tmr_brkdt_config_struct.auto_output_enable = TRUE;
tmr_brkdt_config_struct.deadtime = 0;
tmr_brkdt_config_struct.fcsodis_state = TRUE;
tmr_brkdt_config_struct.fcsoen_state = TRUE;
tmr_brkdt_config_struct.brk_polarity = TMR_BRK_INPUT_ACTIVE_HIGH;
```

```
tmr_brkdt_config_struct.wp_level = TMR_WP_OFF;
tmr_brkdt_config(TMR1, &tmr_brkdt_config_struct);
```

5.20 通用同步异步收发器（USART）

USART 寄存器结构 usart_type，定义于文件“at32f413_usart.h”如下：

```
/**
  * @brief type define usart register all
  */
typedef struct
{
      ...

} usart_type;
```

下表给出了 USART 寄存器总览：

表 438. USART 寄存器对应表

寄存器	描述
sts	状态寄存器
dt	数据寄存器
baudr	波特率寄存器
ctrl1	控制寄存器 1
ctrl2	控制寄存器 2
ctrl3	控制寄存器 3
gdiv	保护时间和预分频寄存器

下表给出了 USART 库函数总览：

表 439. USART 库函数总览

函数名	描述
usart_reset	将指定的 USART 外设的寄存器复位
usart_init	波特率、数据位和停止位等进行设定
usart_parity_selection_config	校验方式进行设定
usart_enable	外设使能设置
usart_transmitter_enable	外设发送使能设置
usart_receiver_enable	外设接收使能设置
usart_clock_config	同步功能的时钟极性、相位等进行设置
usart_clock_enable	同步功能时钟输出使能设置
usart_interrupt_enable	中断使能设置
usart_dma_transmitter_enable	DMA 发送使能设置
usart_dma_receiver_enable	DMA 接收使能设置
usart_wakeup_id_set	唤醒 ID 设置
usart_wakeup_mode_set	唤醒模式设置
usart_receiver_mute_enable	接收器静默模式使能
usart_break_bit_num_set	断开帧长度设置

usart_lin_mode_enable	lin 模式使能设置
usart_data_transmit	数据发送
usart_data_receive	数据接收
usart_break_send	发送断开帧设置
usart_smartcard_guard_time_set	智能卡模式保护时间设置
usart_irda_smartcard_division_set	红外和智能卡模式的分频设置
usart_smartcard_mode_enable	智能卡模式使能设置
usart_smartcard_nack_set	智能卡模式的 NACK 使能设置
usart_single_line_halfduplex_select	单线半双工模式使能设置
usart_irda_mode_enable	红外模式使能设置
usart_irda_low_power_enable	红外模式低功耗使能设置
usart_hardware_flow_control_set	外设硬件流控使能设置
usart_flag_get	检查指定的 flag 状态是否置起
usart_flag_clear	清除指定的 flag 状态标志

5.20.1 函数 usart_reset

下表描述了函数 usart_reset

表 440. 函数 usart_reset

项目	描述
函数名	usart_reset
函数原型	void usart_reset(usart_type* usart_x);
功能描述	将指定的 USART 外设的寄存器复位
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	crm_periph_reset

示例

```
/* reset usart1 */
usart_reset(USART1);
```

5.20.2 函数 usart_init

下表描述了函数 usart_init

表 441. 函数 usart_init

项目	描述
函数名	usart_init
函数原型	void usart_init(usart_type* usart_x, uint32_t baud_rate, usart_data_bit_num_type data_bit, usart_stop_bit_num_type stop_bit);
功能描述	波特率、数据位和停止位等进行设定
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	baud_rate：串口使用的通讯波特率

项目	描述
输入参数 3	data_bit：串口数据位宽度
输入参数 4	stop_bit：串口停止位宽度
输出参数	无
返回值	无
先决条件	外部低速时钟在未使能的情况下进行设定
被调用函数	无

data_bit

串口通讯采用的数据位宽度

USART_DATA_8BITS：数据位宽度为 8-bit

USART_DATA_9BITS：数据位宽度为 9-bit

stop_bit

串口通讯采用的停止位宽度

USART_STOP_1_BIT：　　停止位宽度为 1 个 bit

USART_STOP_0_5_BIT：　　停止位宽度为 0.5 个 bit

USART_STOP_2_BIT：　　停止位宽度为 2 个 bit

USART_STOP_1_5_BIT：　　停止位宽度为 1.5 个 bit

示例

```
/* configure uart param */
usart_init(USART1, 115200, USART_DATA_8BITS, USART_STOP_1_BIT);
```

5.20.3 函数 usart_parity_selection_config

下表描述了函数 usart_parity_selection_config

表 442. 函数 usart_parity_selection_config

项目	描述
函数名	usart_parity_selection_config
函数原型	void usart_parity_selection_config(usart_type* usart_x, usart_parity_selection_type parity);
功能描述	校验方式进行设定
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	parity：串口通讯采用的数据校验方式
输出参数	无
返回值	无
先决条件	无
被调用函数	无

parity

串口通讯采用的数据校验方式

USART_PARITY_NONE：　　无校验

USART_PARITY_EVEN：　　偶校验

USART_PARITY_ODD：　　奇校验

示例

```
/* config usart even parity */
usart_parity_selection_config(USART1, USART_PARITY_EVEN);
```

5.20.4 函数 usart_enable

下表描述了函数 usart_enable

表 443. 函数 usart_enable

项目	描述
函数名	usart_enable
函数原型	void usart_enable(usart_type* usart_x, confirm_state new_state);
功能描述	外设使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable usart1 */
usart_enable(USART1, TRUE);
```

5.20.5 函数 usart_transmitter_enable

下表描述了函数 usart_transmitter_enable

表 444. 函数 usart_transmitter_enable

项目	描述
函数名	usart_transmitter_enable
函数原型	void usart_transmitter_enable(usart_type* usart_x, confirm_state new_state);
功能描述	外设发送使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable usart1 transmitter */
usart_transmitter_enable(USART1, TRUE);
```

5.20.6 函数 usart_receiver_enable

下表描述了函数 usart_receiver_enable

表 445. 函数 usart_receiver_enable

项目	描述
函数名	usart_receiver_enable
函数原型	void usart_receiver_enable(usart_type* usart_x, confirm_state new_state);
功能描述	外设接收使能设置

项目	描述
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable usart1 receiver */
usart_ receiver _enable(USART1, TRUE);
```

5.20.7 函数 usart_clock_config

下表描述了函数 usart_clock_config

表 446. 函数 usart_clock_config

项目	描述
函数名	usart_clock_config
函数原型	void usart_clock_config(usart_type* usart_x, usart_clock_polarity_type clk_pol, usart_clock_phase_type clk_pha, usart_lbcp_type clk_lb);
功能描述	同步功能的时钟极性、相位等进行设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	clk_pol：同步功能所使用的时钟极性
输入参数 3	clk_pha：同步功能所使用的时钟相位
输入参数 4	clk_lb：同步功能发送一笔数据的最后一个 bit 数据（最高位）的时钟是否输出
输出参数	无
返回值	无
先决条件	无
被调用函数	无

clk_pol

同步功能时钟极性

USART_CLOCK_POLARITY_LOW：时钟极性为低电平

USART_CLOCK_POLARITY_HIGH：时钟极性为高电平

clk_pha

同步功能时钟相位

USART_CLOCK_PHASE_1EDGE：时钟相位为第一个沿

USART_CLOCK_PHASE_2EDGE：时钟相位为第二个沿

clk_lb

同步功能数据最后一个 bit 时钟设定

USART_CLOCK_LAST_BIT_NONE： 无时钟输出

USART_CLOCK_LAST_BIT_OUTPUT：有时钟输出

示例

```
/* config synchronous mode */
usart_clock_config(USART1, USART_CLOCK_POLARITY_HIGH, USART_CLOCK_PHASE_2EDGE,
USART_CLOCK_LAST_BIT_OUTPUT);
```

5.20.8 函数 usart_clock_enable

下表描述了函数 usart_clock_enable

表 447. 函数 usart_clock_enable

项目	描述
函数名	usart_clock_enable
函数原型	void usart_clock_enable(usart_type* usart_x, confirm_state new_state);
功能描述	同步功能时钟输出使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable clock */
usart_clock_enable(USART1, TRUE);
```

5.20.9 函数 usart_interrupt_enable

下表描述了函数 usart_interrupt_enable

表 448. 函数 usart_interrupt_enable

项目	描述
函数名	usart_interrupt_enable
函数原型	void usart_interrupt_enable(usart_type* usart_x, uint32_t usart_int, confirm_state new_state);
功能描述	中断使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	usart_int：指定的中断类型
输入参数 3	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

usart_int

指定的外设中断。

USART_IDLE_INT： 总线空闲中断
USART_RDBF_INT： 接收数据 buff 满中断
USART_TDC_INT： 发送数据完成中断
USART_TDBE_INT： 发送数据 buff 空中断
USART_PERR_INT： 校验错误中断
USART_BF_INT： 断开帧接收中断
USART_ERR_INT： 错误中断
USART_CTSCF_INT： CTS（Clear To Send 清除发送）变化中断

示例

```
/* enable usart1 transmit complete interrupt */
usart_interrupt_enable (USART1, USART_TDC_INT, TRUE);
```

5.20.10 函数 usart_dma_transmitter_enable

下表描述了函数 usart_dma_transmitter_enable

表 449. 函数 usart_dma_transmitter_enable

项目	描述
函数名	usart_dma_transmitter_enable
函数原型	void usart_dma_transmitter_enable(usart_type* usart_x, confirm_state new_state);
功能描述	DMA 发送使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable dma transmitter */
usart_dma_transmitter_enable (USART1, TRUE);
```

5.20.11 函数 usart_dma_receiver_enable

下表描述了函数 usart_dma_receiver_enable

表 450. 函数 usart_dma_receiver_enable

项目	描述
函数名	usart_dma_receiver_enable
函数原型	void usart_dma_receiver_enable(usart_type* usart_x, confirm_state new_state);
功能描述	DMA 接收使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable dma receiver */
usart_dma_receiver_enable (USART1, TRUE);
```

5.20.12 函数 usart_wakeup_id_set

下表描述了函数 usart_wakeup_id_set

表 451. 函数 usart_wakeup_id_set

项目	描述
函数名	usart_wakeup_id_set
函数原型	void usart_wakeup_id_set(usart_type* usart_x, uint8_t usart_id);
功能描述	唤醒 ID 设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	usart_id：需要设置的唤醒 ID
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* config wakeup id */
usart_wakeup_id_set (USART1, 0x88);
```

5.20.13 函数 usart_wakeup_mode_set

下表描述了函数 usart_wakeup_mode_set

表 452. 函数 usart_wakeup_mode_set

项目	描述
函数名	usart_wakeup_mode_set
函数原型	void usart_wakeup_mode_set(usart_type* usart_x, usart_wakeup_mode_type wakeup_mode);
功能描述	唤醒模式设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	wakeup_mode：设置的唤醒模式
输出参数	无
返回值	无
先决条件	无
被调用函数	无

wakeup_mode

从静默状态的下唤醒的模式设置

USART_WAKEUP_BY_IDLE_FRAME： 接收到空闲帧唤醒

USART_WAKEUP_BY_MATCHING_ID： 接收到匹配 ID 进行唤醒

示例

```
/* config usart1 wakeup mode */
usart_wakeup_mode_set (USART1, USART_WAKEUP_BY_MATCHING_ID);
```

5.20.14 函数 usart_receiver_mute_enable

下表描述了函数 usart_receiver_mute_enable

表 453. 函数 usart_receiver_mute_enable

项目	描述
函数名	usart_receiver_mute_enable

项目	描述
函数原型	void usart_receiver_mute_enable(usart_type* usart_x, confirm_state new_state);
功能描述	接收器静默模式使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* config receiver mute */
usart_receiver_mute_enable (USART1, TRUE);
```

5.20.15 函数 usart_break_bit_num_set

下表描述了函数 usart_break_bit_num_set

表 454. 函数 usart_break_bit_num_set

项目	描述
函数名	usart_break_bit_num_set
函数原型	void usart_break_bit_num_set(usart_type* usart_x, usart_break_bit_num_type break_bit);
功能描述	断开帧长度设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	break_bit：断开帧长度类型
输出参数	无
返回值	无
先决条件	无
被调用函数	无

break_bit

断开帧长度设定

USART_BREAK_10BITS：断开帧长度设定为 10 个 bit

USART_BREAK_11BITS：断开帧长度设定为 11 个 bit

示例

```
/* config break frame length 10bits */
usart_break_bit_num_set (USART1, USART_BREAK_10BITS);
```

5.20.16 函数 usart_lin_mode_enable

下表描述了函数 usart_lin_mode_enable

表 455. 函数 usart_lin_mode_enable

项目	描述
函数名	usart_lin_mode_enable
函数原型	void usart_lin_mode_enable(usart_type* usart_x, confirm_state new_state);
功能描述	lin 模式使能设置

项目	描述
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable usart1 lin mode */
usart_lin_mode_enable (USART1, TRUE);
```

5.20.17 函数 usart_data_transmit

下表描述了函数 usart_data_transmit

表 456. 函数 usart_data_transmit

项目	描述
函数名	usart_data_transmit
函数原型	void usart_data_transmit(usart_type* usart_x, uint16_t data);
功能描述	数据发送
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	data：需要发送数据
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* transmit data */
uint16_t data = 0x88;
usart_data_transmit (USART1, data);
```

5.20.18 函数 usart_data_receive

下表描述了函数 usart_data_receive

表 457. 函数 usart_data_receive

项目	描述
函数名	usart_data_receive
函数原型	uint16_t usart_data_receive(usart_type* usart_x);
功能描述	数据接收
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	无
输出参数	无
返回值	uint16_t：返回接收到的数据
先决条件	无
被调用函数	无

示例

```
/* receive data */
uint16_t data = 0;
data = usart_data_receive (USART1);
```

5.20.19 函数 usart_break_send

下表描述了函数 usart_break_send

表 458. 函数 usart_break_send

项目	描述
函数名	usart_break_send
函数原型	void usart_break_send(usart_type* usart_x);
功能描述	发送断开帧设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* send break frame */
usart_break_send (USART1);
```

5.20.20 函数 usart_smartcard_guard_time_set

下表描述了函数 usart_smartcard_guard_time_set

表 459. 函数 usart_smartcard_guard_time_set

项目	描述
函数名	usart_smartcard_guard_time_set
函数原型	void usart_smartcard_guard_time_set(usart_type* usart_x, uint8_t guard_time_val);
功能描述	智能卡模式保护时间设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	guard_time_val：需要设置的保护时间，范围：0x00~0xFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* usart guard time set to 2 bit */
usart_smartcard_guard_time_set(USART1, 0x2);
```

5.20.21 函数 usart_irda_smartcard_division_set

下表描述了函数 usart_irda_smartcard_division_set

表 460. 函数 usart_irda_smartcard_division_set

项目	描述
函数名	usart_irda_smartcard_division_set
函数原型	void usart_irda_smartcard_division_set(usart_type* usart_x, uint8_t div_val);
功能描述	红外和智能卡模式的分频设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	div_val：分频值
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* usart clock set to (apbclk / (2 * 20)) */
usart_irda_smartcard_division_set(USART1, 20);
```

5.20.22 函数 usart_smartcard_mode_enable

下表描述了函数 usart_smartcard_mode_enable

表 461. 函数 usart_smartcard_mode_enable

项目	描述
函数名	usart_smartcard_mode_enable
函数原型	void usart_smartcard_mode_enable(usart_type* usart_x, confirm_state new_state);
功能描述	智能卡模式使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable the smartcard mode */
usart_smartcard_mode_enable(USART1, TRUE);
```

5.20.23 函数 usart_smartcard_nack_set

下表描述了函数 usart_smartcard_nack_set

表 462. 函数 usart_smartcard_nack_set

项目	描述
函数名	usart_smartcard_nack_set
函数原型	void usart_smartcard_nack_set(usart_type* usart_x, confirm_state new_state);
功能描述	智能卡模式的 NACK 使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）

项目	描述
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable the nack transmission */
usart_smartcard_nack_set(USART1, TRUE);
```

5.20.24 函数 usart_single_line_halfduplex_select

下表描述了函数 usart_single_line_halfduplex_select

表 463. 函数 usart_single_line_halfduplex_select

项目	描述
函数名	usart_single_line_halfduplex_select
函数原型	void usart_single_line_halfduplex_select(usart_type* usart_x, confirm_state new_state);
功能描述	单线半双工模式使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable halfduplex */
usart_single_line_halfduplex_select(USART1, TRUE);
```

5.20.25 函数 usart_irda_mode_enable

下表描述了函数 usart_irda_mode_enable

表 464. 函数 usart_irda_mode_enable

项目	描述
函数名	usart_irda_mode_enable
函数原型	void usart_irda_mode_enable(usart_type* usart_x, confirm_state new_state);
功能描述	红外模式使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable irda mode */
```

```
usart_irda_mode_enable(USART1, TRUE);
```

5.20.26 函数 usart_irda_low_power_enable

下表描述了函数 usart_irda_low_power_enable

表 465. 函数 usart_irda_low_power_enable

项目	描述
函数名	usart_irda_low_power_enable
函数原型	void usart_irda_low_power_enable(usart_type* usart_x, confirm_state new_state);
功能描述	红外模式低功耗使能设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	new_state：设置的新状态，使能（TRUE）失能（FALSE）
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
/* enable irda lowpower mode */
usart_irda_low_power_enable (USART1, TRUE);
```

5.20.27 函数 usart_hardware_flow_control_set

下表描述了函数 usart_hardware_flow_control_set

表 466. 函数 usart_hardware_flow_control_set

项目	描述
函数名	usart_hardware_flow_control_set
函数原型	void usart_hardware_flow_control_set(usart_type* usart_x,usart_hardware_flow_control_type flow_state);
功能描述	外设硬件流控设置
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	flow_state：流控类型设置
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flow_state

USART_HARDWARE_FLOW_NONE： 无硬件流控
USART_HARDWARE_FLOW_RTS： 硬件流控仅使用 rts
USART_HARDWARE_FLOW_CTS： 硬件流控仅使用 cts
USART_HARDWARE_FLOW_RTS_CTS： 硬件流控 rts 与 cts 共同使用

示例

```
/* hardware flow set none */
usart_hardware_flow_control_set (USART1, USART_HARDWARE_FLOW_NONE);
```

5.20.28 函数 usart_flag_get

下表描述了函数 usart_flag_get

表 467. 函数 usart_flag_get

项目	描述
函数名	usart_flag_get
函数原型	flag_status usart_flag_get(usart_type* usart_x, uint32_t flag);
功能描述	检查指定的 flag 状态是否置起
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	flag：需检查的 flag 标志
输出参数	无
返回值	flag_status：返回状态标志，置起（SET），未置起（RESET）
先决条件	无
被调用函数	无

flag

USART_CTSCF_FLAG： CTS（Clear To Send 清除发送）变化标志
USART_BFF_FLAG： 断开帧接收标志
USART_TDBE_FLAG： 发送 buff 空标志
USART_TDC_FLAG： 发送完成标志
USART_RDBF_FLAG： 接收数据 buff 满标志
USART_IDLEF_FLAG： 空闲帧标志
USART_ROERR_FLAG： 接收溢出标志
USART_NERR_FLAG： 噪声错误标志
USART_FERR_FLAG： 帧错误标志
USART_PERR_FLAG： 数据校验错误标志

示例

```
/* wait data transmit complete flag */
while(usart_flag_get (USART1, USART_TDC_FLAG) == RESET);
```

5.20.29 函数 usart_flag_clear

下表描述了函数 usart_flag_clear

表 468. 函数 usart_flag_clear

项目	描述
函数名	usart_flag_clear
函数原型	void usart_flag_clear(usart_type* usart_x, uint32_t flag);
功能描述	清除指定的 flag 状态标志
输入参数 1	usart_x：指定的串口外设，如：USART1、USART2、USART3…
输入参数 2	flag：指定清除的 flag 标志
输出参数	无
返回值	无
先决条件	无
被调用函数	无

flag

USART_CTSCF_FLAG：CTS（Clear To Send 清除发送）变化标志
USART_BFF_FLAG： 断开帧接收标志
USART_TDC_FLAG： 发送完成标志
USART_RDBF_FLAG： 接收数据 buff 满标志
示例

```
/* clear data transmit complete flag */
usart_flag_clear (USART1, USART_TDC_FLAG );
```

5.21 看门狗（WDT）

WDT 寄存器结构 wdt_type，定义于文件“at32f413_wdt.h”如下：

```
/**
  * @brief type define wdt register all
  */
typedef struct
{


} wdt_type;
```

下表给出了 WDT 寄存器总览：

表 469. WDT 寄存器对应表

寄存器	描述
cmd	命令寄存器
div	预分频寄存器
rld	重装载寄存器
sts	状态寄存器

下表给出了 WDT 库函数总览：

表 470. WDT 库函数总览

函数名	描述
wdt_enable	看门狗使能
wdt_counter_reload	重载计数器
wdt_reload_value_set	设置重载值
wdt_divider_set	设置分频值
wdt_register_write_enable	解锁 WDT_DIV、WDT_RLD 寄存器写保护
wdt_flag_get	获取标志

5.21.1 函数 wdt_enable

下表描述了函数 wdt_enable

表 471. 函数 wdt_enable

项目	描述
函数名	wdt_enable

项目	描述
函数原型	void wdt_enable(void);
功能描述	看门狗使能
输入参数 1	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wdt_enable();
```

5.21.2 函数 wdt_counter_reload

下表描述了函数 wdt_counter_reload

表 472. 函数 wdt_counter_reload

项目	描述
函数名	wdt_counter_reload
函数原型	void wdt_counter_reload(void);
功能描述	重载计数器
输入参数 1	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wdt_counter_reload();
```

5.21.3 函数 wdt_reload_value_set

下表描述了函数 wdt_reload_value_set

表 473. 函数 wdt_reload_value_set

项目	描述
函数名	wdt_reload_value_set
函数原型	void wdt_reload_value_set(uint16_t reload_value);
功能描述	设置重载值
输入参数 1	reload_value：重载值，范围 0x000~0xFFF
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wdt_reload_value_set(0xFFF);
```

5.21.4 函数 wdt_divider_set

下表描述了函数 wdt_divider_set

表 474. 函数 wdt_divider_set

项目	描述
函数名	wdt_divider_set
函数原型	void wdt_divider_set(wdt_division_type division);
功能描述	设置分频值
输入参数 1	division：看门狗分频值 参阅章节：division 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

division

看门狗分频值

WDT_CLK_DIV_4： 4 分频
WDT_CLK_DIV_8： 8 分频
WDT_CLK_DIV_16： 16 分频
WDT_CLK_DIV_32： 32 分频
WDT_CLK_DIV_64： 64 分频
WDT_CLK_DIV_128： 128 分频
WDT_CLK_DIV_256： 256 分频

示例

```
wdt_divider_set(WDT_CLK_DIV_4);
```

5.21.5 函数 wdt_register_write_enable

下表描述了函数 wdt_register_write_enable

表 475. 函数 wdt_register_write_enable

项目	描述
函数名	wdt_register_write_enable
函数原型	void wdt_register_write_enable(confirm_state new_state);
功能描述	解锁 WDT_DIV、WDT_RLD 寄存器写保护
输入参数 1	new_state：寄存器解锁使能 该参数可以选取自其中之一：TRUE、FALSE
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wdt_register_write_enable(TRUE);
```

5.21.6 函数 wdt_flag_get

下表描述了函数 wdt_flag_get

表 476. 函数 wdt_flag_get

项目	描述
函数名	wdt_flag_get
函数原型	flag_status wdt_flag_get(uint16_t wdt_flag);
功能描述	获取标志位状态
输入参数 1	flag：需要获取状态的标志选择 该参数详细描述见 flag
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET、RESET
先决条件	无
被调用函数	无

flag

用于选择需要获取状态的标志，其可选参数罗列如下

WDT_DIVF_UPDATE_FLAG： 分频值更新完成标志

WDT_RLDF_UPDATE_FLAG： 重载值更新完成标志

示例

```
wdt_flag_get(WDT_DIVF_UPDATE_FLAG);
```

5.22 窗口看门狗（WWDT）

WWDT 寄存器结构 wwdt_type，定义于文件“at32f413_wwdt.h”如下：

```
/**
  * @brief type define wwdt register all
  */
typedef struct
{


} wwdt_type;
```

下表给出了 WWDT 寄存器总览：

表 477. WWDT 寄存器对应表

寄存器	描述
ctrl	控制寄存器
cfg	配置寄存器
sts	状态寄存器

下表给出了 WWDT 库函数总览：

表 478. WWDT 库函数总览

函数名	描述

wwdt_reset	窗口看门狗寄存器复位
wwdt_divider_set	分频器设置
wwdt_flag_clear	清除重载计数器中断标志
wwdt_enable	窗口看门狗使能
wwdt_interrupt_enable	重载计数器中断使能
wwdt_flag_get	标志获取
wwdt_counter_set	计数值设置
wwdt_window_counter_set	窗口值设置

5.22.1 函数 wwdt_reset

下表描述了函数 wwdt_reset

表 479. 函数 wwdt_reset

项目	描述
函数名	wwdt_reset
函数原型	void wwdt_reset(void);
功能描述	窗口看门狗复位，将窗口看门狗寄存器复位成初始值
输入参数 1	无
输出参数	无
返回值	无
先决条件	无
被调用函数	void crm_periph_reset(crm_periph_reset_type value, confirm_state new_state);

示例

```
wwdt_reset();
```

5.22.2 函数 wwdt_divider_set

下表描述了函数 wwdt_divider_set

表 480. 函数 wwdt_divider_set

项目	描述
函数名	wwdt_divider_set
函数原型	void wwdt_divider_set(wwdt_division_type division);
功能描述	分频器设置
输入参数 1	division：窗口看门狗分频值 参阅章节：division 查阅更多该参数允许取值范围
输出参数	无
返回值	无
先决条件	无
被调用函数	无

division

窗口看门狗分频值

WWDT_PCLK1_DIV_4096： 4096 分频

WWDT_PCLK1_DIV_8192： 8192 分频

WWDT_PCLK1_DIV_16384： 16384 分频

WWDT_PCLK1_DIV_32768： 32768 分频

示例

```
wwdt_divider_set(WWDT_PCLK1_DIV_4096);
```

5.22.3 函数 wwdt_enable

下表描述了函数 wwdt_enable

表 481. 函数 wwdt_enable

项目	描述
函数名	wwdt_enable
函数原型	void wwdt_enable(uint8_t wwdt_cnt);
功能描述	窗口看门狗使能
输入参数 1	wwdt_cnt：窗口看门狗计数器初值，范围 0x40~0x7F
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wwdt_enable(0x7F);
```

5.22.4 函数 wwdt_interrupt_enable

下表描述了函数 wwdt_interrupt_enable

表 482. 函数 wwdt_interrupt_enable

项目	描述
函数名	wwdt_interrupt_enable
函数原型	void wwdt_interrupt_enable(void);
功能描述	重载计数器中断使能
输入参数 1	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wwdt_interrupt_enable();
```

5.22.5 函数 wwdt_counter_set

下表描述了函数 wwdt_counter_set

表 483. 函数 wwdt_counter_set

项目	描述
函数名	wwdt_counter_set
函数原型	void wwdt_counter_set(uint8_t wwdt_cnt);
功能描述	计数值设置

项目	描述
输入参数 1	wwdt_cnt：窗口看门狗计数值，范围 0x40~0x7F
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wwdt_counter_set(0x7F);
```

5.22.6 函数 wwdt_window_counter_set

下表描述了函数 wwdt_window_counter_set

表 484. 函数 wwdt_window_counter_set

项目	描述
函数名	wwdt_window_counter_set
函数原型	void wwdt_window_counter_set(uint8_t window_cnt);
功能描述	窗口值设置
输入参数 1	wwdt_cnt：窗口看门狗窗口值，范围 0x40~0x7F
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wwdt_window_counter_set(0x6F);
```

5.22.7 函数 wwdt_flag_get

下表描述了函数 wwdt_flag_get

表 485. 函数 wwdt_flag_get

项目	描述
函数名	wwdt_flag_get
函数原型	flag_status wwdt_flag_get(void);
功能描述	获取重载计数器中断标志状态
输入参数 1	无
输出参数	无
返回值	flag_status：标志位的状态 该返回值可为其中之一：SET、RESET
先决条件	无
被调用函数	无

示例

```
wwdt_flag_get();
```

5.22.8 函数 wwdt_flag_clear

下表描述了函数 wwdt_flag_clear

表 486. 函数 wwdt_flag_clear

项目	描述
函数名	wwdt_flag_clear
函数原型	void wwdt_flag_clear(void);
功能描述	清除重载计数器中断标志
输入参数 1	无
输出参数	无
返回值	无
先决条件	无
被调用函数	无

示例

```
wwdt_flag_clear();
```

6 注意事项

6.1 型号切换

如若需要在已有的工程或 demo 中进行型号切换时需注意型号宏定义及 device 名的同步修改，在修改前请详细查看文中**型号宏定义对应表**内容，其中详细罗列了 MCU 型号与宏定义的对应关系。
接下来将对两种常用的开发环境的修改方法来进行介绍（以下内容以 at32f403avgt7 作为示例与图示，其余系列或型号的修改方法与此类似），修改方法主要有两步：1、改 device，2、改型号宏定义。

6.1.1 KEIL 上型号切换

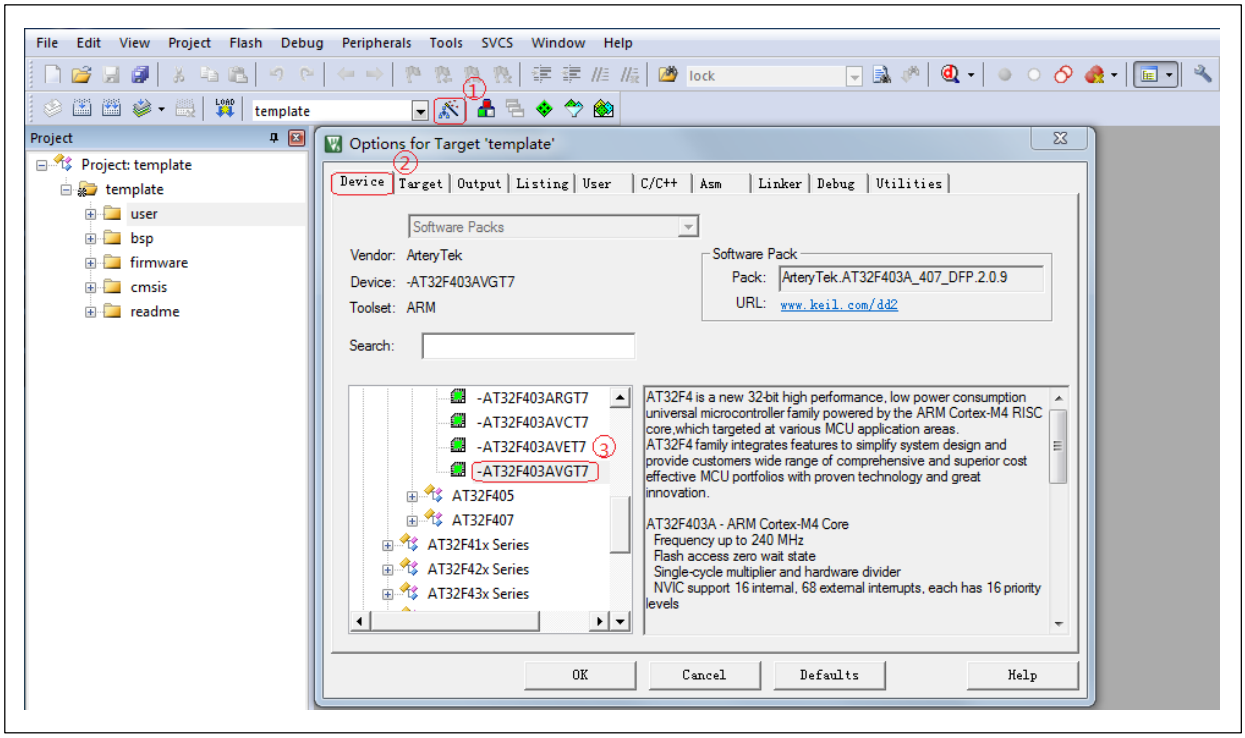
修改 device，操作步骤和图示如下：

① 点击魔术棒“Options for Target”。

② 点击 Device 选项卡。

③ 选择需要切换的 Device 型号。

图 29. Keil 改 device



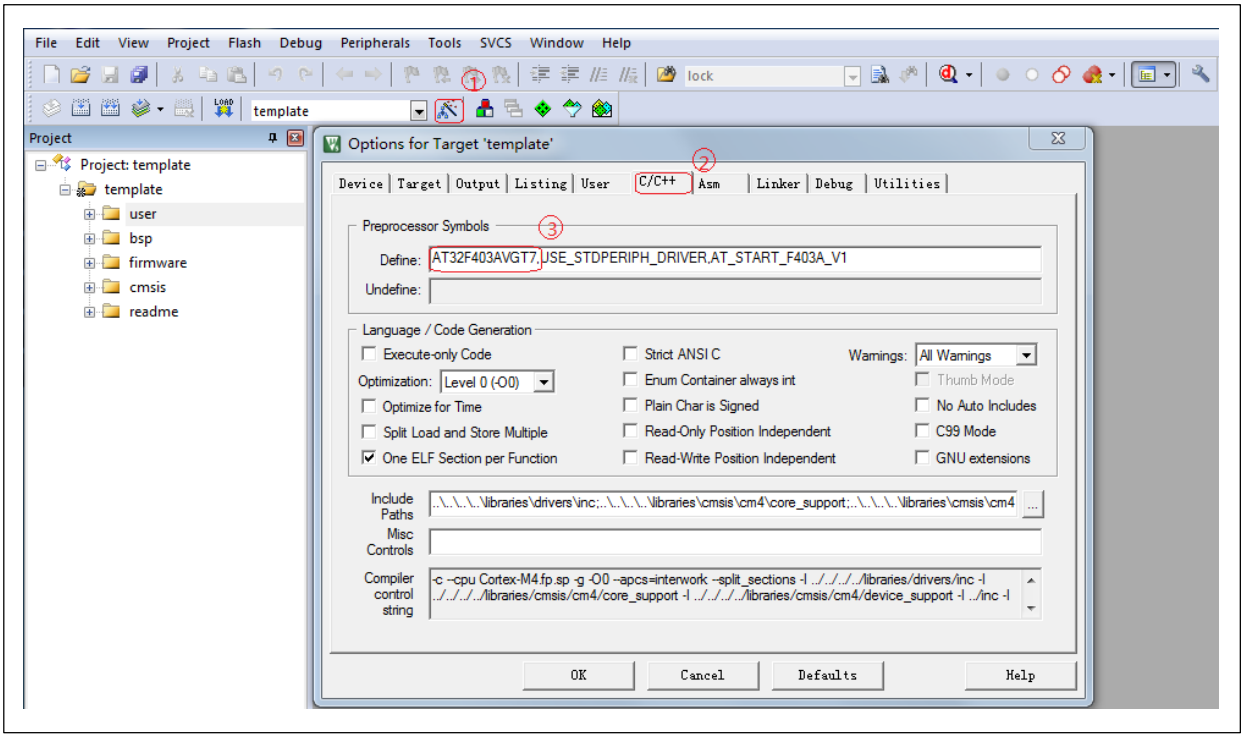
修改型号宏定义，操作步骤和图示如下：

① 点击魔术棒“Options for Target”。

② 点击 C/C++选项卡。

③ 将 Define 栏中将原有的型号宏定义删除，根据表 1 内容写入需要切换型号的宏定义。

图 30. Keil 改宏定义



6.1.2 IAR 上型号切换

修改 device，操作步骤和图示如下：

① 鼠标右键点击工程名，并选择 Options…。

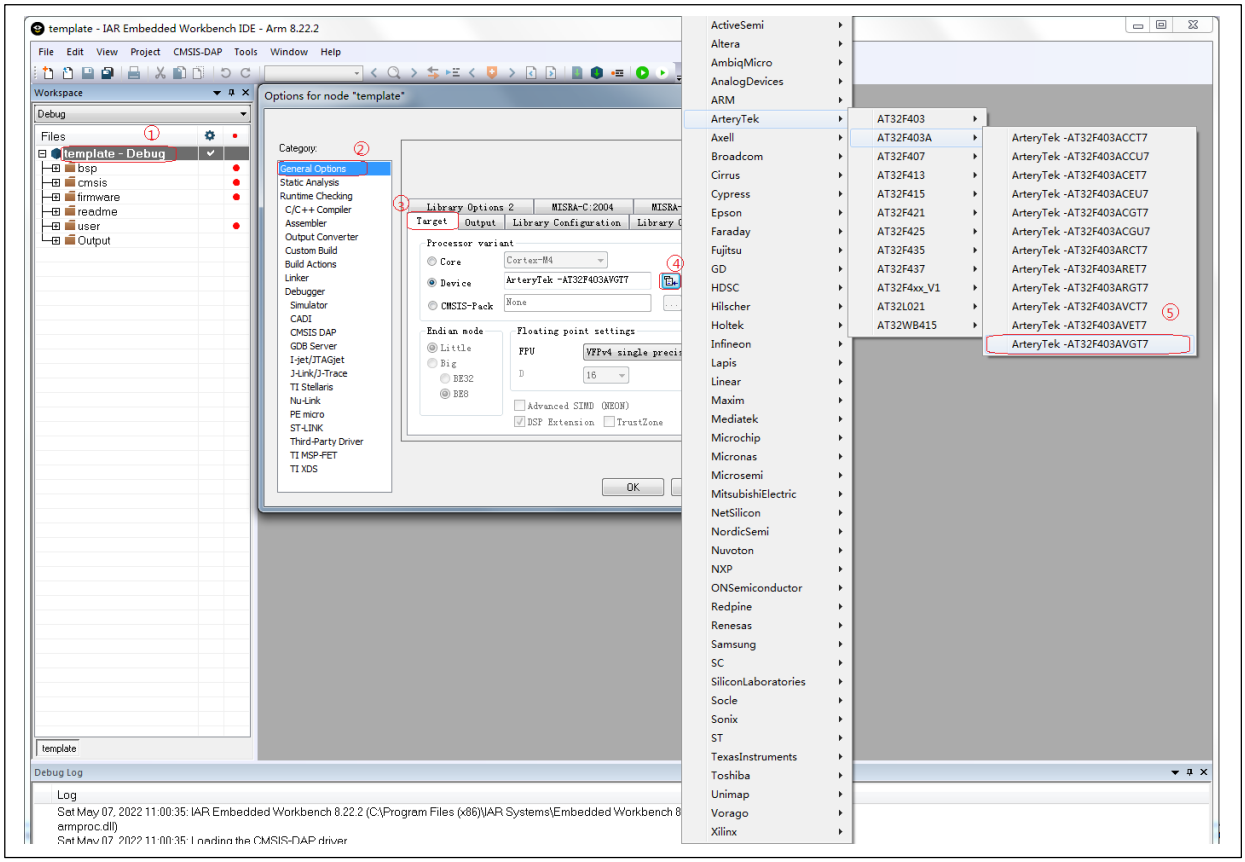
② 选择 General Options。

③ 选择 Target。

④ 点选复选框。

⑤ 选择需要切换的 Device 型号。

图 31. IAR 改 device



修改型号宏定义，操作步骤和图示如下：

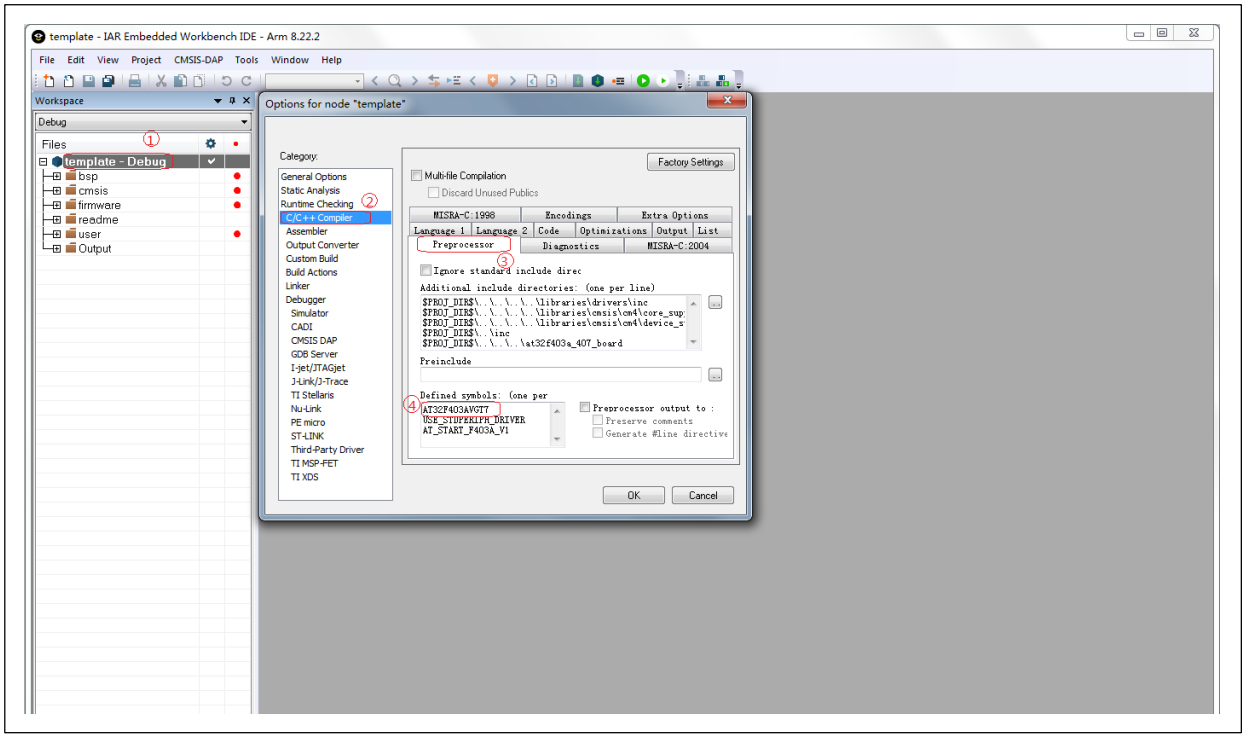
① 鼠标右键点击工程名，并选择 Options…。

② 选择 C/C++ Compiler。

③ 点击 Preprocessor 选项卡。

④ 将 Defined symbols 栏中将原有的型号宏定义删除，根据表 1 内容写入需要切换型号的宏定义。

图 32. IAR 改宏定义



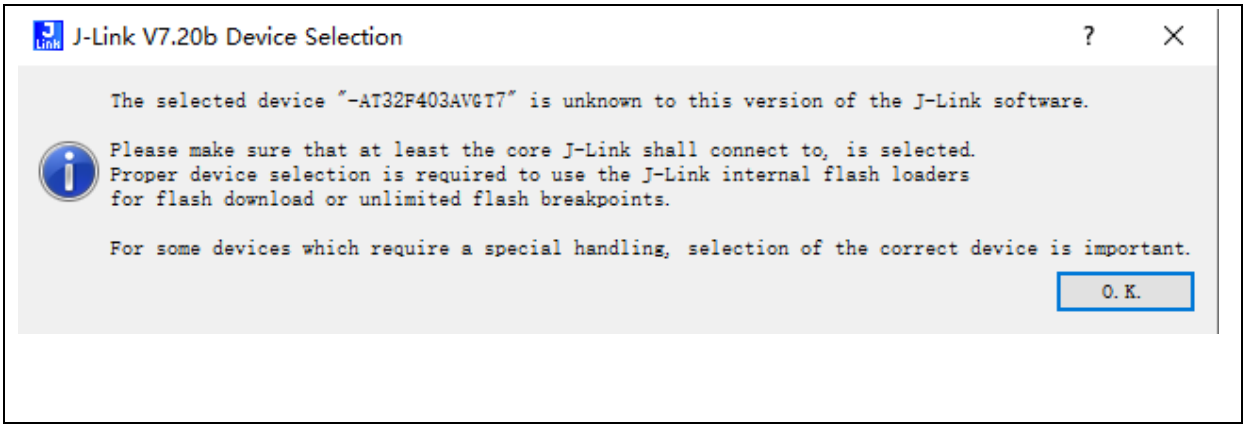
6.2 Keil 项目内 Jlink 无法找到 IC 问题

在一些特殊情况，工程师编译好的 Keil 下的工程项目给到其他工程人员后发现程序可以编译通过，使用 ICP 软件可以正常找到 IC，但 Jlink 找不到 IC。

例如出现如下警告

错误警告情形一

图 33. 错误警告情形一



错误警告情形二

图 34. 错误警告情形二

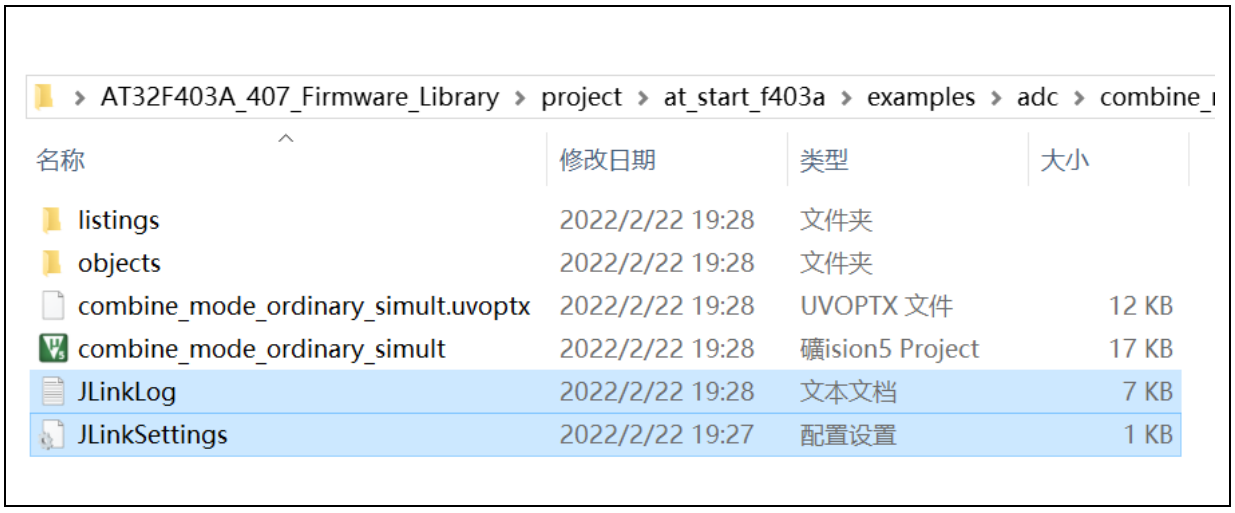
错误警告情形三

图 35. 错误警告情形三

解决方法：

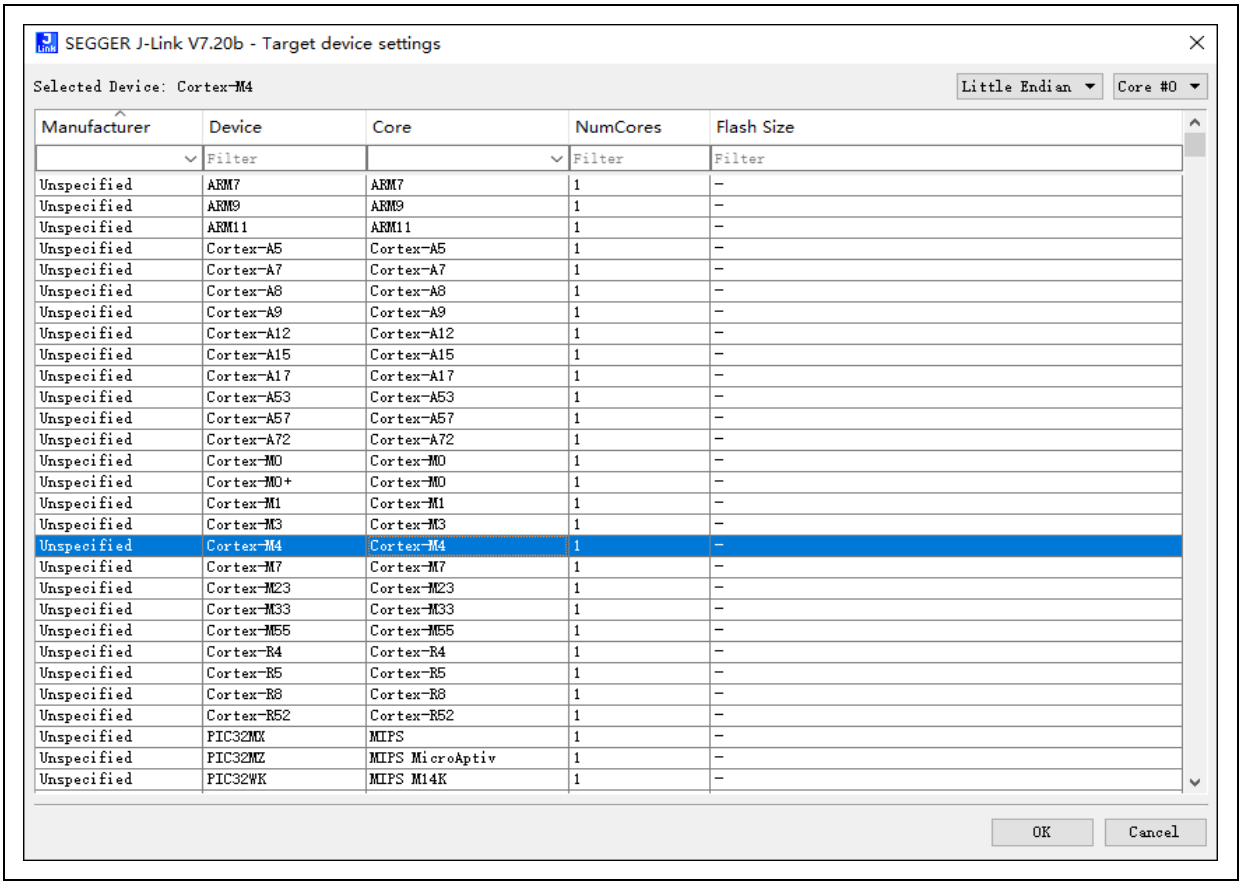
Step 1: 在工程路径内找到 JLinkLog，JLinkSettings 两个文件并删除它

图 36. JLinkLog 和 JLinkSettings



Step 2: 再点击魔法棒->Debug，选择“Unspecified Cortex–M4”

图 37. Unspecified Cortex–M4



6.3 更换外部高速晶振后异常

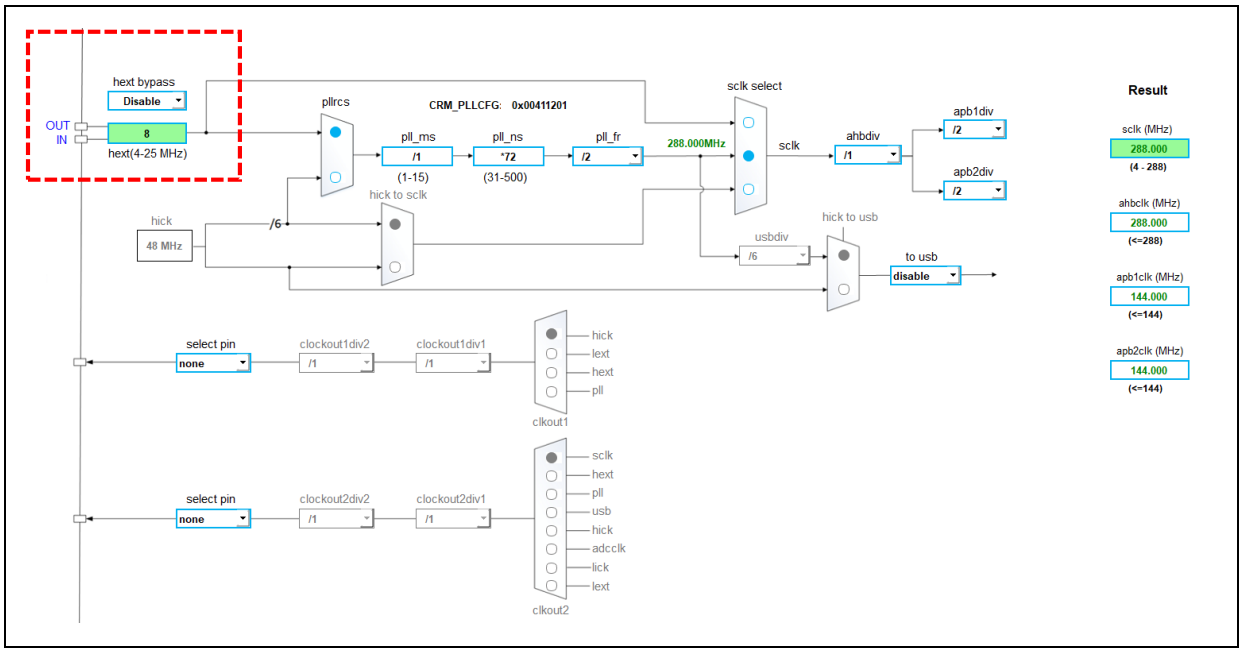
BSP 所有案例都是搭配开发板上 8MHz 的外部高速晶振进行倍频的。
在实际应用中如果采用了非 8 MHz 的外部晶振的话，需注意修改 BSP 中时钟配置以保证时钟频率的正确及稳定。

为此，雅特力专门开发了 AT32_New_Clock_Configuration 工具（可于雅特力官网 TOOL 目录获取），用于生成用户期望的 BSP 系统时钟代码文件。如下图红框所示，外部时钟源参数、分频系数、倍频系数、时钟源选择等参数均可配置，配置完成后点击生成代码即可，避免了修改代码时繁杂的注意事项。

用户只需使用该工具新生成的时钟代码文件（at32f4xx_clock.c/ at32f4xx_clock.h/ at32f4xx_conf.h）将原 BSP demo 中的对应文件替换，在 main 函数中进行 system_clock_config 函数调用即可。

其中 at32f4xx_conf.h 中有外部高速晶振的宏定义 HEXT_VALUE，因此也需要替换。以 AT32F403A 举例，at32f403a_407_conf.h 中 HEXT_VALUE 宏定义如下
#define HEXT_VALUE ((uint32_t)8000000) /*!< value of the high speed exernal crystal in hz */
AT32_New_Clock_Configuration 工具界面如下：

图 38. AT32_New_Clock_Configuration 界面



关于 AT32_New_Clock_Configuration 工具的使用以及 AT32 时钟配置流程、代码解析等详细介绍，请参考各型号的 AN，下表所列 AN 均可从雅特力官网获取。

表 487. 时钟配置应用指南

型号	应用指南
AT32F403A/407时钟配置	AN0082
AT32F435/437时钟配置	AN0084
AT32F421时钟配置	AN0116
AT32F415时钟配置	AN0117
AT32F413时钟配置	AN0118
AT32F425时钟配置	AN0121

7 版本历史

表 488. 文档版本历史

日期	版本	变更
2021.11.26	2.0.0	最初版本
2022.06.15	2.0.1	增加外设库函数概述等
2022.11.15	2.0.2	修正“外设缩写”章节I2C描述错误
2023.07.18	2.0.3	新增CRC特性功能寄存器及函数

重要通知 - 请仔细阅读

买方自行负责对本文所述雅特力产品和服务的选择和使用，雅特力概不承担与选择或使用本文所述雅特力产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为雅特力授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在雅特力的销售条款中另有说明，否则，雅特力对雅特力产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途(及其依据任何司法管辖区的法律的对应情况)，或侵犯任何专利、版权或其他知识产权的默示保证。

雅特力产品并非设计或专门用于下列用途的产品：(A) 对安全性有特别要求的应用，例如：生命支持、主动植入设备或对产品功能安全有要求的系统；(B) 航空应用；(C) 航天应用或航天环境；(D) 武器，且/或(E)其他可能导致人身伤害、死亡及财产损害的应用。如果采购商擅自将其用于前述应用，即使采购商向雅特力发出了书面通知，风险及法律责任仍将由采购商单独承担，且采购商应独力负责在前述应用中满足所有法律和法规要求。

经销的雅特力产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致雅特力针对本文所述雅特力产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大雅特力的任何责任。