BL1824X 应用手册

修改历史

版本号	描 述	日期	作者	版本校对
V1.0	初稿	2024/07/21	Chengfj	

目录

缩略语清单

缩略语	英文全名	中文解释
GPIO	General-purpose input/output	通用输入输出
UART	Universal Asynchronous Receiver/Transmitter	通用异步收发传输器
DMA	Direct Memory Access	直接存储器访问
SFLASH Controller	SPI Flash Controller	串行外设设备接口 FLASH控制器
TIMER	TIMER	定时器
WDT	Watchdog Timer	看门狗计时器
IIC/I2C	inter-integrated circuit	集成电路总线
IIS/ I2S	integrated inter-chip sound	集成芯片音频
AUDIO	AUDIO	音频，声音
GPADC	Analog-to-digital converter	模拟数字转换器
EFUSE	EFUSE	一次性可编程存储器
IR_TX	Infrared Remote Transmit	红外遥控信号传送

本文档介绍了BL1824X SDK中的各外设模块相关的变量及函数，并描述了对应的使用例程。用于指导各外设模块的代码编写和测试工作。

1. GPIO

1.1. 简介

GPIO：General-purpose input/output 通用输入输出。GPIO 是芯片的部分引脚，并不是所有的引脚都是 GPIO 口。GPIO 口作为 GPIO 外设使用的时候是普通 IO 功能；GPIO 口作为其他外设使用的时候是复用 IO 功能。BL1824X 中 GPIO 组只用到了第 0 组，因此 gpio_ex 的库函数并没有用到。

GPIO 电路图如下图所示：

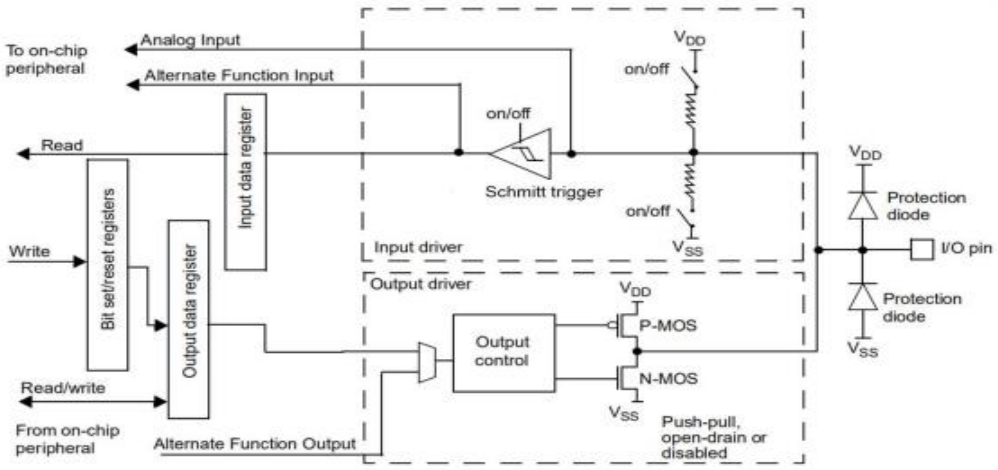


图 1.1 BL1824XGPIO 电路图

BL1824X 共有 23 个客户可用的 GPIO。其中：gpio0 到 gpio22 在一组，输入输出功能都支持。

其中 gpio0 到 gpio22 所在的 GPIO 组对应的基地址为：

GPIO 编号	基地址
GPIO0	0x41200000

GPIO0 在代码中被宏定义为 HS_GPIO0，PMU 在代码中被宏定义为 HS_PMU_BASE。其他芯片，例如 BL1824X 有两组 GPIO，则分别定义为 HS_GPIO0 和 HS_GPIO1。

BL1824X 一共对 GPIO 定义了 6 种工作方式：

PMU_PIN_MODE_FLOAT：浮空输入

PMU_PIN_MODE_PP：推挽输出

PMU_PIN_MODE_PU：上拉输入

PMU_PIN_MODE_PD：下拉输入

PMU_PIN_MODE_OD：开漏输出

PMU_PIN_MODE_OD_PU：开漏上拉输出

其中定义如下：

```
/// Mode
typedef enum
{
    /// Float (Input)
    PMU_PIN_MODE_FLOAT,
    /// Push pull (Output)
    PMU_PIN_MODE_PP,
    /// Pull up (221kOHM~394kOHM)
    PMU_PIN_MODE_PU,
    /// Pull down (141kOHM~303kOHM)
    PMU_PIN_MODE_PD,
    /// Open drain
    PMU_PIN_MODE_OD,
    /// Open drain, pull up
    PMU_PIN_MODE_OD_PU,
}pmu_pin_mode_t;
```

1.2. API 介绍

1.2.1.GPIO 寄存器结构

```
/// HS_GPIO_Type
typedef struct
{
    __I  uint32_t DATA;            // offset:0x00
    __IO uint32_t DATAOUT;         // offset:0x04
         uint32_t RESERVED0[2];
    __IO uint32_t OUTENSET;        // offset:0x10
    __IO uint32_t OUTENCLR;        // offset:0x14
    __IO uint32_t ALTFUNCSET;      // offset:0x18
    __IO uint32_t ALTFUNCCLR;      // offset:0x1C
    __IO uint32_t INTENSET;        // offset:0x20
    __IO uint32_t INTENCLR;        // offset:0x24
    __IO uint32_t INTTYPESET;      // offset:0x28
    __IO uint32_t INTTYPECLR;      // offset:0x2C
    __IO uint32_t INTPOLSET;       // offset:0x30
    __IO uint32_t INTPOLCLR;       // offset:0x34
    __IO uint32_t INTSTATUS;       // offset:0x38
         uint32_t RESERVED1;
    __IO uint32_t INTBOTHSET;      // offset:0x40
    __IO uint32_t INTBOTHCLR;      // offset:0x44
```

```
        uint8_t   RESERVED3[0x1000-0x0048];

    __IO uint32_t MASK_0_7[0x100];    // offset:0x1000
    __IO uint32_t MASK_8_15[0x100];   // offset:0x1400
    __IO uint32_t MASK_16_23[0x100]; // offset:0x1800
    __IO uint32_t MASK_24_31[0x100]; // offset:0x1C00
}HS_GPIO_Type;
```

注：BL1824X 只有 23 个 GPIO，即 0-22 号，所以 MASK_16_23 的最后一位和 MASK_24_31 没有使用，考虑兼容其他型号芯片，代码中未进行删除。以下介绍不会对此进行介绍。

Offset	寄存器	描述
0x00	DATA	数据寄存器
0x04	DATAOUT	数据输出寄存器
	Reserved0[2]	保留
0x10	OUTENSET	输出使能设置寄存器
0x14	OUTENCLR	输出使能清除寄存器
0x18	ALTFUNCSET	保留
0x1C	ALTFUNCCLR	保留
0x20	INTENSET	中断使能设置寄存器
0x24	INTENCLR	中断使能清除寄存器
0x28	INTTYPESET	中断类型设置寄存器
0x2C	INTTYPECLR	中断类型清除寄存器
0x30	INTPOLSET	中断极性设置寄存器
0x34	INTPOLCLR	中断极性清除寄存器
0x38	INTSTATUS	中断状态寄存器
	RESERVED1	保留
0x40	INTBOTHSET	中断双边沿设置
0x44	INTBOTHCLR	中断双边沿清除
	RESERVED3	保留
0x1000-0x13FC	MASKBYTE0	字节 0 掩码访问
0x1400-0x17FC	MASKBYTE1	字节 1 掩码访问
0x1800-0x1BFC	MASKBYTE2	字节 2 掩码访问

DATA address offset：0x0000

Bit	R/W	Reset	Name	Description
22:0	R	N/A	DATA	数据寄存器值 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：低电平

				1：高电平

DATAOUT address offset：0x0004

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	DATAOUT	数据输出寄存器值 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：低电平 1：高电平

OUTENSET address offset：0x00010

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	OUTENSET	输出使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：不使能 1：使能

OUTENCLR address offset：0x00014

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	OUTENCLR	输出清除使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：不使能 1：使能

ALTFUNCSET address offset：0x00018

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	ALTFUNCSET	复用功能使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0

				… Bit22：该组 GPIO22 每比特取值含义： 0：不使能 1：使能

ALTFUNCCLR address offset：0x0001C

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	ALTFUNCCLR	复用功能清除使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：不使能 1：使能

INTENSET address offset：0x00020

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTENSET	中断使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：不使能 1：使能

INTENCLR address offset：0x00024

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTENCLR	中断清除使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：不使能 1：使能

INTTYPESET address offset：0x00028

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTTYPESET	中断类型配置 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：电平触发（低电平或高电平） 1：边沿触发（下降沿或上升沿）

INTTYPECLR address offset：0x0002c

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTTYPECLR	中断类型清除使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：不使能 1：使能

INTPOLSET address offset：0x00030

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTPOLSET	电平极性、边沿中断配置 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：低电平或下降沿 1：高电平或上升沿

INTPOLCLR address offset：0x00034

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTPOLCLR	电平极性、边沿中断配置清除 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义：

				0：不清除 1：清除

INTSTATUS address offset：0x00038

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTSTATUS	GPIO中断状态 写1清除中断请求 每一位代表GPIO组的哪一个引脚来的中断。 例如：第0位代表该组GPIO组的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：无中断 1：有中断 例如： 0x81 表示 GPIO0 和 GPIO7 来了中断

INTTYPE1SET address offset：0x00040

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTTYPE1SET	双边沿中断触发使能 每一位代表GPIO组的哪一个引脚来的中断。 例如：第0位代表该组GPIO组的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值含义： 0：不使能 1：使能

INTTYPE1CLR address offset：0x00044

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	INTTYPE1CLR	双边沿中断触发清除 每一位代表GPIO组的哪一个引脚来的中断。 例如：第0位代表该组GPIO组的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22

				每比特取值含义： 0：不清除 1：清除

MASKBYTE0 address offset：0x01000-0x013FC

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
7:0	RW	0x0	MASKBYTE0	GPIO组的0-7引脚的功能屏蔽使能 Bit0： 0：不使能（GPIO0引脚正常工作） 1：使能（GPIO0不能正常工作） …… Bit7： 0：不使能（GPIO7引脚正常工作） 1：使能（GPIO7不能正常工作）

MASKBYTE1 address offset：0x01400-0x017FC

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
7:0	RW	0x0	MASKBYTE1	GPIO组的8-15引脚的功能屏蔽使能 Bit0： 0：不使能（GPIO8引脚正常工作） 1：使能（GPIO8不能正常工作） …… Bit7： 0：不使能（GPIO15引脚正常工作） 1：使能（GPIO15不能正常工作）

MASKBYTE2 address offset：0x01800-0x01BFC

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
6:0	RW	0x0	MASKBYTE2	GPIO组的16-22引脚的功能屏蔽使能 Bit0： 0：不使能（GPIO16引脚正常工作） 1：使能（GPIO16不能正常工作） …… Bit6： 0：不使能（GPIO22引脚正常工作） 1：使能（GPIO22不能正常工作）

1.2.2.与 GPIO 相关的 PMU 寄存器结构

PMU 中 GPIO 的相关寄存器：

```
/// HS_PMU_Type
typedef struct
{
    __IO uint32_t GPIO_POL;             // offset:0x1C
    ……
    __IO uint32_t PMU_MISC_CTRL;        // offset:0x24
    ……
    __IO uint32_t GPIO_OE_CTRL;         // offset:0x2C
    ……
    __IO uint32_t GPIO_PU_CTRL;         // offset:0x34
    ……
    __IO uint32_t GPIO_ODA_CTRL;        // offset:0x40
    ……
    __IO uint32_t GPIO_IE_CTRL;         // offset:0x54
    ……
    __IO uint32_t GPIO_STATUS_READ;     // offset:0x88
    ……
    __IO uint32_t GPIO_STATUS_READ_2; // offset:0x90
    ……
    __IO uint32_t GPIO_WAKEUP;          // offset:0x98
    ……
    __IO uint32_t GPIO_ODE_CTRL;        // offset:0xA0
    ……
    __IO uint32_t GPIO_PD_CTRL;         // offset:0xA8
    ……
    __IO uint32_t GPIO_LATCH;           // offset:0xB0
    ……
    __IO uint32_t GPIO_NOCLK_LATCH;     // offset:0xB8
    ……
    __IO uint32_t GPIO_DRV_CTRL_0;      // offset:0xC4
    ……
    __IO uint32_t GPIO_DRV_CTRL_2;      // offset:0xCC
}HS_PMU_Type;
```

Offset	寄存器	描述
0x001C	PMU_GPIO_POL	GPIO 极性控制寄存器
0x0020	GPIO_PU_CTRL_1	GPIO 上拉控制
0x0024	PMU_MISC_CTRL	PMU 控制寄存器
0x002C	GPIO_OE_CTRL	GPIO 输出使能控制寄存器
0x0034	GPIO_PU_CTRL	GPIO 上拉控制寄存器
0x0040	GPIO_ODA_CTRL	GPIO 输出数据寄存器
0x0054	GPIO_IE_CTRL	GPIO 输入使能控制寄存器
0x0088	PMU_GPIO_STATUS_READ	GPIO 状态读取寄存器
0x0090	PMU_GPIO_STATUS_READ_2	GPIO 状态读取寄存器 2

0x0098	PMU_GPIO_MASK	GPIO 掩码
0x00A0	GPIO_ODE_CTRL	GPIO 开漏控制寄存器
0x00A8	GPIO_PD_CTRL	GPIO 下拉控制寄存器
0x00B0	PMU_GPIO_LATCH	GPIO 事件锁存器
0x00B8	PMU_GPIO_NOCLK_LATCH	无时钟 GPIO 事件锁存器
0x00C4	GPIO_DRV_CTRL_0	GPIO 驱动控制寄存器 0
0x00CC	GPIO_DRV_CTRL_2	GPIO 驱动控制寄存器 2

PMU_GPIO_POL address offset：0x001C

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	REG_GPIO_POL [22:0]	GPIO引脚电源唤醒电平配置 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO组的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值具体含义： 0：高电平唤醒 1：低电平唤醒

GPIO_PU_CTRL_1 address offset：0x0020

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	GPIO_PL_CTRL _1[22:0]	GPIO[22:0]上拉阻抗选择（bit1） Bit0：GPIO0的上拉阻抗 …… Bit22：GPIO22的上拉阻抗 00：4Kohm 01：10Kohm 10：300Kohm 11：2Mohm

PMU_MISC_CTRL address offset: 0x0024

Bit	R/W	Reset	Name	Description
12	RW	0x0	GPIO_AUTO_LA TCH_CTRL	GPIO 自动锁控制 0：唤醒后写入 0 以释放 GPIO 控制信号 1：睡眠前写入 1 以锁定 GPIO 控制信号

GPIO_OE_CTRL address offset：0x002c

Bit	R/W	Reset	Name	Description
23	RW	0x0	GPIO_OEB_SEL	GPIO_OEB 和 GPIO_OUT 的控制归属 1：通过reg控制GPIO_OEB和

				GPIO_OUT。 0：通过HW或者 GPIO_AUTO_LATCH_CTRL控制 GPIO_OEB和GPIO_OUT。
22:0	RW	0x7fffff	GPIO_OEB_REG[22:0]	GPIO输出使能控制 每一位代表 GPIO 组的哪一个引脚 例如：第 0 位代表该组 GPIO 组的第 0 个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 仅当 gpio_oeb_sel = 1 时有效 每比特取值具体含义： 0：使能 1：不使能

GPIO_PU_CTRL address offset：0x0034

Bit	R/W	Reset	Name	Description
22:0	RW	0x010	GPIO_PU_REG[22:0]	GPIO上拉使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO组的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组GPIO22 每比特取值具体含义： 0：不使能 1：使能

GPIO_ODA_CTRL address offset：0x0040

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	GPIO_ODA_CTRL[22:0]	GPIO 输出数据配置（PMU 控制） 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：低电平 1：高电平

GPIO_IE_CTRL address offset：0x0054

Bit	R/W	Reset	Name	Description
28:23	RW	0x3f	gpio_sf_ctrl	内部FLASH管脚输入使能 0：不使能

				1：使能 **注：不建议修改！！！**
22:0	RW	0x1f	gpio_ie_ctrl[22:0]	GPIO输入使能控制 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：不使能 1：使能

PMU_GPIO_STATUS_READ address offset：0x0088

Bit	R/W	Reset	Name	Description
22:0	R	0x0	GPIO_OEB_AUTO_LATCH[22:0]	GPIO状态读取 当gpio_auto_latch_ctrl值为1时，该寄存器内容代表gpio_oeb[22:0]锁存值。 每比特取值含义： 0：低电平 1：高电平

PMU_GPIO_STATUS_READ_2 address offset：0x0090

Bit	R/W	Reset	Name	Description
22:0	R	0x0	GPIO_OUT_AUTO LATCH[22:0]	GPIO状态读取 当gpio_auto_latch_ctrl值为1时，该寄存器内容代表gpio_out[22:0]锁存值。 每比特取值含义： 0：低电平 1：高电平

PMU_GPIO_MASK address offset：0x0098

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	GPIO_WAKE_EN[22:0]	GPIO唤醒使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：不使能 1：使能

GPIO_ODE_CTRL address offset：0x00a0

Bit	R/W	Reset	Name	Description

23	RW	0x0	gpio_pmu_dbg	用 debug bus 方式调试使能 0：不使能 1：使能
22:0	RW	0x0	GPIO_ODE_REG[22:0]	GPIO开漏控制使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：不使能 1：使能

GPIO_PD_CTRL address offset：0x00a8

Bit	R/W	Reset	Name	Description
22:0	RW	0x0	GPIO_PD_REG[22:0]	GPIO下拉使能 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：不使能 1：使能

PMU_GPIO_LATCH address offset：0x00b0

Bit	R/W	Reset	Name	Description
22:0	R	N/A	GPIO_INT_LATCH[22:0]	GPIO唤醒中断事件状态 每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：无中断 1：有中断

PMU_GPIO_NOCLK_LATCH address offset：0x00b8

Bit	R/W	Reset	Name	Description
22:0	R	N/A	GPIO_INT_NOCLK_LATCH[22:0]	GPIO睡眠唤醒中断事件状态，唤醒时无32K时钟。

				每一位代表GPIO组的哪一个引脚 例如：第0位代表该组GPIO的第0个引脚。 Bit 0：该组 GPIO0 … Bit22：该组 GPIO22 每比特取值含义： 0：无中断 1：有中断

GPIO_DRV_CTRL_0 address offset：0x00c4

Bit	R/W	Reset	Name	Description
28:23	RW	0x0	gpio_drv_ctrl_0[28:23]	GPIO[28:23]驱动能力选择 内部FLASH管脚GPIO驱动电流档位配置。 GPIO_drv_ctrl_0与GPIO_drv_ctrl_2对应同样的比特位值，共同组成驱动能力寄存器[ctrl_2][ctrl_0] =[1:0]配置。 驱动能力寄存器[1:0]对应的驱动电流为： 00：2mA 01：4mA 10：6mA 11：12mA
22:0	RW	0x0	GPIO_DRV_CTRL[22:0]	GPIO[22:0]驱动电流档位配置 Bit0：GPIO0的电流档位 …… Bit22：GPIO22的电流档位 0：8mA 1：13mA

GPIO_DRV_CTRL_2 address offset：0x00cc

Bit	R/W	Reset	Name	Description
28:23	RW	0x0	GPIO_DRV_CTRL_1[28:23]	GPIO[28:23]驱动能力选择 内部FLASH管脚GPIO驱动电流档位配置。 GPIO_drv_ctrl_0与GPIO_drv_ctrl_2对应同样的比特位值，共同组成驱动能力寄存器[ctrl_2][ctrl_0] =[1:0]配置。 驱动能力寄存器[1:0]对应的驱动电流为： 00：2mA 01：4mA 10：6mA 11：12mA
22:0	RW	0x0	GPIO_PL_CTRL_0[22:0]	GPIO[22:0]上拉阻抗选择（bit0） Bit0：GPIO0的上拉阻抗

				…… Bit22：GPIO22的上拉阻抗 00：4Kohm 01：10Kohm 10：300Kohm 11：2Mohm

1.2.3.变量参数

1.2.3.1. gpio_level_t

```
/// Level
typedef enum
{
    GPIO_LOW = 0,
    GPIO_HIGH = ~0,
}gpio_level_t;
```

GPIO 电平。

成员名称	描述
GPIO_LOW	GPIO 低电平
GPIO_HIGH	GPIO 高电平

1.2.3.2. gpio_direction_t

```
/// Direction
typedef enum
{
    GPIO_INPUT = 0,
    GPIO_OUTPUT = ~0,
}gpio_direction_t;
```

GPIO 方向。

成员名称	描述
GPIO_INPUT	GPIO 输入
GPIO_OUTPUT	GPIO 输出

1.2.3.3. gpio_trigger_type_t

```
/// Interrupt trigger type
typedef enum
{
    GPIO_FALLING_EDGE,
```

```
    GPIO_RISING_EDGE,
    GPIO_BOTH_EDGE,
    GPIO_LOW_LEVEL,
    GPIO_HIGH_LEVEL,
    GPIO_TRIGGER_DISABLE,
}gpio_trigger_type_t;
```

GPIO 触发类型。

成员名称	描述
GPIO_FALLING_EDGE	GPIO 下降沿
GPIO_RISING_EDGE	GPIO 上升沿
GPIO_BOTH_EDGE	GPIO 双边沿
GPIO_LOW_LEVEL	GPIO 低电平
GPIO_HIGH_LEVEL	GPIO 高电平
GPIO_TRIGGER_DISABLE	GPIO 关闭中断或禁能中断

1.2.3.4. gpio_env_t

```
typedef struct
{
    gpio_callback_t callback;
    uint32_t pin_single_edge_flag;
    uint32_t pin_single_edge_level;

#ifndef CONFIG_GPIO1_ABSENT
    gpio_callback_t callback_ex;
    uint32_t pin_single_edge_flag_ex;
    uint32_t pin_single_edge_level_ex;
#endif

    struct
    {
        struct
        {
            struct
            {
                uint32_t DATAOUT;
                uint32_t OUTENSET;
                uint32_t INTENSET;
                uint32_t INTPOLSET;
                uint32_t INTTYPESET;
                uint32_t INTBOTHSET;
            }gpio0;
#ifndef CONFIG_GPIO1_ABSENT
```

```
            struct
            {
                uint8_t DATAOUT;
                uint8_t OUTENSET;
                uint8_t INTENSET;
                uint8_t INTPOLSET;
                uint8_t INTTYPESET;
                uint8_t INTBOTHSET;
            }gpio1;
#endif
        }store_reg;
    }lowpower;
}gpio_env_t;
```

GPIO 结构体类型。

注：由于 BL1824X 的 GPIO 只有一组，即 GPIO0 组，里面有 23 个 pin。没有 GPIO1 组。所以该结构体类型中的宏 **CONFIG_GPIO1_ABSENT** 控制的代码是不生效的。

成员名称				描述
gpio_callback_t callback				GPIO0 回调函数指针
pin_single_edge_flag				GPIO0 单边沿标志
pin_single_edge_level				GPIO0 单边沿电平
gpio_callback_t callback_ex				GPIO1 回调函数指针,该芯片没有 GPIO1 组，可忽略
pin_single_edge_flag_ex				GPIO1 单边沿标志,该芯片没有 GPIO1 组，可忽略
pin_single_edge_level_ex				GPIO1 单边沿电平,该芯片没有 GPIO1 组，可忽略
lowpower	**store_reg**	**gpio0**	DATAOUT	保存 GPIO0 数据输出寄存器的参数
			OUTENSET	保存 GPIO0 输出使能设置寄存器的参数
			INTENSET	保存 GPIO0 输入使能设置寄存器的参数
			INTPOLSET	保存 GPIO0 中断优先级设置寄存器的参数
			INTTYPESET	保存 GPIO0 中断类型设置
			INTBOTHSET	保存 GPIO0 中断双边沿设置
		gpio1	DATAOUT	保存 GPIO1 数据输出寄存器的参数,该芯片没有 GPIO1 组，可忽略
			OUTENSET	保存 GPIO1 输出使能设置寄存器的参数,该芯片没有 GPIO1 组，可忽略
			INTENSET	保存 GPIO1 输入使能设置寄存器的参数,该芯片没有 GPIO1 组，可忽略
			INTPOLSET	保存 GPIO1 中断优先级设置寄存器的参数,该芯片没有 GPIO1 组，可忽略

			INTTYPESET	保存 GPIO1 中断类型设置,该芯片没有 GPIO1 组，可忽略
			INTBOTHSET	保存 GPIO1 中断双边沿设置,该芯片没有 GPIO1 组，可忽略

1.2.4.库函数

1.2.4.1. pmu_pin_mode_set

函数名	pmu_pin_mode_set
函数原型	void pmu_pin_mode_set(uint32_t pin_mask, pmu_pin_mode_t mode)
功能描述	GPIO0 组的 PIN 管脚 GPIO 模式设置
输入参数 1	pin_mask：待被配置的管脚掩码, GPIO0_0 的管脚掩码 为 1<<0 GPIO0_1 的管脚掩码 为 1<<1 ..... GPIO0_22 的管脚掩码 为 1<<22
输入参数 2	mode：pmu_pin_mode_t 枚举类型，用于 GPIO 工作模式
输出参数	无
返回值	无

1.2.4.2. pinmux_config

函数名	pinmux_config
函数原型	void pinmux_config(uint32_t pin, pinmux_t func)
功能描述	GPIO 模式参数配置
输入参数 1	pin：待被配置是 GPIO 管脚编号
输入参数 2	func：pinmux_t 枚举类型，GPIO 模式参数
输出参数	无
返回值	无

1.2.4.3. gpio_open

函数名	gpio_open
函数原型	void gpio_open(void)
功能描述	打开 GPIO，使用 GPIO 口之前必须调用这个函数
输入参数	无
输出参数	无
返回值	无

1.2.4.4. gpio_open_clock

函数名	gpio_open_clock
函数原型	void gpio_open_clock (void)
功能描述	打开 GPIO 时钟
输入参数	无
输出参数	无
返回值	无

1.2.4.5. gpio_set_interrupt

函数名	gpio_set_interrupt
函数原型	void gpio_set_interrupt(uint32_t pin_mask, gpio_trigger_type_ttrigger_type)
功能描述	为 GPIO0 组对应 GPIO 设置中断触发类型
输入参数 1	pin_mask：待被配置的管脚掩码 GPIO0_0 的管脚掩码 为 1<<0 GPIO0_1 的管脚掩码 为 1<<1 ..... GPIO0_22 的管脚掩码 为 1<<22
输入参数 2	trigger_type：gpio_trigger_type_t 枚举类型，用于设置中断触发方式
输出参数	无
返回值	无

1.2.4.6. gpio_set_interrupt_callback

函数名	gpio_set_interrupt_callback
函数原型	void gpio_set_interrupt_callback(gpio_callback_t callback)
功能描述	为 GPIO0 组对应 GPIO 设置中断回调函数
输入参数	callback：gpio_callback_t 类型函数指针，用于指向待被调用的函数
输出参数	无
返回值	无

1.2.4.7. gpio_set_direction

函数名	gpio_set_direction
函数原型	void gpio_set_direction(uint32_t pin_mask, gpio_direction_t dir)
功能描述	为 GPIO0 组对应 GPIO 设置 GPIO 的方向，是输入还是输出
输入参数 1	pin_mask：待被配置的管脚掩码 GPIO0_0 的管脚掩码 为 1<<0 GPIO0_1 的管脚掩码 为 1<<1 ..... GPIO0_22 的管脚掩码 为 1<<22
输入参数 2	dir：gpio_direction_t 枚举类型，用于设置 GPIO 的方向
输出参数	无
返回值	无

1.2.4.8. gpio_read

函数名	gpio_read
函数原型	uint32_t gpio_read(uint32_t pin_mask)
功能描述	用于读取 GPIO0 组对应 GPIO 相应管脚的数值
输入参数	pin_mask：待被读取数值的 GPIO 管脚掩码 GPIO0_0 的管脚掩码 为 1<<0 GPIO0_1 的管脚掩码 为 1<<1 ..... GPIO0_22 的管脚掩码 为 1<<22
输出参数	无
返回值	对应 GPIO 管脚的数值

1.2.4.9. gpio_read_output_status

函数名	gpio_read_output_status
函数原型	uint32_t gpio_read_output_status(uint32_t pin_mask)
功能描述	读取 GPIO0 组 GPIO 相应管脚的输出状态
输入参数	pin_mask：待读取状态的 GPIO 的管脚掩码 GPIO0_0 的管脚掩码 为 1<<0 GPIO0_1 的管脚掩码 为 1<<1 ..... GPIO0_22 的管脚掩码 为 1<<22
输出参数	无
返回值	gpio 管脚的状态值

1.2.4.10. gpio_write

函数名	gpio_write
函数原型	void gpio_write(uint32_t pin_mask, uint32_t level)
功能描述	为 GPIO0 组对应 GPIO 管脚写值
输入参数 1	pin_mask：待被写值的 GPIO 管脚掩码 GPIO0_0 的管脚掩码 为 1<<0 GPIO0_1 的管脚掩码 为 1<<1 ..... GPIO0_22 的管脚掩码 为 1<<22
输入参数 2	level：待写入的数据，0 为拉低，1 为拉高
输出参数	无
返回值	无

1.2.4.11. gpio_toggle

函数名	gpio_toggle
函数原型	void gpio_toggle(uint32_t pin_mask);
功能描述	用于翻转 GPIO0 组对应 GPIO 指定管脚的值，如果当前为低电平，翻转后为高
输入参数	pin_mask：待翻转的 gpio 的管脚掩码 GPIO0_0 的管脚掩码 为 1<<0 GPIO0_1 的管脚掩码 为 1<<1 ..... GPIO0_22 的管脚掩码 为 1<<22
输出参数	无
返回值	无

1.2.4.12. gpio_clear_interrupt_pending

函数名	gpio_clear_interrupt_pending
函数原型	void gpio_clear_interrupt_pending(uint32_t pin_mask)
功能描述	清除 GPIO0 组相应 GPIO 管脚的中断状态 例如： 若 GPIO0_3 来中断，对应的中断状态位为 pin_mask=1<<3;
输入参数	pin_mask：待清除 gpio 中断状态的引脚
输出参数	无
返回值	无

1.2.4.13. gpio_store

函数名	gpio_store
函数原型	void gpio_store(void)
功能描述	用于保存 GPIO 寄存器的状态
输入参数	无
输出参数	无
返回值	无

1.2.4.14. gpio_restore

函数名	gpio_restore
函数原型	void gpio_restore(void)
功能描述	恢复 GPIO 寄存器的状态
输入参数	无
输出参数	无
返回值	无

1.2.4.15. GPIO0_IRQHandler

函数名	GPIO0_IRQHandler
函数原型	void GPIO0_IRQHandler(void)
功能描述	GPIO0 组 GPIO 的中断请求，无论哪个 GPIO0_0 到 GPIO0_22 的那个 PIN 来的中断，都进入该中断服务函数。
输入参数	无
输出参数	无
返回值	无

1.3. GPIO 应用例程

定义 GPIO 参数变量：

```
#define PIN_MIN             0
#define PIN_MAX             22
#define PIN_LED             4
#define RESET_NO_ERROR      0x00000000
```

定义了引脚最小值、引脚最大值和 LED 灯引脚值。

1.3.1.例程 1：GPIO 推挽方式输出高、低电平功能测试

软件实现：

```
//GPIO 推挽方式输出高、低电平功能测试
static void example_gpio_output_PP_HIGH(void)
{
    /* gpio led/output test */
    for (uint32_t PIN_NUM = PIN_MIN; PIN_NUM <= PIN_MAX; PIN_NUM++) {
        pmu_pin_mode_set(BIT_MASK(PIN_NUM), PMU_PIN_MODE_PP);
        pinmux_config(PIN_NUM, PINMUX_GPIO_MODE_CFG);
    }     //GPIO 0-22 设置为推挽模式，并调整为复用模式
    gpio_open();      //打开 GPIO
    for (uint32_t PIN_NUM = PIN_MIN; PIN_NUM <= PIN_MAX; PIN_NUM++) {
        gpio_set_direction(BIT_MASK(PIN_NUM), GPIO_OUTPUT);
    }     //GPIO 0-22 设置输出模式
    for (uint32_t PIN_NUM = PIN_MIN; PIN_NUM <= PIN_MAX; PIN_NUM++) {
          gpio_write(BIT_MASK(PIN_NUM), GPIO_HIGH);
    }     //将 GPIO 0-22 写入高电平
}

static void example_gpio_output_PP_LOW(void)
{
    /* gpio led/output test */
    for (uint32_t PIN_NUM = PIN_MIN; PIN_NUM <= PIN_MAX; PIN_NUM++) {
        pmu_pin_mode_set(BIT_MASK(PIN_NUM), PMU_PIN_MODE_PP);
        pinmux_config(PIN_NUM, PINMUX_GPIO_MODE_CFG);
    }
    gpio_open();
    for (uint32_t PIN_NUM = PIN_MIN; PIN_NUM <= PIN_MAX; PIN_NUM++) {
        gpio_set_direction(BIT_MASK(PIN_NUM), GPIO_OUTPUT);
    }

    for (uint32_t PIN_NUM = PIN_MIN; PIN_NUM <= PIN_MAX; PIN_NUM++) {
        gpio_write(BIT_MASK(PIN_NUM), GPIO_LOW);
    }     //将 GPIO 0-22 写入低电平
}

void gpio_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);

    switch (case_num)
    {
```

```
        //-----------------------------------------
        //推挽输出和上下拉输入
        case GPIO_TEST_CASE_010000:      //GPIO  推挽方式输出高电平功能测试
            example_gpio_output_PP_HIGH();   //01 00 00
            break;
        case GPIO_TEST_CASE_010001:      //GPIO  推挽方式输出低电平功能测试
            example_gpio_output_PP_LOW();                    //01 00 01
            break;
        default:
            log_debug("[%s][%d][The input test number is not support!]\n",
__FUNCTION__,   __LINE__);
            // Do something you want to warning.
            break;
    }
}
int main(void)
{
    uint8_t         cmd[SHELL_FIFO_SIZE];
    uint8_t*        pCmd = &cmd[0];
    volatile bool ret = false;

    // Enble all IRQ quikly.
    __set_PRIMASK(0);

    // Disable WDT.
    wdt_enable(0);

    // Init interactive UART.
    uart_interaction_init();
    shell_init(HS_UART1);
    while (1)
    {
        ret = shell_get_cmd(&pCmd);
        if (ret == true)
        {
            log_debug("\r\ncmd : %s.\n", cmd);
            gpio_test_case(pCmd);
        }
    }
    shell_uninit();
    return 0;
}
```

本例程主要实现对 GPIO 推挽方式输出高、低电平功能测试。

在函数 gpio_test_case(pCmd)中，首先选择测试历程编号，即 01 00 00 推挽输出高

电平，01 00 01 推挽输出低电平，分别进入到 example_gpio_output_PP_HIGH(void)和 example_gpio_output_PP_LOW(void)函数内。利用 for 循环对 GPIO0 到 GPIO22 设置为推挽模式，并把每个引脚调为引脚复用模式；然后打开 GPIO，设置为输出模式；最后利用 for 循环将 GPIO0-22 写入高低电平。

1.3.2.例程 2：GPIO 输入高低电平测试

软件实现：

```
//GPIO 输入高低电平测试
static void example_gpio_input_HIGH_LOW(void)
{
    bTest_intr = false;

    pmu_pin_mode_t pmu_pin_mode = PMU_PIN_MODE_FLOAT;
    for (uint32_t PIN_NUM = PIN_MIN; PIN_NUM <= PIN_MAX; PIN_NUM++) {
        if (PIN_NUM == PIN_LED ) {  //&& PIN_NUM != PIN_BOOT)
            continue;              //LED 用于指示测试结果
        }
        pinmux_config(PIN_NUM, PINMUX_GPIO_MODE_CFG);
        pmu_pin_mode_set(BIT_MASK(PIN_NUM), pmu_pin_mode);
    }

    //LED 用于指示测试结果
    pinmux_config(PIN_LED, PINMUX_GPIO_MODE_CFG);
    pmu_pin_mode_set(BIT_MASK(PIN_LED), PMU_PIN_MODE_PP);

    gpio_open();

    //LED
    gpio_set_direction(BIT_MASK(PIN_LED), GPIO_OUTPUT);
    gpio_write(BIT_MASK(PIN_LED), GPIO_HIGH);
    gpio_set_interrupt_callback(input_handler);

    for (uint32_t PIN_NUM = PIN_MIN; PIN_NUM <= PIN_MAX; PIN_NUM++) {
        if (PIN_NUM != PIN_LED ) {
            gpio_set_direction(BIT_MASK(PIN_NUM), GPIO_INPUT);
            gpio_set_interrupt(BIT_MASK(PIN_NUM), GPIO_BOTH_EDGE);
        }
    }

    __set_PRIMASK(0);
    while(1);
```

```
}

void gpio_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);

    switch (case_num)
    {
        //-----------------------------------------
        case GPIO_TEST_CASE_010002:                 //GPIO 输入高低电平测试
        case GPIO_TEST_CASE_010003:
            example_gpio_input_HIGH_LOW();          //01 00 02/03
            break;
        default:
            log_debug("[%s][%d][The input test number is not support!]\n",
__FUNCTION__,   __LINE__);
            // Do something you want to warning.
            break;
    }
}
int main(void)
{
    uint8_t         cmd[SHELL_FIFO_SIZE];
    uint8_t*        pCmd = &cmd[0];
    volatile bool ret = false;

    // Enble all IRQ quikly.
    __set_PRIMASK(0);

    // Disable WDT.
    wdt_enable(0);

    // Init interactive UART.
    uart_interaction_init();
    shell_init(HS_UART1);
    while (1)
    {
        ret = shell_get_cmd(&pCmd);
        if (ret == true)
        {
            log_debug("\r\ncmd : %s.\n", cmd);
            gpio_test_case(pCmd);
        }
    }
```

```
    shell_uninit();
    return 0;
}
```

本例程主要实现对 GPIO 输入高低电平测试。

在函数 example_gpio_input_HIGH_LOW(void)函数中，首先对 bTest_intr 设置为 true，然后利用 for 循环设置 LED 用于指示测试结果，然后，再对 LED 引脚配置为推挽模式，并配置引脚为复用模式；最后打开 GPIO。

每个可用的 GPIO 设置为输入下拉，并设置中断使能，分别配置为高低电平触发中断；外部输入一个周期性高低电平信号，外接 LED 灯，高电平点亮、低电平熄灭 LED 灯。

1.3.3.例程 3：GPIO 睡眠唤醒能力测试

软件实现：

```
//GPIO 睡眠唤醒功能测试
static void wakeup_gpio_handler(uint32_t pin_mask)
{
    static int counter = 0;

    if(gpio_read(pin_mask))
    {
        counter++;
        log_debug("gpio: %08X [%d]\n", pin_mask, counter);
    }
}

static void pinmux_init(void)
{
    // default all is pull up
     pmu_pin_mode_set(0x7FFFF, PMU_PIN_MODE_PU);

    // KEY
    pinmux_config(PIN_WAKEUP_0,  PINMUX_GPIO_MODE_CFG);
    pinmux_config(PIN_WAKEUP_1,  PINMUX_GPIO_MODE_CFG);
     pmu_pin_mode_set(pin_wakeup_mask, PMU_PIN_MODE_PU);

    // UART1
    pinmux_config(PIN_UART1_TX,  PINMUX_UART1_SDA_O_CFG);
    pinmux_config(PIN_UART1_RX,  PINMUX_UART1_SDA_I_CFG);
    pmu_pin_mode_set(BIT_MASK(PIN_UART1_TX), PMU_PIN_MODE_PP);
     pmu_pin_mode_set(BIT_MASK(PIN_UART1_RX), PMU_PIN_MODE_PU);
}
static void peripheral_init(void)
```

```
{
    // Init GPIO
     gpio_open();

    // Wakeup Pin
    gpio_set_direction(pin_wakeup_mask, GPIO_INPUT);
    gpio_set_interrupt(pin_wakeup_mask, GPIO_BOTH_EDGE);
     gpio_set_interrupt_callback(wakeup_gpio_handler);

#if defined(CONFIG_SLEEP_SUPPORT)
    pmu_wakeup_pin_set(pin_wakeup_mask, PMU_PIN_WAKEUP_LOW_LEVEL);
    pmu_wakeup_pin_register_callback(wakeup_gpio_handler,
wakeup_gpio_handler);
#endif

    // Init UART
    uart_open(DEBUG_UART, DEBUG_UART_BAUDRATE,
UART_FLOW_CTRL_DISABLED, NULL);
}

static void peripheral_restore(void)
{
    // Init UART
    uart_open(DEBUG_UART, DEBUG_UART_BAUDRATE,
UART_FLOW_CTRL_DISABLED, NULL);
}

static void power_sleep_event_handler(co_power_sleep_state_t sleep_state,
co_power_status_t power_status)
{
    switch(sleep_state)
    {
        case POWER_SLEEP_ENTRY:
            break;

        case POWER_SLEEP_LEAVE_TOP_HALF:
            // Peripheral restore
            peripheral_restore();
            break;

        case POWER_SLEEP_LEAVE_BOTTOM_HALF:
            break;
    }
}
```

```
co_power_status_t power_sleep_status_handler(void)
{
    co_power_status_t pmu_status;

    pmu_status = pmu_power_status();

  return pmu_status;
}

static void hardware_init(void)
{
    pinmux_init();

    peripheral_init();

  co_power_register_user_status(power_sleep_status_handler);
  co_power_register_sleep_event(power_sleep_event_handler);
}

static void on_ble_evt(ble_evt_t* p_ble_evt)
{
  if (p_ble_evt->header.evt_id == BLE_GAP_EVT_DISCONNECTED) {
      log_debug("[GAP]: Disonnected. reason:0x%02X\n",
p_ble_evt->evt.gap_evt.params.disconnected.reason);
  } else if (p_ble_evt->header.evt_id == BLE_GAP_EVT_CONNECTED) {
      ble_gap_addr_t *addr =
&p_ble_evt->evt.gap_evt.params.connected.peer_addr;
      ble_gap_conn_params_t *param =
&p_ble_evt->evt.gap_evt.params.connected.conn_params;
      log_debug("[GAP]: Connected. %02x %02X:%02X:%02X:%02X:%02X:%02X
Intv:%d Lat:%d Tout:%d\n",
          addr->addr_type, addr->addr[5], addr->addr[4], addr->addr[3],
addr->addr[2], addr->addr[1], addr->addr[0],
          param->max_conn_interval, param->slave_latency,
param->conn_sup_timeout
      );
  } else if (p_ble_evt->header.evt_id == BLE_GAP_EVT_CONN_PARAM_UPDATE) {
      ble_gap_conn_params_t *p =
&p_ble_evt->evt.gap_evt.params.conn_param_update.conn_params;
      log_debug("[GAP]: Conn Param Updated: Intv:%d Lat:%d Tout:%d\n",
          p->max_conn_interval, p->slave_latency, p->conn_sup_timeout);
  } else if (p_ble_evt->header.evt_id == BLE_GATTC_EVT_MTU_CHANGED) {
      log_debug("[GATT]: MTU changed:%d\n",
```

```
p_ble_evt->evt.gattc_evt.params.mtu_changed.mtu);
    }
    app_sec_evt_cb(p_ble_evt);
    app_adv_evt_cb(p_ble_evt);
    service_common_evt_cb(p_ble_evt);
    service_onmicro_dfu_evt_cb(p_ble_evt);
    service_hid_evt_cb(p_ble_evt);
    ancs_evt_cb(p_ble_evt);
    ams_evt_cb(p_ble_evt);
    service_wechat_lite_evt_cb(p_ble_evt);
    service_tspp_evt_cb(p_ble_evt);
}

static void ble_stack_config(void)
{
     wdt_enable(0);

    // Select 32k clock for stack
     cc_sca_set(CC_SCA_1000PPM);

    // Enable sleep, SWD will be closed.
     co_power_sleep_enable(true);

    // xtal32m param
     pmu_xtal32m_change_param(20);

    // tx pwoer
    rf_tx_power_set(RF_TX_POWER_NORMAL);
}

/**********************************************************************
 * PUBLIC FUNCTIONS
 */

int main(void)
{
    hardware_init();
    ble_stack_config();

    om_ble_enable(NULL);
    softdevice_ble_evt_handler_set(on_ble_evt);

    service_common_init();
    service_tspp_init();
```

```
    service_onmicro_dfu_init();
    service_wechat_lite_init();
    service_hid_init();
     app_sec_init();

    const static ble_gap_addr_t gap_addr = {
        BLE_GAP_ADDR_TYPE_RANDOM_STATIC, {0x0d, 0xBF, 0x66, 0x21, 0xD0,
0xdd}
     };

    // Enter main loop.
     co_sche();

    return 0;
}
```

本例程主要实现对 GPIO 睡眠唤醒测试。

在主函数中首先对蓝牙栈配置和硬件初始化，在 ble_stack_config()中，对 SLEEP、进行了配置，在 hardware_init()中，对 23 个引脚配置为上拉，再对 UART0 和 UART1 进行配置，在 peripheral_init()中，初始化 GPIO，设置 GPIO 方向、双边沿中断方式和中断回调函数，唤醒引脚设置和唤醒引脚回调函数；然后 RWIP 初始化和重置；最后通过寄存器睡眠事件回调函数，来设置睡眠事件处理函数。

在 PC 端，Jlink 仿真器和上位机 Keil 连接后，将 GPIO 睡眠唤醒测试代码烧录至芯片中，烧录完成后芯片重新上电。当芯片处于睡眠状态时，将每一个引脚接通过高低电平的变化产生双边沿中断，进而将其唤醒；通过电源串联一个电流表，电流若为 10uA 以下则处于睡眠状态，若为大于 1mA 则处于唤醒状态。

1.3.4.例程 3：GPIO 上拉阻抗选择测试

BL1824X 的 23 个 IO 可以根据需要配置上拉阻抗，由 GPIO_PU_CTRL_1 和 GPIO_DRV_CTRL_0[22:0]共同作用控制。其中 GPIO_DRV_CTRL_0 为 bit1，GPIO_PU_CTRL_1 位 bit0。

```
typedef enum {
    PU_4K           ,
    PU_10K          ,
    PU_300K         ,
    PU_2M           ,
}pu_res_mode_t;

static void example_gpio_resistance_pull_up(pu_res_mode_t pu_res_mode)
{
    for (uint8_t i = PIN_MIN; i <= PIN_MAX; i++)
```

```
    {
        pinmux_config(i,PINMUX_GPIO_MODE_CFG);
        pmu_pin_mode_set(BIT_MASK(i), PMU_PIN_MODE_PU);
        gpio_open();
        gpio_set_direction(BIT_MASK(i), GPIO_INPUT);

        switch (pu_res_mode)
        {
            case PU_4K:
                HS_PMU->GPIO_DRV_CTRL_2 &= ~(1<<i);
                HS_PMU->GPIO_POL_1 &= ~(1<<i);
                break;

            case PU_10K:
                HS_PMU->GPIO_DRV_CTRL_2 |= (1<<i);
                HS_PMU->GPIO_POL_1 &= ~(1<<i);
                break;

            case PU_300K:
                HS_PMU->GPIO_DRV_CTRL_2 &= ~(1<<i);
                HS_PMU->GPIO_POL_1 |= (1<<i);
                break;

            case PU_2M:
                HS_PMU->GPIO_DRV_CTRL_2 |= (1<<i);
                HS_PMU->GPIO_POL_1 |= (1<<i);
                break;

            default:
                break;
        }
    }
}
```

1.4. GPIO 使用注意事项

- 目前，SDK 中提供的 GPIO 库函数，默认是不支持单边沿触发的，其目的是为了低功耗。如果客户不在乎功耗，可以修改 GPIO 相关函数 gpio_set_interrupt，从而可以达到单边沿触发的目的；
- 如果测试程序下载不到芯片中时，可以尝试 GPIO4（boot）引脚接地再下载；下载程序完成后，重新上电跑程序时，需要把 GPIO4（boot）引脚接地引脚拔掉。

- 在睡眠的时候，当设置的电平和睡眠时候的电平相同，不会阻止睡眠。
- 睡眠的时候 JLINK 脚会断开时，不能进行调试。
- 通过 pmu_dump(printf);函数打印信息来判断是否有 GPIO 阻止睡眠或漏电，打印信息如下图所示，从图中可得第二行中 wakeup_pin=cur_level 时，没有 GPIO 阻止睡眠；pull_up=cur_level 时，在睡眠中没有漏电情况；相反的，当不相等的时候，通过判断 cur_level 的 16 进制的哪一位不等，进而判断哪个 GPIO 影响睡眠或漏电。

```
running 0
[PMU] prevent_status=00000000
[PMU] wakeup_pin=01000004(cur_level=01000004 sleep_level=01000004)
[PMU] pull_up=FFFFFFDF(cur_level=FFFFFFDF) pull_down=00000000(cur_level=00000000)
all_cur_level=FFFFFFFF
[PMU] clocking: CPU(32MHz) SRAM(FF) SF0 SF1 SF2 UART0 GPIO
```

- BL1824X 的 23 个 IO 可以根据需要配置上拉阻抗，由 GPIO_PU_CTRL_1 和 GPIO_DRV_CTRL_0[22:0]共同作用控制。其中 GPIO_DRV_CTRL_0 为 bit1，GPIO_PU_CTRL_1 位 bit0。
- IO2/3 作为 audio 的语音输入引脚，与 audio 电路连接。IO2/3 不要另作他用。

2. UART1

2.1. 简介

UART 是 Universal Synchronous Asynchronous Receiver and Transmitter 的缩写，又称为通用异步收发传输器，它可以与外部设备进行全双工数据交换。UART 是一种通用串行数据总线，用于异步通信。该总线双向通信，可以实现全双工传输。

BL1824X 配有两个 UART：UART0 和 UART1。UART0 提供了一个灵活的全双工同步/异步收发器，它可以在四种模式下工作（一种同步模式和三种异步模式）。在模式 0 下，UART0 作为同步发射器/接收器工作；在模式 1 中，具有 8 个数据位和可编程的波特率；模式 2 有 9 个数据位，波特率固定为 Fclkper/32 或 Fclkper/64；模式 3 有 9 个数据位和可编程的波特率。UART1 支持以下波特率：4800、9600、14400、19200、38400、57600、76800、115200、230400、256000、460800、500000、512000、600000、750000、921600。可以设置 5 到 8 位的有效比特数，支持奇偶校验，可以设置 1 至 2 位的停止位，支持流控，支持 ISO7816 协议。

BL1824X 共有 2 个 UART 组，两个 UART 组对应的基地址为：

UART 编号	基地址
UART0	0x40080000
UART1	0x40040000

UART0 在代码中被宏定义为 HS_UART0，UART1 在代码中被宏定义为 HS_UART1。

2.2. API 介绍

2.2.1.UART 寄存器结构

```
typedef struct
{
    union
    {
        __I  uint32_t RBR;
        __O  uint32_t THR;
        __IO uint32_t DLL;
    };                              // offset：0x00
    union
    {
```

```
    __IO uint32_t DLH;
    __IO uint32_t IER;
};                              // offset：0x04
union
{
    __I  uint32_t IIR;
    __O  uint32_t FCR;
};                                // offset：0x08
__IO uint32_t LCR;                // offset：0x0C
__IO uint32_t MCR;                // offset：0x10
__I  uint32_t LSR;                // offset：0x14
__I  uint32_t MSR;                // offset：0x18
__IO uint32_t SCR;                // offset：0x1C
__IO uint32_t LPDLL;              // offset：0x20
__IO uint32_t LPDLH;              // offset：0x24
struct
{
    // ISO7816_CTRL0 offset：0x28
    __IO uint32_t ENABLE     :1;
    __IO uint32_t NOACK_EN   :1;      // In RX, don't return ack
    __IO uint32_t TRX_EN     :1;      // 0:TX 1:RX
    __IO uint32_t RETRANS_EN:1;   // In TX, enable re-tx on check error
    __IO uint32_t SAMPLE_DLY:8;
    __I  uint32_t TX_DONE    :1;
         uint32_t RESERVE1   :19;

    // ISO7816_CTRL1 offset：0x2C
    __I  uint32_t ERROR_CNT; //0-7bit:RX_ERROR 8-15bit:TX_ERROR
}ISO7816;
union
{
    __I  uint32_t SRBR[16];
    __O  uint32_t STHR[16];
};                                // offset：0x30
__IO uint32_t FAR;                // offset：0x70
__I  uint32_t TFR;
__O  uint32_t RFW;
__I  uint32_t USR;
__I  uint32_t TFL;
__I  uint32_t RFL;
__O  uint32_t SRR;
__IO uint32_t SRTS;
__IO uint32_t SBCR;
__IO uint32_t SDMAM;
```

```
    __IO uint32_t SFE;
    __IO uint32_t SRT;
    __IO uint32_t STET;
    __IO uint32_t HTX;
    __O   uint32_t DMASA;
    uint8_t Reservedac_f0[0xf4-0xac];
    __I uint32_t CPR;
    __I uint32_t UCV;
    __I uint32_t CTR;
} HS_UART_Type;//uart1

typedef struct
{
    union {
        uint32_t CON;
        struct
        {
            __IO uint32_t RX_INT_FLG    :1;
            __IO uint32_t TX_INT_FLG    :1;
            __IO uint32_t RX_BT         :1;
            __IO uint32_t TX_BT         :1;
            __IO uint32_t RX_EN         :1;
            __IO uint32_t SM_EN         :1;
            __IO uint32_t SM_MD         :2;
            __IO uint32_t RX_IRQ_EN     :1;
            __IO uint32_t TX_IRQ_EN     :1;
                 uint32_t RESERVE1      :22;
        }CTL;
    };

    union
    {
        __I   uint32_t RBR;
        __O   uint32_t THR;
    };
    __IO   uint32_t RELL;
    __IO   uint32_t RELH;
    __IO   uint32_t PCON;
    __IO   uint32_t ADON;
} HS_UART_EX_Type;//uart0
```

2.2.1.1. UART1 寄存器表

Offset	寄存器	描述
0x0000	RBR	接收缓冲寄存器
0x0000	THR	发送保持寄存器
0x0000	DLL	分频锁存器（低）
0x0004	DLH	分频锁存器（高）
0x0004	IER	中断使能寄存器
0x0008	IIR	中断标识寄存器
0x0008	FCR	FIFO 控制寄存器
0x000C	LCR	Line 控制控寄存器
0x0010	MCR	Modem 控制寄存器
0x0014	LSR	线路状态寄存器
0x0018	MSR	Modem 状态寄存器
0x0020	LPDLL	低功耗分频数锁寄存器（L）
0x0024	LPDLH	低功耗分频数锁寄存器（H）
0x0028	ISO7816_CTRL0	ISO7816 控制寄存器 0
0x002C	ISO7816_CTRL1	ISO7816 控制寄存器 1
0x0030--0x006C	SRBR	接收缓冲影子寄存器
0x0030--0x006C	STHR	发送保持影子寄存器
0x0070	FAR	FIFO 访问寄存器
0x0074	TFR	FIFO 传输寄存器
0x0078	RFW	FIFO 接收寄存器
0x007C	USR	UART 状态寄存器
0x0080	TFL	发送 FIFO 深度
0x0084	RFL	接收 FIFO 深度
0x0088	SRR	软件复位寄存器
0x008C	SRTS	请求发送影子寄存器
0x0090	SBCR	中断控制影子寄存器
0x0094	SDMAM	DMA 模式影子寄存器
0x0098	SFE	FIFO 使能影子寄存器
0x009C	SRT	RCVR 触发影子寄存器
0x00A0	STET	TX 空触发影子寄存器
0x00A4	HTX	停止 TX 寄存器
0x00A8	DMASA	DMA 软件确认寄存器

2.2.1.2. UART0 寄存器表

Offset	寄存器	描述
0x0000	CON	控制寄存器
0x0004	BUF	数据缓冲区
0x0040	RELL	波特率发生器重载寄存器（低位字节）
0x0044	RELH	波特率发生器重载寄存器（高位字节）
0x0400	PCON	功率控制寄存器
0x0404	ADCON	波特率选择寄存器

RBR address offset：0x0000

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	R	0x0	RBR	接收缓冲寄存器 UART 模式下串行输入端口（sin）或红外模式下串行红外输入端口（sir_in）上接收的数据字节。只有当线路状态寄存器（LCR）中的数据就绪（DR）位被设置时，该寄存器中的数据才有效。

THR address offset：0x0000

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	W	0x0	THR	发送保持寄存器 UART 模式下串行输出端口（sout）或红外模式下串行红外输出端口（sir_out_n）上传输的数据。只有当 THR 空（THRE）位（LSR[5]）被设置时，数据才应写入 THR。

DLL address offset：0x0000

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	RW	0x0	DLL	分频锁存器（低） 包含 UART 波特率除数的 16 位读/写除数锁存寄存器的低 8 位。 输出波特率等于串行时钟（如果是一个时钟设计，则为 pclk；如果是两个时钟设计（clock_MODE==启用），则为 sclk）频率除以波特率除数值的 16 倍，如下所示：波特率=（串行时钟频率）/（16*除数）。

DLH address offset：0x0004

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	RW	0x0	DLH	分频锁存器（高） 16 位读/写除数锁存寄存器的高 8 位，包含 UART 的波特率除数。 输出波特率等于串行时钟（如果是一个时钟设计，则为 pclk；如果是两个时钟设计（clock_MODE==启用），则为 sclk）频率除以波特率除数值的 16 倍，如下所示： 波特率=（串行时钟频率）/（16*除数）。

IER address offset：0x0004

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7	RW	0x0	PTIME	使能/禁用 THRE 中断的生成 0：禁用 1：使能
6:4	N/A	0x0	N/A	保留
3	RW	0x0	EDSSI	使能 Modem 状态中断 使能/禁用调制解调器状态中断的生成，第四高优先级中断。 0：禁用 1：使能
2	RW	0x0	ELSI	使能接收线路状态中断 使能/禁用接收线路状态中断的生成，最高优先级中断。 0：禁用 1：使能
1	RW	0x0	ETBEI	使能传输保持寄存器空中断 使能/禁用发送器保持寄存器空中断的生成，第三高优先级中断。 0：禁用 1：使能
0	RW	0x0	ERBFI	使能接收数据可用中断 使能/禁用接收数据可用中断和字符超时中断(如果在 FIFO 模式和 FIFOs 使能)的生成，第二高优先级中断。 0：禁用 1：使能

IIR address offset：0x0008

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留

7:6	R	0x0	FIFOSE	FIFO 使能 显示 FIFO 是使能还是禁用 00：禁用 11：使能
5:4	N/A	0x0	N/A	保留
3:0	R	0x1	IID	中断 ID。 优先级最高的挂起中断，有以下类型： 0000：调制解调器状态 0001：无中断等待处理 0010：THR 为空 0100：接收到可用的数据 0110：接收线路状态 0111：忙于检测 1100：字符超时

FCR address offset：0x0008

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:6	W	0x0	RT	RCVR 触发 用来选择在接收 FIFO 时触发电平，在接收数据时会生成可用中断。在自动流量控制模式中，它用于确定 rts_n 信号何时失效。有关 DMA 支持的详细信息，支持以下触发器级别： 00：FIFO 达到一个字符 01：FIFO 1/4 满时 10：FIFO 半满时 11：FIFO 差两个字符变满
5:4	W	0x0	TET	TX 空触发 用于选择空阈值水平，在此阈值水平上，当模式生效时产生 THRE 中断。有关 DMA 支持的详细信息，支持以下触发器级别: 00：FIFO 空 01：FIFO 剩下 2 个字符触发中断 10：FIFO 剩下 1/4 时触发中断 11：FIFO 剩下 1/2 时触发中断
3	W	0x0	DMAM	DMA 模式 0：模式 0，一次支持单个 DMA 数据传输 1：模式 1，支持多 DMA 数据传输
2	W	0x0	XFIFOR	XMIT FIFO 重置 0：禁用 1：使能 重置了传输 FIFO 的控制部分，并将 FIFO

				视为空。这个部分是“自我清理”。
1	W	0x0	RFIFOR	RCVR FIFO 重置 0：禁用 1：使能 重置接收 FIFO 的控制部分，并将 FIFO 视为空。这个部分是“自我清理”。
0	W	0x0	FIFOE	FIFO 使能 这将使能/禁用发送(XMIT)和接收(RCVR) FIFOs。当改变此位的值时，FIFO 的 XMIT 和 RCVR 控制器部分将会重置。 0：禁用 1：使能

LCR address offset：0x000C

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7	RW	0x0	DLAB	分频锁存访问位 设置 UART 的波特率之前，即操作分频锁存寄存器(DLL 和 DLH)，要将该位置 1。 0：禁用 1：使能
6	RW	0x0	BC	跳出控制位
5	N/A	0x0	N/A	保留
4	RW	0x0	EPS	奇偶校验选择 0：奇校验 1：偶校验
3	RW	0x0	PEN	校验使能 0：校验禁用 1：校验使能
2	RW	0x0	STOP	停止位 0：1 停止位 1：1.5 停止位(DLS==0) 1：2 停止位(DLS!=0)
1:0	RW	0x0	DLS	数据长度选择 选择外设传输和接收每个字符的数据位数。可以被选择区域的比特数如下： 00：5 比特 01：6 比特 10：7 比特 11：8 比特

MCR address offset：0x0010

Bit	R/W	Reset	Name	Description
31:7	N/A	0x0	N/A	保留
6	RW	0x0	SIRE	SIR 模式使能。 0：禁用 1：使能
5	RW	0x0	AFCE	自动流量控制使能。 0：禁用 1：使能
4	RW	0x0	LB	回环位 将 UART 放入诊断模式以进行测试 0：禁用 1：使能
3	RW	0x0	OUT2	输出 2 直接控制用户指定的 Output2(out2_n)输出。写入此位置的值会被反转，并在 out2_n 上被驱动输出，即： 0：out2_n 失效(逻辑 1) 1：out2_n 生效(逻辑 0)
2	RW	0x0	OUT1	输出 1 直接控制用户指定的 Output1(out1_n)输出。写入这个位置的值被反转，并在 out1_n 上被驱动输出，即: 0：out1_n 失效(逻辑 1) 1：out1_n 生效(逻辑 0)
1	RW	0x0	RTS	请求发送 直接控制请求发送(rts_n)输出。发送请求(rts_n)输出，用于通知调制解调器或数据集，UART 已准备好交换数据。 0：禁用 1：使能
0	RW	0x0	DTR	数据终端就绪 直接控制数据终端准备(dtr_n)输出。写入到此位置的值将被反转，并在 dtr_n 上被驱动输出，即： 0：dtr_n 失效(逻辑 1) 1：dtr_n 生效(逻辑 0)

LSR address offset：0x0014

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7	R	0x0	RFE	接收 FIFO 错误位 0：接收 FIFO 正确 1：接收 FIFO 错误

6	R	0x1	TEMT	发送空位
5	R	0x1	THRE	发送并保持寄存器空位
4	R	0x0	BI	跳出中断位
3	R	0x0	FE	帧错误位。 0：没有帧错误 1：帧错误
2	R	0x0	PE	校验错误位 0：校验正确 1：校验错误
1	R	0x0	OE	溢出错位 0：没有溢出错位 1：溢出错位
0	R	0x0	DR	准备数据位 指示接收器在 RBR 或接收器 FIFO 中，至少包含一个字符。 0：数据没有准备 1：数据准备

MSR address offset：0x0018

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7	R	0x0	DCD	数据载波检测 0：dcd_n 输入失效(逻辑 1) 1：dcd_n 输入生效(逻辑 0)
6	R	0x0	RI	振铃指示 0：ri_n 输入失效(逻辑 1) 1：ri_n 输入生效(逻辑 0)
5	R	0x0	DSR	数据设置就绪 0：dsr_n 输入失效 (逻辑 1) 1：dsr_n 输入生效 (逻辑 0)
4	R	0x0	CTS	清除发送 0：cts_n 输入失效(逻辑 1) 1：cts_n 输入生效(逻辑 0)
3	R	0x0	DDCD	Delta 数据载波检测 0：读取 MSR 后，dcd_n 没有变化 1：读取 MSR 后，dcd_n 有变化
2	R	0x0	TERI	环指示器后缘 0：读取 MSR 后，ri_n 没有变化 1：读取 MSR 后，ri_n 有变化
1	R	0x0	DDSR	Delta 数据设置就绪 0：读取 MSR 后，dsr_n 没有变化 1：读取 MSR 后，dsr_n 有变化
0	R	0x0	DCTS	Delta 清除发送 0：读取 MSR 后，cts_n 没有变化

				1：读取 MSR 后，cts_n 有变化

LPDLL address offset：0x0020

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	RW	0x0	LPDLL	低功耗分频数锁寄存器（L） 此寄存器 16 位中的低 8 位，可读写，包含 UART 的波特率分频数。

LPDLH address offset：0x0024

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	RW	0x0	LPDLH	低功耗分频数锁寄存器（H） 此寄存器 16 位中的高 8 位，可读写，包含 UART 的波特率分频数。

ISO7816_CTRL0 address offset：0x0028

Bit	R/W	Reset	Name	Description
31:13	N/A	0x0	N/A	保留
12	R	0x0	tx_done	TX 已完成
11:4	RW	0x0	sample_dly	采样 delay 值 用于调整 SIN 的采样时序
3	RW	0x0	retrans_en	奇偶校验错误重新转换 0：禁用 1：使能
2	RW	0x0	trx_oen	0：TX 1：RX
1	RW	0x0	nack_enable	使能 NACK 0：禁用 1：使能
0	RW	0x0	iso7816_en	使能 ISO7816 0：禁用 1：使能

ISO7816_CTRL1 address offset：0x002C

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:8	R	0x0	tx_perr_cnt	TX 奇偶校验错误计数器
7:0	R	0x0	rx_perr_cnt	RX 奇偶校验错误计数器

SRBR address offset：0x0030--0x006C

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留

7:0	R	0x0	SRBR	接收缓冲影子寄存器 RBR 的一个影子寄存器，已经分配了 16 个 32 位的位置，以便容纳来自主机的突发访问。该寄存器包含在 UART 模式下的串行输入端口(sin)。 当线路状态寄存器（LSR）中的数据就绪（DR）位被设置时，该寄存器中的数据才有效。

STHR address offset：0x0030--0x006C

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	W	0x0	STHR	发送保持影子寄存器 THR 的一个影子寄存器，已经分配了 16 个 32 位的位置，以适应来自主机的突发访问。该寄存器包含在 UART 模式下串行输出端口(sout)。 只有当设置 THR Empty (THRE)位(LSR[5])时，数据才会写入 THR。

FAR address offset：0x0070

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	N/A	保留
0	RW	0x0	FIFO Access Register	FIFO 访问寄存器 当 FIFO_ACCESS=No 时，写入无效，始终可读。该寄存器用于启用 FIFO 访问模式进行测试，以便主设备可以写入接收 FIFO 当 FIFO 被实现和启用时。当 FIFO 未实现或未启用时，它允许 RBR 由主机写入，THR 由主机读取。 0=FIFO 访问模式已禁用 1=启用 FIFO 访问模式 注意，当启用/禁用 FIFO 访问模式时，接收 FIFO 和发送 FIFO 的控制部分被重置，FIFO 被视为空。

TFR address offset：0x0074

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	R	0x0	Transmit FIFO Read	读取 FIFO 传输寄存器 传输 FIFO 读取。这些位仅在启用 FIFO 访问模式时有效（FAR[0]设置为 1）。 当 FIFO 被实现并启用时，读取此寄存器

				将在发送 FIFO 的顶部给出数据。每次连续读取都会弹出传输 FIFO，并给出当前位于 FIFO 顶部的下一个数据值。 当 FIFO 未实现或未启用时，读取此寄存器会在 THR 中给出数据。

RFW address offset：0x0078

Bit	R/W	Reset	Name	Description
31:10	N/A	0x0	N/A	保留和读取为零。
9	W	0x0	RFFE	接收 FIFO 帧错误 这些位仅在启用 FIFO 访问模式时有效（FAR[0]设置为 1）。当实施 FIFO 时该位用于将帧错误检测信息写入接收 FIFO。当 FIFO 未实现或未启用时，该位用于将帧错误检测信息写入 RBR。
8	W	0x0	RFPE	接收 FIFO 奇偶校验错误 这些位仅在启用 FIFO 访问模式时有效（FAR[0]设置为 1）。当 FIFO 被实现和启用时，该位用于将奇偶校验错误检测信息写入接收 FIFO。 当 FIFO 未实现或未启用时，该位用于将奇偶校验错误检测信息写入 RBR。
7:0	W	0x0	RFWD	接收 FIFO 写入数据 这些位仅在启用 FIFO 访问模式时有效（FAR[0]设置为 1）。当 FIFO 被实现和启用时，写入 RFWD 的数据被推入接收 FIFO。每次连续写入都将新数据推送到接收 FIFO 中的下一个写入位置。当 FIFO 未实现或未启用时，写入 RFWD 的数据被推入 RBR。

USR address offset：0x007C

Bit	R/W	Reset	Name	Description
31:5	N/A	0x0	N/A	保留
4	R	0x0	RFF	接收 FIFO 溢出 此位仅在 FIFO_STAT == YES 时有效，这用于指示接收的 FIFO 已经完全满了。 0：接收 FIFO 未满 1：接收 FIFO 满
3	R	0x0	RFNE	接收 FIFO 不为空 此位仅在 FIFO_STAT == YES 时有效，这用于指示接收 FIFO 包含一个或多个项。 0：接收 FIFO 为空

				1：接收 FIFO 不为空
2	R	0x01	TFE	传输的 FIFO 为空 此位仅在 FIFO_STAT == YES 时有效，这是用来表明发送 FIFO 是完全空的。 0：传输 FIFO 不为空 1：发送 FIFO 为空
1	R	0x01	TFNF	传输 FIFO 未满 此位仅在 FIFO_STAT == YES 时有效，这是用来表示传输的 FIFO 不完全。 0：发送 FIFO 已满 1：发送 FIFO 未满
0	R	0x0	BUSY	UART 繁忙 该位仅在 UART_16550_COMPATIBLE == NO 时有效，表示正在进行串行传输。清除时，表示 DW_apb_uart 处于空闲或非活动状态。 0：uart 处于空闲或非活动状态 1：uart 正在忙(正在传输数据)

TFL address offset：0x0080

Bit	R/W	Reset	Name	Description
31:5	N/A	0x0	N/A	保留
4:0	R	0x0	TFL	传输 FIFO 的水平 该参数表示发送 FIFO 的数据项的数量

RFL address offset：0x0084

Bit	R/W	Reset	Name	Description
31:5	N/A	0x0	N/A	保留
4:0	R	0x0	RFL	接收 FIFO 的水平 接收 FIFO 的数据项的数量

SRR address offset：0x0088

Bit	R/W	Reset	Name	Description
31:3	N/A	0x0	N/A	保留
2	W	0x0	XFR	XMIT FIFO 重置 0：禁用 1：使能 XMIT FIFO 复位位（FCR[2]）的影子寄存器。 注意，这部分是“自我清除”的，没有必要清除此位。
1	W	0x0	RFR	RCVR FIFO 重置 0：禁用 1：使能

				影子寄存器的 RCVR FIFO 复位位（FCR[1]）。 注意，这部分是“自我清除”的，没有必要清除此位。
0	W	0x0	UR	UART 重置 0：禁用 1：使能 这将异步地重置串口，并同步地删除重置断言。

SRTS address offset：0x008C

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	N/A	保留
0	RW	0x0	SRTS	请求发送影子寄存器 0：禁用 1：使能 RTS 位(MCR[1])的影子寄存器，可以用它来消除在 MCR 上执行读-修改-写的负担，这用于直接控制请求发送(rts_n)输出。发送请求(rts_n)输出用于通知调制解调器或数据集 DW_apb_uart 已准备好交换数据。 当自动 RTS 流量控制未启用(MCR[5] = 0)，rts_n 信号被设置低，可以通过编程 MCR[1] (RTS)调高。 在自动流量控制中，AFCE_MODE == Enabled，有效(MCR[5] = 1)和 FIFOs 使能 (FCR[0] = 1)，rts_n 输出被以同样的方式控制，但可以与接收 FIFO 阈值触发器(rts_n 在超过阈值时是无效高电平)一起门控。 注意，在环回模式(MCR[4]=1)中，rts_n 输出保持为无效高电平，而该位置的值在内部循环回输入。

SBCR address offset：0x0090

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	N/A	保留
0	RW	0x0	SBCR	阴影中断控制位 0：禁用 1：使能 中断位(LCR[6])的影子寄存器可以用它来消除必须在 LCR 上执行读、修改、写的负担。这用于使中断条件被发送到接收

				设备。 如果设置为 1，则串行输出被强制为间隔(逻辑 0)状态。当不在环回模式(由 MCR[4]确定)时，sout 线被强制降低，直到中断位被清除。 如果 SIR_MODE 使能和激活(MCR[6] = 1)，则 sir_out_n 线路连续脉冲。当在环回模式，中断条件是内部环回接收器。

SDMAM address offset：0x0094

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	N/A	保留
0	RW	0x0	SDMAM	DMA 模式影子寄存器 DMA 模式位(FCR[3])的阴影寄存器，这可以用来消除必须将以前写入的值存储到内存中的 FCR 中，并且必须屏蔽这个值以便只更新 DMA Mode 位的负担。 当没有选择额外的 DMA 握手信号时(DMA_EXTRA == NO)，这决定了 DMA 用于 dma_tx_req_n 和 dma_rx_req_n 输出信号的方式。 0：模式 0 1：模式 1

SFE address offset：0x0098

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	N/A	保留
0	RW	0x0	SFE	FIFO 使能影子寄存器 0：禁用 1：使能 FIFO 使能位（FCR[0]）的阴影寄存器，这可以用来消除以前写入的值存储到内存中的 FCR 的多余数据，并必须屏蔽这个值，以便只有 FIFO 使能位得到更新，这将使能/禁用发送(XMIT)和接收(RCVR) FIFO。 若在使能后，将该位设置为零(禁用)，则 FIFO 的 XMIT 和 RCVR 控制器部分都将重置。

SRT address offset：0x009C

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	保留
1:0	RW	0x0	SRT	RCVR 触发影子寄存器

				RCVR 触发位(FCR[7:6])的影子寄存器，这可以用来消除以前写入的值存储到内存中 FCR 的多余数据，并且必须屏蔽这个值，以便只更新 RCVR 触发器位。 此寄存器是用来选择在接收 FIFO 的触发电平，进而接收数据和可用中断被产生。它还确定当 DMA Mode (FCR[3])=1 时，dma_rx_req_n 信号何时失效。支持以下触发器级别： 00：FIFO 达到一个字时 01：FIFO 1/4 满时 10：FIFO 半满时 11：FIFO 差两个字变满时

STET address offset：0x00A0

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	保留
1:0	RW	0x0	STET	阴影 TX 空触发影子寄存器 TX 空触发位的影子寄存器(FCR[5:4])，这可以用来消除之前写入存储到内存中的 FCR 中的值，并且必须屏蔽这个值，以便只更新 TX 空触发器位。 这用于选择空阈值水平，在此阈值水平上，当模式处于激活状态时生成 THRE 中断。支持以下触发器级别： 00：FIFO 空 01：FIFO 达到两个字时 10：FIFO 1/4 满时 11：FIFO 半满时 依赖项：当 THRE_MODE_USER==Disabled 时，写入无效。

HTX address offset：0x00A4

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	N/A	保留
0	RW	0x0	HTX	停止传输测试寄存器 当 FIFO 执行和使能时，发送 FIFO 可以被主机填充。 0：停止发送已禁用 1：停止发送使能 注意，如果执行了 FIFO 但没有使能，则暂停 TX 寄存器的设置对操作没有影响。 依赖关系：当 FIFO_MODE==None 时，写入不起作用。

DMASA address offset：0x00A8

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	N/A	保留
0	W	0x0	DMASA	DMA 软件执行确认寄存器 该寄存器用于执行 DMA 软件确认，若出现错误条件，则传输需要终止。例如，如果 DMA 禁用了通道，那么 uart 应该清除它的请求。 这导致了 TX 请求，单个 TX，RX 请求和单个 RX 信号失效。 注意，这部分是“自我清除”的，没有必要清除此位。

CON address offset: 0x0000

Bit	R/W	Reset	Name	Description
31:10	N/A	0x0	N/A	保留
9	R/W	0x0	Txen	Tx 中断使能寄存器
8	R/W	0x0	Rxen	Rx 中断使能寄存器
7:6	R/W	0x0	Mode	型号选择寄存器 01：模式 1，8 位 uart，带宽速率=（（2^smod）*pclk/64）*（2^10 rel），rel 是 S0REL 寄存器（s0relh，s0rell）的内容 10： 模式 2，9 位 uart，在模式 2 中，uart 作为异步发射机/接收机工作，具有 9 个数据位，波特率固定为 pclkper/32 或 pclkper/64，具体取决于 PCON 寄存器的“smod”位的设置。 通过写入 BUF 寄存器开始传输。“txd0”引脚输出数据。传输的第一位是起始位（总是 0），然后 9 位数据继续进行，第 9 位从 CON 寄存器的位“tb80”中取出，之后传输停止位（总是 1）。 “rxd0i”引脚输入数据。当接收开始时，串行端口 0 与引脚“rxd0”处检测到的下降沿同步。输入数据在“buf”寄存器中完成接收后可用，第 9 位可用作 CON 寄存器中的“rb80”标志。在接收过程中，“s0buf”和“rb80”保持不变，直到完成。 11： 模式 3，8 位 uart，带宽速率可变，如果 ADCON[7]为 0, 带宽速率=（2^smod）*pclk/64）*（2^10rel），rel 是 S0REL 寄存器（s0relh，s0rell）的内容

5	R/W	0x0	SM20	多处理器通信启用，UART0的模式2和模式3中接收9位的功能可用于多处理器通信。当CON寄存器的“sm20”位被设置时，仅当第9个接收位（“CON”的“rb80”）为1时才产生接收中断。否则，接收时不会产生中断。
4	RW	0x0	REN0	串行接收使能 如果设置为高串行接收，则启用uart。否则，uart的串行接收被禁用。
3	RW	0x0	TB80	**Transmitter bit 8** 在模式2和模式3中通过uart传输数据时使用此位。此位的状态与第9个传输位的状态相对应（例如奇偶校验或多处理器通信）。它由软件控制。
2	R/W	0x0	RB80	**Received bit 8** 该位在模式2和模式3中通过UART接收数据时使用。它反映了第9个接收位的状态。在模式1中，如果启用了多处理器通信（sm20=0），则该位是接收到的停止位
1	R/W	0x0	TI0	传输中断标志 它表示在UART完成串行传输。 在所有模式下，它都位于停止位的开头。 必须通过软件清除。
0	R/W	0x0	RI0	接收中断标志 它在UART完成串行接收后由硬件设置。 在所有模式下，它都处于停止位的中间。 必须通过软件清除。

BUF address offset: 0x0004

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7**:0**	R/W	0x0	DATA BUF	将数据写入该寄存器将数据设置在串行输出缓冲器中，并通过UART开始传输。 从BUF读取从串行接收缓冲器读取数据。

RELL address offset: 0x0040

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7**:0**	R/W	0xD9	RELL	UART重新加载寄存器用于生成UART波特率。仅使用10位。来自RELL的8位作为低位，来自RELH的2位（RELH.1，RELH.0）作为高位。

RELH address offset: 0x0044

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:**0**	R/W	0x03	RELH	UART重新加载寄存器用于生成UART波特率。仅使用10位。来自RELL的8位作为低位，来自RELH的2位（RELH.1，RELH.0）作为高位。

PCON address offset: 0x0400

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	N/A	保留
0	R/W	0x03	SMOD	UART波特率选择（波特率倍增器）（在模式1和模式3中）

ADCON address offset: 0x0404

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:**0**	R/W	0x0	BD	Uart波特率选择（模式1和3）当为1时，使用额外的内部波特率发生器（需要设置为1）

2.2.2.变量参数

2.2.2.1. uart_rx_callback_t

```
typedef void (*uart_rx_callback_t)(uint8_t data);
```

接收事件回调。

成员名称	描述
uart_rx_callback_t	UART 接收回调函数指针类型

2.2.2.2. uart_tx_cmp_callback_t

```
typedef void (*uart_tx_cmp_callback_t)(void);
```

发送完成回调。

成员名称	描述
uart_tx_cmp_callback_t	UART 发送回调函数指针类型

2.2.2.3. uart_flow_ctrl_t

```
typedef enum
{
    UART_FLOW_CTRL_DISABLED,
    UART_FLOW_CTRL_ENABLED,
}uart_flow_ctrl_t;
```

流控。

成员名称	描述
UART_FLOW_CTRL_DISABLED	UART 流控禁用
UART_FLOW_CTRL_ENABLED	UART 流控使能

2.2.3.库函数

2.2.3.1. uart_open

函数名	uart_open
函数原型	void uart_open(HS_UART_Type *uart, uint32_t baud_rate, uart_flow_ctrl_t flow_ctrl, uart_rx_callback_t uart_rx_cb)
功能描述	uart 初始化
输入参数 1	uart：指向结构 HS_UART_Type 的指针，包含 uart 控制器相关信息
输入参数 2	baud_rate：波特率设置
输入参数 3	flow_ctrl：指向结构 uart_flow_ctrl_t 的指针，流控使能选择
输入参数 4	uart_rx_cb：指向结构 uart_rx_callback_t 的指针，包含接收中断函数指针。
输出参数	无
返回值	无

2.2.3.2. uart_close

函数名	uart_close
函数原型	void uart_close(HS_UART_Type *uart)
功能描述	uart 关闭
输入参数	uart：指向结构 HS_UART_Type 的指针，包含 uart 控制器相关信息
输出参数	无
返回值	无

2.2.3.3. uart_send_noblock

函数名	uart_send_noblock
函数原型	void uart_send_noblock(HS_UART_Type *uart, uint8_t **buf, unsigned *length, uart_tx_cmp_callback_t txcb)
功能描述	uart 无阻塞发送
输入参数 1	uart：指向结构 HS_UART_Type 的指针，包含 uart 控制器相关信息
输入参数 2	buf：发送数据的指针
输入参数 3	length：发送数据的长度
输入参数 4	txcb：指向结构 uart_tx_cmp_callback_t 的指针，包含发送完成回调项
返回值	无

2.2.3.4. uart_wait_send_finish

函数名	uart_wait_send_finish
函数原型	void uart_wait_send_finish(HS_UART_Type *uart)
功能描述	UART 等待发送完成
输入参数	uart：指向结构 HS_UART_Type 的指针，包含 uart 控制器相关信息
返回值	无

2.2.3.5. uart_send_block

函数名	uart_send_block
函数原型	uart_send_block(HS_UART_Type *uart, const uint8_t *buf, unsigned length)
功能描述	uart 阻塞发送
输入参数 1	uart：指向结构 HS_UART_Type 的指针，包含 uart 控制器相关信息
输入参数 2	buf：发送数据的指针
输入参数 3	length：发送数据的长度
返回值	无

2.2.3.6. uart_empty_fifo

函数名	uart_empty_fifo
函数原型	uart_empty_fifo(HS_UART_Type *uart)
功能描述	UART 清空 FIFO
输入参数	uart：指向结构 HS_UART_Type 的指针，包含 uart 控制器相关信息
返回值	无

2.2.3.7. iso7816_init

函数名	iso7816_init
函数原型	void iso7816_init(uint32_t clock, uint32_t etu, uart_rx_callback_t rx_cb)
功能描述	iso7816 初始化
输入参数 1	clock：设置时钟
输入参数 2	etu：波特率设置
输入参数 3	rx_cb：指向结构 uart_rx_callback_t 的指针，包含接收事件回调项
返回值	无

2.2.3.8. iso7816_send_block

函数名	iso7816_send_block
函数原型	void iso7816_send_block(const uint8_t *buf, unsigned length)
功能描述	iso7816 阻塞发送数据
输入参数 1	buf：发送数据的指针
输入参数 2	length：发送数据的长度
返回值	无

2.2.3.9. uart_ex_open

函数名	uart_ex_open
函数原型	void uart_ex_open(HS_UART_EX_Type *uart, const uart_ex_cfg_t *uart_ex_cfg);
功能描述	uart0 初始化
输入参数 1	uart：指向结构 HS_UART_EX_Type 的指针，包含 uart 控制器相关信息
输入参数 2	uart_ex_cfg：指向结构体 uart_ex_cfg_t 的指针
返回值	无

2.2.3.10. uart_ex_send_block

函数名	uart_ex_send_block
函数原型	void uart_ex_send_block(HS_UART_EX_Type *uart, const uint8_t *buf, uint32_t length);
功能描述	uart0 阻塞发送
输入参数 1	uart：指向结构 HS_UART_EX_Type 的指针，包含 uart 控制器相关信息
输入参数 2	buf：发送数据的指针
输入参数 3	length：发送数据的长度
返回值	无

2.3. UART 应用例程

定义 GPIO 参数变量和结构体变量数组：

```
#define PIN_UART0_TX                    7
#define PIN_UART0_TX                    8
#define PIN_UART1_TX                    5
#define PIN_UART1_RX                    6
#define UART_LSR_THRE    0x20
#define UART_LSR_TEMT    0x40
#define UART_LCR_STB     0x04
#define UART_MODE_IRQ
#define BUF_RX_SIZE 100
#define UART_LSR_THRE    0x20        /* Xmit holding register empty */
#define UART_LSR_TEMT    0x40        /* Xmitter empty */
typedef enum {
    UART_LCR_5N1    = 0    ,
    UART_LCR_6N1    = 0x01,
    UART_LCR_7N1    = 0x02,
}data_bit_t;
typedef enum {
    EVEN_PARITY     ,
    ODD_PARITY      ,
}parity_t;
typedef enum {
    FLOW_CTRL_ENABLE_TX     ,
    FLOW_CTRL_ENABLE_RX     ,
    FLOW_CTRL_ENABLE_TX_RX  ,
}flow_ctrl_mode_t;
uint32_t uart0_baud_rates[] =
{1000,2000,4000,5000,10000,20000,25000,50000,100000,500000,1000000};
// 0    1    2    3     4    5     6     7      8     9       10

uint32_t uart1_baud_rates[] =
{4800,9600,14400,19200,38400,57600,76800,115200,230400,256000,460800,
// 0    1    2     3     4     5     6      7     8       9      10
                              500000,512000,600000,750000,921600};
                             //  11    12     13      14     15
```

定义了串口 0、串口 1 的发送和接收引脚、串口模式的中断和接收 BUF 的大小；定义线路状态寄存器中的 THR 位、TEMT 位和停止位；定义结构体数据位、结构体奇偶校验位和流控模式结构体；定义数据接收数组初值、数据接收 ID 初值、串口序号初值和串口波特率数组。

2.3.1.例程 1：不同波特率收发

软件实现：

```
//uart1 test
static void test_uart1_all_baudrates(void)
{
    pinmux_config(5, PINMUX_UART1_SDA_O_CFG);
    pinmux_config(6, PINMUX_UART1_SDA_I_CFG);

    uint8_t param_num = sizeof(uart1_baud_rates)
/sizeof(uart1_baud_rates[0]);
    for (uint8_t i = 1; i < param_num; i++)
    {
        uart_open(HS_UART1, uart1_baud_rates[i],
UART_FLOW_CTRL_ENABLED, receive_handler);
        log_debug("baudrate now is : %d \n\n",uart1_baud_rates[i]);
        if (i + 1 < param_num )
        {
            log_debug("next baudrate: %d test will begin in
15s\n\n",uart1_baud_rates[i + 1]);
            log_debug("please change XCOM/SSCOM baudrate to %d
\n\n",uart1_baud_rates[i + 1]);
        }
        delay_s(15);
    }
    log_debug("\n\n!!!baudrate 4800 test will begin in 15s \n\n");
    log_debug("please change XCOM/SSCOM baudrate to 4800  \n\n");

    /* switch to 32M for uart baud 4800 */
    /*-------------------------------------*/
    CPM_ANA_CLK_ENABLE();
    pmu_topclk_rc32m_power_enable(true);
    pmu_topclk_switch_to_rc32m();
    pmu_topclk_xtal32m_power_enable(false);
    CPM_ANA_CLK_RESTORE();
    /*-------------------------------------*/

    uart_open(HS_UART1, uart1_baud_rates[0], UART_FLOW_CTRL_ENABLED,
receive_handler);

    delay_s(15);
    log_debug("uart1 test has finished \r\n");
}
```

在不同波特率收发测试程序中，首先配置串口的数据输入和输出引脚，然后利用 for 循环测试不同波特率的发送和接收能力，最后打印串口测试完成。

在不同波特率收发测试函数时，先设置串口调试工具的波特率，即可以测试 tx，也可以测试 rx；根据串口输出提示来设置不同波特率，使用串口助手发送数据看 UART 打印信息如下图：

```
baudrate now is : 4800

next baudrate: 9600 test will begin in 20s

please change XCOM/SSCOM baudrate to 9600

1234567
baudrate now is : 9600

next baudrate: 14400 test will begin in 20s

please change XCOM/SSCOM baudrate to 14400

1234567
baudrate now is : 14400

next baudrate: 19200 test will begin in 20s

please change XCOM/SSCOM baudrate to 19200

1234567
baudrate now is : 19200

next baudrate: 38400 test will begin in 20s

please change XCOM/SSCOM baudrate to 38400

1234567
baudrate now is : 38400

next baudrate: 57600 test will begin in 20s
```

图 2.1 不同波特率收发测试

如上图所示，根据串口打印信息观察串口打印是否正确，如果全都打印正确，则不同波特率收发测试通过。

2.3.2.例程 2：不同有效位收发

软件实现：

```
static void test_uart1_data_bit(uint8_t data_num)
{
    co_assert(data_num >= 5 && data_num <= 7);
    uart_index = 1;

    pinmux_config(5, PINMUX_UART1_SDA_O_CFG);
    pinmux_config(6, PINMUX_UART1_SDA_I_CFG);

    uart_open(HS_UART1, 115200, UART_FLOW_CTRL_DISABLED,
receive_handler);

    if (5 == data_num) {
```

```
        HS_UART1->LCR = UART_LCR_5N1;
    }
    else if (6 == data_num)
    {
        HS_UART1->LCR = UART_LCR_6N1;
    }
    else if (7 == data_num)
    {
        HS_UART1->LCR = UART_LCR_7N1;
    }
    uint8_t buff[] = {0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xF8};

    for (uint8_t i = 0; i < sizeof(buff)/sizeof(uint8_t); i++)
    {
        HS_UART1->THR = buff[i];
        while (!(HS_UART1->LSR & UART_LSR_THRE));
    }
    while (!(HS_UART1->LSR & UART_LSR_TEMT));
}
```

在不同有效位收发测试程序中，首先配置串口的数据输入引脚、输出引脚和打开串口，然后根据函数输入参数，来配置串口数据有效位和设置发送数据 BUFF，最后利用 for 循环利用 for 循环把发送数据的 BUFF 赋值给 THR 寄存器，进而来发送数据，并通过判断串口中线路状态寄存器中的 TEMT 位是否为 1，若为 1 则跳出循环。

```
//uart1 不同有效位收发
test_uart1_data_bit(5);
```

UART 数据构成为：起始位、数据位、校验位和停止位；对串口数据位的有效位收发函数，入参配置为 UART1，有效位设置为 5，6 或 7；例如有效位设置为 5，芯片发送 F1F2F3F4，串口助手的波特率调为 115200，数据位调为 5 位，上位机串口打印若图所示：

(a) 串口助手上电打印 **(b) 串口助手发送数据打印**

图 2.2 不同有效位收发测试

上位机串口助手上电时打印 11、12、13、14、15、16、17 和 18；串口 16 进制发送 F1F2F3F4，串口显示 11、12、14 和 14，则不同有效位收发测试通过。其他有效位收发

测试操作与举例类似。

2.3.3.例程 3：不同校验位收发

软件实现：

```
static void test_uart1_parity(parity_t parity)
{
    co_assert(EVEN_PARITY == parity || ODD_PARITY == parity);

    uart_index = 1;
    pinmux_config(5, PINMUX_UART1_SDA_O_CFG);
    pinmux_config(6, PINMUX_UART1_SDA_I_CFG);

    uart_open(HS_UART1, 115200, UART_FLOW_CTRL_DISABLED,
receive_handler);

    if (EVEN_PARITY == parity)
    {
        HS_UART1->LCR = UART_LCR_8N1| UART_LCR_PEN |
UART_LCR_EPS;   //even
        //log_debug("EVEN check pass **\n");
    }
    else if (ODD_PARITY == parity)
    {
        HS_UART1->LCR = UART_LCR_8N1|
UART_LCR_PEN;                 //odd
        //log_debug("ODD check pass **\n");
    }

    uint8_t buff[] = {0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xF8};
    for (uint8_t i = 0; i < sizeof(buff)/sizeof(uint8_t); i++)
    {
        HS_UART1->THR = buff[i];
        while (!(HS_UART1->LSR & UART_LSR_THRE));
    }
    while (!(HS_UART1->LSR & UART_LSR_TEMT));
}
```

在不同校验位收发测试程序中，首先配置串口的数据输入、输出引脚和打开串口，然后根据函数入口参数来配置串口的奇偶校验位和设置发送数据 BUFF，最后利用 for 循环把发送数据的 BUFF 赋值给 THR 寄存器，进而来发送数据，并通过判断串口中线路状态寄存器中的 TEMT 位是否为 1，若为 1 则跳出循环。

```
//uart1 不同校验方式收发
```

```
test_uart_parity(HS_UART0,EVEN_PARITY);
test_uart_parity(HS_UART0,ODD_PARITY);
```

对串口校验位收发函数的入参配置为 UART1，校验位选择 EVEN_PARITY 或 ODD_PARITY；

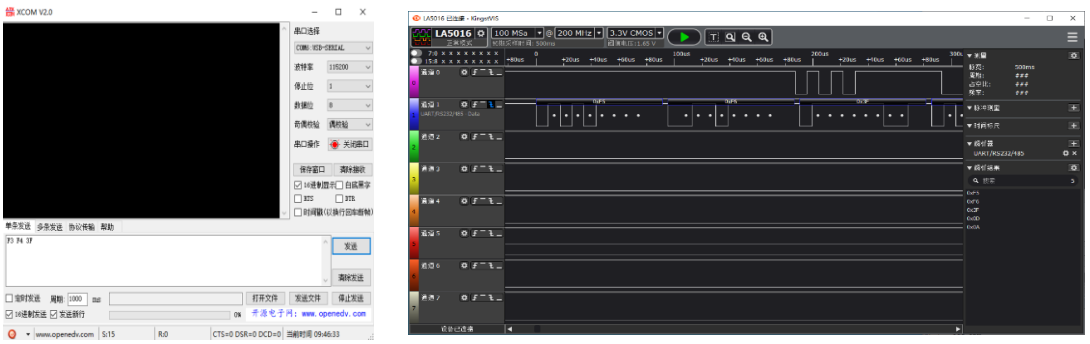
例如选择 UART1，偶校验，对于偶校验，数据中 1 的个数位偶数，则校验位为 0，否则为 1；借助逻辑分析仪，将 UART 的 TX 引脚接到逻辑分析仪的上位机，上位机配置为自动检测波形；逻辑分析仪一路连接板子 UART TX，另一路连接板子 UART RX 同时连接 USB 串口的 TX（三者相连）。

程序运行后 UART 首先发送数据，逻辑分析仪看波形是否正确，如下图所示：

(a) 逻辑分析仪配置 **(b) 逻辑分析仪抓取结果**

图 2.3 芯片发送数据逻辑分析仪抓取

程序发送 F1 F2 F3 F4 F7 F8，如图所示，逻辑分析仪抓取结果表明串口发送数据正确；然后上位机通过串口助手来发送数据 F3 F4 3F，芯片从 RX 接收到数据然后再通过 TX 发送，通过逻辑分析仪抓取数据来比较，是否发送正确，如图所示：



(a) 上位机发送数据 **(b) 逻辑分析仪抓取结果**

图 2.4 上位机发送数据逻辑分析仪抓取

通过逻辑分析仪结果可知，上位机发送数据和逻辑分析仪抓取结果一致，则 UART1 的偶校验测试通过。其他校验位收发测试操作与举例类似。

2.3.4.例程 4：不同停止位收发

软件实现：

```
static void test_uart1_stop_bit(uint8_t data_bit)
{
    co_assert(data_bit >= 5 && data_bit <= 8);
    uart_index = 1;
    pinmux_config(5, PINMUX_UART1_SDA_O_CFG);
    pinmux_config(6, PINMUX_UART1_SDA_I_CFG);
    uart_open(HS_UART1, 115200, UART_FLOW_CTRL_DISABLED,
receive_handler);
    if (5 == data_bit)
    {
        HS_UART1->LCR = UART_LCR_5N1 | UART_LCR_STB;
    }
    else if (6 == data_bit)
    {
        HS_UART1->LCR = UART_LCR_6N1 | UART_LCR_STB;
    }
    else if (7 == data_bit)
    {
        HS_UART1->LCR = UART_LCR_7N1 | UART_LCR_STB;
    }
    else if (8 == data_bit)
    {
        HS_UART1->LCR = UART_LCR_8N1 | UART_LCR_STB;
    }


    uint8_t buff[] = {0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7, 0xF8};
    for (uint8_t i = 0; i < sizeof(buff)/sizeof(uint8_t); i++)
    {
        HS_UART1->THR = buff[i];
        while (!(HS_UART1->LSR & UART_LSR_THRE));
    }
    while (!(HS_UART1->LSR & UART_LSR_TEMT));
}
```

在不同停止位收发程序中，首先配置串口的数据输入和输出引脚，并打开串口，然后根据函数入口参数来配置串口的数据位和停止位，并设置发送数据 BUFF，最后利用 for 循环把发送数据的 BUFF 赋值给 THR 寄存器，进而来发送数据，并通过判断串口中线路状态寄存器中的 TEMT 位是否为 1，若为 1 则跳出循环。

```
//uart 不同停止位 收发
//测试方法以及连接，与测试不同校验方式一致。
    test_uart_stop_bit(HS_UART1, 5);    //(数据位:5, 停止位:1.5)
    test_uart_stop_bit(HS_UART1, 6);    //(数据位:6, 停止位:2)
    test_uart_stop_bit(HS_UART1, 7);    //(数据位:7, 停止位:2)
    test_uart_stop_bit(HS_UART1, 8);    //(数据位:8, 停止位:2)
```

对串口校验位收发函数的入参配置为UART1，停止位可以选择1.5或2位；测试方法、连接方式和测试不同校验位方式一致。

例如选择数据位为8，停止位为2，配置为UART1，程序发出的数据为F1 F2 F3 F4 F7 F8，如图所示：

(a) 逻辑分析仪配置 **(b) 逻辑分析仪抓取结果**

图 2.5 芯片发送数据逻辑分析仪抓取

如上图所示，逻辑分析仪抓取结果表明串口发送数据正确；然后上位机通过串口助手来发送数据F5 F6 3F，芯片从RX接收到数据然后再通过TX发送，通过逻辑分析仪抓取数据来比较，是否发送正确，如下图所示：

(a) 上位机发送数据 **(b) 逻辑分析仪抓取结果**

图 2.6 上位机发送数据逻辑分析仪抓取

通过逻辑分析仪结果可知，上位机发送数据和逻辑分析仪抓取结果一致，则 UART1的数据位为8，停止位为2测试通过。其他校验位收发测试操作与举例类似。

2.3.5.例程 5：使用流控收发

软件实现：

```
static void test_uart1_flow_ctrl(flow_ctrl_mode_t flow_ctrl_mode)
{
    uart_index = 1;
    pinmux_config(5, PINMUX_UART1_SDA_O_CFG);
    pinmux_config(6, PINMUX_UART1_SDA_I_CFG);

    uart_open(HS_UART1, 115200, UART_FLOW_CTRL_ENABLED,
receive_handler);

    //发送流控
    if (FLOW_CTRL_ENABLE_TX == flow_ctrl_mode)
    {
        pinmux_config(7, PINMUX_UART1_CTS_I_N_CFG);
        while(1)
        {
            log_debug("uart1 flow ctrl test tx begin\n");
        }
    }
    //接收流控
    if (FLOW_CTRL_ENABLE_RX == flow_ctrl_mode)
    {
        pinmux_config(8, PINMUX_UART1_RTS_O_N_CFG);
        log_debug("uart1 flow ctrl test rx begin\n");
    }
    //发送与接收都流控
    if (FLOW_CTRL_ENABLE_TX_RX == flow_ctrl_mode)
    {
        pinmux_config(7, PINMUX_UART1_CTS_I_N_CFG);
        pinmux_config(8, PINMUX_UART1_RTS_O_N_CFG);
        log_debug("uart1 flow ctrl test rx begin\n");
    }
}
```

在使用流控收发函数中，首先根据函数入参配置串口的数据输入和输出引脚，并打开串口，然后根据流控模式来配置发送流控、接收流控、发送和接收流控，最后根据配置的方式来打印流控方式。

```
//uart 使用流控收发
    test_uart_flow_ctrl(HS_UART1, FLOW_CTRL_ENABLE_TX);
    test_uart_flow_ctrl(HS_UART1, FLOW_CTRL_ENABLE_RX);
    test_uart_flow_ctrl(HS_UART1, FLOW_CTRL_ENABLE_TX_RX);
```

对使用流控收发函数的入参配置只能为 UART1，流控可以配置为发送使能、接收使能，发送和接收都使能。

例如选择 UART1，流控接收和发送使能；程序运行时，芯片开始发送大量数据到串口助手，将 CTS 接到 3.3V，发送数据停止，拔掉 CTS 后发送继续；串口助手发送大量数据到芯片，观察芯片是否将 RTS 的电平拉高。

2.3.6.例程 6：UART0 不同模式下的收发

以模式 1/2/3 为例，UART0 的模式 1 和模式 3 支持以下波特率 1000, 2000, 4000, 5000, 10000, 20000, 25000, 50000, 100000, 500000，当 smod=1 时最大支持波特率为 1000000。模式 2 为固定波特率，当 smod=0 时 baud_rate = Fclk/64；当 smod=1 时为 baud_rate = Fclk/32。

```
void test_uart0_all_baudrates(HS_UART_EX_Type *uart, enum uart_ex_mode_values mode)
{
    uart_index = 0;
    pinmux_config(5, PINMUX_UART0_SDA_O_CFG);
    pinmux_config(6, PINMUX_UART0_SDA_I_CFG);

    uart_ex_cfg_t uart0_cfg =
    {
        .baud_rate = uart0_baud_rates[0],
        .mode = mode,
        .tb80_val = 1,
        .rb80_val = 1,
        .smod_val = SMOD_1,
        .rx_cb = receive_handler,
        .tx_cb = NULL,
    };
    uint8_t param_num = sizeof(uart0_baud_rates) /sizeof(uart0_baud_rates[0]);

    //用于测试 model_2
    if(UART_EX_MD_2 == mode) {
        uint8_t smod_cnt = 2;
        for (uint8_t i = 0; i < smod_cnt; i++) {
            uart0_cfg.baud_rate = uart0_baud_rates[10 - i];
            uart_ex_open(uart, &uart0_cfg);
            log_debug("baudrate now : %d, smod value: SMOD_%d\r\n",uart0_baud_rates[10 - i], uart0_cfg.smod_val);
            if (0 == i) {
                log_debug("next baudrate: %d test will begin in
```

```
15s\r\n",uart0_baud_rates[10 - i - 1]);
                log_debug("please change XCOM/SSCOM baudrate to %d
\r\n\n",uart0_baud_rates[10 - i - 1]);
                uart0_cfg.smod_val = SMOD_0;
            }
            delay_s(20);
        }
        log_debug("uart model_%d test has finished \r\n",mode);
        return;
    }

    //用于测试 model_1 / model_3
    for (uint8_t i = 0; i < param_num; i++) {        //依次测试不同波特率
        uart0_cfg.baud_rate = uart0_baud_rates[i];
        uint8_t smod_cnt = 2;
        while(smod_cnt--) { //smod == 1  与 smod == 0
            //smod = 1;
            uart_ex_open(uart, &uart0_cfg);
            log_debug("baudrate now : %d, smod value:
SMOD_%d\r\n",uart0_baud_rates[i],uart0_cfg.smod_val);
            //切换 smod=0
            if(1000000 == uart0_baud_rates[i]) {  //波特率为 1000000 时，只有 smod1
                break;
            } else {
                uart0_cfg.smod_val = SMOD_0;
            }
            delay_s(10);
        }
        uart0_cfg.smod_val = SMOD_1;
        if (i + 1 < param_num ) {  //非最后一个波特率
            log_debug("\n\nnext baudrate: %d test will begin in
15s\r\n",uart0_baud_rates[i + 1]);
            log_debug("please change XCOM/SSCOM baudrate to %d
\r\n\n",uart0_baud_rates[i + 1]);
        }
        delay_s(15);
    }
    log_debug("uart model_%d test has finished \r\n",mode);
}
```

在不同波特率收发测试程序中，首先配置串口的模式，以及数据输入/输出引脚，然后利用 for 循环测试不同波特率的发送和接收能力，最后打印串口测试完成。

```
typedef uint8_t uart_ex_mode_enum;
enum uart_ex_mode_values
{
```

```
    UART_EX_MD_1 = 1,
    UART_EX_MD_2,
    UART_EX_MD_3,
    UART_EX_MD_MAX,
};

//uart0 model_1 不同波特率收发
test_uart0_all_baudrates(HS_UART0, UART_EX_MD_1);
//uart0 model_2 不同波特率收发
test_uart0_all_baudrates(HS_UART0, UART_EX_MD_2);
//uart0 model_3 不同波特率收发
test_uart0_all_baudrates(HS_UART0, UART_EX_MD_3);
```

2.3.7.例程 7：将 UART 接收的数据搬移到内存

以下示例代码可以实现在上位机串口助手上发送数据至 BL1824X 串口接收寄存器 RBR 后，BL1824X 通过 dma 将 RBR 中的数据搬运到内存中，然后再通过串口输出打印。

```
void test_dma_uart_to_mem(void)
{
    pinmux_config(5, PINMUX_UART1_SDA_O_CFG);
    pinmux_config(6, PINMUX_UART1_SDA_I_CFG);
    uart_open(HS_UART1, 115200, UART_FLOW_CTRL_DISABLED, NULL);

    // Allocate and init buffers
    uint8_t buf[3][BUF_NUM];
    for (int i = 0; i < BUF_NUM; i++) {
        buf[0][i] = 0;
        buf[1][i] = 0;
        buf[2][i] = 0;
    }

    log_debug("\r\n uart1 to mem multiple block transfer test:\n");
    log_debug("Transfer: uart1 => buf[0].\n");
    log_debug("Transfer: uart1 => buf[1].\n");
    log_debug("Transfer: uart1 => buf[2].\n");

    // Build blocks
    dma_block_t block[3];
    dma_dev_t dma_mem = {
        .id          = MEM_DMA_ID,
        .addr        = buf[0],
        .addr_ctrl   = DMA_ADDR_CTRL_INC,
        .bus_width   = DMA_SLAVE_BUSWIDTH_8BITS,
```

```
        .burst_size = DMA_BURST_LEN_1UNITS,
    };
    dma_dev_t dma_uart = {
        .id          = UART1_RX_DMA_ID,
        .addr        = (void *)&HS_UART1->RBR,
        .addr_ctrl   = DMA_ADDR_CTRL_FIX,
        .bus_width   = DMA_SLAVE_BUSWIDTH_8BITS,
        .burst_size = DMA_BURST_LEN_1UNITS,
    };
    dma_block_config_t block_config = {
        .src                 = &dma_uart,
        .dst                 = &dma_mem,
        .block_size_in_bytes = 1,
        .priority            = 0,
        .flow_controller     = DMA_FLOW_CONTROLLER_USE_NONE,
        .intr_en             = false,
    };

    //每个 block 控制一个 src-->dst 的传输线路
    //block0
    dma_build_block(&block[0], &block_config);//第一个 DMA block 用于控制传输 1 个字
节
    //block1
    dma_mem.addr = buf[1];
    block_config.block_size_in_bytes = 3;
    dma_build_block(&block[1], &block_config);//第一个 DMA block 用于控制传输 3 个字
节
    //block2
    dma_mem.addr = buf[2];
    block_config.block_size_in_bytes = 5;
    dma_build_block(&block[2], &block_config);//第一个 DMA block 用于控制传输 5 个字
节
    //将三个 DMA block 使用链表串联起来，数据来临时，三个 block 依次执行，也即三条
DMA 传输线路依次执行
    dma_append_block(block, block+1);
    dma_append_block(block, block+2);

    // Init DMA and start transfer.
    dma_init();
    /**
     * 1. DMA Software Ackknowledge must be executed before start using UART1 DMA
after manual UART1 transfers,
     * because UART1 has no DMA enable/disable control, and its DMA interface is
always enabled from the start.
```

```
     * So any transfer before connecting to DMA module will make the dma_tx/rx_req asserted and keeping, and a
     * dma_tx/rx_ack is required to deassert the corresponding req signal before using DMA.
     * 2. uart.dma_tx_req will always be asserted when the tx fifo is empty, so it can't and there's no need to be cleared.
     * 3. Do not use HS_UART->FCR.bit1 to clear the UART.dma_rx_req signal, cause it reset whole control portion of rx fifo.
     */
    HS_UART1->DMASA = 1;
    //不用 DMA 传输回调函数,或者回调函数中不要做过多打印等耗费时间的事，否则影响 DMA 传输，造成数据传输错误——yangxs
    HS_DMA_CH_Type *dma_ch = dma_start_transfer(NULL, block, NULL);
    if (dma_ch == NULL) {
        log_debug("DMA uart1-to-mem transfer fails.\r\n");
        return;
    }
    dma_wait_stop(dma_ch);
    dma_release(dma_ch);

    // Check result by manual
    //若串口调试助手中发送 112312345,共 9 个字节的数据
    print_buf(buf[0], 0, "buf[0]");      //将打印 1       ,前面设置 DMA 传输 1 个字节
    print_buf(buf[1], 0, "buf[1]");      //将打印 123     ,前面设置 DMA 传输 3 个字节
    print_buf(buf[2], 0, "buf[2]");      //将打印 12345   ,前面设置 DMA 传输 5 个字节
}
```

如果上位发送“112312345”等 9 个字符。dma 会将第 1 个字符“1”搬运到 buf[0]所指向的内存中，将接下来的 3 个字符“123”搬运到 buf[1]所指向的内存中，将接下来的 5 个字符“12345”搬运到 buf[2]所指向的内存中。

2.3.8. 例程 8：将数据从内存搬移到 UART

以下示例代码是将内存中的字符串通过 dma 搬移到 uart 的发送寄存器中，并通过串口工具发送到上位机最终显示出来。

```
void test_dma_mem_to_uart(void)
{
    pinmux_config(5, PINMUX_UART1_SDA_O_CFG);
    pinmux_config(6, PINMUX_UART1_SDA_I_CFG);
    uart_open(HS_UART1, 115200, UART_FLOW_CTRL_DISABLED, NULL);

    // Allocate and init buffers
    uint32_t buf_32bits_aligned[2][BUF_NUM/4];
    char *buf[2] = {(char *)buf_32bits_aligned[0], (char *)buf_32bits_aligned[1]};
    for (int i = 0; i < BUF_NUM; i++) {
        buf[0][i] = i;
        buf[1][i] = ~i;
    }
    sprintf(buf[0], "dma [MEM ==> UART1] ");
    sprintf(buf[1], "pass **\n");

    // Build blocks
    dma_block_t block[2];
    dma_dev_t dma_mem = {
        .id          = MEM_DMA_ID,
        .addr        = buf[0],
        .addr_ctrl   = DMA_ADDR_CTRL_INC,
        .bus_width   = DMA_SLAVE_BUSWIDTH_8BITS,
        .burst_size = DMA_BURST_LEN_1UNITS,
    };
    dma_dev_t dma_uart = {
        .id          = UART1_TX_DMA_ID,
        .addr        = (uint8_t *)&HS_UART1->THR,
        .addr_ctrl   = DMA_ADDR_CTRL_FIX,
        .bus_width   = DMA_SLAVE_BUSWIDTH_8BITS,
        .burst_size = DMA_BURST_LEN_1UNITS,
    };
    dma_block_config_t block_config = {
        .src                = &dma_mem,
        .dst                = &dma_uart,
        .block_size_in_bytes = strlen(buf[0]),
        .priority           = 0,
        .flow_controller    = DMA_FLOW_CONTROLLER_USE_NONE,
        .intr_en            = false,
```

```
    };
    dma_build_block(&block[0], &block_config);

    dma_mem.addr = buf[1];
    block_config.block_size_in_bytes = strlen(buf[1]);
    dma_build_block(&block[1], &block_config);

    dma_append_block(block, block+1);

    dma_init();
    HS_DMA_CH_Type *dma_ch = dma_start_transfer(NULL, block, NULL);
    if (NULL == dma_ch) {
        log_debug("DMA mem-to-uart1 transfer fails.\r\n");
        return;
    }
    dma_wait_stop(dma_ch);
    dma_release(dma_ch);

    // Check result by manual
}
```

函数 test_dma_mem_to_uart()的目的是将两个指针 buf[0]与 buf[1]所指向的两组字符串依次通过 dma 搬移到 uart 的 THR 中并发送出去。

数据准备。定义一个 uint32_t 类型的二维数组用于存放两个字符串，使用包含 2 个元素的 char*型指针数组分别指向这两个字符串,对数组进行数据初始化。

建立 dma 块链表。（1）首先建立两个 dma 块节点。（2）然后将这两个 dma 块节点组成链表。定义两个 dma_dev_t 类型的变量 dma_mem 与 dma_uart，分别包含 dma 传输的源信息以及目标信息。接着定义 dma_block_config_t 类型的 dma 块配置变量 block_config，使用 dma_mem 与 dma_uart 对其初始化。接着调用 dma_build_block()，用 dma 块配置变量 block_config 对 dma 块节点 block[0]进行赋值，最终将 block[0]设置为链表节点。同理设置了链表节点 block[1]。最后调用 dma_append_block()将这两个节点链接起来形成链表。

初始化 dma。调用 dma_init()初始化 dma。

dma 开始传输数据。调用 dma_start_transfer()启动 dma 传输，它会从 dma 块链表的头节点开始执行，执行完头节点的源与目标的数据传输后，就会开始执行下一个节点的源与目标的数据传输，以此类推，直至链表的所有节点的信息执行完毕为止。

最后调用 dma_wait_stop()等待 dma 传输完毕后终止 dma 传输，并调用 dma_release()释放 dma 通道资源。

2.4. UART 使用注意事项

- UART0 没有 FIFO，是单纯的寄存器读写，所以不建议用 UART0 去做大量且吞吐率要求高的应用；
- UART0 不支持 DMA；UART1 支持 DMA；
- UART0 不支持流控，UART1 支持流控；
- UART0 仅支持整数波特率，最大波特率为 1000000，最小波特率 1000，其中 mode1 和 mode3 波特率可变，mode2 为固定波特率，波特率计算公式如下：

  mode1 和 mode3：

$$baud\ rate = \frac{2^{SMOD} * Fclk}{64 * (2^{10} - s0rel)}$$

  mode2：

smod	Baud Rate
0	Fclk/64
1	Fclk/32

  注：s0rel 为 10 比特，由 RELH 的低 2 比特和 RELL 的 8 比特组成。
- UART1 默认最大波特率 921600 和最小波特率 4800；如果需要配置波特率 1200、2400，需要在 uart_open()函数下加以下代码

  配置波特率 1200：

```
if(1200 == DEBUG_UART_BAUDRATE)
{
  uint16_t baud_divisor = 8;

  cpm_set_clock(CPM_UART1_CLK,baud_divisor*DEBUG_UART_BAUDRATE*16);

  /* Baud rate setting.*/

  HS_UART1->LCR = UART_LCR_DLAB;

  HS_UART1->DLL = baud_divisor & 0xff;

  HS_UART1->DLH = (baud_divisor >> 8) & 0xff;

  /* 8 data, 1 stop, no parity */

  HS_UART1->LCR = UART_LCR_8N1;
}
```

配置波特率 2400：

```
if(2400 == DEBUG_UART_BAUDRATE)
{
    uint16_t baud_divisor = 4;

    cpm_set_clock(CPM_UART1_CLK,baud_divisor*DEBUG_UART_BAUDRATE*16);

    /* Baud rate setting.*/

    HS_UART1->LCR = UART_LCR_DLAB;

    HS_UART1->DLL = baud_divisor & 0xff;

    HS_UART1->DLH = (baud_divisor >> 8) & 0xff;

    /* 8 data, 1 stop, no parity */

    HS_UART1->LCR = UART_LCR_8N1;
}
```

3. UART0/UART2

4. DMA

4.1. 简介

DMA 传输将数据从一个地址空间复制到另外一个地址空间。当 CPU 初始化这个传输动作，传输动作本身是由 DMA 控制器来实行和完成。在实现 DMA 传输时，是由 DMA 控制器直接掌管总线，因此，存在着一个总线控制权转移问题。即 DMA 传输前，CPU 要把总线控制权交给 DMA 控制器，而在结束 DMA 传输后，DMA 控制器应立即把总线控制权再交回给 CPU。一个完整的 DMA 传输过程必须经过 DMA 请求、DMA 响应、DMA 传输、DMA 结束 4 个步骤。

BL1824X 一共有 4 个 DMA 通道，这 4 个 DMA 通道与基地址的对应关系为：

DMA 通道	基地址
channel 0	0x41100044
channel 1	0x41100044 + 1 * 0x14 = 0x41100058
channel 2	0x41100044 + 2 * 0x14 = 0x4110006C
channel 3	0x41100044 + 3 * 0x14 = 0x41100080

4.2. API 介绍

4.2.1.DMA 寄存器结构

DMA 寄存器定义的代码如下：

```
typedef union
{
    uint32_t all;
    struct {
        uint32_t Enable         : 1;
        uint32_t IntTCMask      : 1;
        uint32_t IntErrMask     : 1;
        uint32_t IntAbtMask     : 1;
        uint32_t DstReqSel      : 4;
        uint32_t SrcReqSel      : 4;
        uint32_t DstAddrCtrl    : 2;
        uint32_t SrcAddrCtrl    : 2;
        uint32_t DstMode        : 1;
        uint32_t SrcMode        : 1;
        uint32_t DstWidth       : 2;
        uint32_t SrcWidth       : 2;
        uint32_t SrcBurstSize : 3;
        uint32_t                : 4;
        uint32_t Priority       : 1;
        uint32_t                : 2;
```

```
    };
} HS_DMA_CH_CTRL_REG_Type;

typedef struct HS_DMA_CH_Type
{
    __IO HS_DMA_CH_CTRL_REG_Type Ctrl;
    __IO uint32_t SrcAddr;
    __IO uint32_t DstAddr;
    __IO uint32_t TranSize;
    __IO struct HS_DMA_CH_Type* LLPointer;
} HS_DMA_CH_Type;
```

一共定义了 5 个寄存器：通道控制寄存器 Ctrl、源地址寄存器 SrcAddr、目的地址寄存器 DstAddr、传输大小寄存器 TranSize 和链表指针寄存器 LLPointer。

0x10	DMACfg	DMAC 配置寄存器
0x20	DMACtrl	DMAC 控制寄存器
0x30	IntStatus	中断状态寄存器
0x34	ChEN	通道启用寄存器
0x40	ChAbort	通道中止寄存器
0x44 + n*0x14	ChnCtrl	通道 n 控制寄存器
0x48 + n*0x14	ChnSrcAddr	通道 n 源地址寄存器
0x4c + n*0x14	ChnDstAddr	通道 n 目的地址寄存器
0x50 + n*0x14	ChnTranSize	通道 n 传输大小寄存器
0x54 + n*0x14	ChnLLPointer	通道 n 链表指针寄存器

ID and Revision Register offset: 0x00

该寄存器保存 ID 号和修订号。两个修订字段的重置值取决于修订。

Bit	R/W	Reset	Name	Description
31:12	R	0x01021	ID	DMAC 的 ID 号
11:4	R	Revision dependent	RevMajor	主要修订号
3:0	R	Revision dependent	RevMinor	次要修订号

DMAC Configuration Register offset: 0x10

Bit	R/W	Reset	Name	Description
31	R	Configuration dependent	ChainXfr	链条传送 0x0:未配置链传输 0x1:链传输已配置
30	R	Configuration depende	ReqSync	DMA 请求同步。 DMA 请求同步应配置为在请求时避免信号完整性问题

		nt		信号不由 DMA 控制逻辑操作的系统总线时钟计时。如果未配置请求同步 请求信号直接采样而不同步。 0x0:未配置请求同步 0x1:请求同步已配置
29:15	-	-	-	Reserved
14:10	R	Configuration dependent	ReqNum	请求/确认号码
9:4	R	Configuration dependent	FIFODepth	FIFO 深度
3:0	R	Configuration dependent	ChannelNum	通道编号

DMAC Control Register offset: 0x20

Bit	R/W	Reset	Name	Description
31:1	-	-	-	Reserved
0	W	0x0	Reset	软件复位控制。 将此位设置为 1 以重置 DMA 核心并禁用所有通道

Interrupt Status Register offset: 0x30

此寄存器包含终端计数、错误和中止状态。当通道遇到终端计数器事件时，通道的终端计数状态被断言。当通道遇到错误/中止事件时，通道的错误/中止状态被断言。每个通道有一个状态位，当相应通道未配置时，状态位为零。

Bit	R/W	Reset	Name	Description
31:24	-	-	-	Reserved
23:16	R/W_1C	0x0	TC	DMA 信道的终端计数状态，每个信道一位。当信道传输完成而没有中断或错误事件时，终端计数状态被断言。 0x0:通道 N 无终端计数状态 0x1:通道 N 具有终端计数状态
15:8	R/W_1C	0x0	Abort	通道的中止状态，每个通道一位。 当通道传输被中止时，中止状态被断言。 0x0:通道 N 没有中止状态 0x1:通道 N 具有中止状态
7:0	R/W_1C	0x0	Error	错误状态，每个通道一位。当通道传输遇到以下错误事件时，将断言错误状态：

				总线错误 未对齐的地址 未对齐的传送宽度 保留的配置 0x0:通道 N 无错误状态 0x1:通道 N 具有错误状态

Channel Enable Register offset: 0x34

寄存器显示 DMA 通道启用状态。状态字段仅在配置相应通道时存在。此寄存器是所有 ChnCtrl 寄存器的启用字段的别名。

Bit	R/W	Reset	Name	Description
N:0	R	0x0	ChEN	所有 ChnCtrl 寄存器的 Enable 字段的别名

Channel Abort Register offset: 0x40

寄存器控制 DMA 信道传输的中止，每个信道一位。写入 1 以停止相应通道的当前传输。触发通道中止事件后，硬件自动清除中止位。

Bit	R/W	Reset	Name	Description
N:0	W	0x0	ChAbort	将 1 写入该字段以停止信道传输。 只有当相应的 启用通道。否则，将忽略未启用的通道的写入。

Channel n Control Register offset: 0x44 + n*0x14

Bit	R/W	Reset	Name	Description
31:30	-	-	-	Reserved
29	RW	0x0	Priority	通道优先级。 0x0：低优先级 0x1：较高优先级
28:25	-	-	-	Reserved
24:22	RW	0x0	SrcBurstSize	源脉冲串大小。此字段表示 DMA 信道重新仲裁之前的传输次数。 突发的总字节为 SrcBurstSize*SrcWidth。 0x0:1 次传输 0x1:2 次传输 0x2:4 次传输 0x3:8 次传输 0x4:16 次传输 0x5:32 次传输 0x6:64 次传输 0x7:128 次传输
21:20	RW	0x2	SrcWidth	源传输宽度 0x0：字节传输

				0x1：半字传输 0x2：字传输 0x3：保留，使用此值设置字段 触发错误异常
19:18	RW	0x2	DstWidth	目标传输宽度。 总传输字节和总突发 字节应与目标传输宽度对齐；否则将触发错误事件。例如，如果总传输字节未与字或半字对齐，则应将目标传输宽度设置为字节传输。 有关总突发字节的定义，请参见上面的 SrcBurstSize 字段，有关总传输字节的定义请参见第 3.12 节。 0x0：字节传输 0x1：半字传输 0x2：字传输 0x3：保留，将字段设置为该值触发错误异常
17	RW	0x0	SrcMode	源 DMA 握手模式 0x0：正常模式 0x1：握手模式
16	RW	0x0	DstMode	目标 DMA 握手模式 0x0：正常模式 0x1：握手模式
15:14	RW	0x0	SrcAddrCtrl	源地址控制 0x0：增量地址 0x1：递减地址 0x2：固定地址 0x3：保留，使用此值设置字段 触发错误异常
13:12	RW	0x0	DstAddrCtrl	目标地址控制 0x0：增量地址 0x1：递减地址 0x2：固定地址 0x3：保留，使用此值设置字段 触发错误异常
11:8	RW	0x0	SrcReqSel	源 DMA 请求选择。选择 请求/确认握手对 设备已连接到。
7:4	RW	0x0	DstReqSel	目标 DMA 请求选择。选择目标设备连接到的请求/确认握手对
3	RW	0x0	IntAbtMask	通道中止中断掩码。 0x0：允许触发中止中断 0x1：禁用中止中断

2	RW	0x0	IntErrMask	通道错误中断掩码。 0x0：允许触发错误中断 0x1：禁用错误中断
1	RW	0x0	IntTCMask	通道终端计数中断掩码 0x0：允许触发终端计数中断 0x1：禁用终端计数中断
0	RW	0x0	Enable	通道启用位 0x0：禁用 0x1：启用

Channel n Source Address Register offset: 0x48 + n*0x14

Bit	R/W	Reset	Name	Description
31:0	RW	0x0	SrcAddr	源起始地址。传输完成后，其值将更新为结束地址+sizeof（SrcWidth）。 此地址必须与源传输大小对齐；否则，将触发错误事件。

Channel n Destination Address Register offset: 0x4C + n*0x14

Bit	R/W	Reset	Name	Description
31:0	RW	0x0	DstAddr	目标起始地址。传输完成后，其值将更新为结束地址+sizeof（DstWidth）。 此地址必须与目标传输大小对齐；否则将触发错误事件。

Channel n Transfer Size Register offset: 0x50 + n*0x14

Bit	R/W	Reset	Name	Description
31:22	-	-	-	Reserved
21:0	RW	0x0	TranSize	来自源的总传输大小。传输的字节总数为TranSize*SrcWidth。DMA 传输完成后，该值更新为零。 如果启用了总传输大小为零的通道，则将触发错误事件并终止传输。

Channel n Linked List Pointer Register offset: 0x54 + n*0x14

Bit	R/W	Reset	Name	Description
31:2	RW	0x0	LLPointer	指向下一个块描述符的指针。指针必须与单词对齐。
1:0	-	-	-	Reserved

4.2.2.变量参数

4.2.2.1. dma_dir_t

```
typedef enum
{
    /// Async memcpy mode
    DMA_MEM_TO_MEM,
    /// Slave mode & From Memory to Device
    DMA_MEM_TO_DEV,
    /// Slave mode & From Device to Memory
    DMA_DEV_TO_MEM,
    /// Slave mode & From Device to Device, not supported currently, due to
dma_config_t now has only one slave_id.
    DMA_DEV_TO_DEV,
    /// Error indicator
    DMA_DIR_MAX,
}dma_dir_t;
```

枚举变量类型 dma_dir_t，定义了 DMA 搬移数据的方向，每个成员的具体描述如下：

成员名	描述
DMA_MEM_TO_MEM	将数据从内存搬移到内存
DMA_MEM_TO_DEV	将数据从内存搬移到外部设备
DMA_DEV_TO_MEM	将数据从外部设备搬移到内存
DMA_DEV_TO_DEV	将数据从外部设备搬移到外部设备
DMA_DIR_MAX	用于指示错误

4.2.2.2. dma_addr_ctrl_t

```
typedef enum
{
    DMA_ADDR_CTRL_INC       = 0,
    DMA_ADDR_CTRL_DEC         ,
    DMA_ADDR_CTRL_FIX         ,
    DMA_ADDR_CTRL_RESERVED    ,
} dma_addr_ctrl_t;
```

成员名	描述
DMA_ADDR_CTRL_INC	DMA 地址控制器地址递增
DMA_ADDR_CTRL_DEC	DMA 地址控制器地址递减
DMA_ADDR_CTRL_FIX	DMA 地址控制器地址固定
DMA_ADDR_CTRL_RESERVED	保留

4.2.2.3. dma_handshake_mode_t

```
typedef enum
{
    DMA_HANDSHAKE_MODE_NORMAL       = 0,
    DMA_HANDSHAKE_MODE_HANDSHAKE        ,
} dma_handshake_mode_t;
```

成员名	描述
DMA_HANDSHAKE_MODE_NORMAL	DMA 握手模式：一般模式（内存之间数据搬移）
DMA_HANDSHAKE_MODE_HANDSHAKE	DMA 握手模式：握手模式（内存和外设之间，外设与外设之间的数据搬移）

4.2.2.4. dma_slave_buswidth_t / dma_buswidth_t

```
/// defines bus with of the DMA slave device, source or target buses
typedef enum
{
    DMA_SLAVE_BUSWIDTH_8BITS      = 0,
    DMA_SLAVE_BUSWIDTH_16BITS       ,
    DMA_SLAVE_BUSWIDTH_32BITS       ,
    DMA_SLAVE_BUSWIDTH_MAXBITS      ,
} dma_slave_buswidth_t, dma_buswidth_t;
```

成员名	描述
DMA_SLAVE_BUSWIDTH_8BITS	用来设置 DMA 传输位宽为 8bit
DMA_SLAVE_BUSWIDTH_16BITS	用来设置 DMA 传输位宽为 16bit
DMA_SLAVE_BUSWIDTH_32BITS	用来设置 DMA 传输位宽为 32bit
DMA_SLAVE_BUSWIDTH_ MAXBITS	保留

4.2.2.5. dma_status_t

```
/// DMA status
typedef enum
{
    DMA_STATUS_BLOCK_OK,
    DMA_STATUS_ERROR,
    DMA_STATUS_ABORT,
    DMA_STATUS_UNKNOWN,
}dma_status_t;
```

成员名	描述
DMA_STATUS_BLOCK_OK	DMA 传输状态为传输 OK
DMA_STATUS_ERROR	DMA 传输状态为传输错误
DMA_STATUS_ABORT	DMA 传输状态为传输丢弃

DMA_STATUS_UNKNOWN	DMA 传输状态为未知

4.2.2.6. dma_burstlen_t

```
/// DMA burst length
typedef enum
{
    DMA_BURST_LEN_1UNITS       = 0,      /* a unit length equal to
SRC/DST_TR_WIDTH */
    DMA_BURST_LEN_2UNITS         ,
    DMA_BURST_LEN_4UNITS         ,
    DMA_BURST_LEN_8UNITS         ,
    DMA_BURST_LEN_16UNITS        ,
    DMA_BURST_LEN_32UNITS        ,
    DMA_BURST_LEN_64UNITS        ,
    DMA_BURST_LEN_128UNITS       ,

    DMA_BURST_LEN_RESERVE        ,
}dma_burstlen_t;
```

成员名	描述
DMA_BURST_LEN_1UNITS	DMA 突发传输单元为 1
DMA_BURST_LEN_2UNITS	DMA 突发传输单元为 2
DMA_BURST_LEN_4UNITS	DMA 突发传输单元为 4
DMA_BURST_LEN_8UNITS	DMA 突发传输单元为 8
DMA_BURST_LEN_16UNITS	DMA 突发传输单元为 16
DMA_BURST_LEN_32UNITS	DMA 突发传输单元为 32
DMA_BURST_LEN_64UNITS	DMA 突发传输单元为 64
DMA_BURST_LEN_128UNITS	DMA 突发传输单元为 128

4.2.2.7. dma_id_t

```
/// DMA ID for support peripheral
/**
 * BL1824X DMA req/ack allocation (according to LaoXing):
 *
 * F: w_i2c2_dma_rx_req
 * E: w_i2c2_dma_tx_req
 * D: w_i2s_dma_rx_req
 * C: gadc_dma_ahb_req
 * B: w_i2s_dma_tx_req
 * A: w_timer3_dma_req
 * 9: w_timer2_dma_req
 * 8: w_timer1_dma_req
 * 7: i2c3_spi1_dma_sel(HS_SYS->MON[16], default=0)? i2c3_dma_rx :
spi_mst1_dma_rx;
```

```
 * 6: i2c3_spi1_dma_sel(HS_SYS->MON[16], default=0)? i2c3_dma_tx :
spi_mst1_dma_tx;
 * 5: ****
 * 4: ****
 * 3: w_spi_mst0_dma_rx_req
 * 2: w_spi_mst0_dma_tx_req
 * 1: w_uart1_dma_rx_req
 * 0: w_uart1_dma_tx_req
 *
 */
typedef enum
{
    UART1_TX_DMA_ID              = 0,
    UART1_RX_DMA_ID              = 1,
    I2C0_TX_DMA_ID               = 2,
    I2C0_RX_DMA_ID               = 3,
    TIMER0_DMA_ID                = 4,
    TIMER1_DMA_ID                = 5,
    TIMER2_DMA_ID                = 6,
    ADC_DMA_ID                   = 7,
    I2S_TX_DMA_ID                = 8,
    I2S_RX_DMA_ID                = 9,

    MEM_DMA_ID                   = 10, // extended for identifying memory, but note that
memory have no req/ack for DMA.

    DMA_MAX_ID                       ,

}dma_id_t;
```

成员名	描述
UART1_TX_DMA_ID	uart1 tx（发送）的 DMA ID：0
UART1_RX_DMA_ID	uart1 rx（接收）的 DMA ID：1
I2C0_TX_DMA_ID	i2c tx（发送）的 DMA ID：2
I2C0_RX_DMA_ID	i2c rx（接收）的 DMA ID：3
TIMER0_DMA_ID	timer0 的 DMA ID：4
TIMER1_DMA_ID	timer1 的 DMA ID：5
TIMER2_DMA_ID	timer2 的 DMA ID：6
ADC_DMA_ID	adc 的 DMA ID：7
I2S_DMA_ID	i2s tx（发送）的 DMA ID 为 8（对于 1824X）
I2S_TX_DMA_ID	i2s tx（发送）的 DMA ID 为 9（对于 1824X 系列）
MEM_DMA_ID	10, extended for identifying memory, but note that memory have no req/ack for DMA.

注意：

1. 对于 BL1824X 芯片，DMA ID = 6 被固定为 SPI_MST1_TX_DMA_ID，DMA ID = 7 被固定为 SPI_MST1_RX_DMA_ID。

2. 对于 BL1824X 芯片，DMA ID = 6 与 DMA ID = 7 是可以复用的：

(1) HS6620_MON_AD[16] = 1 时，DMA ID = 6 被用作 I2C2_TX_DMA_ID；HS6620_MON_AD[16] = 0 时，DMA ID = 7 被用作 SPI_MST1_TX_DMA_ID。

(2) HS6620_MON_AD[16] = 1 时，DMA ID = 6 被用作 I2C2_RX_DMA_ID；HS6620_MON_AD[16] = 0 时，DMA ID = 7 被用作 SPI_MST1_RX_DMA_ID。

4.2.2.8. dma_block_t

```
typedef HS_DMA_CH_Type dma_block_t;
```

定义 DMA 通道类型变量 HS_DMA_CH_Type 别名为 dma_block_t。

4.2.2.9. dma_llip_t

```
typedef dma_block_t dma_llip_t;
```

将 3.2.2.8 中 HS_DMA_CH_Type 的别名 dma_block_t 重定义别名为 dma_llip_t，DMA 链表指针类型。

4.2.2.10. dma_lli_t / dma_lli_of_consecutive_mem_buffers_t

```
/// Linked list item config
/// Note: This lli config can only be used for DMA transfers of which the memory side is consecutive memory buffers.
typedef struct
{
    /// Enabel link list item
    bool        enable;
    /// Fifo enable (ring buffer), name misleading, should be named "circular"
    bool        use_fifo;
    /// Source address
    uint32_t    src_addr;
    /// Destination address
    uint32_t    dst_addr;
    /// Link list item block num
    uint32_t    block_num;
    /// Link list item block length, MAX 4095*src_width/8 for BL1824X / (2**22 - 1)*src_width/8 for BL1824X, in bytes.
    uint32_t    block_len;
    /// Linked List Item struct
    dma_llip_t *llip;
}dma_lli_t, dma_lli_of_consecutive_mem_buffers_t;
```

定义 dma 链表的结构体类型变量 dma_lli_t。

成员名	描述

enable	是否使能 dma 链表。 true：使能 false：不使能
use_fifo	是否使用 fifo true：使用 fifo false：不使用 fifo
src_addr	dma 源地址
dst_addr	dma 目的地址
block_num	dma 要搬运数据块的数量
block_len	每个要搬运数据块的大小
llip	指向 dma_llip_t 结构体类型的指针

4.2.2.11. dma_config_t

```
/// DMA configuration
typedef struct
{
    /// direction
    dma_dir_t       direction;
    /// source bus width
    dma_slave_buswidth_t src_addr_width; // the name "addr_width" leads to
misunderstanding.
                                         // "bus_width" is the right selection.
    /// dest bus width
    dma_slave_buswidth_t dst_addr_width;
    /// source burst length, in elements
    dma_burstlen_t src_burst;
    /// dest burst length, in elements
    dma_burstlen_t dst_burst;
    /// behave as flow controller by the device other than by DMAC.
    bool            dev_flow_ctrl;
    /// peripheral ID
    dma_id_t         slave_id;
    /// Priority
    uint32_t        priority;
    /// Linked List Item
    dma_lli_t       lli;
    /// Event callback
    dma_callback_t callback;
}dma_config_t;
```

成员名	描述
direction	dma 数据传输的方向
src_addr_width	源地址传输数据的位宽
dst_addr_width	目的地址传输数据的位宽
src_burst	源突发传输的单元数
dst_burst	目的突发传输单元数
dev_flow_ctrl	是否使用数据流控 true：使用数据流控

	false：不使用数据流控
slave_id	slave DMA ID（前面描述过的 dma_id_t 类型）
priority	设置优先级（位域为 1，意味着优先级只能 设置为 0 或者 1）
lli	dma_lli_t 类型变量 lli.enable = false 不使用链表方式，lli 其他成员标量无需配置（即使配置不生效）。 lli.enable = true 使用链表方式，需要对 lli 的其他成员变量做相应配置
callback	dma_callback_t 类型的回调函数。dma 传输数据完毕会进入此回调函数。

4.2.2.12. dma_dev_t

```
/// DMA device config.
typedef struct
{
    /// device ID
    dma_id_t          id;
    /// address
    void              *addr;
    /// address control
    dma_addr_ctrl_t addr_ctrl;
    /// bus width
    dma_buswidth_t   bus_width;
    /// burst length, in bus events
    dma_burstlen_t   burst_size;
} dma_dev_t;
```

成员名	描述
id	dma_id_t 类型成员变量，用于配置 dma ID
addr	作为 dma 源或者目的的地址
addr_ctrl	用于配地址控制模式：递增、递减或固定
bus_width	用于配置传输位宽
burst_size	用于配置突发传输的数据大小

4.2.2.13. dma_block_config_t

```
typedef struct
{
    /// dma source device
    dma_dev_t         *src;
    /// dma dest device
    dma_dev_t         *dst;
    /// Link list item block length, in bytes.
    /// BL1824X:   max src_width_in_bytes * 4095;
    /// BL1824X: max src_width_in_bytes * (2**22 - 1).
    uint32_t          block_size_in_bytes;
    /// Priority
    uint32_t          priority;
```

```
    /// specify DMAC, src, dst or none as the flow controller.
    dma_flow_controller_t flow_controller;

    /// interrupt enabling
    bool                  intr_en;
} dma_block_config_t;
```

成员名	描述
src	dma_dev_t 类型的结构体指针，用于指向源的配置参数
dst	dma_dev_t 类型的结构体指针，用于指向目标的配置参数
block_size_in_bytes	dma 传输一个数据块的字节大小
priority	用于配置优先级
flow_controller	用于配置流控
intr_en	是否使能中断 true：使能中断，dma 传输完毕后会进入中断 false：不使能中断

4.2.2.14. dma_callback_t

```
typedef void (*dma_callback_t)(dma_status_t status, uint32_t cur_src_addr, uint32_t
cur_dst_addr, uint32_t xfer_size);
```

定义函数指针 dma_callback_t。

4.2.3.库函数

4.2.3.1. dma_init

函数名	dma_init
函数原型	void dma_init(void)
功能描述	用于初始化 dma，在配置 dma 参数（dma_config()函数）之前使用
输入参数	无
输出参数	无
返回值	无

4.2.3.2. dma_allocate

函数名	dma_allocate
函数原型	HS_DMA_CH_Type *dma_allocate(void)
功能描述	用于分配空闲的 dma 通道资源
输入参数	无
输出参数	无
返回值	DMA 通道指针，指向新分配的 DMA 通道资源

4.2.3.3. dma_release

函数名	dma_release
函数原型	void dma_release(HS_DMA_CH_Type *ch)
功能描述	释放 dma 通道资源
输入参数	ch：指向 dma 通道的指针
输出参数	无
返回值	无

4.2.3.4. dma_config

函数名	dma_config
函数原型	bool dma_config(HS_DMA_CH_Type *ch, dma_config_t *config)
功能描述	用于对 dma 的通道进行参数配置
输入参数 1	ch：指向 dma 通道资源的指针
输入参数 2	config：指向待配置的 dma 结构体参数
输出参数	无
返回值	返回 bool 型数值。dma 参数配置成功返回 true，配置失败返回 false。

4.2.3.5. dma_start

函数名	dma_start
函数原型	bool dma_start(HS_DMA_CH_Type *ch, uint32_t src_addr, uint32_t dst_addr, uint32_t trans_count)
功能描述	初始化并配置完 dma 后，使用 dma_start()启动 dma，开始用非链表的方式传输数据
输入参数 1	ch：指向待使用的 dma 通道的指针
输入参数 2	src_addr：源地址
输入参数 3	dst_addr：目标地址
输入参数 4	trans_count：代码传输的数据个数
输出参数	无
返回值	返回 bool 类型数值 true：dma 传输成功 false：dma 传输失败

4.2.3.6. dma_start_with_lli

函数名	dma_start_with_lli
函数原型	void dma_start_with_lli(HS_DMA_CH_Type *ch)
功能描述	初始化并配置完 dma 后，使用 dma_start_with_lli ()启动 dma，开始用链表的方式传输数据
输入参数	ch：指向待使用的 dma 通道的指针

输出参数	无
返回值	无

4.2.3.7. dma_stop

函数名	dma_stop
函数原型	void dma_stop(HS_DMA_CH_Type *ch)
功能描述	立即强制停止 dma 传输，清空此通道参数配置的环境变量，关闭此通道的 dma 电源。可在 dma_stop()之后使用 dma_release()释放此通道资源
输入参数	ch：指向 dma 通道的指针
输出参数	无
返回值	无

4.2.3.8. dma_wait_stop

函数名	dma_wait_stop
函数原型	void dma_wait_stop(HS_DMA_CH_Type *ch)
功能描述	等待 dma 将当前数据传输完毕之后，清空此通道参数配置的环境变量，关闭此通道的 dma 电源。可在 dma_wait_stop()之后使用 dma_release()释放此通道资源
输入参数	ch：指向 dma 通道的指针
输出参数	无
返回值	无

4.2.3.9. dma_build_block

函数名	dma_build_block
函数原型	dma_block_t * dma_build_block(dma_block_t *block, dma_block_config_t *config);
功能描述	参数 config 包含了 dma 传输的源以及目标等信息，然后使用 config 初始化 block，block 可以作为一个链表的节点块，并且其成员变量包含了一个指向下一个节点的指针。
输入参数 1	block：待建立的 dma_block_t *类型的块指针
输入参数 2	config：包含了源和目标等信息的 dma_block_config_t *类型指针
输出参数	block：建立过的 dma_block_t *类型的块指针
返回值	返回建立好的 dma_block_t *类型的块指针

4.2.3.10. dma_append_block

函数名	dma_append_block
函数原型	dma_block_t * dma_append_block(dma_block_t *list, dma_block_t *block);

功能描述	使用 dma_build_block()建立各个链表的节点块之后，可以使用 dma_append_block()将这些节点块依次连接到一个链表上。
输入参数 1	list：链表指针
输入参数 2	block：包含 dma 传输源和目标等信息
输出参数	list：输出添加新节点的链表头指针
返回值	dma_block_t*类型数据：返回添加了新节点的链表头指针

4.2.3.11. dma_start_transfer

函数名	dma_start_transfer
函数原型	HS_DMA_CH_Type * dma_start_transfer(HS_DMA_CH_Type *ch, dma_block_t *block, dma_callback_t cb)
功能描述	启动 dma 块传输
输入参数 1	ch：指向 dma 通道的指针，如果未提前分配通道资源，即 ch==NULL，则在函数内部会首先为其分配通道资源；如果已经提前分配了通道资源，即 ch 不等于 NULL，则在函数内部不再为其分配 dma 通道。
输入参数 2	block：包含 dma 传输源和目标等信息的块
输入参数 3	cb：dma_callback_t 类型的回调函数。dma 传输完毕后会进入此回调函数
输出参数	无
返回值	返回传输数据的 dma 通道指针。

4.2.3.12. dma_get_direction

函数名	dma_get_direction
函数原型	dma_dir_t dma_get_direction(dma_id_t src_id, dma_id_t dst_id)
功能描述	获取 dma 数据传输的方向
输入参数	src_id：dma 传输的源 ID
输出参数	dst_id：dma 传输的目标 ID
返回值	返回 dma_dir_t 类型的数值，dma_dir_t 是数据传输方向的枚举变量类型

4.2.3.13. DMA_SETUP_CONFIG

DMA_SETUP_CONFIG()是宏定义函数。

函数名	DMA_SETUP_CONFIG
函数原型	DMA_SETUP_CONFIG(config, \ dir, \ dev_id, \ src_bus_width, \ dst_bus_width, \ src_burst_len, \ dst_burst_len, \

	flow_ctrl, \ prio, \ cb)
功能描述	使用其他的入参对 dma_config_t 类型的参数 config 进行赋值
输入参数 1	config：dma_config_t 类型的结构体
输入参数 2	dir：dma_dir_t 类型的枚举变量
输入参数 3	dev_id：dma_id_t 类型的枚举变量
输入参数 4	src_bus_width：dma_buswidth_t 类型的枚举变量
输入参数 5	dst_bus_width：dma_buswidth_t 类型的枚举变量
输入参数 6	src_burst_len：dma_burstlen_t 类型的枚举变量
输入参数 7	dst_burst_len：dma_burstlen_t 类型的枚举变量
输入参数 8	flow_ctrl：bool 型，是否使用流控。true：使用流控；false：不使用流控
输入参数 9	prio：设置优先级别
输入参数 10	cb：回调函数。如果为 NULL，则不使用回调函数；非 NULL，则 DMA 传输完毕后会进入次回调函数
输出参数	输出赋值后的 config 结构体
返回值	无

4.2.3.14. DMA_SETUP_LLI

DMA_SETUP_LLI ()是宏定义函数。

函数名	DMA_SETUP_LLI
函数原型	DMA_SETUP_LLI(lli, \ lli_enable, \ lli_use_fifo, \ lli_src_addr, \ lli_dst_addr, \ lli_block_num, \ lli_block_len, \ lli_llip)
功能描述	对 dma_lli_t 类型的参数 lli 进行赋值
输入参数 1	lli：dma_lli_t 类型的参数
输入参数 2	lli_enable：bool 类型，是否使用链表方式。true：使用链表；false：不使用链表
输入参数 3	lli_use_fifo：bool 类型，是否使用 fifo。true：使用 fifo；false：不使用 fifo
输入参数 4	lli_src_addr：uint32_t 类型，用于对结构体 lli 的成员 src_addr 进行赋值
输入参数 5	lli_dst_addr：uint32_t 类型，用于对结构体 lli 的成员 dst_addr 进行赋值
输入参数 6	lli_block_num：uint32_t 类型，用于对结构体 lli 的成员 block_num 进行赋值
输入参数 7	lli_block_len：uint32_t 类型，用于对结构体 lli 的成员 block_len 进行赋值
输入参数 8	lli_llip：dma_llip_t *类型指针。用于对结构体 lli 的成员 llip 进行赋值
输出参数	lli：dma_lli_t 类型的参数
返回值	无

4.3. DMA 应用例程

4.3.1.内存之间的单块搬移

以下示例代码为使用 DMA 将一块 memory 中的数据搬移到另一块 memory 中。

```
void test_dma_mem_to_mem_single_block(void)
{
    log_debug("\r\n mem to mem single block transfer test:\r\n");
    log_debug("Transfer: buf[0] => buf[1].\r\n");

    // Init buffers
    uint32_t buf_32bits_aligned[BUF_SIZE/4];
    uint32_t buf_8bits_aligned[BUF_SIZE];
    uint8_t *buf[2] = {(uint8_t *)buf_32bits_aligned, (uint8_t *)buf_8bits_aligned};
    for (int i = 0; i < BUF_SIZE; i++)
        buf[0][i] = i;
    memset(buf[1], 0, BUF_SIZE);

    // Setup DMA and start transfer.
    dma_init();
    HS_DMA_CH_Type *dma_ch = dma_allocate();
    dma_config_t config;
    DMA_SETUP_CONFIG(config,
                     DMA_MEM_TO_MEM,
                     MEM_DMA_ID,
                     DMA_SLAVE_BUSWIDTH_32BITS,
                     DMA_SLAVE_BUSWIDTH_8BITS,
                     DMA_BURST_LEN_4UNITS,
                     DMA_BURST_LEN_32UNITS,
                     false,
                     0,
                     NULL);

    if (!dma_config(dma_ch, &config)) {
        log_debug("DMA config fails.\r\n");
        return;
    }

    /* memcpy(buf[1], buf[0], BUF_SIZE); */
    dma_start(dma_ch, (uint32_t)buf[0], (uint32_t)buf[1], BUF_SIZE/4);
```

```
    dma_wait_stop(dma_ch);
    dma_release(dma_ch);

    // Check result
    if (memcmp(buf[0], buf[1], BUF_NUM) == 0)
    {
        log_debug("DMA transfer Mem-to-Mem succeeds **\n");
    }
    else
    {
        log_debug("DMA transfer Mem-to-Mem fails **\n");
        print_buf(buf[0], BUF_NUM, "buf[0]:");
        print_buf(buf[1], BUF_NUM, "buf[1]:");
    }
    log_debug("\n\n\n");
}
```

函数 test_dma_mem_to_mem_single_block()可实现将一块 memory 中的数据通过 DMA 搬运到另外一块 memory 中。

在示例函数中，首先定义两个 uint8_t(unsigned char)类型数组 buf0 和 buf1，并对 buf0 赋值。

接下来使用 dma_init()接口函数初始化 dma。

然后定义 HS_DMA_CH_Type *类型的指针变量 dma_ch，并调用 dma_allocate()为其分配当前可用的 dma 通道资源。

宏定义函数 DMA_SETUP_CONFIG()对 dma 参数变量 config 进行赋值。

调用 dma_config()来配置 dma 通道，使用 dma_start()启动 dma 进行非链表方式的数据传输（非链表方式即数据传输一次完毕后，不再传输）；dma_wait_stop()等待 dma 通道数据传输完毕后停止 dma，清空此通道参数配置的环境变量，关闭此通道的 dma 电源；最后调用 dma_release()释放此通道的 dma 资源。

4.3.2.内存之间的多块搬移

以下示例代码是让两块内存的数据通过 dma 搬移到另外两块内存中。

```
 void test_dma_mem_to_mem_multi_blocks( void )
{
    // Allocate and init buffers:
    // Buffer must be aligned to source/destination's bus width.
    // block transfer size must be aligned to source/destination's (bus_width * burst_size)
    // Or, in other words, src/dst burst size must be selected according to src/dst's bus
width and block transfer size.
    uint32_t buf_32bits_aligned[2][BUF_SIZE/4];
    uint8_t  buf_8bits_aligned[2][BUF_SIZE];
```

```
uint8_t *buf[4] = {(uint8_t *)buf_32bits_aligned[0], (uint8_t *)buf_32bits_aligned[1],
                   (uint8_t *)buf_8bits_aligned[0],  (uint8_t *)buf_8bits_aligned[1]};
for (int i = 0; i < BUF_SIZE; i++) {
    buf[0][i] = rand() % 256;
    buf[1][i] = rand() % 256;
     buf[2][i] = 0;
     buf[3][i] = 0;
}

log_debug("\r\n mem to mem multiple block transfer test:\r\n");
log_debug("Transfer: buf[0] => buf[2].\r\n");
log_debug("Transfer: buf[1] => buf[3].\r\n");

// Build blocks
dma_block_t block[2];
dma_dev_t src = {
    .id          = MEM_DMA_ID,
    .addr        = buf[0],
    .addr_ctrl   = DMA_ADDR_CTRL_INC,
    .bus_width   = DMA_SLAVE_BUSWIDTH_32BITS,
    .burst_size = DMA_BURST_LEN_4UNITS,
};
dma_dev_t dst = {
    .id          = MEM_DMA_ID,
    .addr        = buf[2],
    .addr_ctrl   = DMA_ADDR_CTRL_INC,
    .bus_width   = DMA_SLAVE_BUSWIDTH_8BITS,
    .burst_size = DMA_BURST_LEN_1UNITS,
};
dma_block_config_t block_config = {
    .src                = &src,
    .dst                = &dst,
    .block_size_in_bytes = BUF_SIZE,
    .priority           = 0,
    .flow_controller    = DMA_FLOW_CONTROLLER_USE_NONE,
    .intr_en            = false,
};
dma_build_block(&block[0], &block_config);

src.addr = buf[1];
dst.addr = buf[3];
dst.bus_width = DMA_SLAVE_BUSWIDTH_32BITS;
dma_build_block(&block[1], &block_config);
```

```
    dma_append_block(block, block+1);
    dma_print_block(block);

    // Init DMA and start transfer.
    dma_init();
    HS_DMA_CH_Type *dma_ch = dma_start_transfer(NULL, block, dma_cb);
    if (dma_ch == NULL) {
        log_debug("DMA transfer fails.\r\n");
        return;
    }
    dma_wait_stop(dma_ch);
    dma_release(dma_ch);

    // Check result
    if (memcmp(buf[0], buf[2], BUF_NUM) == 0 && memcmp(buf[1], buf[3], BUF_NUM /
2) == 0)
    {
        log_debug("DMA transfer Mem-to-Mem succeeds **\n");
    }
    else
    {
        log_debug("DMA transfer Mem-to-Mem fails !!\n");
        print_buf(buf[0], BUF_NUM, "buf[0]");
        print_buf(buf[2], BUF_NUM, "buf[2]");
        print_buf(buf[1], BUF_NUM / 2, "buf[1]");
        print_buf(buf[3], BUF_NUM / 2, "buf[3]");
    }
 }
```

4.4. DMA 使用注意事项

```
#define HS_DMA_CH0              ((HS_DMA_CH_Type *) HS_DMA_CH0_BASE)
#define HS_DMA_CH1              ((HS_DMA_CH_Type *) HS_DMA_CH1_BASE)
#define HS_DMA_CH2              ((HS_DMA_CH_Type *) HS_DMA_CH2_BASE)
#define HS_DMA_CH3              ((HS_DMA_CH_Type *) HS_DMA_CH3_BASE)

#define HS_DMA_CH0_BASE             (HS_DMAC_BASE + 0x44)
#define HS_DMA_CH1_BASE             (HS_DMA_CH0_BASE + 0x14)
```

程序中只定义了 channel0 和 channel1 的基地址。若要使用 channel2 和 channel3 的基地址和序号，则需要使用 channel0 和 channel1 的基地址之差来进行求解。

```
HS_DMA_CH_Type *ch = DMA_INDEX2CH(i);
dma_status_t status = DMA_STATUS_UNKNOWN
```

```
#define DMA_CH2INDEX(ch) ((((uint32_t)ch) - HS_DMA_CH0_BASE) /
(HS_DMA_CH1_BASE - HS_DMA_CH0_BASE))
#define DMA_INDEX2CH(i) ((HS_DMA_CH_Type *)(HS_DMA_CH0_BASE + i *
(HS_DMA_CH1_BASE - HS_DMA_CH0_BASE)))
```

5. SFLASH Controller

5.1. 简介

SFLASH 控制器用于支持 MCU 访问外部存储器 SFLASH，支持 1/2/4 线模式，支持 DMA 读写，支持透明读功能。BL1824X 内部一共有 2 个 SFLASH Controller，分别为 HS_SF，HS_SF1。其中 HS_SF 固定连接内部 flash，HS_SF1 可以根据需要连接外部 SFLASH，PSRAM，或者 LCD 等不同的外设。

SFLASH Controller 对应的基地址如下：

SFLASH Controller 编号	基地址
SFLASH Controller0	0x51000000
SFLASH Controller1	0x53000000

SFLASH Controller0 在代码中被宏定义为 HS_SF，SFLASH Controller1 为 HS_SF1。

5.2. API 介绍

5.2.1.SFlash 控制寄存器结构

```
typedef struct
{
    __I  uint32_t INTR_STATUS;
    __IO uint32_t RAW_INTR_STATUS;
    __IO uint32_t INTR_MASK;
    __IO uint32_t COMMAND;
    __IO uint32_t COMMAND_DATA0_REG;
    __IO uint32_t COMMAND_DATA1_REG;
    __IO uint32_t READ0_REG;
    __IO uint32_t READ1_REG;
    __IO uint32_t ADDRESS_REG;
    __IO uint32_t READ_OPCODE_REG;
    struct {
        __IO uint32_t CTRL;
        __IO uint32_t CS;
    }CONFIGURATION[2];

    __IO uint32_t TRANS_REMAP_REG;
    __IO uint32_t WP_HOLD_REG;
```

```
    __IO uint32_t SW_SPI_CFG0_REG;
    __IO uint32_t SW_SPI_CFG1_REG;
} HS_SF_Type;
```

Offset	寄存器	描述
0x00000	SPI_INTR_STATUS	SPI 中断状态寄存器
0x00004	SPI_RAW_INTR_STATUS	SPI 原始中断状态寄存器
0x00008	SPI_INTR_MASK	SPI 中断屏蔽寄存器
0x0000c	SPI_COMMAND	SPI 命令寄存器
0x00010	SPI_COMMAND_DATA0_REG	SPI 命令数据寄存器 0
0x00014	SPI_COMMAND_DATA1_REG	SPI 命令数据寄存器 1
0x00018	SPI_READ0_REG	SPI 读寄存器 0
0x0001c	SPI_READ1_REG	SPI 读寄存器 1
0x00020	SPI_ADDRESS_REG	SPI 地址寄存器
0x00024	SPI_READ_OPCODE_REG	SPI 读取操作码寄存器
0x00028	SPI_CONFIGURATION_0	SPI 配置寄存器 0
0x0002c	SPI_CS_CONFIGURATION_0	SPI CS 信号配置寄存器 0
0x00030	SPI_CONFIGURATION_1	SPI 配置寄存器 1
0x00034	SPI_CS_CONFIGURATION_1	SPI CS 信号配置寄存器 1

SPI_INTR_STATUS address offset: 0x000

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	RESERVED	reserved
0	R	0x0	SPI_CMD_DONE	SPI 命令完成

SPI_RAW_INTR_STATUS address offset: 0x004

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	RESERVED	reserved
0	RW	0x0	SPI_RAW_INTR_STATUS	SPI 命令完成中断（屏蔽前的内部中断值）。SPI 命令完成时设置。 写入 1 以清除中断状态.

SPI_INTR_MASK address offset: 0x008

Bit	R/W	Reset	Name	Description
31:1	N/A	0x0	RESERVED	reserved
0	RW	0x0	SPI_CMD_DONE_MASK	SPI 命令完成中断掩码。当为 1 时，允许中断断言中断。

SPI_COMMAND address offset: 0x00c

Bit	R/W	Reset	Name	Description
31:12	RW	0x0	DATA_BYTES	该字段指定命令数据位发送后要传输的数据字节数。 有效值为 0-65535。

				对于读命令，如果该字段不是 4 的倍数，在数据被写入系统内存之前，将被填充零。 这个值在 SPI 交换过程中递减，一直到 0。
11:5	RW	0x0	CMD_BITS	该字段指定要发送的命令数据的位数。有效值为 0-64。 该值在 SPI 交换过程中递减，直到它达到 0。 请注意： 64 个命令数据位被定义在 2 个 32 位的寄存器中。 前 32 个命令数据位从 spi_command_data0_reg 寄存器发送，后 32 个命令数据位从 spi_command_data1_reg 寄存器中发送。
4	RW	0x0	KEEP_CS	CS 保持使能 0：不使能 1：使能
3	RW	0x0	DATA_2_LANE_EN	仅当 spi_if_mode 为 3 线模式 1 时，二线数据通道模式使能。 0：不使能 1：使能
2	RW	0x0	CHIP_SELECT	CS 选择 0：选择 CS0 1：选择 CS1 注：内部 flash 时选择 CS0；外部 flash 时选择 CS1
1:0	RW	0x0	COMMAND	读/写命令指示 该字段在 SPI 转换完成后自动清零 0x0：NOP 0x1：读命令。在命令数据位被发送后，数据被传输到存储器。 0x2：写命令。在命令数据位被发送后，数据将传输到 SPI 设备。

SPI_COMMAND_DATA0_REG address offset: 0x010

Bit	R/W	Reset	Name	Description
31:0	RW	0x0	COMMAND_DATA	这是前 32 个 SPI 时钟周期发送的命令数据，具体取决于 CMD_BITS 字段。它首先被发送到 MSB（数据从位 31 左移）。该值在 SPI 事务期间保持。

SPI_COMMAND_DATA1_REG address offset: 0x014

Bit	R/W	Reset	Name	Description
31:0	RW	0x0	COMMAND_DATA	这是前 32 个 SPI 时钟周期发送的命令数据，具体取决于 CMD_BITS 字段。它首先被发送到 MSB（数据从位 31 左移）。该值在 SPI 事务期间保持。

SPI_READ_DATA0_REG address offset: 0x018

Bit	R/W	Reset	Name	Description
31:0	RW	0x0	READ_DATA0	该寄存器保存在前 32 个 SPI 时钟周期期间捕获的数据。首先捕获 MSB（数据从位 0 左移）。未使用的前导位将为 0。

SPI_READ_DATA1_REG address offset: 0x01c

Bit	R/W	Reset	Name	Description
31:0	RW	0x0	READ_DATA1	该寄存器保存在前 32 个 SPI 时钟周期期间捕获的数据。首先捕获 MSB（数据从位 0 左移）。未使用的前导位将为 0.

SPI_ADDRESS_REG address offset: 0x020

Bit	R/W	Reset	Name	Description
31:0	RW	0x0	ADDRESS	该寄存器保存用于数据传输的系统内存地址。当数据从系统内存中读取时（在写入命令的情况下），或当数据写入系统内存时（在读取命令的情况中），它将递增 4。该寄存器的低两位始终为 0，以强制字对齐。

SPI_READ_OPCODE_REG address offset: 0x024

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	RESERVED	reserved
15:8	RW	0x3b	CS1_OPCODE	该寄存器保存 OPCODE，当 CS1 地址空间中发生透明读取时，该 OPCODE 用于从串行闪存设备读取
7:0	RW	0x3b	CS0_OPCODE	该寄存器保存 OPCODE，当 CS0 地址空间中发生透明读取时，该 OPCODE 用于从串行闪存设备读取

SPI_CONFIGURATION_0 address offset: 0x028

Bit	R/W	Reset	Name	Description
31:24	N/A	0x0	RESEVED	reserved
23	RW	0x0	LCD_RD_EN	0x0：闪存读写，液晶写 0x1:lcd 读取
22:21	RW	0x0	RGB_MODE	0x0：闪存，RGB565 1 线/2 线数据通道 0x1:RGB666 1 线/2 线数据通道

				0x2:RGB888 1 线/2 线数据通道
20:18	RW	0x0	LCD_SPI_CTRL	0x0：闪烁模式； 0x1:RGB565/RGB888 3 线、1 线数据通道 0x2:RGB565/RGB888 4 线、1 线数据通道 0x3:RGB666 3 线、1 线数据通道/RGB565 3 线、2 线数据通道 0x4:RGB666 4 线、1 线数据通道/RGB666 3 线、2 线数据通道 0x5:RGB888 3 线、2 线数据通道 （仅适用于 SFLASH2）
17:16	RW	0x0	WIDTH	数据读取/写入的宽度。 该字段使数据正确写入小端系统中小于 32 位宽的设备。大端系统应使用 32 位数据设置。 00:8 位数据 01:16 位数据 1X：32 位数据 注：RGB565 为 0x1； RGB666 和 RGB888 的 0x2
15	N/A	0x0	RESERVED	reserved
14	RW	0x0	FE_DLY_SAMPLE	下降沿延迟采样。 1： 在延迟采样时钟（SPI_CLK_DLY）的下降沿采样 SPI_DI。 0：在内部采样时钟（SPI_CLK）的下降沿采样 SPI_DI。启用此位将允许更高频率的操作。 如果设置，dly_sample[13:12]必须设置为 1 或更大。这仅对 SPI 模式 0 和 3 有效。对于模式 1 和 2，这些位必须设置为 0。
13:12	RW	0x0	DLY_SAMPLE	延迟采样，SPI_CLK 下降沿后采样 SPI_DI 的 REF_CLK 数。 设置这些位将允许更高频率的操作。如果设置了 fe_dly_sample[14]，则该字段必须设置为 1 或更大。这仅对 SPI 模式 0 和 3 有效。对于模式 1 和 2，这些位必须设置为 0。
11	N/A	0x0	RESERVED	reserved
10	RW	0x0	BP_CLOCK_DIV	旁路时钟分频器 （仅适用于 SFLASH2）
9	RW	0x0	CPOL	时钟极性。 0：时钟在空闲时间为低，每个时钟脉冲由上升沿和下降沿组成。 1： 时钟在空闲时间为高电平，每个时钟脉冲由下降沿和上升沿组成。
8	RW	0x0	CPHA	时钟相位。

				0：输入数据在每个时钟脉冲的第一边缘上计时。 1：输入数据在每个时钟脉冲的第二边缘上计时。
7:0	RW	0x2	CLOCK_DIV	该寄存器是系统时钟的分频器，用于生成 SPI 时钟。只有偶数值才能编程，以保持 50%的占空比时钟。此寄存器的最小值为 2。

SPI_CS_CONFIGURATION_0 address offset: 0x02c

Bit	R/W	Reset	Name	Description
31:24	RW	0x0a	CS_RECOVER	芯片选择恢复时间。这是在芯片选择被取消断言之后必须经过的系统周期数，然后才能断言相同或任何其他芯片选择。
23:16	RW	0x0a	CS_HOLD	芯片选择保持时间。这是最后一个时钟和芯片选择的取消断言之间的系统时钟周期数。
15:8	RW	0x0a	CS_SETUP	芯片选择设置时间。这是芯片选择的断言和第一时钟脉冲之间的系统时钟周期数。
7:1	N/A	0x0	RESERVED	reserved
0	RW	0x0	CS_POL	芯片选择极性。 0：芯片选择处于低电平激活状态。 1： 芯片选择处于高电平

SPI_CONFIGURATION_1 address offset: 0x030

Bit	R/W	Reset	Name	Description
31:24	N/A	0x0	RESERVED	reserved
23	RW	0x0	LCD_RD_EN	0x0：闪存读写，液晶写 0x1:lcd 读取
22:21	RW	0x0	RGB_MODE	0x0：闪存，RGB565 1 线/2 线数据通道 0x1:RGB666 1 线/2 线数据通道 0x2:RGB888 1 线/2 线数据通道
20:18	RW	0x0	LCD_SPI_CTRL	0x0：闪烁模式； 0x1:RGB565/RGB888 3 线、1 线数据通道 0x2:RGB565/RGB888 4 线、1 线数据通道 0x3:RGB666 3 线、1 线数据通道/RGB565 3 线、2 线数据通道 0x4:RGB666 4 线、1 线数据通道/RGB666 3 线、2 线数据通道 0x5:RGB888 3 线、2 线数据通道 （仅适用于 SFLASH2）
17:16	RW	0x0	WIDTH	数据读取/写入的宽度。 该字段使数据正确写入小端系统中小于 32 位宽的设备。大端系统应使用 32 位数据设置。

				00:8 位数据 01:16 位数据 1X：32 位数据 注：RGB565 为 0x1； RGB666 和 RGB888 的 0x2
15	N/A	0x0	RESERVED	reserved
14	RW	0x0	FE_DLY_SAMPLE	下降沿延迟采样。 1： 在延迟采样时钟（SPI_CLK_DLY）的下降沿采样 SPI_DI。 0：在内部采样时钟（SPI_CLK）的下降沿采样 SPI_DI。启用此位将允许更高频率的操作。 如果设置，dly_sample[13:12]必须设置为 1 或更大。这仅对 SPI 模式 0 和 3 有效。对于模式 1 和 2，这些位必须设置为 0。
13:12	RW	0x0	DLY_SAMPLE	延迟采样，SPI_CLK 下降沿后采样 SPI_DI 的 REF_CLK 数。设置这些位将允许更高频率的操作。如果设置了 fe_dly_sample[14]，则该字段必须设置为 1 或更大。这仅对 SPI 模式 0 和 3 有效。对于模式 1 和 2，这些位必须设置为 0。
11	N/A	0x0	RESEVED	reserved
10	RW	0x0	BP_CLOCK_DIV	旁路时钟分频器 （仅适用于 SFLASH2）
9	RW	0x0	CPOL	时钟极性。 0：时钟在空闲时间为低，每个时钟脉冲由上升沿和下降沿组成。 1： 时钟在空闲时间为高电平，每个时钟脉冲由下降沿和上升沿组成。
8	RW	0x0	CPHA	时钟相位。 0：输入数据在每个时钟脉冲的第一边缘上计时。 1：输入数据在每个时钟脉冲的第二边缘上计时。
7:0	RW	0x2	CLOCK_DIV	该寄存器是系统时钟的分频器，用于生成 SPI 时钟。只有偶数值才能编程，以保持 50%的占空比时钟。此寄存器的最小值为 2。

SPI_CS_CONFIGURATION_1 address offset: 0x034

Bit	R/W	Reset	Name	Description
31:24	RW	0x0a	CS_RECOVER	芯片选择恢复时间。这是在芯片选择被取消断言之后必须经过的系统周期数，然后才能断言相同或任何其他芯片选择。

23:16	RW	0x0a	CS_HOLD	芯片选择保持时间。这是最后一个时钟和芯片选择的取消断言之间的系统时钟周期数。
15:8	RW	0x0a	CS_SETUP	芯片选择设置时间。这是芯片选择的断言和第一时钟脉冲之间的系统时钟周期数。
7:1	N/A	0x0	RESERVED	reserved
0	RW	0x0	CS_POL	芯片选择极性。 0：芯片选择处于低电平激活状态。 1： 芯片选择处于高电平

5.2.2.变量参数

5.2.2.1. sf_width_t

```
typedef enum
{
    SF_WIDTH_1LINE = 1,
    SF_WIDTH_2LINE = 2,
    SF_WIDTH_4LINE = 4,
}sf_width_t;
```

FLASH 线宽。

成员名称	描述
SF_WIDTH_1LINE	1 线
SF_WIDTH_2LINE	2 线
SF_WIDTH_4LINE	4 线

5.2.2.2. sf_status_t

```
typedef enum
{
    SF_STATUS_NONE,
    SF_STATUS_ABSENT,
    SF_STATUS_PRESENT,
}sf_status_t;
```

FLASH 状态。

成员名称	描述
SF_STATUS_NONE	未检测到 FLASH
SF_STATUS_ABSENT	检测到 FLASH 但返回数据（ID）不正确
SF_STATUS_PRESENT	检测到 FLASH

5.2.2.3. sf_config_t

```
typedef struct
{
    uint32_t freq_hz;
    sf_width_t width;
    uint8_t delay;
}sf_config_t;
```

FLASH 配置参数。

成员名称	描述
freq_hz	频率
width	线宽
delay	延迟

5.2.3.库函数

5.2.3.1. sf_read_sr

函数名	sf_read_sr
函数原型	uint8_t sf_read_sr(HS_SF_Type *sf, uint32_t cs);
功能描述	读取 FLASH 的状态寄存器低八位的值
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	FLASH 的状态寄存器低八位的值。

5.2.3.2. sf_read_sr2

函数名	sf_read_sr2
函数原型	uint8_t sf_read_sr2(HS_SF_Type *sf, uint32_t cs);
功能描述	读取 FLASH 的状态寄存器高八位的值
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	FLASH 的状态寄存器高八位的值。

5.2.3.3. sf_read_sr_16bits

函数名	sf_read_sr_16bits
函数原型	uint16_t sf_read_sr_16bits(HS_SF_Type *sf, uint32_t cs);
功能描述	读取 FLASH 的状态寄存器的值（16 位）
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	16 位 FLASH 的状态寄存器的值

5.2.3.4. sf_wait_sr_no_busy

函数名	sf_wait_sr_no_busy
函数原型	void sf_wait_sr_no_busy(HS_SF_Type *sf, uint32_t cs);
功能描述	读取状态寄存器 SR1[0]的值，判断 flash 是否完成编程、擦除、写状态寄存器操作
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.5. sf_write_enable

函数名	sf_write_enable
函数原型	void sf_write_enable(HS_SF_Type *sf, uint32_t cs);
功能描述	FLASH 写使能
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.6. sf_write_sr

函数名	sf_write_sr
函数原型	void sf_write_sr(HS_SF_Type *sf, uint32_t cs, uint8_t sr);
功能描述	向 FLASH 的状态寄存器写入指定的值
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	sr：写入 FLASH 状态寄存器中的值
输出参数	无
返回值	无

5.2.3.7. sf_write_sr_16bits

函数名	sf_write_sr_16bits
函数原型	void sf_write_sr_16bits(HS_SF_Type *sf, uint32_t cs, uint16_t sr);
功能描述	向 FLASH 状态寄存器写 16 位指定数据
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	sr：写入 FLASH 状态寄存器中的值
输出参数	无
返回值	无

5.2.3.8. sf_write_sr_mask

函数名	sf_write_sr_mask
函数原型	void sf_write_sr_mask(HS_SF_Type *sf, uint32_t cs, uint8_t mask, uint8_t value);
功能描述	用掩码方式向 FLASH 状态寄存器写值
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	mask：8 位掩码
输入参数 4	value：写入 FLASH 状态寄存器中的值（8 位）
输出参数	无
返回值	无

5.2.3.9. sf_write_sr_mask_16bits

函数名	sf_write_sr_mask_16bits
函数原型	void sf_write_sr_mask_16bits(HS_SF_Type *sf, uint32_t cs, uint16_t mask, uint16_t value);
功能描述	用掩码方式向 FLASH 状态寄存器写入 16 位的值
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	mask：16 位掩码
输入参数 4	value：写入 FLASH 状态寄存器中的值（16 位）
输出参数	无
返回值	无

5.2.3.10. sf_quad_enable

函数名	sf_quad_enable
函数原型	void sf_quad_enable(HS_SF_Type *sf, uint32_t cs, bool enable);
功能描述	四线 FLASH 使能
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	enable：使能与否
输出参数	无
返回值	无

5.2.3.11. sf_otp_set

函数名	sf_otp_set
函数原型	void sf_otp_set(HS_SF_Type *sf, uint32_t cs, uint8_t lb_mask);
功能描述	设置 FLASH 状态寄存器的 LB 位（用于指示 OTP 的情况）
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	lb_mask：lb 位的掩码
输出参数	无
返回值	无

5.2.3.12. sf_otp_get

函数名	sf_otp_get
函数原型	uint8_t sf_otp_get(HS_SF_Type *sf, uint32_t cs);
功能描述	获得 FLASH 状态寄存器中的 LB 位上的值（用于指示 OTP 的情况）
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.13. sf_lowpower_enter

函数名	sf_lowpower_enter
函数原型	void sf_lowpower_enter (HS_SF_Type *sf, uint32_t cs);
功能描述	FLASH 进入低功耗模式
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.14. sf_lowpower_leave

函数名	sf_lowpower_leave
函数原型	void sf_lowpower_leave (HS_SF_Type *sf, uint32_t cs);
功能描述	退出低功耗模式
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.15. sf_unlock_all

函数名	sf_unlock_all
函数原型	void sf_unlock_all(HS_SF_Type *sf, uint32_t cs);
功能描述	解除写保护
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.16. sf_read_id

函数名	sf_read_id
函数原型	uint32_t sf_read_id(HS_SF_Type *sf, uint32_t cs);
功能描述	读 FLASH ID
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	FLASH ID（低 24 位）

5.2.3.17. sf_read_uid_ex

函数名	sf_read_uid_ex
函数原型	void sf_read_uid_ex(HS_SF_Type *sf, uint32_t cs, void *data, uint32_t length);
功能描述	读取标准 UID 值
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	data：指向 FLASH 读取的 UID 值的首地址
输入参数 4	length：读取数据的长度
输出参数	data：指向 FLASH 读取的 UID 值的首地址

返回值	无

5.2.3.18. sf_read_uid

函数名	sf_read_uid
函数原型	uint32_t sf_read_uid(HS_SF_Type *sf, uint32_t cs);
功能描述	读取 32 位 UID 值
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	FLASH uid 32 位的异或值（如：puya 的 FLASH uid 一共 128 位，分为 4 个 32 位）

5.2.3.19. sf_erase_chip

函数名	sf_erase_chip
函数原型	void sf_erase_chip(HS_SF_Type *sf, uint32_t cs);
功能描述	擦除整个 FLASH
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.20. sf_erase_sector

函数名	sf_erase_sector
函数原型	void sf_erase_sector(HS_SF_Type *sf, uint32_t cs, uint32_t addr);
功能描述	擦除 FLASH 一个扇区
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要擦除扇区的地址 注：地址必须是 sector 大小对齐的
输出参数	无
返回值	无

5.2.3.21. sf_erase_block

函数名	sf_erase_block
函数原型	void sf_erase_block(HS_SF_Type *sf, uint32_t cs, uint32_t addr);
功能描述	擦除 FLASH 一个块（64K）
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要擦除块的地址
输出参数	无
返回值	无

5.2.3.22. sf_erase_sec

函数名	sf_erase_sec
函数原型	void sf_erase_sec(HS_SF_Type *sf, uint32_t cs, uint32_t addr);
功能描述	擦除 FLASH 安全寄存器
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要擦除的安全寄存器地址
输出参数	无
返回值	无

5.2.3.23. sf_erase

函数名	sf_erase
函数原型	void sf_erase(HS_SF_Type *sf, uint32_t cs, uint32_t addr, uint32_t length);
功能描述	FLASH 擦除指定地址的部分长度
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要擦除的 FLASH 地址（实际擦除地址为输入地址所在扇区的起始地址）
输入参数 4	length：擦除的长度（擦除长度单位为一个扇区，如果 length=0，将擦除整个 FLASH）
输出参数	无
返回值	无

5.2.3.24. sf_write_page_nodma

函数名	sf_write_page_nodma
函数原型	void sf_write_page_nodma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const void *data, uint32_t length);
功能描述	使用非 DMA 方式向 FLASH 指定地址写数据（不支持跨页写）
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要写入 FLASH 地址
输入参数 4	data：指向写入数据 buffer 的首地址
输入参数 5	length：数据长度
输出参数	无
返回值	无

5.2.3.25. sf_write_page_dma

函数名	sf_write_page_dma
函数原型	void sf_write_page_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const void *data, uint32_t length);
功能描述	使用 DMA 方式向 FLASH 指定地址写数据（不支持跨页写）
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要写入 FLASH 地址 注：该地址必须 page 对齐
输入参数 4	data：指向写入数据 buffer 的首地址 注：该地址必须 4 字节对齐
输入参数 5	length：数据长度
输出参数	无
返回值	无

5.2.3.26. sf_write_page

函数名	sf_write_page
函数原型	void sf_write_page(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const void *data, uint32_t length);
功能描述	向 FLASH 指定地址写数据（不支持跨页写）
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要写入 FLASH 地址 注：该地址必须 page 对齐
输入参数 4	data：指向写入数据 buffer 的首地址
输入参数 5	length：数据长度

输出参数	无
返回值	无

5.2.3.27. sf_write_sec

函数名	sf_write_sec
函数原型	void sf_write_sec(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const void *data, uint32_t length);
功能描述	向安全寄存器写数据
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要写入的地址
输入参数 4	data：指向写入数据 buffer 的首地址
输入参数 5	length：数据长度
输出参数	无
返回值	无

5.2.3.28. sf_write

函数名	sf_write
函数原型	void sf_write(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const void *data, uint32_t length);
功能描述	向 FLASH 指定地址写数据（支持跨页写）
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：将要写入的地址值
输入参数 4	data：指向写入数据 buffer 的首地址
输入参数 5	length：数据长度
输出参数	无
返回值	无

5.2.3.29. sf_read_normal_nodma

函数名	sf_read_normal_nodma
函数原型	void sf_read_normal_nodma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data, uint32_t length);
功能描述	使用一线非 DMA 方式读 FLASH 内部偏移地址为 addr、长度为 length 的内容到 data 指向的 buffer 地址中
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：要读取的地址
输入参数 4	data：指向读出数据 buffer 的首地址

	注：该地址无字节对齐要求
输入参数 5	length：数据长度
输出参数	data：指向读出数据 buffer 的首地址
返回值	无

5.2.3.30. sf_read_normal_dma

函数名	sf_read_normal_dma
函数原型	void sf_read_normal_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data, uint32_t length);
功能描述	使用一线 DMA 方式读 FLASH 内部偏移地址为 addr、长度为 length 的内容到 data 指向的 buffer 地址中
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：要读取的地址
输入参数 4	data：指向读出数据 buffer 的首地址 注：该地址必须 4 字节对齐
输入参数 5	length：数据长度
输出参数	data：指向读出数据 buffer 的首地址
返回值	无

5.2.3.31. sf_read_fast_dma

函数名	sf_read_fast_dma
函数原型	void sf_read_fast_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data, uint32_t length);
功能描述	使用一线 DMA 方式快读 FLASH 内部偏移地址为 addr、长度为 length 的内容到 data 指向的 buffer 地址中。
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：要读取的地址
输入参数 4	data：指向读出数据 buffer 的首地址 注：该地址必须 4 字节对齐
输入参数 5	length：数据长度
输出参数	data：指向读出数据 buffer 的首地址
返回值	无

5.2.3.32. sf_read_fast_dual_dma

函数名	sf_read_fast_dual_dma
函数原型	void sf_read_fast_dual_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data, uint32_t length);
功能描述	使用两线 DMA 方式读取 FLASH 内部偏移地址为 addr、长度为 length 的内容到 data 指向的 buffer 地址中。
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：要读取的地址
输入参数 4	data：指向读出数据 buffer 的首地址 注：该地址必须 4 字节对齐
输入参数 5	length：数据长度
输出参数	data：指向读出数据 buffer 的首地址
返回值	无

5.2.3.33. sf_read_fast_quad_naked_dma

函数名	sf_read_fast_quad_naked_dma
函数原型	void sf_read_fast_quad_naked_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data, uint32_t length);
功能描述	使用四线 DMA 方式读取 FLASH 内部偏移地址为 addr、长度为 length 的内容到 data 指向的 buffer 地址中 在使用该函数前要使能 sf_quad_enable 函数
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：要读取的地址
输入参数 4	data：指向读出数据 buffer 的首地址 注：该地址必须 4 字节对齐
输入参数 5	length：数据长度
输出参数	data：指向读出数据 buffer 的首地址
返回值	无

5.2.3.34. sf_read_fast_quad_dma

函数名	sf_read_fast_quad_dma
函数原型	void sf_read_fast_quad_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data, uint32_t length);
功能描述	使用四线 DMA 方式读取 FLASH 内部偏移地址为 addr、长度为 length 的内容到 data 指向的 buffer 地址中
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择

输入参数 3	addr：要读取的地址
输入参数 4	data：指向读出数据 buffer 的首地址 注：该地址必须 4 字节对齐
输入参数 5	length：数据长度
输出参数	data：指向读出数据 buffer 的首地址
返回值	无

5.2.3.35. sf_read_sec

函数名	sf_read_sec
函数原型	void sf_read_sec(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data, uint32_t length);
功能描述	读取 FLASH 安全寄存器的值
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：要读取的地址
输入参数 4	data：指向读出数据 buffer 的首地址
输入参数 5	length：数据长度
输出参数	data：指向读出数据 buffer 的首地址
返回值	无

5.2.3.36. sf_read

函数名	sf_read
函数原型	void sf_read(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data, uint32_t length);
功能描述	读取 FLASH 内部偏移地址为 addr、长度为 length 的内容到 data 指向的 buffer 地址中
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	addr：要读取的地址
输入参数 4	data：指向读出数据 buffer 的首地址
输入参数 5	length：数据长度
输出参数	data：指向读出数据 buffer 的首地址
返回值	无

5.2.3.37. sf_config

函数名	sf_config
函数原型	void sf_config(HS_SF_Type *sf, uint32_t cs, const sf_config_t *config);
功能描述	sf 控制器参数配置
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输入参数 3	config：FLASH 配置参数，包括频率、位宽和延迟采样时间
输出参数	无
返回值	无

5.2.3.38. sf_enable

函数名	sf_enable
函数原型	void sf_enable(HS_SF_Type *sf, uint32_t cs);
功能描述	sf controllor 使能
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.39. sf_disable

函数名	sf_disable
函数原型	void sf_disable(HS_SF_Type *sf, uint32_t cs);
功能描述	sf controllor 关闭
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	无

5.2.3.40. sf_iflash_auto_close

函数名	sf_iflash_auto_close
函数原型	void sf_iflash_auto_close(uint32_t delay_ms)
功能描述	经过 dealy_ms 自动关闭 sf controllor
输入参数	delay_ms：设定时间，单位 ms
输出参数	无
返回值	无

5.2.3.41. sf_iflash_extra_open_delay_set

函数名	sf_iflash_extra_open_delay_set
函数原型	void sf_iflash_extra_open_delay_set(uint32_t delay_10us);
功能描述	sf controller 使能的额外延迟设置
输入参数	delay_10us：延迟时间，单位 10us
输出参数	无
返回值	无

5.2.3.42. sf_detect

函数名	sf_detect
函数原型	bool sf_detect(HS_SF_Type *sf, uint32_t cs);
功能描述	检测 FLASH ID 是否有效，并存储 FLASH ID 相关数据
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	true 或者 false

5.2.3.43. sf_status

函数名	sf_status
函数原型	sf_status_t sf_status(HS_SF_Type *sf, uint32_t cs);
功能描述	读取 FLASH 状态
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	FLASH 的状态，返回值包括 SF_STATUS_NONE， SF_STATUS_ABSENT， SF_STATUS_PRESENT

5.2.3.44. sf_capacity

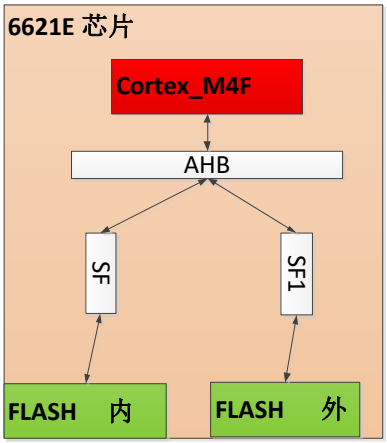
函数名	sf_capacity
函数原型	uint32_t sf_capacity(HS_SF_Type *sf, uint32_t cs);
功能描述	读取 FLASH 容量
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	FLASH 的容量

5.2.3.45. sf_id

函数名	sf_id
函数原型	uint32_t sf_id(HS_SF_Type *sf, uint32_t cs);
功能描述	在执行完 sf_detect 函数后，读取 FLASH ID
输入参数 1	sf：可以是 HS_SF、HS_SF1 或者 HS_SF2，来选择 FLASH 外设控制器
输入参数 2	cs：CS 信号选择
输出参数	无
返回值	FLASH ID 值

5.3. SFLASH 控制器应用例程

5.3.1.SFLASH 控制器接口以及连线方式



		总线访问基址	Cmd访问基址
内部FLASH	HS_SF CS=0	0x50000000	0x00000000
外部FLASH	HS_SF1 CS=1	0x52000000	0x00000000

总线访问方式（仅支持读）	例:读取片内FLASH地址a (*(volatile uint8_t *)(0x50000000+a))
CMD访问方式	读取片内FLASH地址a，长度为100字节的数据，放到数组sf_read_buf中。 sf_read(HS_SF, 0, a, (uint8_t *)sf_read_buf, 10);
	将长度为100字节的数组sf_read_buf内容写到片内FLASH地址a中。 sf_write(HS_SF, 0, a, (uint8_t *)sf_read_buf, 10);
	将FLASH内部偏移地址为128*1024长度为4*1024的内容擦除。 sf_erase(HS_SF, 0, 128*1024, 4*1024);

图 5.1 SFLASH 接口

如上图所示，BL1824X 芯片内部集成了 2 个 SFLASH Controller，其中 SF 连接内部 FLASH，SF1 连接外部 FLASH。其对应地址范围分别为 0x50000000+FLASH 总容量、0x52000000+FLASH 总容量。支持总线访问方式（也叫 CPU 透明读方式）和 CMD 访问方式。用户自己使用的时候，通常是通过 CMD 访问方式（即调用 SF 相关封装接口函数）对 FLASH 进行读写操作。

FLASH 控制器和 FLASH 设备之间的连线方式如下图所示。

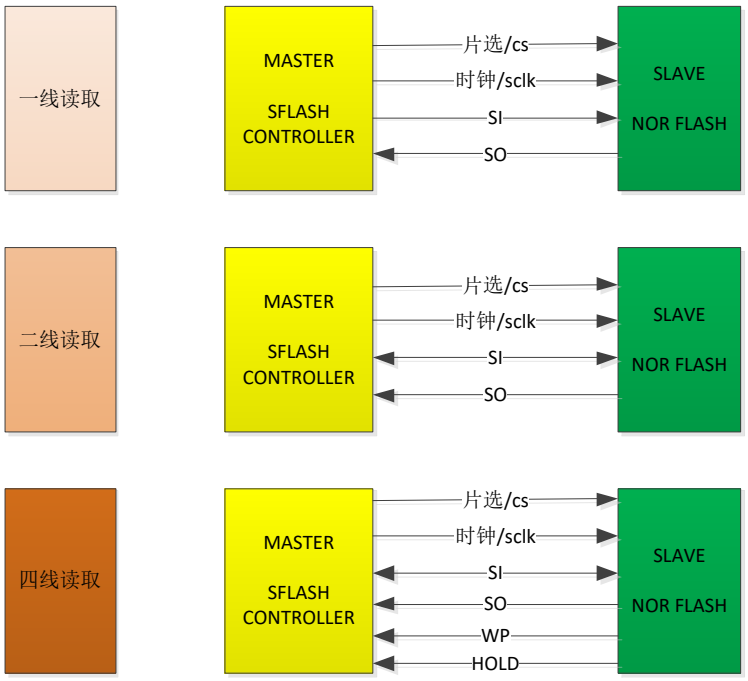


图 5.2 FLASH 连线方式

5.3.2.SFLASH Controller 工作流程

初始化	步骤一	SF软复位 SF时钟使能	sfb_enable
	步骤二	给SF上电	sf_iflash_power_enable(true)
	步骤三	配置SF控制寄存器	sf_config(sf,cs,config)
具体	步骤四	必要步骤，检测是否存FLASH。若存在则读取FLASH ID并储存	sf_detect(sf, cs)
	步骤N	读、写、擦操作	sf_erase(sf, cs, addr, length) sf_write(cs, addr, data, length) sf_read(sf, cs, addr, data, length)等

CS=0

CS=1

0x00028	spi_configuration_0	Spi configuration Register 0
0x0002C	spi_cs_configuration_0	Spi CS configuration Register 0

0x00030	spi_configuration_1	Spi configuration Register 1
0x00034	spi_cs_configuration_1	Spi CS configuration Register 1

图 5.3 工作流程

SFLASH Controller 工作流程如图所示，库函数接口 sf_enable(sf,cs, addr, data, length)中包括了步骤一和步骤二。

5.3.3.擦操作

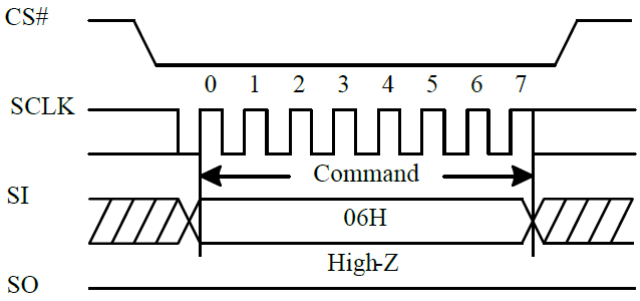


图 5.4 写使能顺序（命令 06H）

发出写使能指令的顺序为：CS#低→发送写使能指令码→CS#高。

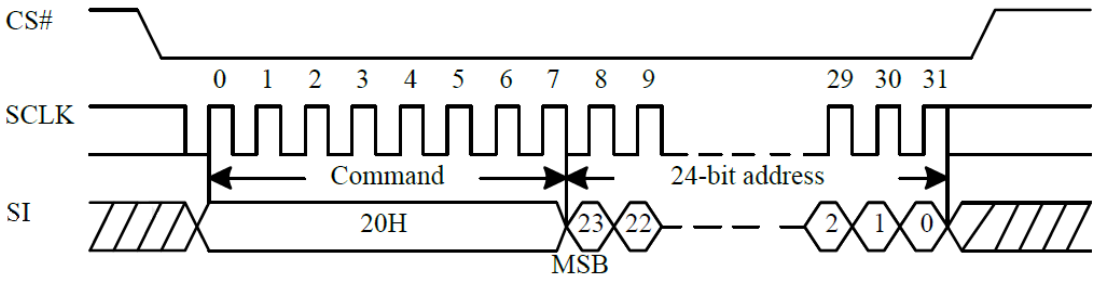


图 5.5 扇区擦除顺序（命令 20H）

发出扇区擦除指令的顺序为：CS#低→发送扇区擦除指令码→SI 上的 3 字节地址→CS#高。

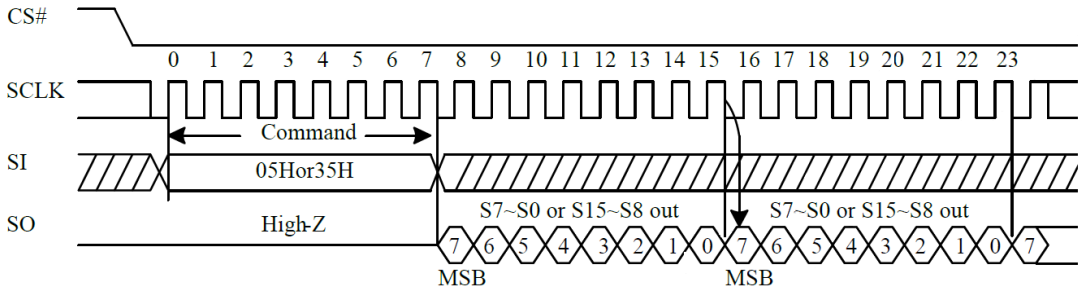


图 5.6 读状态寄存器顺序（05H 或 35H）

发出读状态寄存器指令的顺序为：CS#低→发送读状态寄存器的指令码→状态寄存器数据在 SO 上输出。

写使能	sf->COMMAND_DATA0_REG = 0x06 <<24;
	sf->COMMAND = (0<<12)|(8<<5)|(0<<4)|(0<<2)|(2<<0);
	while(!(sf->RAW_INTR_STATUS & 1));
	sf->RAW_INTR_STATUS = 1;
		0x06为写允许操作；
		对应COMMAND中的cmd_bits字段为8；
		对应COMMAND中的command字段为2（即：写）

擦除	sf->COMMAND_DATA0_REG = (0x20 <<24) | (addr);
	sf->COMMAND = (0<<12)|(32<<5)|(0<<4)|(0<<2)|(2<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
		0x20为Sector擦除操作；
		对应COMMAND中的cmd_bits字段为8；
		对应COMMAND中的command字段为2（即：写）
		我们用的norFlash，对应的sector大小为4k字节；

等待完毕	uint32_t rdata[1];
	while (1)
	{
	// ctrl
	sf->COMMAND_DATA0_REG = 0x05<<24;
	sf->COMMAND = (0<<12)|(16<<5)|(0<<4)|(0<<2)|(1<<0);
	// wait done
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
	rdata[0] = sf->READ0_REG & 0xFF;
	if ((rdata[0] & 1) == 0)
	break;
	else
	delay_ms(1);
	}
		0x05为读状态寄存器操作；
		对应COMMAND中的cmd_bits字段为16；
		对应COMMAND中的command字段为1（即：读）

对 Flash 的其他操作可以查阅 Flash 手册。

5.3.4.写操作

5.3.4.1. 写操作—非 DMA 方式

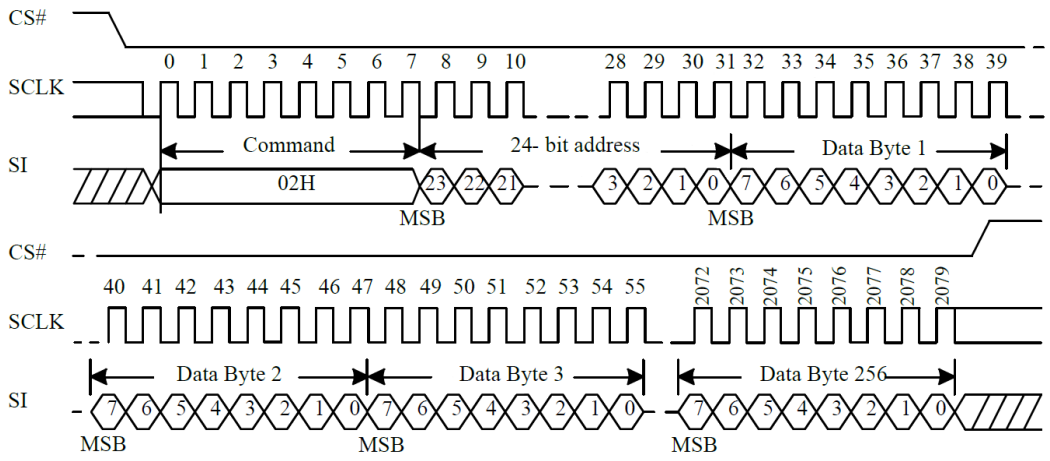


图 5.7 页编程顺序（02H）

发出页编程指令的顺序为：CS#低→发送页编程指令码→SI 上的 3 个字节地址→SI 上的至少 1 个字节数据→CS#高。

写使能	sf->COMMAND_DATA0_REG = 0x06 <<24;
	sf->COMMAND = (0<<12)|(8<<5)|(0<<4)|(0<<2)|(2<<0);
	while(!(sf->RAW_INTR_STATUS & 1));
	sf->RAW_INTR_STATUS = 1;
		0x06为写允许操作；
		对应COMMAND中的cmd_bits为8；
		对应COMMAND中的command为2（即：写）；

写操作	sf->COMMAND_DATA0_REG = (0x02<<24) | (addr);
	sf->ADDRESS_REG = (uint32_t) data;
	sf->COMMAND = (0<<12)|(32<<5)|(1<<4)|(0<<2)|(2<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
	
	循环
	sf->COMMAND_DATA0_REG = data[i];
	sf->COMMAND_DATA1_REG = data[i+1];
	sf->COMMAND = (0<<12)|(64<<5)|(1<<4)|(0<<2)|(2<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
	
	最后一次
	sf->COMMAND_DATA0_REG = data[i];
	sf->COMMAND_DATA1_REG = data[i+1]; //视情况而定，可能没有
	sf->COMMAND = (0<<12)|(LEN<<5)|(1<<4)|(0<<2)|(2<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
		0x02为页写（page programme）操作；
		addr为被写的地址；
		data为写的数据源地址；
		对应COMMAND中的cmd_bits为32；
		对应COMMAND中的command为2；
		对应COMMAND中的data_bytes为0；
		
		不用DMA方式，而是一个一个写进去的。

等待完毕	uint32_t rdata[1];
	while (1)
	{
	// ctrl
	sf->COMMAND_DATA0_REG = 0x05<<24;
	sf->COMMAND = (0<<12)|(16<<5)|(0<<4)|(0<<2)|(1<<0);
	// wait done
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
	rdata[0] = sf->READ0_REG & 0xFF;
	if ((rdata[0] & 1) == 0)
	break;
	else
	delay_ms(1);
	}
		0x05为读状态寄存器操作；
		对应COMMAND中的cmd_bits为16；
		对应COMMAND中的command为1（即：读）；

5.3.4.2. 写操作—DMA 方式

写使能	sf->COMMAND_DATA0_REG = 0x06 <<24;
	sf->COMMAND = (0<<12)|(8<<5)|(0<<4)|(0<<2)|(2<<0);
	while(!(sf->RAW_INTR_STATUS & 1));
	sf->RAW_INTR_STATUS = 1;
		0x06为写允许操作；
		对应COMMAND中的cmd_bits为8；
		对应COMMAND中的command为2（即：写）；

写操作	sf->COMMAND_DATA0_REG = (0x02<<24) | (addr);
	sf->ADDRESS_REG = (uint32_t) data;
	sf->COMMAND = (256<<12)|(32<<5)|(0<<4)|(0<<2)|(2<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
		0x02为页写（page programme）操作；
		addr为写的地址；
		data为写的数据源地址；
		对应COMMAND中的cmd_bits为32；
		对应COMMAND中的command为2；
		对应COMMAND中的data_bytes为256；
		我们用的norFlash，对应的page大小为256字节；

等待完毕	uint32_t rdata[1];
	while (1)
	{
	// ctrl
	sf->COMMAND_DATA0_REG = 0x05<<24;
	sf->COMMAND = (0<<12)|(16<<5)|(0<<4)|(0<<2)|(1<<0);
	
	// wait done
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
	
	rdata[0] = sf->READ0_REG & 0xFF;
	
	if ((rdata[0] & 1) == 0)
	break;
	else
	delay_ms(1);
	}
		0x05为读状态寄存器操作；
		对应COMMAND中的cmd_bits为16；
		对应COMMAND中的command为1（即：读）；

5.3.5.读操作

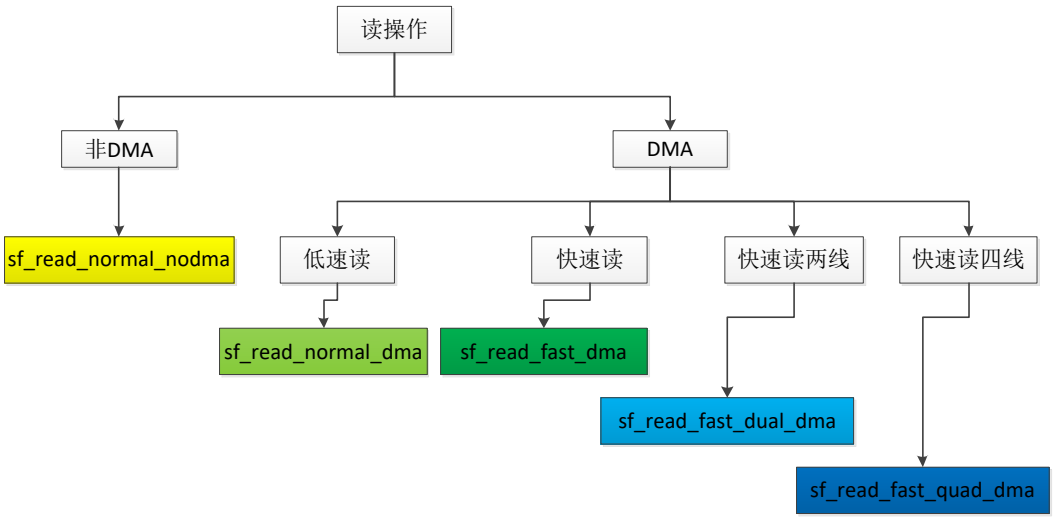


图 5.8 读操作流程

5.3.5.1. 读操作—非 DMA

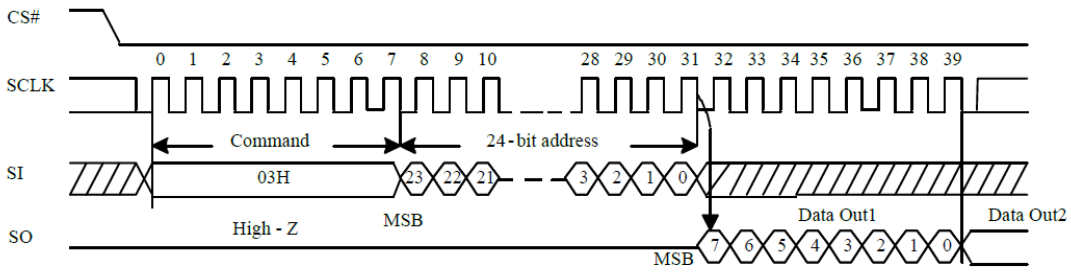


图 5.9 读顺序（03H）

发出 READ 指令的顺序为：CS#低→发送 READ 指令码→SI 上的 3 个字节地址→在 SO 上读取数据，在数据读取过程中，可随时将 CS#拉高。

读操作命令发送	sf->COMMAND_DATA0_REG = (0x03<<24) | (addr);
	sf->ADDRESS_REG = (uint32_t) data;
	sf->COMMAND = (LEN<<12)|(32<<5)|(1<<4)|(0<<2)|(1<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
		0x03为读操作；时钟<=20M
		addr为读的源地址；
		data为读出来数据存放的首地址；
		对应COMMAND中的cmd_bits为32；
		对应COMMAND中的command为1；
		对应COMMAND中的data_bytes为LEN；
		对应COMMAND中的keep_cs为1；

具体读内容	sf->COMMAND_DATA0_REG = 0x0 ;
	sf->COMMAND = (0<<12)|((LEN<<3)<<5)|(0<<4)|(0<<2)|(1<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
	data[0] = sf->READ0_REG;
	data[1] = sf->READ1_REG;
		对应COMMAND中的cmd_bits为(LEN<<3)；
		对应COMMAND中的command为1；
		对应COMMAND中的keep_cs为0；

5.3.5.2. 读操作—DMA

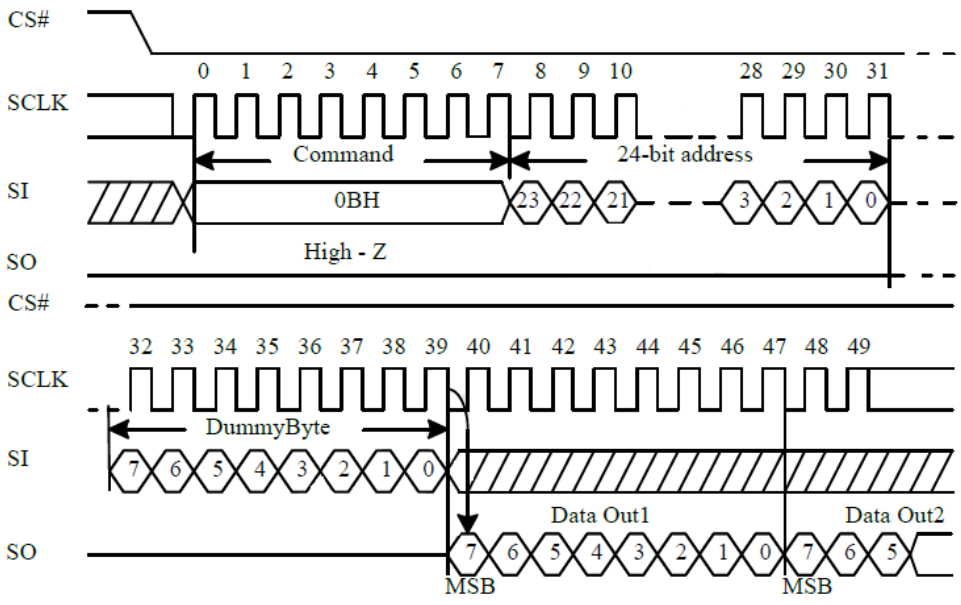


图 5.10 Fast Read 顺序（0BH）

发出快读（Fast Read）指令的顺序是：CS#低→发送 Fast Read 指令码→SI 上的 3 个字节地址→SI 上的 8 个空指令周期→在 SO 上读取数据→可以在数据读取过程中随时拉高 CS#，以结束 Fast Read 操作。

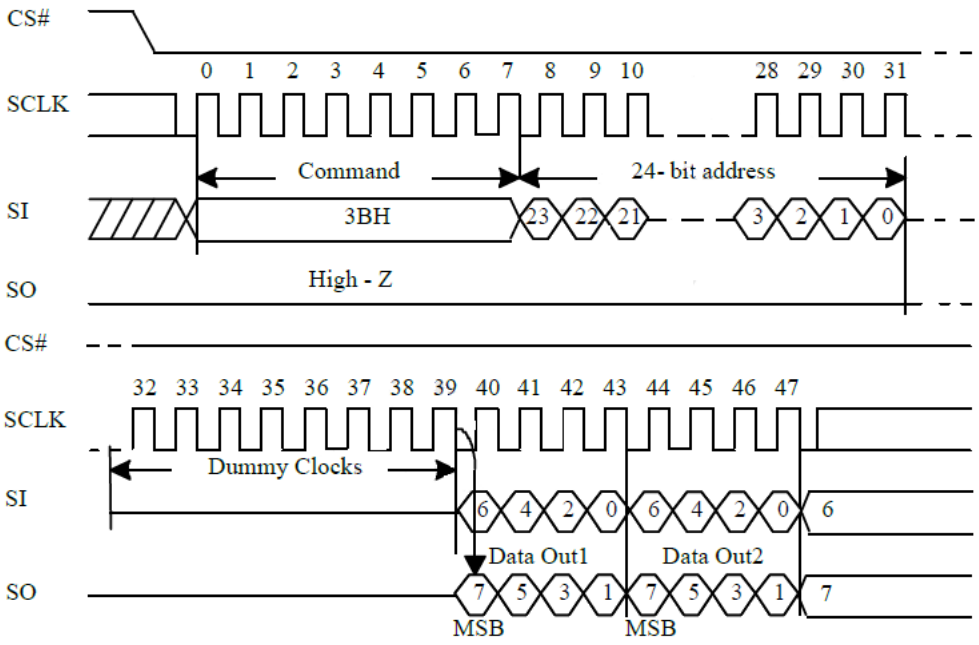


图 5.11 Dual Read 顺序（3BH）

发出两线读（Dual Read）指令的顺序是：CS#低→发送 Dual Read 指令码→SI 上的 3 个字节地址→SI 上的 8 个空指令周期→在 SO1 和 SO0 上交错读取数据→可以在数据读取过程中随时拉高 CS#，以结束 Dual Read 操作。

读操作	sf->COMMAND_DATA0_REG = (0x03<<24) | (addr);
	sf->ADDRESS_REG = (uint32_t) data;
	sf->COMMAND = (LEN<<12)|(32<<5)|(0<<4)|(0<<2)|(1<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
		0x03为读操作；时钟<=20M
		addr为读的源地址；
		data为读出来数据存放的首地址；
		对应COMMAND中的cmd_bits为32；
		对应COMMAND中的command为1；
		对应COMMAND中的data_bytes为LEN；

读操作
(快读)
	sf->COMMAND_DATA0_REG = (0x0B<<24) | (addr);
	sf->ADDRESS_REG = (uint32_t) data;
	sf->COMMAND = (LEN<<12)|(40<<5)|(0<<4)|(0<<2)|(1<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
		0x0B为快读操作；时钟>20M
		addr为写的地址；
		data为写的数据源地址；
		对应COMMAND中的cmd_bits为40；
		对应COMMAND中的command为1；
		对应COMMAND中的data_bytes为LEN；

读操作
(两线)
	sf->COMMAND_DATA0_REG = (0x3B<<24) | (addr);
	sf->ADDRESS_REG = (uint32_t) data;
	sf->COMMAND = (LEN<<12)|(40<<5)|(0<<4)|(0<<2)|(1<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
		0x3B为两线快读操作；
		addr为读的源地址；
		data为读出来数据存放的首地址；
		对应COMMAND中的cmd_bits为40；
		对应COMMAND中的command为1；
		对应COMMAND中的data_bytes为LEN；

5.3.5.3. 读操作—四线 DMA

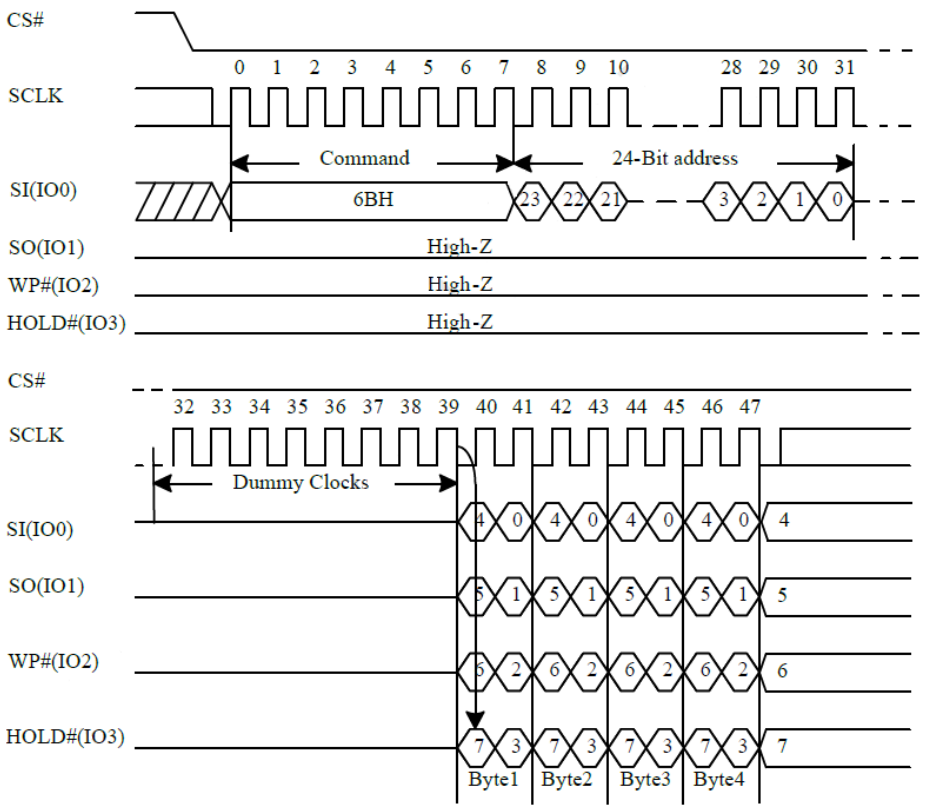


图 5.12 Quad Read 顺序（6BH）

发出两线读（Quad Read）指令的顺序是：CS#低→发送 Quad Reads 指令码→SI 上的 3 个字节地址→SI 上的 8 个空指令周期→在 SO3, SO2, SO1 和 SO0 上交错读取数据→可以在数据读取过程中随时拉高 CS#，以结束 Quad Read 操作。

四线使能	写使能	sf->COMMAND_DATA0_REG = 0x06 <<24;
		sf->COMMAND = (0<<12)|(8<<5)|(0<<4)|(0<<2)|(2<<0);
		while(!(sf->RAW_INTR_STATUS & 1));
		sf->RAW_INTR_STATUS = 1;
			0x06为写允许操作；
			对应COMMAND中的cmd_bits为8；
			对应COMMAND中的command为2（即：写）；

	写状态寄存器命令发送	sf->COMMAND_DATA0_REG = (0x01<<24) ;
		sf->COMMAND = (0<<12)|(8<<5)|(1<<4)|(0<<2)|(2<<0);
		while(!(sf->RAW_INTR_STATUS &1));
		sf->RAW_INTR_STATUS = 1;
			0x01为写状态寄存器操作；
			对应COMMAND中的cmd_bits为8；
			对应COMMAND中的command为2；
			对应COMMAND中的keep_cs为1；

	具体写内容	sf->COMMAND_DATA0_REG = 0x0200 ;
		sf->COMMAND = (0<<12)|(16<<5)|(0<<4)|(0<<2)|(2<<0);
		while(!(sf->RAW_INTR_STATUS &1));
		sf->RAW_INTR_STATUS = 1;
			0x0200为 四线使能；
			对应COMMAND中的cmd_bits为16；
			对应COMMAND中的command为2；
			对应COMMAND中的keep_cs为0；

读操作	sf->COMMAND_DATA0_REG = (0x6B<<24) | (addr);
	sf->ADDRESS_REG = (uint32_t) data;
	sf->COMMAND = (LEN<<12)|(40<<5)|(0<<4)|(0<<2)|(1<<0);
	while(!(sf->RAW_INTR_STATUS &1));
	sf->RAW_INTR_STATUS = 1;
			0x6B为四线快读操作；
			addr为读的源地址；
			data为读出来数据存放的首地址；
			对应COMMAND中的cmd_bits为40；
			对应COMMAND中的command为1；
			对应COMMAND中的data_bytes为LEN；

四线去使能	写使能	sf->COMMAND_DATA0_REG = 0x06 <<24;
		sf->COMMAND = (0<<12)|(8<<5)|(0<<4)|(0<<2)|(2<<0);
		while(!(sf->RAW_INTR_STATUS & 1));
		sf->RAW_INTR_STATUS = 1;
			0x06为写允许操作；
			对应COMMAND中的cmd_bits为8；
			对应COMMAND中的command为2（即：写）；

	写状态寄存器命令发送	sf->COMMAND_DATA0_REG = (0x01<<24) ;
		sf->COMMAND = (0<<12)|(8<<5)|(1<<4)|(0<<2)|(2<<0);
		while(!(sf->RAW_INTR_STATUS &1));
		sf->RAW_INTR_STATUS = 1;
			0x01为写状态寄存器操作；
			对应COMMAND中的cmd_bits为8；
			对应COMMAND中的command为2；
			对应COMMAND中的keep_cs为1；

	具体写内容	sf->COMMAND_DATA0_REG = 0x0000 ;
		sf->COMMAND = (0<<12)|(16<<5)|(0<<4)|(0<<2)|(2<<0);
		while(!(sf->RAW_INTR_STATUS &1));
		sf->RAW_INTR_STATUS = 1;
			0x0000为 四线去使能；
			对应COMMAND中的cmd_bits为16；
			对应COMMAND中的command为2；
			对应COMMAND中的keep_cs为0；

5.3.6.实例应用

```
sf_config_t sfInsideCfg[] = {
    {1000000,   SF_WIDTH_1LINE,  5},
    {8000000,   SF_WIDTH_1LINE,  5},
    {16000000,  SF_WIDTH_1LINE,  5},
    {32000000,  SF_WIDTH_1LINE,  5},
    {64000000,  SF_WIDTH_1LINE,  5},
    {1000000,   SF_WIDTH_2LINE,  5},
    {8000000,   SF_WIDTH_2LINE,  5},
    {16000000,  SF_WIDTH_2LINE,  5},
    {32000000,  SF_WIDTH_2LINE,  5},
    {64000000,  SF_WIDTH_2LINE,  5},
    {1000000,   SF_WIDTH_4LINE,  5},
    {8000000,   SF_WIDTH_4LINE,  5},
    {16000000,  SF_WIDTH_4LINE,  5},
    {32000000,  SF_WIDTH_4LINE,  5},
    {64000000,  SF_WIDTH_4LINE,  5},
};
```

```
void flash_cmd_dma_64m_over_sector(HS_SF_Type *sf, uint32_t cs)
{
    uint32_t j=0;
    uint32_t flashAddr = (ADDR_OFFSET + LEN_SECTOR - 100 ); // make sure the
address is over the two sector

    sf_config_t sfCfg = {64000000,   SF_WIDTH_4LINE,   5};

    log_debug("\r\n--------------------------------------\r\n");
    log_debug("flash config clk=%d, width=%x,delay=%x\r\n",
                        sfCfg.freq_hz,sfCfg .width,sfCfg .delay);

    memset((void*)r_buf, 0, sizeof(r_buf));
    for(j=0; j<sizeof(w_buf); j++)
        w_buf[j] = rand();

    log_debug_array_ex("w_buf", w_buf, sizeof(w_buf));

    if( HS_SF != sf )
    {
        flash_out_pinmux_cfg(sf, cs);
    }

    // open
    sf_enable(sf, cs);

    // config
    sf_config(sf, cs, &sfCfg);

    // Flash ID
    flash_id = sf_read_id(sf, cs);
    log_debug("flash ID: %08X\n", flash_id);
    if( (( HS_SF == sf ) && (flash_id != FLASH_ID_1_INSIDE) && (flash_id !=
FLASH_ID_2_INSIDE))
        || ( ( HS_SF1 == sf ) && (flash_id != FLASH_ID_1_OUTSIDE) && (flash_id !=
FLASH_ID_2_OUTSIDE) ) )
    {
        log_debug("Error flash ID, exit!!! Please change delay\n");
        return;
    }


    // detect
```

```
    sf_detect(sf, cs);

    log_debug("erase sector \r\n");
    sf_erase(HS_SF, 0, ADDR_OFFSET, LEN_SECTOR*2);

    // Read data (no DMA)
    sf_read_fast_dma(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != 0xFF)
        {
            log_debug("Error!!! read_normal_dma id=%d, V=%02x != 0xFF----------\r\n",
j, r_buf[j]);
            log_debug_array_ex("read_normal_dma", r_buf, sizeof(r_buf));
            break;
        }
    }
    if (j == sizeof(r_buf)) log_debug("read_normal_dma over sector ok\r\n");
    memset((void*)r_buf, 0, sizeof(r_buf));

    log_debug("write 200 bytes data over sector\r\n");
    // Write data (DMA)
    sf_write(sf, cs, flashAddr, (const void*)w_buf, sizeof(w_buf));

    // Read data (DMA)
    sf_read_fast_dma(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != w_buf[j])
        {
            log_debug("Error!!! read_normal_dma id=%d, rV=%02x,
wV=%02x----------\r\n",j,r_buf[j], w_buf[j]);
            log_debug_array_ex("read_normal_dma", r_buf, sizeof(r_buf));
            break;
        }
    }
    if (j == sizeof(r_buf)) log_debug("read_normal_dma over sector ok\r\n");
    memset((void*)r_buf, 0, sizeof(r_buf));

    // Read data (DMA)
    sf_read_fast_dual_dma(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != w_buf[j])
```

```
        {
            log_debug("Error!!! read_fast_dual_dma id=%d, rV=%02x,
wV=%02x----------\r\n",j,r_buf[j], w_buf[j]);
            log_debug_array_ex("read_fast_dual_dma", r_buf, sizeof(r_buf));
            break;
        }
    }
    if (j == sizeof(r_buf)) log_debug("read_fast_dual_dma over sector ok\r\n");
    memset((void*)r_buf, 0, sizeof(r_buf));

    // Read data (DMA)
    sf_read_fast_quad_dma(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != w_buf[j])
        {
            log_debug("Error!!! read_fast_quad_dma id=%d, rV=%02x,
wV=%02x---------\r\n",j, r_buf[j], w_buf[j]);
            log_debug_array_ex("read_fast_quad_dma", r_buf, sizeof(r_buf));
            break;
        }
    }
    memset((void*)r_buf, 0, sizeof(r_buf));

    sf_erase(HS_SF, 0, ADDR_OFFSET, LEN_SECTOR*2);
    // Write data (DMA)
    sf_write(sf, cs, flashAddr, (const void*)w_buf, sizeof(w_buf));

    // Read data (DMA)
    sf_read(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != w_buf[j])
        {
            log_debug("Fail !!!! Over sector read id=%d, rV=%02x,
wV=%02x---------\r\n",j, r_buf[j], w_buf[j]);
            log_debug_array_ex("sf_read", r_buf, sizeof(r_buf));
            break;
        }
    }

    if (j == sizeof(r_buf))
        log_debug("Pass ****** Over sector read\r\n");
}
```

```
void flash_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);

    switch (case_num)
    {
        case FLASH_TEST_CASE_200103:
            log_debug("FLASH_TEST_CASE_200103\n");

            flash_cmd_dma_64m_over_sector(HS_SF,0);

            log_debug("FLASH_TEST_CASE_200103_end\n");
            break;
        default:
            log_debug("[%s][%d][The input test number is not support!]\n",
__FUNCTION__, __LINE__);

            break;
    }
}

int main(void)
{
    uint8_t         cmd[SHELL_FIFO_SIZE];
    uint8_t*        pCmd = &cmd[0];
    volatile bool ret = false;

    // Enble all IRQ quikly.
    __set_PRIMASK(0);

    // Disable WDT.
    wdt_enable(0);

    pmu_xtal32m_x2_startup();
    co_delay_ms(100);

    // Init interactive UART.
    uart_interaction_init();
    shell_init(HS_UART1);

    uint32_t uartClk = cpm_get_clock(CPM_UART1_CLK);
    uint32_t topClk = cpm_get_clock(CPM_TOP_CLK);
    uint32_t cpuClk = cpm_get_clock(CPM_CPU_CLK);
```

```
    log_debug("BL1824X_Chip Test, topClk =%d,
cpuClk=%d %d\n",topClk,cpuClk,uartClk);

    while (1)
    {
        ret = shell_get_cmd(&pCmd);

        if (ret == true)
        {
            log_debug("\r\ncmd : %s.\n", cmd);

            flash_test_case(pCmd);
        }
    }

    shell_uninit();

    return 0;
}
```

以上代码使用了 flash_cmd_dma_64m_over_sector 4 线 dma 的方式读取内部 FLASH。

如果要读写外部 FLASH，则需要添加下面的代码。

```
sf_config_t sfOutsideCfg[] = {
    {1000000,   SF_WIDTH_1LINE,  5},
    {8000000,   SF_WIDTH_1LINE,  5},
    {16000000,  SF_WIDTH_1LINE,  5},
    {32000000,  SF_WIDTH_1LINE,  5},
    {64000000,  SF_WIDTH_1LINE,  5},
    {1000000,   SF_WIDTH_2LINE,  5},
    {8000000,   SF_WIDTH_2LINE,  5},
    {16000000,  SF_WIDTH_2LINE,  5},
    {32000000,  SF_WIDTH_2LINE,  5},
    {64000000,  SF_WIDTH_2LINE,  5},
    {1000000,   SF_WIDTH_4LINE,  5},
    {8000000,   SF_WIDTH_4LINE,  5},
    {16000000,  SF_WIDTH_4LINE,  5},
    {32000000,  SF_WIDTH_4LINE,  5},
    {64000000,  SF_WIDTH_4LINE,  5},
};
void flash_cmd_dma_64m_over_sector(HS_SF_Type *sf, uint32_t cs)
{
    uint32_t j=0;
    uint32_t flashAddr = (ADDR_OFFSET + LEN_SECTOR - 100 ); // make sure the
address is over the two sector
```

```
    sf_config_t sfCfg = {64000000,   SF_WIDTH_4LINE,   5};

    log_debug("\r\n--------------------------------------\r\n");
    log_debug("flash config clk=%d, width=%x,delay=%x\r\n",
                        sfCfg.freq_hz,sfCfg .width,sfCfg .delay);

    memset((void*)r_buf, 0, sizeof(r_buf));
    for(j=0; j<sizeof(w_buf); j++)
        w_buf[j] = rand();

    log_debug_array_ex("w_buf", w_buf, sizeof(w_buf));

    if( HS_SF != sf )
    {
        flash_out_pinmux_cfg(sf, cs);
    }

    // open
    sf_enable(sf, cs);

    // config
    sf_config(sf, cs, &sfCfg);

    // Flash ID
    flash_id = sf_read_id(sf, cs);
    log_debug("flash ID: %08X\n", flash_id);
    if( (( HS_SF == sf ) && (flash_id != FLASH_ID_1_INSIDE) && (flash_id !=
FLASH_ID_2_INSIDE))
        || ( ( HS_SF1 == sf ) && (flash_id != FLASH_ID_1_OUTSIDE) && (flash_id !=
FLASH_ID_2_OUTSIDE) ) )
    {
        log_debug("Error flash ID, exit!!! Please change delay\n");
        return;
    }

    // detect
    sf_detect(sf, cs);

    log_debug("erase sector \r\n");
    sf_erase(HS_SF, 0, ADDR_OFFSET, LEN_SECTOR*2);


    // Read data (no DMA)
```

```
    sf_read_fast_dma(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != 0xFF)
        {
            log_debug("Error!!! read_normal_dma id=%d, V=%02x != 0xFF----------\r\n",
j, r_buf[j]);
            log_debug_array_ex("read_normal_dma", r_buf, sizeof(r_buf));
            break;
        }
    }
    if (j == sizeof(r_buf)) log_debug("read_normal_dma over sector ok\r\n");
    memset((void*)r_buf, 0, sizeof(r_buf));

    log_debug("write 200 bytes data over sector\r\n");
    // Write data (DMA)
    sf_write(sf, cs, flashAddr, (const void*)w_buf, sizeof(w_buf));

    // Read data (DMA)
    sf_read_fast_dma(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != w_buf[j])
        {
            log_debug("Error!!! read_normal_dma id=%d, rV=%02x,
wV=%02x----------\r\n",j,r_buf[j], w_buf[j]);
            log_debug_array_ex("read_normal_dma", r_buf, sizeof(r_buf));
            break;
        }
    }
    if (j == sizeof(r_buf)) log_debug("read_normal_dma over sector ok\r\n");
    memset((void*)r_buf, 0, sizeof(r_buf));

    // Read data (DMA)
    sf_read_fast_dual_dma(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != w_buf[j])
        {
            log_debug("Error!!! read_fast_dual_dma id=%d, rV=%02x,
wV=%02x----------\r\n",j,r_buf[j], w_buf[j]);
            log_debug_array_ex("read_fast_dual_dma", r_buf, sizeof(r_buf));
            break;
        }
```

```
    }
    if (j == sizeof(r_buf)) log_debug("read_fast_dual_dma over sector ok\r\n");
    memset((void*)r_buf, 0, sizeof(r_buf));

    // Read data (DMA)
    sf_read_fast_quad_dma(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != w_buf[j])
        {
            log_debug("Error!!! read_fast_quad_dma id=%d, rV=%02x,
wV=%02x---------\r\n",j, r_buf[j], w_buf[j]);
            log_debug_array_ex("read_fast_quad_dma", r_buf, sizeof(r_buf));
            break;
        }
    }
    memset((void*)r_buf, 0, sizeof(r_buf));

    sf_erase(HS_SF, 0, ADDR_OFFSET, LEN_SECTOR*2);
    // Write data (DMA)
    sf_write(sf, cs, flashAddr, (const void*)w_buf, sizeof(w_buf));

    // Read data (DMA)
    sf_read(sf, cs, flashAddr, (uint8_t *)r_buf, sizeof(r_buf));
    for(j=0;j<sizeof(r_buf);j++)
    {
        if (r_buf[j] != w_buf[j])
        {
            log_debug("Fail !!!! Over sector read id=%d, rV=%02x,
wV=%02x---------\r\n",j, r_buf[j], w_buf[j]);
            log_debug_array_ex("sf_read", r_buf, sizeof(r_buf));
            break;
        }
    }

    if (j == sizeof(r_buf))
        log_debug("Pass ****** Over sector read\r\n");
}

void flash_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);

    switch (case_num)
```

```
    {
        case FLASH_TEST_CASE_210103:
            log_debug("FLASH_TEST_CASE_210103\n");

            // out flash access with dma in the condition of sram address over setor

            flash_cmd_dma_64m_over_sector(HS_SF1,1);
            log_debug("FLASH_TEST_CASE_210103_end\n");
            break;
        default:
            log_debug("[%s][%d][The input test number is not support!]\n",
__FUNCTION__, __LINE__);

            break;
    }
}

int main(void)
{
    uint8_t         cmd[SHELL_FIFO_SIZE];
    uint8_t*        pCmd = &cmd[0];
    volatile bool ret = false;

    // Enble all IRQ quikly.
    __set_PRIMASK(0);

    // Disable WDT.
    wdt_enable(0);

    pmu_xtal32m_x2_startup();
    co_delay_ms(100);

    // Init interactive UART.
    uart_interaction_init();
    shell_init(HS_UART1);

    uint32_t uartClk = cpm_get_clock(CPM_UART1_CLK);
    uint32_t topClk = cpm_get_clock(CPM_TOP_CLK);
    uint32_t cpuClk = cpm_get_clock(CPM_CPU_CLK);

    log_debug("BL1824X_Chip Test, topClk =%d,
cpuClk=%d %d\n",topClk,cpuClk,uartClk);

    while (1)
```

```
    {
        ret = shell_get_cmd(&pCmd);

        if (ret == true)
        {
            log_debug("\r\ncmd : %s.\n", cmd);

            flash_test_case(pCmd);
        }
    }

    shell_uninit();

    return 0;
}
```

5.4. SFLASH Controller 使用注意事项

- 在进行 flash 读写之前，请先使用 sf_detect(sf, cs)接口。
- 使用下列函数时注意 data 的首地址要四字节对齐，length 为 4 的倍数：

```
void sf_write_page_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const void
*data, uint32_t length);
void sf_read_fast_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void *data,
uint32_t length);
void sf_read_fast_dual_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void
*data, uint32_t length);
void sf_read_fast_quad_naked_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr,
void *data, uint32_t length);
void sf_read_fast_quad_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, void
*data, uint32_t length);
```

- 使用 sf_read_fast_quad_naked_dma 接口之前要使能四线 sf_quad_enable。
- 使用下列函数时注意不要擦、写到最后一个扇区(FT)：

```
void sf_erase_sector(HS_SF_Type *sf, uint32_t cs, uint32_t addr);
void sf_erase_block(HS_SF_Type *sf, uint32_t cs, uint32_t addr);
void sf_write_page_nodma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const
void *data, uint32_t length);
void sf_write_page(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const void
*data, uint32_t length);
void sf_write_page_dma(HS_SF_Type *sf, uint32_t cs, uint32_t addr, const void
*data, uint32_t length);
```

- 有些时候读到的 flash id 错误或者死机，有可能是 delay 值不匹配，调整 delay 可能会解决问题。

6. SPI

6.1. 简介

BL1824X 内部没有硬件 SPI，为了方便应用开发，使用 SFLASH Controller1 模拟硬件 SPI。因此该 SPI 和外部 flash 不可以同时使用。

该 SPI 支持模式如下：

- 仅支持 master 模式
- mode0 工作模式，即 cpol=0，cpha=0
- 轮询传输方式
- 最高 4M 传输速率

6.2. 函数介绍

6.2.1.spi_sw_pinmux_cfg

函数名	spi_sw_pinmux_cfg
函数原型	void spi_sw_pinmux_cfg(uint8_t cs_pin, uint8_t ck_pin, uint8_t si_pin, uint8_t so_pin);
功能描述	SPI 引脚定义
输入参数	cs_pin：cs 引脚 ck_pin：时钟 clk 引脚 si_pin：MOSI 引脚 so_pin：MISO 引脚
输出参数	无
返回值	无

6.2.2.spi_sw_open

函数名	spi_sw_open
函数原型	void spi_sw_open(spi_sw_mode_t mode, spi_sw_transmode_t transmode, uint32_t speed);
功能描述	SPI 初始化，并打开时钟
输入参数	mode：仅支持 master 模式，即 SPI_SW_MODE_MASTER

	Transmode：工作模式，仅支持 mode0 模式，即 SPI_SW_TRANS_MODE_0 Speed：传输速率，最大 4M
输出参数	无
返回值	无

6.2.3.spi_sw_master_exchange

函数名	spi_sw_master_exchange
函数原型	void spi_sw_master_exchange(const void *txbuf, void *rxbuf, uint32_t length)
功能描述	SPI 传输
输入参数	txbuf：tx buffer 指针 rxbuf：rx buffer 指针 length：传输长度
输出参数	无
返回值	无

6.3. 应用例程

```
#define PIN_SF_CS               7
#define PIN_SF_CK               8
#define PIN_SF_SI               9
#define PIN_SF_SO               20

void test_spi(void)
{
    CO_ALIGN(4) uint8_t txbuf[DATA_NUM] = {0};
    CO_ALIGN(4) uint8_t rxbuf[DATA_NUM] = {0};

    memset(rxbuf, 0, sizeof(rxbuf));

    spi_sw_pinmux_cfg(PIN_SF_CS, PIN_SF_CK, PIN_SF_SI, PIN_SF_SO);

    spi_sw_open(SPI_SW_MODE_MASTER, SPI_SW_TRANS_MODE_0, 1000000);


  for (uint16_t i = 0; i < DATA_NUM; i++)
  {
```

```
    txbuf[i] = i % 256;
  }

  spi_sw_master_exchange(txbuf, rxbuf, DATA_NUM);

  log_debug_array_ex("txbuf = ", txbuf, DATA_NUM);
  log_debug_array_ex("rxbuf = ", rxbuf, DATA_NUM);
}
```

6.4. 注意事项

- 使用的是 SFLASH Controller1 模拟硬件 SPI，因此该 SPI 和外部 flash 不可以同时使用。
- 该 SPI 支持模式如下：
  - 仅支持 master 模式
  - mode0 工作模式，即 cpol=0，cpha=0
  - 轮询传输方式
  - 最高 4M 传输速率

7. Timer

7.1. 简介

BL1824X 共有 3 个 timer，这 3 个 timer 使用同一个 IP。本 IP 的 timer 有三种工作模式：普通定时模式、pwm 模式以及 capture 模式。

普通定时模式是最基本的工作模式，在设定定时时长后会进入一次内部中断，用户可在中断回调函数中自定义需要完成的任务。

pwm 模式可以输出不同频率、不同占空比的方波，同时还具有输出带死区互补方波的功能。Timer0/2 共有 4 个通道，ch0~ch3 这 4 个通道均支持 pwm 功能，其中 ch0~ch2 还支持死区互补。Timer1 仅 ch0 支持 pwm 和死区互补输出功能。

capture 模式用边沿触发的方式捕捉输入的待测试信号的边沿，并能以此得到信号的频率周期等信息，仅 Timer0 支持捕获功能，其中，ch0~ch2 支持上升沿、下降沿、双边沿捕获。Ch3 仅支持上升、下降沿捕获，不支持双边沿捕获。

这 3 个 timer 对应的基地址如下：

Timer 编号	**基地址**
Timer0	0x400C0000
Timer1	0x400C0100
Timer2	0x400C0200

timer0 在代码中被宏定义为 HS_TIM0，timer1 为 HS_TIM1，timer2 为 HS_TIM2。

7.2. API 介绍

7.2.1.TIMER 寄存器结构

```
typedef struct
{
    __IO uint16_t CR1;
    uint16_t      RESERVED0;
    __IO uint16_t CR2;
    uint16_t      RESERVED1;
    __IO uint16_t SMCR;
    uint16_t      RESERVED2;
    __IO uint16_t DIER;
     uint16_t      RESERVED3;
```

```
    __IO uint16_t SR;
    uint16_t      RESERVED4;
    __IO uint16_t EGR;
    uint16_t      RESERVED5;
    __IO uint16_t CCMR1;
    uint16_t      RESERVED6;
    __IO uint16_t CCMR2;
    uint16_t      RESERVED7;
    __IO uint16_t CCER;
    uint16_t      RESERVED8;
    __IO uint32_t CNT;
    __IO uint16_t PSC;
    uint16_t      RESERVED9;
    __IO uint32_t ARR;
    __IO uint16_t RCR;
    uint16_t      RESERVED10;
    __IO uint32_t CCR[4];
    __IO uint16_t BDTR;
    uint16_t      RESERVED11;
    __IO uint16_t DCR;
    uint16_t      RESERVED12;
    __IO uint16_t DMAR;
    uint16_t      RESERVED13;
} HS_TIM_Type;

typedef struct
{
    __IO uint32_t CR1;
    __IO uint32_t CR2;
    __IO uint32_t SMCR;
    __IO uint32_t DIER;
    __IO uint32_t SR;
    __IO uint32_t EGR;
    __IO uint32_t CCMR1;
    __IO uint32_t CCER;
    __IO uint32_t CNT;
    __IO uint32_t PSC;
    __IO uint32_t ARR;
    __IO uint32_t RCR;
    __IO uint32_t CCR[1];
    __IO uint32_t BDTR;
    __IO uint32_t DCR;
    __IO uint32_t DMAR;
} HS_TIM_1_Type;
```

7.2.1.1. HS_TIM_Type 寄存器表

Offset	寄存器	描述
0x0000	TIM_CR1	timer 控制寄存器 1
0x0004	TIM_CR2	timer 控制寄存器 2
0x0008	TIM_SMCR	timer 从模式控制寄存器
0x000c	TIM_DIER	timer DMA/中断使能寄存器
0x0010	TIM_SR	timer 状态寄存器
0x0014	TIM_EGR	timer 事件产生寄存器
0x0018	TIM_CCMR1	timer 捕获/比较模式寄存器 1
0x001c	TIM_CCMR2	timer 捕获/比较模式寄存器 2
0x0020	TIM_CCER	timer 捕获/比较使能寄存器
0x0024	TIM_CNT	timer 计数器寄存器
0x0028	TIM_PSC	timer 预分频寄存器
0x002c	TIM_ARR	timer 自动重装载寄存器
0x0030	TIM_RCR	timer 重复计数寄存器
0x0034	TIM_CCR1	timer 捕获/比较寄存器 1
0x0038	TIM_CCR2	timer 捕获/比较寄存器 2
0x003c	TIM_CCR3	timer 捕获/比较寄存器 3
0x0040	TIM_CCR4	timer 捕获/比较寄存器 4
0x0044	TIM_BDTR	timer 刹车和死区寄存器
0x0048	TIM_DCR	timer DMA 控制寄存器
0x004c	TIM_DMAR	timer 全传输 DMA 地址

7.2.1.2. HS_TIM_1_Type 寄存器表

Offset	寄存器	描述
0x0000	CR1	timer 控制寄存器 1
0x0004	CR2	timer 控制寄存器 2
0x0008	SMCR	timer 从模式控制寄存器
0x000c	DIER	timer DMA/中断使能寄存器
0x0010	SR	timer 状态寄存器
0x0014	EGR	timer 事件产生寄存器
0x0018	CCMR1	timer 捕获/比较模式寄存器 1
0x001c	CCMR2	timer 捕获/比较模式寄存器 2
0x0020	CCER	timer 捕获/比较使能寄存器
0x0024	CNT	timer 计数器寄存器
0x0028	PSC	timer 预分频寄存器
0x002c	ARR	timer 自动重装载寄存器
0x0030	RCR	timer 重复计数寄存器
0x0034	CCR1	timer 捕获/比较寄存器 1
0x0044	BDTR	timer 刹车和死区寄存器

0x0048	DCR	timer DMA 控制寄存器
0x004c	DMAR	timer 全传输 DMA 地址

HS_TIM_1_Type 和 HS_TIM_Type 的区别主要是，HS_TIM_1_Type 寄存器只有 CCR1 一个捕获/比较寄存器，而 HS_TIM_Type 有四个。

TIM_CR1 address offset: 0x0000

Bit	R/W	Reset	Name	Description
31:10	N/A	0x0	N/A	保留
9:8	RW	0x0	CKD	时钟分频 该位字段表示定时器时钟 (tCK_INT) 频率与死区时间发生器和数字滤波器使用的死区时间和采样时钟 (tDTS) 之间的分频比 0x0: t_{DTS} =t_{CK_INT} 0x1: t_{DTS} =2*t_{CK_INT} 0x2: t_{DTS} =4*t_{CK_INT} 0x3: 保留
7	RW	0x0	ARPE	自动重新加载预加载 0：不缓冲 TIM_ARR 寄存器 1：缓冲 TIM_ARR 寄存器
6:5	RW	0x0	CMS	中心对齐模式选择 0x0：边缘对齐模式。 计数器根据方向位 (DIR) 向上或向下计数。 0x1：中心对齐模式 1。计数器交替上下计数。 通道的输出比较中断标志仅在计数器倒计时时设置。 0x2：居中对齐模式 2。计数器上下交替计数。 通道的输出比较中断标志仅在计数器向上计数时设置。 0x3：中心对齐模式 3。 计数器上下交替计数。 通道的输出比较中断标志在计数器向上或向下计数时都设置。 注意：只要启用计数器（CEN=1），就不允许从边缘对齐模式切换到中心对齐模式
4	RW	0x0	DIR	计数方向 0：计数器用作递增计数器 1：计数器用作递减计数器 注：当定时器配置为中心对齐模式或编码器模式时，此位为只读
3	RW	0x0	OPM	单脉冲模式 0：计数器在更新事件时不停止 1：计数器在下一次更新事件时停止计数（清零 CEN 位）
2	RW	0x0	URS	更新请求源

				该位由软件设置和清除以选择 UEV 事件源。 0：在使能的情况下，以下任何一个事件都会一个更新中断或者 DMA 请求 - 计数器向上或向下溢出 - 寄存器 TIM_EGR 的 UG 置位 - 通过 slave 模式的控制器生成更新事件 1：只有计数器向上或者向下溢出才会产生更新中断和 DMA 请求（在使能的情况下）
1	RW	0x0	UDIS	更新不使能 该位由软件置位或清除以使能/不使能 UEV 事件生成。 0：UEV 使能，更新事件（UEV）由以下情况之一生成 - 计数器向上或向下溢出 - 寄存器 TIM_EGR 的 UG 置位 - 通过 slave 模式的控制器生成更新事件 然后缓冲寄存器加载其预装载的值。 1：UEV 不使能，更新事件不产生，影子寄存器保持它们的值(ARR, PSC, CCRx) 但是如果 TIM_EGR 寄存器的 UG 置位或者收到 slave 模式控制器的硬件重置信号，计数器和预分频器将会重新初始化。
0	RW	0x0	CEN	计数器使能 0：不使能 1：使能 注：外部时钟、门控模式和编码器模式只能在软件预先设置了 CEN 位的情况下工作。然而，触发模式可以通过硬件自动设置 CEN 位

TIM_CR2 address offset: 0x0004

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7	RW	0x0	TI1S	TI1 选择 0：TIM_CH1 引脚连接到 TI1 输入 1：TIM_CH1、CH2 和 CH3 引脚连接到 TI1 输入（XOR 组合）
6:4	RW	0x0	MMS	主模式选择 这些位允许选择要在主模式下发送到从定时器以进行同步（TRGO）的信息。 组合如下：

				000：复位-来自 TIMx_EGR 寄存器的 UG 位用作触发输出（TRGO）。如果复位是由触发输入产生的（从模式控制器配置为复位模式），则 TRGO 上的信号与实际复位相比延迟。 001：启用-计数器启用信号 CNT_EN 用作触发输出（TRGO）。同时启动多个定时器或控制启用从属定时器的窗口很有用。当配置为选通模式时，计数器使能信号由 CEN 控制位和触发器输入之间的逻辑或产生。当计数器启用信号由触发器输入控制时，TRGO 上有延迟，除非选择了主/从模式（参见 TIMx_SMCR 寄存器中的 MSM 位描述）。 010：更新-选择更新事件作为触发输出（TRGO）。例如，主控 定时器然后可以用作从定时器的预分频器。 011：比较脉冲-一旦发生捕获或比较匹配，当要设置 CC1IF 标志时（即使它已经很高），触发器输出将发送一个正脉冲。（TRGO）。 100：比较-OC1REF 信号用作触发输出（TRGO） 101：比较-OC2REF 信号用作触发输出（TRGO） 110：比较-OC3REF 信号用作触发输出（TRGO） 111：比较-OC4REF 信号用作触发输出（TRGO）
3	N/A	0x0	N/A	保留
2	RW	0x0	CCUS	捕捉/比较控件更新选择 0：当预加载捕捉/比较控制位时（CCPC=1），它们仅通过设置 COMG 位来更新 1：当捕捉/比较控制位被预加载（CCPC=1）时，它们通过设置 COMG 位或在 TRGI 上出现上升沿来更新 注意：此位仅作用于具有互补输出的通道。

1	N/A	0x0	N/A	保留
0	RW	0x0	CCPC	捕捉比较预加载控件 0：未预加载 CCE、CCNE 和 OCM 位 1：预加载 CCE、CCNE 和 OCM 位 在被写入之后，它们仅在换向事件（COM）发生时才被更新（COMG 位设置或在 TRGI 上检测到的上升沿，取决于 CCUS 位）。 注：此位仅作用于具有互补输出的通道

TIM_SMCR address offset: 0x0008

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15	RW	0x0	ETP	外部触发极性 该位选择使用 ETR 还是 ETR 进行触发操作 0：ETR 同相，高电平或上升沿有效。 1：ETR 反相，低电平或下降沿有效。
14	RW	0x0	ECE	外部时钟使能。控制使能外部时钟模式 2。 0：不使能 1：使能 注： 1：设置 ECE 位与选择 TRGI 连接到 ETRF 的外部时钟模式 1 具有相同的效果（SMS=111，TS=111）。 2：可以同时使用外部时钟模式 2 和以下从属模式：复位模式、门控模式和触发模式。然而，在这种情况下，TRGI 不得连接到 ETRF（TS 比特不得为 111）。 3：如果同时启用外部时钟模式 1 和外部时钟模式 2，则外部时钟输入为 ETRF。
13:12	RW	0x0	ETPS	外部触发预分频器 外部触发信号 ETRP 频率必须至多为 TIMxCLK 频率的 1/4。可以启用预分频器以降低 ETRP 频率。当输入快速外部时钟时，它很有用。 00：预分频器关闭 01:ETRP 频率除以 2 10： ETRP 频率除以 4 11： ETRP 频率除以 8
11:8	RW	0x0	ETF	外部触发滤波器 该位字段定义用于对 ETRP 信号进行采样的频率以及 数字滤波器应用于 ETRP。数字滤波器由事件计数器构成，其中 N 个事件

				需要验证输出的转换： 0000：无过滤器，采样在 fDTS 完成 0001:fSAMPLING=fCK_INT，N=2 0010:fSAMPLING=fCK_INT，N=4 0011:fSAMPLING=fCK_INT，N=8 0100:f 采样=fDTS/2，N=6 0101:f 采样=fDTS/2，N=8 0110:f 采样=fDTS/4，N=6 0111:fSAMPLING=fDTS/4，N=8 1000:f 采样=fDTS/8，N=6 1001:f 采样=fDTS/8，N=8 1010:f 采样=fDTS/16，N=5 1011:f 采样=fDTS/16，N=6 1100:f 采样=fDTS/16，N=8 1101:f 采样=fDTS/32，N=5 1110:ff 采样=fDTS/32，N=6 1111:f 采样=fDTS/32，N=8
7	N/A	0x0	N/A	保留
6:4	RW	0x0	TS	触发器选择 该位字段选择用于同步计数器的触发输入。 000:内部触发器 0（ITR0） 001：内部触发器 1（ITR1） 010：内部触发器 2（ITR2） 011：内部触发器 3（ITR3） 100:TI1 边缘检测器（TI1F_ED） 101：滤波定时器输入 1（TI1FP1） 110：滤波定时器输入 2（TI2FP2） 111：外部触发输入（ETRF） 注意：只有在不使用这些位时（例如，当 SMS＝000 时）才能更改这些位，以避免在转换时错误的边缘检测。
3	N/A	0x0	N/A	保留
2:0	RW	0x0	SMS	从模式选择 当选择外部信号时，触发信号（TRGI）的有效边沿与外部输入上选择的极性相关联（参见输入控制寄存器和控制寄存器说明）。 000：禁用从模式-如果 CEN=“1”，则预分频器直接由内部时钟计时。 001：编码器模式 1-根据 TI1FP1 级别，计数器在 TI2FP2 边缘向上/向下计数。 010：编码器模式 2-根据 TI2FP2 级别，计数器在 TI1FP1 边缘向上/向下计数。 011:编码器模式 3-根据另一个输入的电平，

				计数器在 TI1FP1 和 TI2FP2 边缘向上/向下计数。 100：重置模式-所选触发器输入（TRGI）的上升沿重新初始化计数器并生成寄存器更新。 101：选通模式-当触发输入（TRGI）为高时，计数器时钟被启用。一旦触发器变低，计数器就会停止（但不会复位）。计数器的启动和停止均受控制。 110：触发器模式-计数器在触发器 TRGI 的上升沿开始（但不复位）。仅控制计数器的启动。 111：外部时钟模式 1-所选触发器（TRGI）的上升沿为计数器计时。 注：如果选择 TI1F_ED 作为触发输入（TS=“100”），则不得使用门控模式。实际上，TI1F_ED 为 TI1F 上的每个转变输出 1 个脉冲，而选通模式检查触发信号的电平。

TIM_DIER address offset: 0x000c

Bit	R/W	Reset	Name	Description
31:15	N/A	0x0	N/A	保留
14	RW	0x0	TDE	触发 DMA 请求使能 0：不使能 1：使能
13	RW	0x0	COMDE	COM DMA 请求启用使能 0：不使能 1：使能
12	RW	0x0	CC4DE	捕捉/比较 4 DMA 请求使能 0：不使能 1：使能
11	RW	0x0	CC3DE	捕捉/比较 3 DMA 请求使能 0：不使能 1：使能
10	RW	0x0	CC2DE	捕捉/比较 2 DMA 请求使能 0：不使能 1：使能
9	RW	0x0	CC1DE	捕捉/比较 1 DMA 请求使能 0：不使能 1：使能
8	RW	0x0	UDE	更新 DMA 请求使能 0：不使能 1：使能
7	RW	0x0	BIE	中断中断使能

				0：不使能 1：使能
6	RW	0x0	TIE	触发中断使能 0：不使能 1：使能
5	RW	0x0	COMIE	COM 中断使能 0：不使能 1：使能
4	RW	0x0	CC4IE	捕捉/比较 4 中断使能 0：不使能 1：使能
3	RW	0x0	CC3IE	捕捉/比较 3 中断使能 0：不使能 1：使能
2	RW	0x0	CC2IE	捕获/比较 2 中断使能 0：不使能 1：使能
1	RW	0x0	CC1IE	捕获/比较 1 中断使能 0：不使能 1：使能
0	RW	0x0	UIE	更新中断使能 0：不使能 1：使能

TIM_SR address offset: 0x0010

Bit	R/W	Reset	Name	Description
31:13	N/A	0x0	N/A	保留
12	RW	0x0	CC4OF	捕捉/比较 4 过捕获标志 0：未检测到过捕获 1：在 TIMx_CCR4 寄存器中，已经捕获到计数器的值，然而未检测到 CC14F 标志位 注：参考 CC1OF 说明
11	RW	0x0	CC3OF	捕捉/比较 3 过捕获标志 0：未检测到过捕获 1：在 TIMx_CCR3 寄存器中，已经捕获到计数器的值，然而未检测到 CC13F 标志位 注：参考 CC1OF 说明
10	RW	0x0	CC2OF	捕捉/比较 2 过捕获标志 0：未检测到过捕获 1：在 TIMx_CCR2 寄存器中，已经捕获到计数器的值，然而未检测到 CC2IF 标志位 注：参考 CC1OF 说明
9	RW	0x0	CC1OF	捕获/比较 1 超过捕获标志 只有在输入中配置了相应通道时，硬件才会

				设置此标志 捕获模式。软件通过将其写入“0”将其清除。 0：未检测到过度捕获。 1： 当 CC1IF 标志已设置时，计数器值已在 TIMx_CCR1 寄存器中捕获
8	N/A	0x0	N/A	保留
7	RW	0x0	BIF	中断中断标志 一旦中断输入激活，硬件就会设置此标志。如果中断输入未激活，可以通过软件清除。 0： 未发生中断事件。 1： 在中断输入上检测到激活电平。
6	RW	0x0	TIF	触发中断标志 该标志由硬件触发事件设置（当从模式控制器在除选通模式外的所有模式下启用时，在 TRGI 输入上检测到活动边缘）。当选择选通模式时，计数器启动或停止时设置该标志。该标志由软件清除。 0： 未发生触发事件。 1： 触发器中断挂起。
5	RW	0x0	COMIF	COM 中断标志 此标志由硬件 on COM 事件设置（当捕获/比较控制位 CCxE、CCxNE、OCxM 已更新时）。它由软件清除。 0： 未发生 COM 事件。 1： COM 中断挂起。
4	RW	0x0	CC4IF	捕捉/比较 4 中断标志 refer to CC1IF description 请参考 CC1IF 的描述
3	RW	0x0	CC3IF	捕捉/比较 3 中断标志 refer to CC1IF description 请参考 CC1IF 的描述
2	RW	0x0	CC2IF	捕捉/比较 2 中断标志 refer to CC1IF description 请参考 CC1IF 的描述
1	RW	0x0	CC1IF	捕获/比较 1 中断标志 如果通道 CC1 配置为输出： 当计数器与比较值匹配时，该标志由硬件设置，但在中心对齐模式下有一些例外（参考 TIMx_CR1 寄存器描述中的 CMS 位）。它由软件清除。 0:不匹配。 1： 计数器 TIMx_CNT 的内容与 TIMx_。当 TIMx_CCR1 的内容大于 TIMx_ARR 的内容时，计数器上溢（在递增计数和递增/

				递减计数模式下）或下溢（在递减计数模式中）时，CC1IF 位变高 如果通道 CC1 配置为输入： 该位由捕获上的硬件设置。通过软件或读取 TIMx_CCR1 寄存器清除。 0:未发生输入捕获 1： 计数器值已在 TIMx_CCR1 寄存器中捕获（在 IC1 上检测到与所选极性匹配的边缘）
0	RW	0x0	UIF	更新中断标志 此位由硬件在更新事件上设置。它由软件清除。 0:未发生更新。 1： 更新中断挂起。当寄存器更新时，该位由硬件设置： 关于重复计数器值的上溢或下溢（如果重复计数器=0，则更新）以及 TIMx_CR1 寄存器中的 UDIS=0。 当软件使用 TIMx_EGR 寄存器中的 UG 位重新初始化 CNT 时，如果 TIMx_CR1 寄存器中 URS=0 且 UDIS=0。 当触发事件重新初始化 CNT 时，如果 TIMx_CR1 寄存器中的 URS=0 和 UDIS=0。

TIM_EGR address offset: 0x0014

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7	W	0x0	BG	中断生成 该位由软件设置，以便生成事件，它由硬件自动清除。 0：无操作 1： 将生成中断事件。清除 MOE 位并设置 BIF 标志。如果启用，可能会发生相关中断或 DMA 传输。
6	W	0x0	TG	触发器生成 该位由软件设置，以便生成事件，它由硬件自动清除。 0：无操作 1： TIF 标志在 TIMx_SR 寄存器中设置。如果启用，可能会发生相关中断或 DMA 传输。
5	W	0x0	COMG	捕获/比较控件更新生成 该位可由软件设置，由硬件自动清除 0：无操作

				1： 设置 CCPC 位时，允许更新 CCxE、CCxNE 和 OCxM 位 注：此位仅作用于具有互补输出的通道。
4	W	0x0	CC4G	捕捉/比较 4 生成 该位由软件置位以产生事件，由硬件自动清零。 0：无动作 1：在通道 4 上产生捕获/比较事件： 注：参见 CC1G 说明
3	W	0x0	CC3G	捕捉/比较 3 生成 该位由软件置位以产生事件，由硬件自动清零。 0：无动作 1：在通道 3 上产生捕获/比较事件： 注：参见 CC1G 说明
2	W	0x0	CC2G	捕捉/比较 2 生成 该位由软件置位以产生事件，由硬件自动清零。 0：无动作 1：在通道 2 上产生捕获/比较事件。 注：参见 CC1G 说明
1	W	0x0	CC1G	捕获/比较 1 代 该位由软件设置，以便生成事件，它由硬件自动清除。 0：无操作 1： 在通道 1 上生成捕获/比较事件： 如果通道 CC1 配置为输出： 设置 CC1IF 标志，如果启用，则发送相应的中断或 DMA 请求。 如果通道 CC1 配置为输入： 计数器的当前值被捕获在 TIMx_CCR1 寄存器中。CC1IF 标志被设置， 如果启用，则发送相应的中断或 DMA 请求。如果 CC1IF 标志已高。
0	W	0x0	UG	更新生成 该位可由软件设置，由硬件自动清除。 0：无操作 1： 重新初始化计数器并生成寄存器更新。请注意，预分频器计数器也被清除（无论如何预分频率不受影响）。如果选择了中心对齐模式或 DIR=0（向上计数），则计数器将被清除，否则如果 DIR=1（向下计数），计数器将采用自动重新加载值（TIMx_ARR）。

TIM_CCMR1 address offset: 0x0018 (Output compare mode)

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15	RW	0x0	OC2CE	输出比较 2 清除使能 0：不使能 1：使能
14:12	RW	0x0	OC2M	输出比较 2 模式
11	RW	0x0	OC2PE	输出比较 2 预加载使能 0：不使能 1：使能
10	N/A	0x0	N/A	保留
9:8	RW	0x0	CC2S	捕获/比较 2 选择 该位字段定义通道的方向（输入/输出）以及使用的输入。 0x0：CC2 信道配置为输出。 0x1：CC2 信道配置为输入，IC2 映射到 TI2 上。 0x2：CC2 信道配置为输入，IC2 映射到 TI1 上。 0x3：CC2 信道配置为输入，IC2 映射到 TRC 上。只有当通过 TS bit（TIMx_SMCR 寄存器）选择内部触发器输入时，这种模式才有效。 注意：CC2S 位仅在通道关闭时才可写入（在 TIMx_CCER 中 CC2E=“0”）。
7	RW	0x0	OC1CE	输出比较 1 清除使能 0：不使能. OC1Ref 不受 ETRF 输入的影响 1：使能. 一旦在 ETRF 输入上检测到高电平，OC1Ref 即被清除
6:4	RW	0x0	OC1M	输出比较 1 模式 这些位定义输出参考信号 OC1REF 的行为导出 OC1N。OC1REF 为有效高电平，而 OC1 和 OC1N 的有效电平取决于 CC1P 和 CC1NP 位。 000:冻结-输出比较寄存器 TIMx_CCR1 与计数器 TIMx_。（此模式用于生成定时基准）。 001：匹配时将通道 1 设置为活动电平。当计数器 TIMx_ CNT 与捕获/比较寄存器 1（TIMx_CCR1）匹配。 010：在匹配时将通道 1 设置为非活动级别。当计数器 TIMx_ CNT 与捕获/比较寄存器 1

				（TIMx_CCR1）匹配时。 011：切换-当 TIMx_CNT=TIMx_CCR1 时，OC1REF 切换。 100：强制非激活电平-OC1REF 强制为低电平。 101：强制激活电平-OC1REF 强制为高电平。 110:PWM 模式 1-在递增计数中，只要 TIMx_CNT<TIMx-CCR1，通道 1 就处于活动状态。在递减计数中，只要 TIMx_CNT>TIMx-CCR1 处于活动状态（OC1REF='1'），通道 1 就处于非活动状态（OC1REF='0'）。 111:PWM 模式 2-在递增计数中，只要 TIMx_CNT＜TIMx-CCR1，通道 1 就不活动。在递减计数中，只要 TIMx_CNT>TIMx_CCR1 处于非活动状态，通道 1 就处于活动状态。 注 1：只要 LOCK 级别 3 已编程（TIMx_BDTR 寄存器中的 LOCK 位）且 CC1S=“00”（通道在输出中配置），这些位就不能修改。 注 2：在 PWM 模式 1 或 2 中，OCREF 电平仅在比较结果发生变化或输出比较模式从“冻结”模式切换到“PWM”模式时发生变化。 注 3：在具有互补输出的通道上，此位字段是预加载的。如果在 TIMx_CR2 寄存器中设置了 CCPC 位，则只有在生成 COM 事件时，OC1M 活动位才从预加载位中获取新值。
3	RW	0x0	OC1PE	输出比较 1 预加载启用 0：TIMx_CCR1 上的预加载寄存器已禁用。TIMx_CCR1 可以随时写入，新值将立即考虑在内。 1：TIMx_CCR1 上的预加载寄存器已启用。读/写操作访问预加载寄存器。TIMx_CCR1 预加载值在每次更新事件时加载到活动寄存器中。 注 1：只要 LOCK 级别 3 已编程（TIMx_BDTR 寄存器中的 LOCK 位）且 CC1S=“00”（通道在输出中配置），这些位就不能修改。 注 2：仅在一个脉冲模式（TIMx_CR1 寄存

				器中设置的 OPM 位）下，可以使用 PWM 模式，而无需验证预加载寄存器。否则，行为无法保证。
2	N/A	0x0	N/A	保留
1:0	RW	0x0	CC1S	捕获/比较 1 选择 该位字段定义通道的方向（输入/输出）以及使用的输入。 00:CC1 通道配置为输出 01:CC1 通道配置为输入，IC1 映射到 TI1 10： CC1 通道配置为输入，IC1 映射到 TI2 11： CC1 信道被配置为输入，IC1 被映射到 TRC 上。仅当通过 TS 位（TIMx_SMCR 寄存器）选择内部触发输入时，此模式才有效 注：CC1S 位仅在通道关闭时才可写（TIMx_CCER 中的 CC1E=“0”）。

TIM_CCMR1 address offset: 0x0018 (Input capture mode)

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:12	RW	0x0	IC2F	输入捕获 2 滤波器
11:10	RW	0x0	IC2PSC	输入捕捉 2 预分频器
9:8	RW	0x0	CC2S	捕获/比较 2 选择 该位字段定义通道的方向（输入/输出）以及使用的输入。 00:CC2 通道配置为输出 01:CC2 通道配置为输入，IC2 映射到 TI2 上 10： CC2 通道配置为输入，IC2 映射到 TI1 11： CC2 信道被配置为输入，IC2 被映射到 TRC 上。此模式仅在 通过 TS 位（TIMx_SMCR 寄存器）选择内部触发输入 注：CC2S 位仅在通道关闭时才可写（TIMx_CCER 中 CC2E=“0”）。
7:4	RW	0x0	IC1F	输入捕获 1 过滤器 该位字段定义了用于对 TI1 输入进行采样的频率和应用的数字滤波器的长度 数字滤波器由事件计数器构成，其中需要 N 个事件来验证输出上的转换： 0000：无过滤器，采样在 fDTS 完成 0001:fSAMPLING=fCK_INT，N=2 0010:fSAMPLING=fCK_INT，N=4 0011:fSAMPLING=fCK_INT，N=8

				0100:f 采样=fDTS/2，N=6 0101:f 采样=fDTS/2，N=8 0110:f 采样=fDTS/4，N=6 0111:fSAMPLING=fDTS/4，N=8 1000:f 采样=fDTS/8，N=6 1001:f 采样=fDTS/8，N=8 1010:f 采样=fDTS/16，N=5 1011:f 采样=fDTS/16，N=6 1100:f 采样=fDTS/16，N=8 1101:f 采样=fDTS/32，N=5 1110:f 采样=fDTS/32，N=6 1111:f 采样=fDTS/32，N=8
3:2	RW	0x0	IC1PSC	输入捕获 1 预分频器 该位字段定义作用于 CC1 输入（IC1）的预分频器的比率。一旦 CC1E=“0”（TIMx_CCER 寄存器），预分频器就会复位。 00：无预分频器，每次在捕获输入上检测到边缘时都进行捕获 01：每 2 个事件捕获一次 10： 每 4 个事件捕获一次 11： 捕获每 8 个事件进行一次
1:0	RW	0x0	CC1S	捕获/比较 1 选择 该位字段定义通道的方向（输入/输出）以及使用的输入。 00:CC1 通道配置为输出 01:CC1 通道配置为输入，IC1 映射到 TI1 10： CC1 通道配置为输入，IC1 映射到 TI2 11： CC1 信道被配置为输入，IC1 被映射到 TRC 上。此模式仅在 通过 TS 位（TIMx_SMCR 寄存器）选择内部触发输入 注：CC1S 位仅在通道关闭时才可写（TIMx_CCER 中的 CC1E=“0”

TIM_CCMR2 address offset: 0x001c (Output compare mode)

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15	RW	0x0	OC4CE	输出比较 4 清除使能 0：不使能 1：使能
14:12	RW	0x0	OC4M	输出比较 4 模式
11	RW	0x0	OC4PE	输出比较 4 预加载使能 0：不使能

				1：使能
10	N/A	0x0	N/A	保留
9:8	RW	0x0	CC4S	捕获/比较 4 选择 该位字段定义通道的方向（输入/输出）以及使用的输入。 0x0:CC4 信道配置为输出。 0x1:CC4 信道配置为输入，IC4 映射到 TI4 上。 0x2：CC4 信道配置为输入，IC4 映射到 TI3 上。 0x3：CC4 信道配置为输入，IC4 映射到 TRC 上。仅当通过 TS 位（TIMx_SMCR 寄存器）选择内部触发器输入时，此模式才工作 **注意**：CC4S 位仅在通道关闭时才可写入（在 TIMx_CCER 中 CC4E=“0”）。
7	RW	0x0	OC3CE	输出比较 3 清除使能 0：不使能 1：使能
6:4	RW	0x0	OC3M	输出比较 3 模式
3	RW	0x0	OC3PE	输出比较 3 预加载使能 0：不使能 1：使能
2	N/A	0x0	N/A	保留
1:0	RW	0x0	CC3S	捕捉/比较 3 选择 该位字段定义通道的方向（输入/输出）以及使用的输入。 0x0:CC3 信道配置为输出。 0x1:CC3 信道配置为输入，IC3 映射到 TI3 上。 0x2： CC3 信道配置为输入，IC3 映射到 TI4 上。 0x3： CC3 信道配置为输入，IC3 映射到 TRC 上。仅当通过 TS 位（TIMx_SMCR 寄存器）选择内部触发器输入时，此模式才工作 **注意**：CC3S 位仅在通道关闭时才可写入（在 TIMx_CCER 中 CC3E=“0”）。

TIM_CCMR2 address offset: 0x001c (Input capture mode)

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:12	RW	0x0	IC4F	输入比较 4 滤波器
11:10	RW	0x0	IC4PSC	输入捕捉 4 预分频器
9:8	RW	0x0	CC4S	捕捉/比较 4 选择

				该位字段定义通道的方向（输入/输出）以及使用的输入。 0x0:CC4 信道配置为输出。 0x1:CC4 信道配置为输入，IC4 映射到 TI4 上。 0x2： CC4 信道配置为输入，IC4 映射到 TI3 上。 0x3： CC4 信道配置为输入，IC4 映射到 TRC 上。仅当通过 TS 位（TIMx_SMCR 寄存器）选择内部触发器输入时，此模式才工作 **注意**：CC4S 位仅在通道关闭时才可写入（在 TIMx_CCER 中 CC4E=“0”）。
7:4	RW	0x0	IC3F	输入比较 3 滤波器
3:2	RW	0x0	IC3PSC	输入捕捉 3 预分频器
1:0	RW	0x0	CC3S	捕捉/比较 3 选择 **Note**: CC3S bits are writable only when the channel is OFF (CC3E = ‘0’ in TIMx_CCER). 该位字段定义通道的方向（输入/输出）以及使用的输入。 0x0:CC3 信道配置为输出。 0x1:CC3 信道配置为输入，IC3 映射到 TI3 上。 0x2： CC3 信道配置为输入，IC3 映射到 TI4 上。 0x3： CC3 信道配置为输入，IC3 映射到 TRC 上。仅当通过 TS 位（TIMx_SMCR 寄存器）选择内部触发器输入时，此模式才工作 **注意**：CC3S 位仅在通道关闭时才可写入（在 TIMx_CCER 中 CC3E=“0”）。

TIM_CCER address offset: 0x0020

Bit	R/W	Reset	Name	Description
31:14	N/A	0x0	N/A	保留
13	RW	0x0	CC4P	捕捉/比较 4 输出极性 参考 CC1P 的描述
12	RW	0x0	CC4E	捕捉/比较 4 输出使能 参考 CC1E 的描述
11	RW	0x0	CC3NP	捕捉/比较 3 互补输出极性 参考 CC1NP 的描述
10	RW	0x0	CC3NE	捕捉/比较 3 互补输出使能 参考 CC1NE 的描述

9	RW	0x0	CC3P	捕捉/比较 3 输出极性 参考 CC1P 的描述
8	RW	0x0	CC3E	捕捉/比较 3 输出使能 参考 CC1E 的描述
7	RW	0x0	CC2NP	捕捉/比较 2 互补输出极性 参考 CC1NP 的描述
6	RW	0x0	CC2NE	捕捉/比较 2 互补输出使能 参考 CC1NE 的描述
5	RW	0x0	CC2P	捕捉/比较 2 输出极性 参考 CC1P 的描述
4	RW	0x0	CC2E	捕捉/比较 2 输出使能 参考 CC1E 的描述
3	RW	0x0	CC1NP	捕获/比较 1 互补输出极性 CC1 通道配置为输出： 0:OC1N 激活高。 1： OC1N 激活低。 CC1 通道配置为输入： 该位与 CC1P 一起使用，以定义 TI1FP1 和 TI2FP1 的极性。请参阅 CC1P 说明。 注意：在具有互补输出的通道上，此位是预加载的。如果在 TIMx_CR2 寄存器中设置了 CCPC 位，则 CC1NP 活动位仅在产生换向事件时才从预加载位中获取新值。 注：一旦编程了 LOCK 级别 2 或 3（TIMx_BDTR 寄存器中的 LOCK 位）且 CC1S=“00”（通道在输出中配置），该位就不可写。
2	RW	0x0	CC1NE	捕获/比较 1 互补输出启用 0：关闭-OC1N 未激活。OC1N 电平是 MOE、OSSI、OSSR、OIS1、OIS 1N 和 CC1E 位的函数。 1： 根据 MOE、OSSI、OSSR、OIS1、OIS 1N 和 CC1E 位，在相应的输出引脚上输出-OC1N 信号。 注意：在具有互补输出的通道上，此位是预加载的。如果在 TIMx_CR2 寄存器中设置了 CCPC 位，则只有在产生换向事件时，CC1NE 活动位才从预加载位中获取新值。
1	RW	0x0	CC1P	捕获/比较 1 输出极性 CC1 通道配置为输出： 0:OC1 激活高 1： OC1 激活低 CC1 通道配置为输入： CC1NP/CC1P 位选择 TI1FP1 和 TI2FP1

				的有效极性用于触发或捕获操作。 00：非反转/上升沿 电路对 TIxFP1 上升沿敏感（复位、外部时钟或触发模式下的捕获或触发操作），TIxFP2 不反转（门控模式或编码器模式下的触发操作）。 01：倒置/下降边缘 该电路对 TIxFP1 下降沿（复位、外部时钟或触发模式下的捕获或触发操作）敏感，TIxFP2 反转（门控模式或编码器模式下的触发操作）。 10： 保留，请勿使用此配置。 11： 非反转/两边 该电路对 TIxFP1 上升沿和下降沿都敏感（复位、外部时钟或触发模式下的捕获或触发操作），TIxFP2 不反转（门控模式下的触发操作）。此配置不得用于编码器模式。 注意：在具有互补输出的通道上，此位是预加载的。如果在 TIMx_CR2 寄存器中设置了 CCPC 位，则只有在产生换向事件时，CC1P 活动位才从预加载位中获取新值。 注：一旦编程了 LOCK 级别 2 或 3（TIMx_BDTR 寄存器中的 LOCK 位），该位就不可写。
0	RW	0x0	CC1E	捕获/比较 1 输出启用 CC1 通道配置为输出： 0：关闭-OC1 未激活。OC1 电平是 MOE、OSSI、OSSR、OIS1、OIS 1N 和 CC1NE 位的函数。 1： 根据 MOE、OSSI、OSSR、OIS1、OIS 1N 和 CC1NE 位，在相应的输出引脚上输出-OC1 信号。 CC1 通道配置为输入： 该位确定计数器值的捕获是否可以实际执行到输入捕获/比较寄存器 1（TIMx_CCR1）中。 0:已禁用捕获。 1： 已启用捕获。 注意：在具有互补输出的通道上，此位是预加载的。如果在 TIMx_CR2 寄存器中设置了 CCPC 位，则只有在产生换向事件时，CC1E 活动位才从预加载位中获取新值。

TIM_CNT address offset: 0x0024

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:0	RW	0x0	CNT	计数器的值

TIM_PSC address offset: 0x0028

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:0	RW	0x0	PSC	预分频器值 计数器时钟频率（CK_CNT）等于 f CK_PSC/（PSC[15:0]+1）。 PSC 包含在每次更新事件（包括在配置为“重置模式”时通过 TIMx_EGR 寄存器的 UG 位或通过触发器控制器清除计数器时）将加载到活动预分频器寄存器中的值

TIM_ARR address offset: 0x002c

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:0	RW	0x0	ARR	预加载的值 ARR 是要加载到实际自动重新加载寄存器中的值。 当自动重新加载值为空时，计数器被阻止。

TIM_RCR address offset: 0x0030

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	保留
7:0	RW	0x0	REP	重复计数器值 这些位允许用户在预加载寄存器启用时设置比较寄存器的更新速率（即，从预加载到活动寄存器的周期性传输），以及如果该中断启用，则设置更新中断生成速率。 每次与 REP_CNT 相关的递减计数器达到零时，都会生成一个更新事件，并从 REP 值重新开始计数。由于 REP_CNT 仅在重复更新事件 U_RC 时用 REP 值重新加载，因此在下一次重复更新事件之前不会考虑对 TIMx_RCR 寄存器的任何写入。 这意味着在 PWM 模式下（REP+1）对应于： 边缘对齐模式下的 PWM 周期数 中心对齐模式下的半 PWM 周期数

TIM_CCR1 address offset: 0x0034

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:0	RW	0x0	CCR1	捕捉/比较 1 的值 如果通道 CC1 配置为输出： CCR1 是要加载到实际捕获/比较 1 寄存器中的值（预加载值）。 如果在 TIMx_CCMR1 寄存器（位 OC1PE）中未选择预加载功能，则将永久加载。否则，当发生更新事件时，预加载值将复制到活动捕获/比较 1 寄存器中。 活动捕获/比较寄存器包含要与计数器 TIMx_CNT 进行比较的值，并在 OC1 输出上发出信号。 如果通道 CC1 配置为输入： CCR1 是由最后一个输入捕获 1 事件（IC1）传输的计数器值

TIM_CCR2 address offset: 0x0038

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:0	RW	0x0	CCR2	捕捉/比较 2 的值 如果通道 CC2 配置为输出： CCR2 是要加载到实际捕获/比较 2 寄存器中的值（预加载值）。 如果在 TIMx_CCMR2 寄存器（位 OC2PE）中未选择预加载功能，则将永久加载。否则，当发生更新事件时，预加载值将复制到活动捕获/比较 2 寄存器中。 活动捕获/比较寄存器包含要与计数器 TIMx_CNT 比较的值，并在 OC2 输出上发出信号。 如果通道 CC2 配置为输入： CCR2 是由最后一个输入捕获 2 事件（IC2）传送的计数器值。

TIM_CCR3 address offset: 0x003c

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:0	RW	0x0	CCR3	捕获/比较 3 值 如果通道 CC3 配置为输出： CCR3 是要加载到实际捕获/比较 3 寄存器中的值（预加载值）。

				如果在 TIMx_CCMR3 寄存器(位 OC3PE)中未选择预加载功能，则将永久加载。否则，当发生更新事件时，预加载值将复制到活动捕获/比较 3 寄存器中。 活动捕获/比较寄存器包含要与计数器 TIMx_CNT 比较的值，并在 OC3 输出上发出信号。 如果信道 CC3 配置为输入： CCR3 是由最后一个输入捕获 3 事件(IC3)传送的计数器值。

TIM_CCR4 address offset: 0x0040

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:0	RW	0x0	CCR4	捕获/比较 4 值 如果通道 CC4 配置为输出： CCR4 是要加载到实际捕获/比较 4 寄存器中的值（预加载值）。 如果在 TIMx_CCMR4 寄存器(位 OC4PE)中未选择预加载功能，则将永久加载。否则，当发生更新事件时，预加载值将复制到活动捕获/比较 4 寄存器中。 活动捕获/比较寄存器包含要与计数器 TIMx_CNT 比较的值，并在 OC4 输出上发出信号。 如果通道 CC4 配置为输入： CCR4 是由最后一个输入捕获 4 事件(IC4)传送的计数器值。

TIM_BDTR address offset: 0x0044

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15	RW	0x0	MOE	主输出使能 一旦刹车输入激活，硬件就会异步清除该位。它由软件设置或根据 AOE 位自动设置。它仅作用于输出中配置的通道。 0：OC 和 OCN 输出不使能或强制为空闲状态。 1：如果设置了 OC 和 OCN 输出各自的使能位（TIMx_CCER 寄存器中的 CCxE、CCxNE），则会启用 OC 和 OCN 输出。有关更多详细信息，请参阅 OC/OCN 启用说明
14	RW	0x0	AOE	自动输出使能

				0:MOE 只能由软件置位 1： MOE 可以通过软件置位，也可以在下次更新事件时自动置位（如果中断输入未激活） 注：只要 LOCK 级别 1 已编程（TIMx_BDTR 寄存器中的 LOCK 位），就不能修改该位。
13	RW	0x0	BKP	刹车极性 0：刹车输入 BRK 低电平有效 1： 刹车输入 BRK 处于高电平 注：只要 LOCK 级别 1 已编程（TIMx_BDTR 寄存器中的 LOCK 位），就不能修改该位。 注：对该位的任何写入操作都需要延迟 1 个 APB 时钟周期才能生效。
12	RW	0x0	BKE	刹车使能 0：刹车输入（BRK 和 CCS 时钟故障事件）不使能 1：刹车输入（BRK 和 CCS 时钟故障事件）使能 注：当 LOCK 级别 1 已编程（TIMx_BDTR 寄存器中的 LOCK 位）时，不能修改该位。 注：对该位的任何写入操作都需要延迟 1 个 APB 时钟周期才能生效。
11	RW	0x0	OSSR	运行模式的关闭状态选择 当 MOE=1 时，在具有配置为输出的互补输出的信道上使用该位。如果计时器中没有实现补充输出，则不会实现 OSSR。 有关更多详细信息，请参阅 OC/OCN 启用说明 0：处于非活动状态时，OC/OCN 输出被禁用（OC/OCN-启用输出信号=0）。 1： 当处于非活动状态时，只要 CCxE=1 或 CCxNE=1，OC/OCN 输出就会以非活动级别启用。然后，OC/OCN 启用输出信号=1 注：一旦 LOCK 级别 2 编程完毕（TIMx_BDTR 寄存器中的 LOCK 位），该位就不能修改。
10	RW	0x0	OSSI	空闲模式的关闭状态选择 当配置为输出的通道上的 MOE=0 时，使用该位。有关更多详细信息，请参阅 OC/OCN 启用说明。 0：处于非活动状态时，OC/OCN 输出被禁用（OC/OCN-启用输出信号=0）。 1： 当处于非活动状态时，一旦 CCxE=1

				或 CCxNE=1，OC/OCN 输出首先以其空闲水平强制。OC/OCN-启用输出信号=1） 注：一旦 LOCK 级别 2 编程完成（TIM_BDTR 寄存器中的 LOCK 位），该位就不能修改
9:8	RW	0x0	LOCK	锁配置 这些位提供针对软件错误的写入保护。 0x0：LOCK OFF（锁定关闭）-没有位被写保护。 0x1:LOCK Level 1= 无法再写入 TIMx_BDTR 寄存器中的 DTG 位、TIMx_CR2 寄存器中的 OISx 和 OISxN 位以及 TIMx-BDTR 注册表中的 BKE/BKP/AOE 位。 0x2：LOCK Level 2=LOCK Level1+CC Polarity 位（TIMx_CCER 寄存器中的 CCxP/CCxNP 位，只要通过 CCxS 位在输出中配置相关通道）以及 OSSR 和 OSSI 位都不能再写入。 0x3：LOCK Level 3=LOCK Level2+CC 控制位（TIMx_CCMRx 寄存器中的 OCxM 和 OCxPE 位，只要通过 CCxS 位在输出中配置了相关通道）不能再被写入。 注：重置后，LOCK 位只能写入一次。一旦写入 TIMx_BDTR 寄存器，其内容将被冻结，直到下次复位。
7:0	RW	0x0	DTG	死区发生器设置 这个位域用于设置在两个互补输出之间延时时间，即死区。DT 代表这个死区的时间。 当 DTG[7:5] = 0xx 时，DT 的计算公式为：DT=DTG[7.0] * Tdtg，Tdtg=tDTS。 当 DTG[7:5] = 10x 时，DT 的计算公式为：DT=(64+DTG[5:0]) * Tdtg，Tdtg=2 * tDTS。 当 DTG[7:5]=110 时，DT 的计算公式为：DT=(32+DTG[4:0]) * Tdtg，Tdtg=8 * tDTS。 当 DTG[7:5]=111 时，DT 的计算公式为：DT=(32+DTG[4:0]) * Tdtg，Tdtg=16 * tDTS。

				上述公式中 tDTS 是死区生成器的采样时钟，tDTS 在寄存器 TIM_CR1[9:8]处配置。详见寄存器 TIM_CR1（0x0000）的描述。 举例说明：如果 DTG[7:0] = 0xFF，那么就属于 DTG[7:5]=111 的情况。此时 DTG[4:0]=0x1F=31，则 DT=(32 + 31) * Tdtg 假如 TIM_CR1[9:8]=0x0， tDTS = tCK_INT=1/32M，那么 DT= (32 + 31) *16*(1/32M) = 31.5us. 注：只要 LOCK 级别 1、2 或 3 已编程，该位字段就不能修改（TIM_BDTR 寄存器中的 LOCK 位）。

TIM_DCR address offset: 0x0048

Bit	R/W	Reset	Name	Description
31:13	N/A	0x0	N/A	保留
12:8	RW	0x0	DBL	DMA Burst 传输长度 这个 5 位值定义了 DMA 传输的次数（当执行对 TIMx_ DMAR 寄存器地址的读或写访问时定时器检测突发传输）。 TIMx_DMAR 地址） 00000:1 次传输 00001:2 次传输 00010:3 次传输 ... 10001:18 次传输
7:5	N/A	0x0	N/A	保留
4:0	RW	0x0	DBA	DMA 基地址 这个5位数值定义了DMA传输的基本地址（当通过 TIMx_DMAR 地址进行读/写访问时）。DBA 定义为从 CR1 寄存器的地址。 00000:TIMx_ CR1， 00001:TIMx_ CR2， 00010:TIMx_SMCR，

TIM_DMAR address offset: 0x004c

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	保留
15:0	RW	0x0	DMAB	DMA Burst 传输控制寄存器： 对 DMAR 寄存器的读或写操作访问位于地

				址处的寄存器 （TIMx_CR1 地址）+（DBA+DMA 索引）x 4 其中，TIMx_CR1 地址是控制寄存器 1 的地址，DBA 是在 TIMx_DCR 寄存器中配置的 DMA 基地址，DMA 索引由 DMA 传输自动控制，范围从 0 到 DBL（在 TIMx_DCR 中配置的 DBL）。

7.2.2.变量参数

7.2.2.1. tim_dma_burstlen_t

```
/// TIM DMA
typedef enum
{
    TIM_DMA_BURST_LEN_1UNIT = 0,
    TIM_DMA_BURST_LEN_2UNIT,
    TIM_DMA_BURST_LEN_3UNIT,
    TIM_DMA_BURST_LEN_4UNIT,
    TIM_DMA_BURST_LEN_5UNIT,
    TIM_DMA_BURST_LEN_6UNIT,
    TIM_DMA_BURST_LEN_7UNIT,
    TIM_DMA_BURST_LEN_8UNIT,
    TIM_DMA_BURST_LEN_9UNIT,
    TIM_DMA_BURST_LEN_10UNIT,
    TIM_DMA_BURST_LEN_11UNIT,
    TIM_DMA_BURST_LEN_12UNIT,
    TIM_DMA_BURST_LEN_13UNIT,
    TIM_DMA_BURST_LEN_14UNIT,
    TIM_DMA_BURST_LEN_15UNIT,
    TIM_DMA_BURST_LEN_16UNIT,
    TIM_DMA_BURST_LEN_17UNIT,
    TIM_DMA_BURST_LEN_18UNIT,

    TIM_DMA_BURST_LEN_RESERVED,

} tim_dma_burstlen_t;
```

TIM 的 DMA 突发传输次数，TIM_DMA_BURST_LEN_1UNIT 为突发传输 1 次。

7.2.2.2. tim_mode_t

```
/// TIM mode
typedef enum
{
    /// Work as timer mode
    TIM_TIMER_MODE,
    /// Work as PWM mode
    TIM_PWM_MODE,
    /// Work as CAP mode
     TIM_CAP_MODE,
}tim_mode_t;
```

TIM 模式选择。timer 有 3 种工作模式：普通定时模式、PWM 模式以及 capture 模式。

7.2.2.3. tim_pwm_pol_t

```
/// PWM polarity
typedef enum
{
    TIM_PWM_POL_HIGH2LOW,
    TIM_PWM_POL_LOW2HIGH,
}tim_pwm_pol_t;
```

TIM_PWM 的极性选项。TIM_PWM_POL_HIGH2LOW 配置 PWM 的波形为先产生高电平，后低电平。TIM_PWM_POL_LOW2HIGH 配置 PWM 的波形为先产生低电平，后高电平。

7.2.2.4. tim_pwm_force_level_t

```
/// PWM force output level
typedef enum
{
    TIM_PWM_FORCE_LOW      = 4,
    TIM_PWM_FORCE_HIGH     = 5,
    TIM_PWM_FORCE_DISABLE = 6,
}tim_pwm_force_level_t;
```

TIM_PWM 强制输出电平选择。TIM_PWM_FORCE_LOW：强制输出低电平；TIM_PWM_FORCE_HIGH：强制输出高电平；TIM_PWM_FORCE_DISABLE：不强制输出。

7.2.2.5. tim_pwm_channel_t

```
/// PWM channels
typedef enum
{
    TIM_PWM_CHANNEL_0,
    TIM_PWM_CHANNEL_1,
    TIM_PWM_CHANNEL_2,
    TIM_PWM_CHANNEL_3,
```

```
    TIM_PWM_CHANNEL_NUM,
}tim_pwm_channel_t;
```

TIM_PWM 通道号的选择。

7.2.2.6. tim_cnt_mode_t

```
typedef enum
{
  UP_COUNT    = 0,
  DOWN_COUNT = 1,
} tim_cnt_mode_t;
```

TIM 计数模式选择。

7.2.2.7. tim_clock_div_ t

```
typedef enum
{
  TIM_CLK_DIV_0          = 0,
  TIM_CLK_DIV_1          = 1,
  TIM_CLK_DIV_2          = 2,
  TIM_CLK_DIV_RESERVED = 3,
} tim_clock_div_t;
```

TIM 的时钟分频选择。

7.2.2.8. tim_align_mode_t

```
typedef enum
{
  EDGE_ALIGNED_MODE       = 0,
  CENTER_ALIGNED_MODE_1 = 1,
  CENTER_ALIGNED_MODE_2 = 2,
  CENTER_ALIGNED_MODE_3 = 3,
} tim_align_mode_t;
```

TIM 的对齐模式选择。

7.2.2.9. tim_timer_config_ t

```
/// timer configuration
typedef struct
{
    /// delay or period, unit is us
    uint32_t period_us;
    /// event callback
    tim_timer_callback_t callback;
}tim_timer_config_t;
```

timer 工作在普通定时模式时的参数配置结构体。

period_us：定时周期；

callback：回调函数；

7.2.2.10. tim_1_timer_config_ t

```
/// timer configuration
typedef struct
{
    /// delay or period, unit is us
    uint32_t period_us;
    /// event callback
    tim_1_callback_t callback;
}tim_1_timer_config_t;
```

timer 工作在普通定时模式时的参数配置结构体。

period_us：定时周期；

callback：回调函数；

7.2.2.11. tim_pwm_channel_config_t

```
/// PWM channel configuration
typedef struct
{
    /// PWM polarity
    tim_pwm_pol_t pol;
    /// pulse counter value
    uint16_t pulse_count;
    //complementary_output_enable
    bool complementary_output_enable;

}tim_pwm_channel_config_t;
```

PWM 的通道配置

pol：PWM 极性选择

pulse_count：用于配置占空比

complementary_output_enable：互补功能使能

7.2.2.12. tim_pwm_config_t

```
/// PWM configuration
typedef struct
{
    /// Frequency for every count
    uint32_t count_freq;
    /// Period counter
    uint16_t period_count;

    // dead_time generate step
    uint16_t dead_time;
    /// Channel configuration
    struct
    {
```

```
        /// channel enable
        bool enable;
        /// channel configuration
        tim_pwm_channel_config_t config;
    }channel[TIM_PWM_CHANNEL_NUM];
    /// Event callback
    tim_pwm_callback_t callback;
}tim_pwm_config_t;
```

TIM 的 PWM 配置参数。

count_freq：计数时钟频率；

period_count：计数器的自动重载值；

dead_time：死区时间；

channel[TIM_PWM_CHANNEL_NUM]：输出通道的具体配置；

callback：回调函数。

7.2.2.13. tim_1_pwm_config_t

```
/// PWM configuration
typedef struct
{
    /// Frequency for every count
    uint32_t count_freq;
    /// Period counter
    uint16_t period_count;

    // dead_time generate step
    uint16_t dead_time;

    /// Channel configuration
    struct
    {
        /// channel enable
        bool enable;
        /// channel configuration
        tim_pwm_channel_config_t config;
    }channel[TIM_PWM_CHANNEL_NUM];
    /// Event callback
    tim_1_callback_t callback;
}tim_1_pwm_config_t;
```

TIM1 的 PWM 配置参数。

count_freq：计数时钟频率；

period_count：计数器的自动重载值；

dead_time：死区时间；

channel[TIM_PWM_CHANNEL_NUM]：输出通道的具体配置；

callback：回调函数。

7.2.2.14. ir_tim_dma_config_t

```
typedef struct
{
    bool              use_fifo;     /**< fifo buffer */
    uint32_t         *buffer;       /**< DMA fifo buffer */
    uint32_t          buffer_len;   /**< DMA buffer len */
    dma_llip_t       *block_llip;   /**< block llip */
    uint32_t          block_num; /**< block num */
    __IO uint32_t *tim_addr;    /**< Absolute address of the TIM register to access */
    uint32_t          num;      /**< number of registers under updating //DMA burst length */
    uint32_t          req;          /**< TIM DMA request source: TIM_DIER_UDE,
                                        TIM_DIER_CC1DE, ..., TIM_DIER_TDE */
    dma_callback_t callback;     /**<   DMA event callback */
} ir_tim_dma_config_t;
```

TIM 的 ir_tim_dma 配置参数。

use_fifo：fifo 缓存器；

buffer：DMA fifo 缓存器；

buffer_len：DMA 缓存器长度；

block_llip：DMA 链表块；

block_num：回调函数；

Tim_addr：TIM 寄存器绝对地址；

Num：更新寄存器数量；

Req：DMA 请求源；

callback：DMA 事件回调函数。

7.2.2.15. cap_mode_t

```
typedef enum
{
    CAP_MODE_NONE     = 0,
    CAP_MODE_FALLING = 1,
    CAP_MODE_RISING   = 2,
    CAP_MODE_BOTH     = 3,
}cap_mode_t;
```

timer 工作于 capture 模式时的边沿捕获/比较模式。

CAP_MODE_NONE：普通捕获；

CAP_MODE_FALLING：下降沿捕获；

CAP_MODE_RISING：上升沿捕获；

CAP_MODE_BOTH：双边沿捕获。

7.2.2.16. cap_prescale_t

```
typedef enum
{
    CAP_PRESCALE_NONE = 0,
    CAP_PRESCALE_2    = 1,
    CAP_PRESCALE_4    = 2,
    CAP_PRESCALE_8    = 3,
} cap_prescale_t;
```

timer 工作于 capture 模式时的预分频系数选择。

7.2.2.17. cap_filter_t

```
typedef enum
{
    CAP_FILTER_NONE = 0,
    CAP_FILTER_1    = 1,
    CAP_FILTER_2    = 2,
    CAP_FILTER_3    = 3,
    CAP_FILTER_4    = 4,
    CAP_FILTER_5    = 5,
    CAP_FILTER_6    = 6,
    CAP_FILTER_7    = 7,
    CAP_FILTER_8    = 8,
    CAP_FILTER_9    = 9,
    CAP_FILTER_10   = 10,
    CAP_FILTER_11   = 11,
    CAP_FILTER_12   = 12,
    CAP_FILTER_13   = 13,
    CAP_FILTER_14   = 14,
    CAP_FILTER_15   = 15,
} cap_filter_t;
```

timer 工作于 capture 模式时的滤波器选择。

7.2.2.18. cap_map_t

```
typedef enum
{
```

```
    CAP_MAP_TI1 = 0,
    CAP_MAP_TI2 = 1,
    CAP_MAP_TI3 = 2,
    CAP_MAP_TI4 = 3,
} cap_map_t;
```

7.2.2.19. CAPChannelConfig

```
typedef struct
{
    cap_mode_t       cap_mode;
    cap_map_t        cap_map;
    cap_prescale_t   prescale;
    cap_filter_t     filter;
    tim_cap_callback_t   callback;
}CAPChannelConfig;
```

timer capture 通道的配置结构体。

7.2.2.20. tim_cap_config_t

```
/// timer configuration
typedef struct
{
    uint16_t          psc;
    uint16_t          arr;
    tim_cnt_mode_t    cnt_mode;
    tim_clock_div_t clk_div;
    uint16_t          repeat_cnt;
    tim_timer_callback_t      callback;

    CAPChannelConfig channels[4];
    uint16_t          dier;    /*CCxOF enable or not*/
}tim_cap_config_t;
```

timer 工作于 capture 模式时的参数结构体。

7.2.2.21. tim_config_t

```
/// TIM configuration
typedef struct
{
    /// TIM mode
    tim_mode_t mode;
    /// TIM config
    union
    {
        /// Use as timer
        tim_timer_config_t timer;
        /// Use as PWM
        tim_pwm_config_t pwm;
        /// Use as CAP
        tim_cap_config_t cap;
    }config;
}tim_config_t;
```

TIM 的配置。

Mode：模式选择；

Config：timer、pwm、cap 配置选择。

7.2.2.22. tim_1_config_t

```
/// TIM configuration
typedef struct
{
    /// TIM mode
    tim_mode_t mode;
    /// TIM config
    union
    {
        /// Use as timer
        tim_timer_config_t timer;
        /// Use as PWM
        tim_pwm_config_t pwm;
        /// Use as CAP
        tim_cap_config_t cap;
    }config;
}tim_1_config_t;
```

TIM 的配置。

Mode：模式选择；

Config：timer、pwm、cap 配置选择。

7.2.2.23. tim_dma_config_t

```
/// TIM DMA config
typedef struct
{
    /// dma direction: true for mem to tim, false for tim to mem.
    bool mem_to_tim;
    /// Absolute address of the TIM register to access
    __IO uint32_t *tim_addr;
    /// address of the memory buffer
    uint32_t *mem_addr;
    /// length of the DMA transfer in bytes
    uint32_t len_in_bytes;
    /// TIM DMA request source: TIM_DIER_UDE, TIM_DIER_CC1DE, ...,
     TIM_DIER_TDE
    uint32_t req;
    /// DMA block transfer to build.
    dma_block_t *block;
} tim_dma_config_t;
```

tim_dma 配置。

mem_to_tim：传输方向；

tim_addr：tim 地址；

mem_addr：mem 地址；

len_in_bytes：每次传输字节大小；

req：DMA 请求源；

block：dma 传输相关配置。

7.2.3.库函数

7.2.3.1. tim_init

函数名	tim_init
函数原型	void tim_init(void)
功能描述	TIM 的初始化
输入参数	无
输出参数	无
返回值	无

7.2.3.2. tim_config

函数名	tim_config
函数原型	void tim_config(HS_TIM_Type *tim, const tim_config_t *config)
功能描述	TIM 的初始化配置
输入参数 1	tim:TIM0 或 TIM2 的初始化配置

输入参数 2	Config：定时器配置结构体
输出参数	无
返回值	无

7.2.3.3. tim_1_config

函数名	tim_config
函数原型	void tim_1_config(HS_TIM_Type *tim, const tim_1_config_t *config)
功能描述	TIM1 的专用初始化配置函数
输入参数 1	tim：选择定时器(HS_TIM1)
输入参数 2	Config：定时器配置结构体
输出参数	无
返回值	无

7.2.3.4. tim_pwm_change_period_count

函数名	tim_pwm_change_period_count
函数原型	void tim_pwm_change_period_count(HS_TIM_Type *tim, uint16_t period_count);
功能描述	PWM 模式下改变 pwm 波形周期
输入参数 1	Tim：指向 HS_TIM_Type 类型的指针
输入参数 2	period_count：uint16_t 类型的数值，写 ARR 寄存器
输出参数	无
返回值	无

7.2.3.5. tim_pwm_channel_change_pulse_count

函数名	tim_pwm_channel_change_pulse_count
函数原型	void tim_pwm_channel_change_pulse_count(HS_TIM_Type *tim, tim_pwm_channel_t channel, uint16_t pulse_count)
功能描述	PWM 一次脉冲数值配置
输入参数 1	tim：指向 HS_TIM_Type 类型的指针
输入参数 2	channel：指向 tim_pwm_channel_t 类型的指针
输入参数 3	pulse_count：uint16_t 类型的数值
输出参数	无
返回值	无

7.2.3.6. tim_pwm_channel_force_output

函数名	tim_pwm_channel_force_output
函数原型	tim_pwm_channel_force_output(HS_TIM_Type *tim, tim_pwm_channel_t channel, tim_pwm_force_level_t level)

功能描述	PWM 强制输出高低电平选择
输入参数 1	tim：指向 HS_TIM_Type 类型的指针
输入参数 2	channel：tim_pwm_channel_t 类型里面的数值
输入参数 3	level：tim_pwm_force_level_t 类型的数值
输出参数	无
返回值	无

7.2.3.7. tim_start

函数名	tim_start
函数原型	void tim_start(HS_TIM_Type *tim)
功能描述	tim 配置完成，开始工作，计数器开始计数。
输入参数	tim：定时器选择(可选 HS_TIM0、HS_TIM2)
输出参数	无
返回值	无

7.2.3.8. tim_1_start

函数名	tim_1_start
函数原型	void tim_start(HS_TIM_1_Type *tim)
功能描述	HS_TIM1 开始工作，计数器开始计时
输入参数	tim：定时器选择(可选 HS_TIM1)
输出参数	无
返回值	无

7.2.3.9. tim_stop

函数名	tim_stop
函数原型	void tim_stop(HS_TIM_Type *tim);
功能描述	停止定时器 tim 工作
输入参数	tim：定时器选择(可选 HS_TIM0、HS_TIM2)
输出参数	无
返回值	无

7.2.3.10. tim_1_stop

函数名	tim_1_stop
函数原型	void tim_1_stop(HS_TIM_1_Type *tim);
功能描述	tim 定时器停止工作，计数器停止计数
输入参数	tim：定时器选择(可选 HS_TIM1)
输出参数	无
返回值	无

7.2.3.11. tim_dma_config

函数名	tim_dma_config
函数原型	void tim_dma_config(HS_TIM_Type *tim, const tim_dma_config_t *config)
功能描述	Tim DMA 传输的初始化设置
输入参数 1	tim：定时器选择(可选 HS_TIM0、HS_TIM2)
输入参数 2	config：指向 const tim_dma_config_t 类型的指针
输出参数	无
返回值	无

7.2.3.12. tim_1_dma_config

函数名	tim_1_dma_config
函数原型	void tim_1_dma_config(HS_TIM_1_Type *tim, const tim_dma_config_t *config)
功能描述	Tim DMA 传输的初始化设置
输入参数 1	tim：定时器选择(可选 HS_TIM1)
输入参数 2	config：指向 const tim_dma_config_t 类型的指针
输出参数	无
返回值	无

7.3. Timer 应用例程

7.3.1.普通定时模式

软件实现：

```
static void tim_timer_handler(HS_TIM_Type *tim)
{
    if( (HS_TIM_Type*)HS_TIM0 == (HS_TIM_Type*)tim )
    {
        log_debug("tick tim0\n");
    } else if( HS_TIM2 == tim )
    {
        log_debug("tick tim2\n");
    }
}
static void tim_timer1_handler(HS_TIM_1_Type *tim)
{
    if( HS_TIM1 == tim ) {
        log_debug("tick tim1\n");
```

```
    }
}
static void example_normal_common(HS_TIM_Type *tim)
{
    const tim_config_t timer_cfg = {
        .mode = TIM_TIMER_MODE,
        .config.timer = {
            .period_us = 1000*1000,
            .callback = tim_timer_handler,
        },
    };
    tim_config(tim, &timer_cfg);
    tim_start(tim);
    while(1);
    //tim_stop(tim);
}
static void example_normal_timer1(HS_TIM_1_Type *tim)
{
    const tim_1_config_t timer_cfg = {
        .mode = TIM_TIMER_MODE,
        .config.timer = {
            .period_us = 1000*1000,
            .callback = tim_timer1_handler,
        },
    };
    tim_1_config(tim, &timer_cfg);
    tim_1_start(tim);
    while(1);
    //tim_1_stop(tim);
}

void timer_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);
    switch (case_num)
switch (case_num)
    {
        //timer0  定时器模式
        case TIMER_TEST_CASE_150000:
            example_normal_common(HS_TIM0);//定时器 0 计数
            break;
        //timer1  定时器模式
        case TIMER_TEST_CASE_150400:
            example_normal_timer1(HS_TIM1); //定时器 1 计数
```

```
            break;
        //timer2  定时器模式
        case TIMER_TEST_CASE_150700:
            example_normal_common(HS_TIM2);//定时器 2 计数
            break;
    }
}
```

在 example_normal_common()中，通过参数 period_us 设置定时周期，通过 callback 设置定时回调函数，每隔 period_us（微秒）便进入一次中断回调函数 tim_timer_handler()。

7.3.2.PWM 模式

软件实现：

```
#define TIM_STRUCT2INDEX(tim)    ((((uint32_t)tim) - HS_TIM0_BASE) /
(HS_TIM1_BASE - HS_TIM0_BASE))
#define PWM_PERIOD_COUNT 10
//HS_TIM0 and HS_TIM2 PWM mode test case
void example_pwm(HS_TIM_Type *tim, uint8_t channel, bool bUse_complementary,
uint32_t freq, double duty)
{
    uint8_t timer_index = TIM_STRUCT2INDEX(tim);
    uint8_t index = TIM_STRUCT2INDEX(tim);
        volatile uint8_t pulse_cnt = 0;
        pulse_cnt = (uint8_t)(duty*PWM_PERIOD_COUNT);
        switch(timer_index){
            case 0:
                pinmux_config(8,PINMUX_TIMER0_IO_0_CFG+channel);
                pinmux_config(12,PINMUX_TIMER0_TOGGLE_N_0_CFG+channel);
                break;
            case 2:
                pinmux_config(8,PINMUX_TIMER2_IO_0_CFG+channel);
                pinmux_config(12,PINMUX_TIMER2_TOGGLE_N_0_CFG+channel);
                break;
            default:
                break;
        }
    tim_config_t pwm_cfg = {
        .mode = TIM_PWM_MODE,
        .config.pwm = {
            /* pwm_frequency = count_freq/period_count   */
            /* t = (psc + 1)*(arr + 1)/clock      psc = (clock/count_freq) - 1      arr =
period_count - 1 */
            .count_freq = PWM_PERIOD_COUNT*freq,
```

```
            .period_count = PWM_PERIOD_COUNT,
            .dead_time = (uint16_t)0x003,
            .channel = {
                {false/*enable*/, {TIM_PWM_POL_HIGH2LOW, pulse_cnt, false}},
                {false/*enable*/, {TIM_PWM_POL_HIGH2LOW, pulse_cnt, false}},
                {false/*enable*/, {TIM_PWM_POL_HIGH2LOW, pulse_cnt, false}},
                {false/*enable*/, {TIM_PWM_POL_HIGH2LOW, pulse_cnt, false}},
            },
            .callback = NULL,
        },
    };
    if (bTest_intr_flag) {
        pwm_cfg.config.pwm.callback = test_all_intr_flag_handler;
    }

    pwm_cfg.config.pwm.channel[channel].enable = true;
    pwm_cfg.config.pwm.channel[channel].config.complementary_output_enable =
bUse_complementary;
    tim_config(tim, &pwm_cfg);
    tim_start(tim);
}

//HS_TIM1 PWM mode test case
#define TIMER1_PERIOD_COUNT 10
void example_pwm_timer1(HS_TIM_1_Type *tim, uint8_t channel, bool
bUse_complementary, uint32_t freq, double duty)
{
        co_assert((HS_TIM1 == tim) );
        uint16_t pulse_count = 0;
        pulse_count = TIMER1_PERIOD_COUNT * duty;
        pinmux_config(8,PINMUX_TIMER1_IO_0_CFG+channel);
        if(bUse_complementary)
            pinmux_config(12,PINMUX_TIMER1_TOGGLE_N_0_CFG+channel);
    tim_1_config_t pwm_cfg =
    {
        .mode = TIM_PWM_MODE,
        .config.pwm =
        {
            /* pwm_frequency = count_freq/period_count   */
            /*
             t = (psc + 1)*(arr + 1)/clock
             psc = (clock/count_freq) - 1
              arr = period_count - 1 */
            .count_freq = TIMER1_PERIOD_COUNT * freq,
```

```
            .period_count = TIMER1_PERIOD_COUNT,
            .dead_time = (uint16_t)0x0003,
            .channel =
            {
                /* duty cycle = pulse_count/period_count */
                {false/*enable*/, {TIM_PWM_POL_HIGH2LOW, pulse_count, false}},
                {false/*enable*/, {TIM_PWM_POL_HIGH2LOW, pulse_count, false}},
                {false/*enable*/, {TIM_PWM_POL_HIGH2LOW, pulse_count, false}},
                {false/*enable*/, {TIM_PWM_POL_HIGH2LOW, pulse_count, false}},
            },
            .callback = NULL,
        },
    };
    pwm_cfg.config.pwm.channel[channel].enable = true;
    pwm_cfg.config.pwm.channel[channel].config.complementary_output_enable =
bUse_complementary;

    /* PWM Init */
    tim_1_config(HS_TIM1, &pwm_cfg);
    tim_1_start(HS_TIM1);
}
```

BL1824X 有三个定时器，其中 TIM0、TIM2 有四个通道支持 PWM，其中 ch0、ch1 和 ch2 支持 PWM 且同时支持死区和互补，ch3 仅支持 PWM。而 TIM1 只有一个通道 CH0 支持 PWM，且同时支持死区和互补输出。

```
void example_pwm(
                    HS_TIM_Type *tim,          \\选择定时器 HS_TIM0 或 HS_TIM2
                    uint8_t channel,           \\工作通道选择
                    bool bUse_complementary, \\是否使能互补输出
                    uint32_t freq,             \\PWM 输出频率
                    double duty);              \\PWM 占空比设置(0~1)
```

7.3.3.Capture 模式

软件实现：

```
#define BUF_DIF_MAXSIZE 101
#define CCR_MAX 0xFFFF
uint32_t cnt_dif[BUF_DIF_MAXSIZE];
static volatile uint32_t cnt_dif_index = 0;
static uint32_t pre_ccr = 0;
volatile uint8_t cap_flag_done = 0;
//calculate input signal frequency and print
```

```
void example_cap_printfreq(uint32_t psc)
{
        float freq_buf[BUF_DIF_MAXSIZE] = { 0 };
        uint32_t j = 0;
        double freq_temp = 0;
        while(1){
            if(cap_flag_done == 1)
            {
                for(j = 1; j<BUF_DIF_MAXSIZE; j++)
                {
                    freq_temp = (32/psc)*1000*1000/cnt_dif[j];
                    freq_buf[j-1] = (float)freq_temp;
                }
                log_debug("\r\n");
                log_debug("cnt_dif: \n");
                for(j = 0; j<BUF_DIF_MAXSIZE-1; j++)
                {
                    log_debug("%.6d ",(uint32_t)cnt_dif[j]);
                }
                log_debug("\r\n");
                log_debug("TIMER0 cap freq: \n");
                for(j = 0; j<BUF_DIF_MAXSIZE-1; j++)
                {
                    log_debug("%.6d ",(uint32_t)freq_buf[j]);
                }
                cap_flag_done = 0;
            }
        }
}

//CCx capture evevt callback function
static void cap_callback(HS_TIM_Type *tim, uint32_t sr_value)
{
         if(cnt_dif_index<BUF_DIF_MAXSIZE)
        {
                cnt_dif[cnt_dif_index++] = (CCR_MAX + HS_TIM0->CCR[0] -
           pre_ccr)%CCR_MAX;
                pre_ccr = HS_TIM0->CCR[0];
                if(cnt_dif_index == BUF_DIF_MAXSIZE) cap_flag_done = 1;
        }
}
//HS_TIM0 Capture mode test case
static void example_cap(HS_TIM_Type *tim,   uint8_t channel,   cap_mode_t
cap_mode, uint32_t psc, tim_cap_callback_t   callback)
```

```
{
    co_assert(HS_TIM0 == tim);
    co_assert(channel >=0 && channel <= 3);
    co_assert(CAP_MODE_FALLING == cap_mode || CAP_MODE_RISING ==
cap_mode || CAP_MODE_BOTH == cap_mode);
    pinmux_config(8, PINMUX_TIMER0_IO_0_CFG);
    tim_config_t cap_cfg = {
        .mode = TIM_CAP_MODE,
        .config.cap = {
            .psc = psc - 1,
            .arr = 0xFFFF,
            .cnt_mode = UP_COUNT,
            .clk_div = TIM_CLK_DIV_0,
            .repeat_cnt = 0,
            .callback = NULL,
            .channels = {
                {CAP_MODE_NONE,   CAP_MAP_TI1, CAP_PRESCALE_NONE,
             CAP_FILTER_NONE, callback},
                {CAP_MODE_NONE,   CAP_MAP_TI2, CAP_PRESCALE_NONE,
             CAP_FILTER_NONE, callback},
                {CAP_MODE_NONE,   CAP_MAP_TI3, CAP_PRESCALE_NONE,
             CAP_FILTER_NONE, callback},
                {CAP_MODE_NONE,   CAP_MAP_TI4, CAP_PRESCALE_NONE,
             CAP_FILTER_NONE, callback},
            },
            .dier = 0,
        },
    };
    cap_cfg.config.cap.channels[channel].cap_mode = cap_mode;
    if (bTest_intr_flag) {
        cap_cfg.config.cap.channels[channel].callback = test_all_intr_flag_handler;
    }
    tim_config(tim, &cap_cfg);
    tim_start(tim);
        while(1)
        {
            if(cap_flag_done == 1)
            {
                    example_cap_printfreq(psc);
                    cap_flag_done = 0;
            }
        }
}
```

BL1824X 只有 TIM0 有 capture 功能，而 TIM1 和 TIM2 没有 capture 功能。且 TIM0

CH0~CH2 三个通道均支持上升、下降、双边沿捕获，CH3 仅支持上升、下降沿捕获。

函数参数说明：

```
static void example_cap(
    HS_TIM_Type *tim,              //定时器选择，仅可选 HS_TIM0
    uint8_t channel,               //捕获通道选择,ch0~ch3
    cap_mode_t cap_mode,           //捕获边沿选择，可选上升沿、下降沿、双边沿
    uint32_t psc,                  //设置计数时钟分频值，clk_cnt = 32M/(psc - 1)
    tim_cap_callback_t  callback)  //捕获回调函数
```

上述代码实现，使用 IO8 作为 PWM 信号输入引脚，通过定时器 HS_TIM0 测试输入信号频率例程，并通过串口打印测试结果。

函数 example_cap()首先做输入参数校验，接着复用定时器输入捕获通道到 IO8，之后通过 cap_cfg 结构体对捕获通道配置参数，配置完成后启动计数器开始捕获。

在捕获回调函数 cap_callback()中，采集 BUF_DIF_MAXSIZE 宏定义数量的捕获寄存器值，之后置位 cap_flag_done 软件标志位。

example_cap_printfreq()函数将采样到的原始捕获寄存器值转换为频率，通过串口打印出来，注意此函数传入参数 psc 需与 example_cap()函数入参 psc 值保持同步，否则计算频率将可能出错。

7.3.4.DMA 辅助功能应用

软件实现：

```
#define PWM_PERIOD_COUNT 10
static void example_dma_modify_pwm_freq_duty(HS_TIM_Type *timer)
{
    uint8_t  index       = TIM_STRUCT2INDEX(timer);
    const uint32_t original_freq = 1000;
    const uint32_t modified_freq = original_freq * 5;
    const double original_duty  = 0.5;
    const double modified_duty = 0.2;
    uint32_t count_freq      = modified_freq * PWM_PERIOD_COUNT;
    uint32_t clock = 32*1000*1000;
    uint32_t psc             = (clock / count_freq) - 1;
    uint32_t ccr             = (uint32_t)(modified_duty*PWM_PERIOD_COUNT);

    if(HS_TIM0 == timer || HS_TIM2 == timer)
    {
        example_pwm(timer, 0, false, original_freq, original_duty);
        log_debug("original pwm freq is %d Hz\n", original_freq);
        log_debug("10s later, pwm freq will be %d Hz\n", modified_freq);
```

```
            co_delay_ms(10*1000);

            tim_dma_config_t psc_config =
            {
                .mem_to_tim          = true,
                .tim_addr            = (uint32_t *)&timer->PSC,
                .mem_addr            = (uint32_t *)&psc,
                .len_in_bytes        =  sizeof(psc),
                .req                 =  TIM_DIER_UDE|TIM_DIER_TDE,
                .block               =  &block[0],
            };
            tim_dma_config_t ccr_config =
            {
                .mem_to_tim          = true,
                .tim_addr            = (uint32_t *)&timer->CCR[0],
                .mem_addr            = (uint32_t *)&ccr,
                .len_in_bytes    = sizeof(ccr),
                .req                      =  TIM_DIER_UDE|TIM_DIER_TDE,
                .block                =  &block[1],
            };
            dma_init();
            //start alter psc register by dma
            tim_dma_config(timer,&psc_config);
            HS_DMA_CH_Type *psc_dma_ch = dma_allocate();
            dma_start_transfer(psc_dma_ch,&block[0],NULL);
            dma_wait_stop(psc_dma_ch);
            dma_release(psc_dma_ch);

            //start alter psc register by dma
            tim_dma_config(timer,&ccr_config);
            HS_DMA_CH_Type *ccr_dma_ch = dma_allocate();
            dma_start_transfer(ccr_dma_ch,&block[1],NULL);
            dma_wait_stop(ccr_dma_ch);
            dma_release(ccr_dma_ch);
        }else{
            /* TIMER1 modified PWM freq and duty at same time test program */
            HS_TIM_1_Type *tim1 = (HS_TIM_1_Type *)HS_TIM1;

            example_pwm_timer1(tim1, 0, false, original_freq,
original_duty);                    log_debug("original pwm freq is %d Hz\n", original_freq);
            log_debug("10s later, pwm freq will be %d Hz\n", modified_freq);
            co_delay_ms(10*1000);

            tim_dma_config_t psc_config =
```

```
        {
            .mem_to_tim          = true,
            .tim_addr             = (uint32_t *)&tim1->PSC,
            .mem_addr             = (uint32_t *)&psc,
            .len_in_bytes         =  sizeof(psc),
            .req                  =  TIM_DIER_UDE|TIM_DIER_TDE,
            .block                =  &block[0],
        };
        tim_dma_config_t ccr_config =
        {
            .mem_to_tim          = true,
            .tim_addr             = (uint32_t *)&tim1->CCR[0],
            .mem_addr             = (uint32_t *)&ccr,
            .len_in_bytes         = sizeof(ccr),
            .req                  =  TIM_DIER_UDE|TIM_DIER_TDE,
            .block                =  &block[1],
        };

        dma_init();
        //start alter psc register by dma
        tim_1_dma_config(tim1,&psc_config);
        HS_DMA_CH_Type *psc_dma_ch = dma_allocate();
        dma_start_transfer(psc_dma_ch,&block[0],NULL);
        dma_wait_stop(psc_dma_ch);
        dma_release(psc_dma_ch);

        //start alter psc register by dma
        tim_1_dma_config(tim1,&ccr_config);
        HS_DMA_CH_Type *ccr_dma_ch = dma_allocate();
        dma_start_transfer(ccr_dma_ch,&block[1],NULL);
        dma_wait_stop(ccr_dma_ch);
        dma_release(ccr_dma_ch);

    }
    log_debug("\ncheck if pwm freq has been changed into %d \n" , modified_freq);
    log_debug("\ncheck if pwm freq duty cycle has been changed into %.2f%%
\n" ,  modified_duty*100);
}
```

上述代码，举例列举使用 DMA 方式修改定时器 PWM 输出频率和占空比的一个例子。首先配置定时器引脚 IO8 复用定时器输出通道，输出 1kHz，占空比 50%的 pwm 波，10s 延时后，通过 DMA 方式修改定时器的 PSC 和 CCR 寄存器值，将频率修改为 5kHz，占空比 20%，可通过串口查看调试信息。

example_dma_modify_pwm_freq_duty()函数，首先计算 5kHz，占空比 20%的 PSC

和 CCR 寄存器目标设定值，再调用 example_pwm()库函数输出 1kHz PWM 波。此后延时 10s，再用 DMA 方式更新 PSC 和 CCR 寄存器以修改频率和占空比。

定时器 DMA 功能的使用，分 3 步：

1. 通过 tim_dma_config_t 结构体变量和 tim_dma_config，初始化 DMA 传输方向、传输字长配置信息、传输目的地址等参数。

2. 使用 dma_allocate()分配传输通道。

3. 使用 dma_start_transfer()开启传输。

7.4. Timer 使用注意事项

- timer0/2 均有 4 个通道：ch0~ch3, 这 4 个通道均支持 pwm 功能，ch0~ch2 还支持互补死区输出。timer1 仅:ch0 通道支持 PWM 和死区互补输出。
- timer 常规 PWM 输出，最高速率 3.17MHz。
- timer0 的 ch0~ch3 均支持 capture 功能，ch0~ch2 支持上升沿、下降沿以及双边沿捕获。ch3 支持上升沿、下降沿捕获，但是不支持双边沿捕获。timer1 与 timer2 不支持 capture 功能。
- **通道功能说明表：**

BL1824Xtimer 通道说明			
外设名	**功能**	**通道**	**补充说明**
HS_TIM0	PWM 输出	CH0~CH3	
	互补输出	CH0~CH2	
	死区输出	CH0~CH2	
	capture 模式	CH0~CH3	CH3 不支持双边沿捕获，其余通道支持
HS_TIM1	PWM 输出	CH0	
	互补输出	CH0	
	死区输出	CH0	
	capture 模式	无	不支持
HS_TIM2	PWM 输出	CH0~CH3	
	互补输出	CH0~CH2	
	死区输出	CH0~CH2	
	capture 模式	无	不支持

- timer0 和 timer2 在 bsp 库中使用 HS_TIM_Type 类型结构体描述寄存器结构，timer1 使用 HS_TIM_1_Type 描述寄存器结构。3 个 timer 的驱动方式类似，只是通道数量和通道所支持的功能存在差异，timer1 因为寄存器结构和 timer0、timer2 有差异，单独使用一套库函数，例如 tim_1_config()。

8. WDT

8.1. 简介

WDT（Watchdog Timer），即看门狗计时器，通常是一个可以在一定时间内重置系统的计数器。当看门狗启动时，计数器开始自动计数，一段时间后，如果没有重新加载，计数器达到零将生成重置信号，以重新启动系统。实际上，它是为了监视主程序的运行。当系统崩溃时，WDT 计数器将达到零，并将生成重置信号以重置系统。当然，为了防止在系统正常运行时，由于看门狗计数器达到零而导致看门狗中断，从而导致意外的系统复位，当使用看门狗时，将在一定的时间段内执行看门狗喂食操作。也就是说，看门狗计数器在看门狗计数降至零之前设置了一个新的重新加载值。如果看门狗降至零，则认为程序工作不正常，整个系统被迫复位。

WDT 的输入时钟，是 32K 的 256 分频，即一个 clk = 30.72*256 = 7.8ms。

8.2. API 介绍

8.2.1.WDT 寄存器结构

```
/// HS_PMU_Type
typedef struct
{
    __IO uint32_t BASIC;              // offset:0x00
    __IO uint32_t PSO_PM;             // offset:0x04
    __IO uint32_t XTAL32M_CNS0;       // offset:0x08
    __IO uint32_t REMOVED01;          // offset:0x0C
    __IO uint32_t RAM_PM_1;           // offset:0x10
    __IO uint32_t RAM_PM_2;           // offset:0x14
    __IO uint32_t ANA_PD;             // offset:0x18
    __IO uint32_t GPIO_POL;           // offset:0x1C
    __IO uint32_t GPIO_POL_1;         // offset:0x20
    __IO uint32_t MISC_CTRL;          // offset:0x24
    __IO uint32_t WAKE_DEB;           // offset:0x28
    __IO uint32_t GPIO_OE_CTRL;       // offset:0x2C
    __IO uint32_t GPIO_OE_CTRL_1;     // offset:0x30
    __IO uint32_t GPIO_PU_CTRL;       // offset:0x34
    __IO uint32_t XTAL32M_CNS1;       // offset:0x38
```

```
__IO uint32_t AHB_REMAP;            // offset:0x3C
__IO uint32_t GPIO_ODA_CTRL;        // offset:0x40
__IO uint32_t XTAL32M_CNS2;         // offset:0x44
__IO uint32_t ANA_REG;              // offset:0x48
__IO uint32_t CLK_CTRL_1;           // offset:0x4C
__IO uint32_t CLK_CTRL_2;           // offset:0x50
__IO uint32_t GPIO_IE_CTRL;         // offset:0x54
__IO uint32_t GPIO_IE_CTRL_1;       // offset:0x58
__IO uint32_t COMP_CTRL;            // offset:0x5C
__IO uint32_t ANA_PD_1;             // offset:0x60
__IO uint32_t MISC_CTRL_1;          // offset:0x64
__IO uint32_t RAM_CTRL_1;           // offset:0x68
__IO uint32_t RAM_CTRL_2;           // offset:0x6C
__IO uint32_t RAM_CTRL_3;           // offset:0x70
__IO uint32_t RAM_CTRL_4;           // offset:0x74
__IO uint32_t RAM_CTRL_5;           // offset:0x78
__IO uint32_t TIMER_VAL;            // offset:0x7C
__IO uint32_t RW_EXT_WAKEUP;        // offset:0x80
__IO uint32_t OSC_INT_CTRL;         // offset:0x84
__IO uint32_t GPIO_STATUS_READ;     // offset:0x88
__IO uint32_t REMOVED0;             // offset:0x8C
__IO uint32_t GPIO_STATUS_READ_2; // offset:0x90
__IO uint32_t WDT_STATUS;           // offset:0x94
__IO uint32_t GPIO_WAKEUP;          // offset:0x98
__IO uint32_t GPIO_WAKEUP_1;        // offset:0x9C
__IO uint32_t GPIO_ODE_CTRL;        // offset:0xA0
__IO uint32_t GPIO_ODE_CTRL_1;      // offset:0xA4
__IO uint32_t GPIO_PD_CTRL;         // offset:0xA8
__IO uint32_t REMOVED2;             // offset:0xAC
__IO uint32_t GPIO_LATCH;           // offset:0xB0
__IO uint32_t GPIO_LATCH_1;         // offset:0xB4
__IO uint32_t GPIO_NOCLK_LATCH;     // offset:0xB8
__IO uint32_t WDT_RLR_CFG;          // offset:0xBC
__IO uint32_t STATUS_READ;          // offset:0xC0
__IO uint32_t GPIO_DRV_CTRL_0;      // offset:0xC4
__IO uint32_t GPIO_DRV_CTRL_1;      // offset:0xC8
__IO uint32_t GPIO_DRV_CTRL_2;      // offset:0xCC
__IO uint32_t GPIO_DRV_CTRL_3;      // offset:0xD0
__IO uint32_t TIMER_CNT;            // offset:0xD4
__IO uint32_t BB_DEEPSDUR;          // offset:0xD8
__IO uint32_t BB_DEEPSLTIME;        // offset:0xDC
__IO uint32_t BB_FINECNT;           // offset:0xE0
__IO uint32_t BB_CLKN;              // offset:0xE4
__IO uint32_t WDT_KR_CFG;           // offset:0xE8
```

```
    __IO uint32_t SW_STATUS;            // offset:0xEC
    __IO uint32_t CPU_STATUS;           // offset:0xF0
    __IO uint32_t LDR_HOOK;             // offset:0xF4
    __IO uint32_t BOOT_SEL;             // offset:0xF8
    __IO uint32_t ANA_CON;              // offset:0xFC
    __IO uint32_t FLASH_LOW_VOL_CTRL_0; // offset:0x0100
    __IO uint32_t FLASH_LOW_VOL_CTRL_1; // offset:0x0104
    __IO uint32_t RC_32M_CTRL;          // offset:0x0108
    __IO uint32_t TIMER_SET;            // offset:0x010C
         uint32_t RESERVE2[0x0140/4 - 0x010C/4 - 1];
    __IO uint32_t PA_GAIN_REG[18];      // offset:0x0140
}HS_PMU_Type;
```

注： WDT 只使用到了 PMU_Type 的 WDT_RLR_CFG、WDT_KR_CFG、WDT_STATUS，故以下重点介绍这三个寄存器。

Offset	寄存器	描述
0x00bc	WDT_RLR_CFG	重置周期寄存器
0x0094	WDT_STATUS	WDT 状态寄存器
0x00e8	WDT_KR_CFG	WDT 键值寄存器

WDT_RLR_CFG address offset: 0x00 bc

Bit	R/W	Reset	Name	Description
11:0	RW	0xfff	WDT_RLR_CFG	复位周期寄存器

WDT_STATUS address offset: 0x0094

Bit	R/W	Reset	Name	Description
27:16	R	0xfff	WDT_TIMER	watchdog 计数器当前值，，当看门狗启动时，计数器从 0xfff 开始递减计数，当计数器计数到末尾 0x000 时，会产生一个复位信号。
10	R	0x0	WDT_FLAG_CLR_STATUS_SYNC	wdt_flag 清除状态同步标志
9	R	0x0	WDT_FLAG_CLR	wdt_flag 清除状态
8	R	0x0	WDT_FLAG_CLR_COMB	wdt_flag 清除状态
4	R	0x0	WDT_FLAG	watchdog 发生复位的标志，wdt_flag=1 表示发生过看门狗复位，向 WDT_STATUS 写任意值清除 wdt_flag。
2	R	0x0	LD_WDT_KR_SYNC	Ld_wdt_kr 同步标志位
1	R	0x0	LD_WDT_KR	向 WDT_KR_REG 寄存器执行写操作时为 1，自清 0
0	R	0x0	LD_WDT_KR	key 寄存器配置状态 0：未配置

				1: 已配置

WDT_KR_CFG address offset: 0x00e8

Bit	R/W	Reset	Name	Description
31:16	N/A	N/A	保留	保留
15:0	RW	0x0	WDT_KR	watchdog 键值寄存器，当写入 0x6666 值时会关闭看门狗；写入 0x5555 表示允许访问 WDT_RLR_CFG 和 WDT_STATUS 寄存器；写入 0xAAAA 时，WDT_RLR_CFG 中的值会被重新加载到计数器中从而避免产生看门狗复位。

8.2.2.库函数

8.2.2.1. wdt_keepalive

函数名	wdt_keepalive
函数原型	void wdt_keepalive(void)
功能描述	喂狗操作，复位看门狗计时器
输入参数	无
输出参数	无
返回值	无

8.2.2.2. wdt_enable

函数名	wdt_enable
函数原型	void wdt_enable(uint32_t timeout)
功能描述	使能 WDT
输入参数	timeout：WDT 复位系统的时间，单位是秒
输出参数	无
返回值	无

8.3. WDT 应用例程

8.3.1.例程 1：WDT 复位操作

```
static void tim_timer_handler(HS_TIM_Type *tim)
{
    log_debug("timer irq\n");

    /* Keep alive */
    wdt_keepalive();
}


static void example_normal_timer(uint32_t period_us)
{
    //const tim_config_t timer_cfg =
    const tim_config_t timer_cfg = {
        .mode = TIM_TIMER_MODE,
        .config.timer = {
            .period_us = period_us,
            .callback = tim_timer_handler,
        },
    };
    tim_config(HS_TIM1, &timer_cfg);
    tim_start(HS_TIM1);
}


static void example_wdt(uint32_t wdt_reset_time, double feed_dog_time)
{
    //最大可设置为 31 s
    wdt_enable(wdt_reset_time);     //(2^12 - 1) / 128 = 32

    uint32_t period_us = (uint32_t) (feed_dog_time * 1000*1000);
    example_normal_timer(period_us);
}

void wdt_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);
    switch (case_num)
    {
```

```
        //设定时间内不喂狗复位
        case WDT_TEST_CASE_240000:
            //设定 1s, 1s  内不喂狗复位
            example_wdt(1 , 2);             //240000
            break;
        case WDT_TEST_CASE_240001:
            //设定 10s,10s 内不喂狗复位
            example_wdt(10, 12);            //240001
            break;
        default:
            log_debug("[%s][%d][The input test number is not support!]\n",
__FUNCTION__, __LINE__);
            // Do something you want to warning.
            break;
    }
}

int main(void)
{
    uint8_t         cmd[SHELL_FIFO_SIZE];
    uint8_t*        pCmd = &cmd[0];
    volatile bool ret = false;

    // Enble all IRQ quikly.
    __set_PRIMASK(0);

    // Disable WDT.
    wdt_enable(0);

    // Init interactive UART.
    uart_interaction_init();

    shell_init(HS_UART1);

    while (1)
    {
        ret = shell_get_cmd(&pCmd);

        if (ret == true)
        {
            log_debug("\r\ncmd : %s.\n", cmd);

            wdt_test_case(pCmd);
        }
```

```
    }

    shell_uninit();

    return 0;
}
```

此代码设置 WDT 1s 和 10s 自动复位系统，输出结果可由串口打印查看，具体如下图所示。

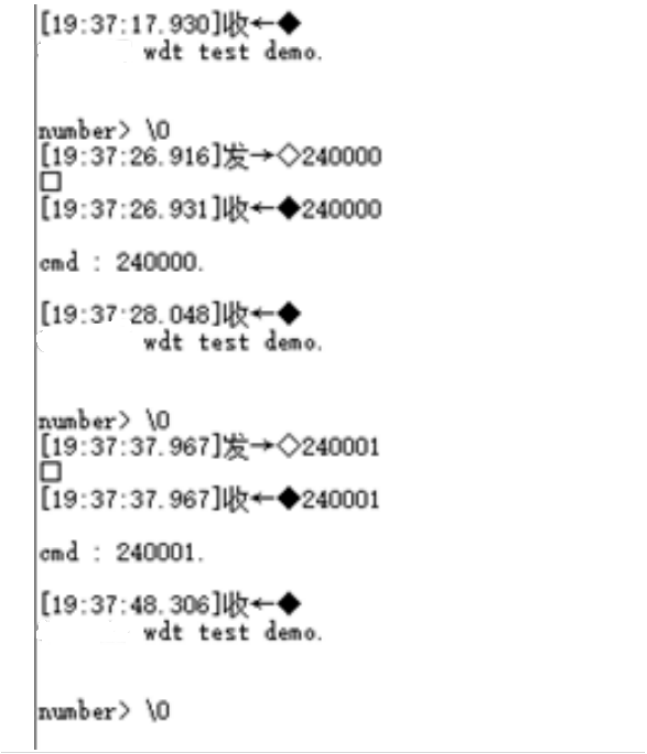


图 8.1 例程 1 串口打印示意图

8.3.2.例程 2：WDT 喂狗操作

```
static void tim_timer_handler(HS_TIM_Type *tim)
{
    log_debug("timer irq\n");

    /* Keep alive */
    wdt_keepalive();
}
static void example_normal_timer(uint32_t period_us)
{
    //const tim_config_t timer_cfg =
    const tim_config_t timer_cfg = {
        .mode = TIM_TIMER_MODE,
```

```
        .config.timer = {
            .period_us = period_us,
            .callback = tim_timer_handler,
        },
    };
    tim_config(HS_TIM1, &timer_cfg);
    tim_start(HS_TIM1);
}
static void example_wdt(uint32_t wdt_reset_time, double feed_dog_time)
{
    //最大可设置为 31 s
    wdt_enable(wdt_reset_time);     //(2^12 - 1) / 128 = 32

    uint32_t period_us = (uint32_t) (feed_dog_time * 1000*1000);
    example_normal_timer(period_us);
}

void wdt_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);
    switch (case_num)
    {
        //设定时间内喂狗不复位
        case WDT_TEST_CASE_240100:
            //设定 1s, 1s  内喂狗不复位
            example_wdt(1, 0.5);            //240100
            break;
        case WDT_TEST_CASE_240101:
            //设定 10s,10s 内喂狗不复位
            example_wdt(10, 8);             //240101
            break;
        default:
            log_debug("[%s][%d][The input test number is not support!]\n",
__FUNCTION__, __LINE__);
            // Do something you want to warning.
            break;
    }
}

int main(void)
{
    uint8_t         cmd[SHELL_FIFO_SIZE];
    uint8_t*        pCmd = &cmd[0];
    volatile bool ret = false;
```

```
    // Enble all IRQ quikly.
    __set_PRIMASK(0);

    // Disable WDT.
    wdt_enable(0);

    // Init interactive UART.
    uart_interaction_init();

    shell_init(HS_UART1);

    while (1)
    {
        ret = shell_get_cmd(&pCmd);

        if (ret == true)
        {
            log_debug("\r\ncmd : %s.\n", cmd);

            wdt_test_case(pCmd);
        }
    }

    shell_uninit();

    return 0;
}
```

此代码设置 0.5s 和 8s 喂狗一次，成功喂狗后，WDT 不再自动复位系统，输出结果可由串口打印查看，具体如下图所示。

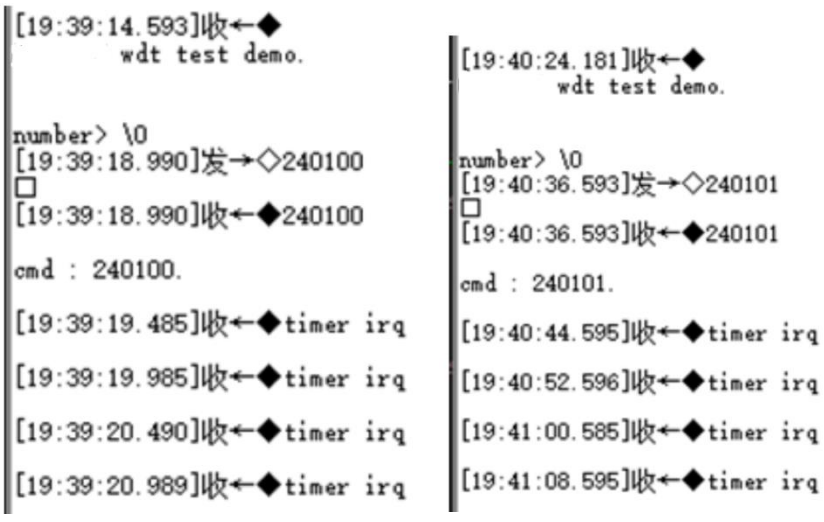


图 8.2 例程 2 串口打印示意图

8.3.3.例程 3：最大超时时间测试

```
static void tim_timer_handler(HS_TIM_Type *tim)
{
    log_debug("timer irq\n");

    /* Keep alive */
    wdt_keepalive();
}
static void example_normal_timer(uint32_t period_us)
{
    //const tim_config_t timer_cfg =
    const tim_config_t timer_cfg = {
        .mode = TIM_TIMER_MODE,
        .config.timer = {
            .period_us = period_us,
            .callback = tim_timer_handler,
        },
    };
    tim_config(HS_TIM1, &timer_cfg);
    tim_start(HS_TIM1);
}
static void example_wdt(uint32_t wdt_reset_time, double feed_dog_time)
{
    //最大可设置为 31 s
    wdt_enable(wdt_reset_time);     //(2^12 - 1) / 128 = 32

    uint32_t period_us = (uint32_t) (feed_dog_time * 1000*1000);
    example_normal_timer(period_us);
}

void wdt_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);
    switch (case_num)
    {
        //最大超时时间
        case WDT_TEST_CASE_240200:
            //设定 1s, 1s  内喂狗不复位
            example_wdt(31, 35);            //240100
            break;
        default:
            log_debug("[%s][%d][The input test number is not support!]\n",
```

```
__FUNCTION__, __LINE__);
            // Do something you want to warning.
            break;
    }
}

int main(void)
{
    uint8_t         cmd[SHELL_FIFO_SIZE];
    uint8_t*        pCmd = &cmd[0];
    volatile bool ret = false;

    // Enble all IRQ quikly.
    __set_PRIMASK(0);

    // Disable WDT.
    wdt_enable(0);

    // Init interactive UART.
    uart_interaction_init();

    shell_init(HS_UART1);

    while (1)
    {
        ret = shell_get_cmd(&pCmd);

        if (ret == true)
        {
            log_debug("\r\ncmd : %s.\n", cmd);

            wdt_test_case(pCmd);
        }
    }

    shell_uninit();

    return 0;
}
```

此代码进行最大超时时间测试，具体如下图所示。

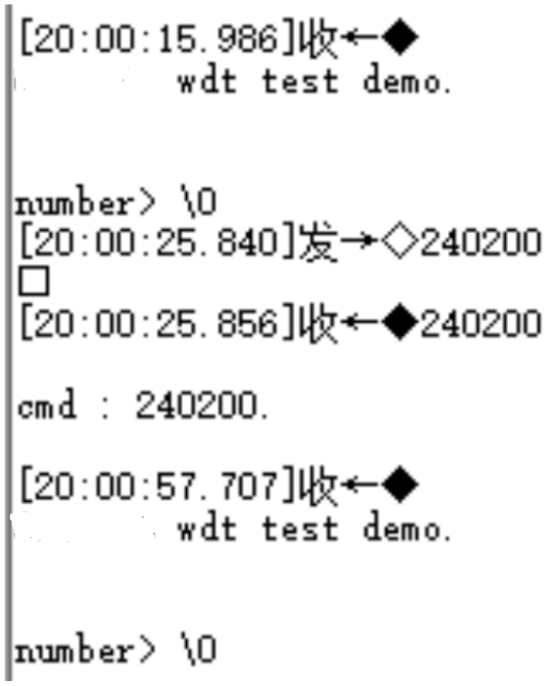


图 8.3 例程 3 串口打印示意图

8.4. WDT 使用注意事项

- BL1824X 的看门狗上电后默认开启。即使在 Boot 模式下，如果没有发指令关闭，也会在 30s 后复位系统。
- 看门狗关闭后，若再次喂狗，看门狗会重新激活运行。

- 看门狗最长超时时间为 31s。

9. GPADC

9.1. 简介

16 位 ADC 是逐次逼近模数转换器。它有多达 10 个多路复用通道，可以测量来自 8 gpio 的信号，以及温度和 VBAT 的两个给定通道。各种通道的 A/D 转换可以在单个、扫描或连续模式下执行。ADC 的结果存储在 16 位数据寄存器中。

ADC 工作时钟由启用寄存器“clk_en”选通的 PCLK 时钟选通，它与输入单 gpadc_out 生成的 16M clk 异步。

模块	基地址
ADC	0x400a2000

9.1.1.ADC 主要特征

- 16 位分辨率
- 转换结束时的中断生成和连续转换期间的提取数据块
- 单一和连续转换模式
- 用于将通道 gpio[0]自动转换为通道 gpio[n]的扫描模式（0<n<8）
- 温度校准
- 逐通道可编程采样时间
- 用于常规和注入转换的外部触发器选项
- 不连续模式
- ADC 电源要求：2.5 V 至 1.5 V
- 常规信道转换期间的 DMA 请求生成

ADC 的框图如图 10.1 所示。

9.1.2.ADC 功能描述

图 10.1 显示了单个 ADC 框图，表 10.1 给出了 ADC 引脚描述。

Name	Signal type	Remarks
ADC_CH*	基于 gpadc_sel 的值来决定 gpadc_out 的信道转换	总共 10 个模拟输入通道
APB_CLK	输入，ADC 工作时钟	来自 apb 总线
TIM*	输入，触发资源	由 trig_hw_sel 寄存器选择
ADC_INTR	输出，添加到状态	可以由相应的 reg 屏蔽。

表 9.1 ADC 引脚

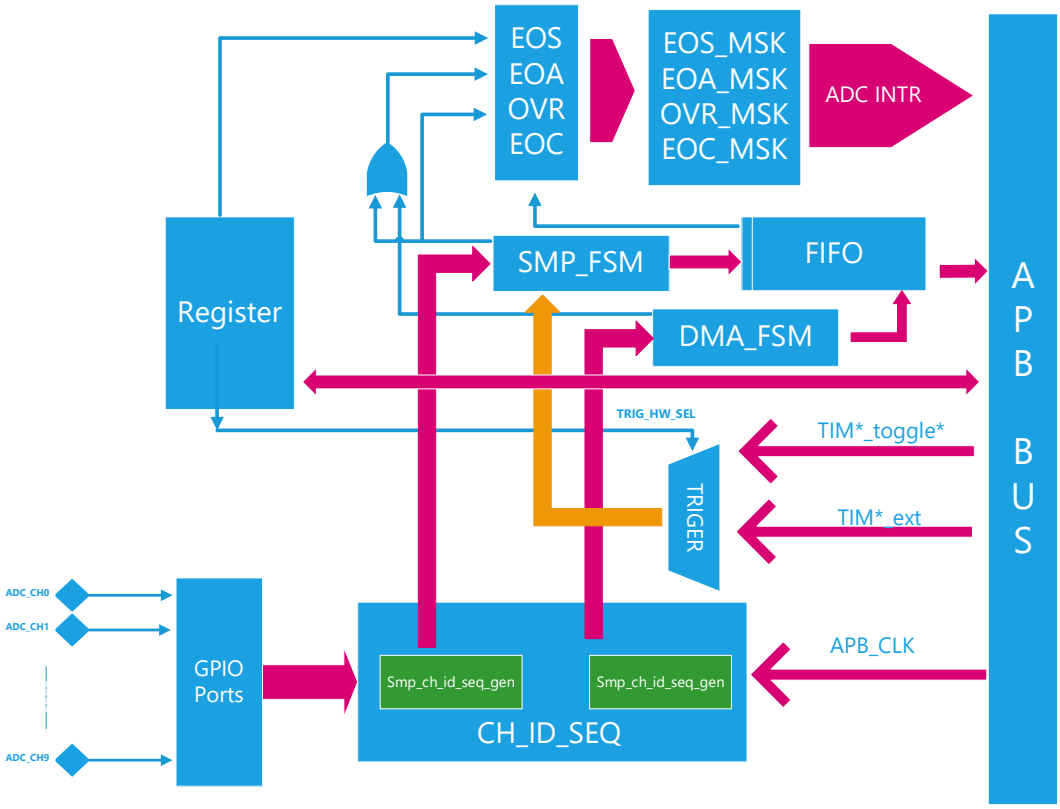


图 9.1 单 ADC 框图

9.1.3.ADC 开关控制

ADC 可以通过设置 ADC_CR0 寄存器中的 gpadc_start 位和 codec_ana_ctrl_2 寄存器中的 ADC_mode 位来通电。当第一次设置这两个位时，它将 ADC 从掉电模式唤醒。一旦设置了 gpadc_start 位，就可以按照软件设置的 adc_mode 进行转换。

通过设置同样在同一 ADC_CFG0 寄存器中的 ADC_stop 位，将 ADC 置于断电模式，可以随时停止转换。在此模式下，ADC 几乎不消耗功率。

9.1.4.ADC 时钟

CPM 控制器提供的 APBCLK 时钟与 gpadc_out（要转换的 16 位模拟输入数据）异步。clk 由 clk_ENS 寄存器（ANA_BASE+0xD0）中的 adc_clk_en 位选通，默认情况下，该位设置为 1。

样本信号是 gp_rdy，一个模拟输入信号，频率几乎为 125Khz。它看起来像一个占空比不同的时钟，但只是模拟和数字之间的脉冲同步。gp_rdy 中的模拟传输 gpadc_out 在正边缘输出，在负边缘进行数字采样。

9.1.5.通道选择

ADC 通路的 10 个通道对应的含义如下表 10.2 所示：

ADC 通道	对应设备	输出	备注
0	温度	温度	-40°C 到 125°C
1	VBAT	电压	电源电压内部默认分压 1/3
2	GPIO13	电压	电压输入范围：0.1V 到 3.5V
3	GPIO8		
4	GPIO9		
5	GPIO10		
6	GPIO11		
7	GPIO12		
8	GPIO2		
9	GPIO3		

表 9.2 ADC 10 通道对应含义说明

有 10 个多路复用信道。可以将转换分为两组：有序和无序。一组由一系列转换组成，这些转换可以在任何频道上以任何顺序进行。

1. 由最多 10 个转换组成。必须在 ADC_CFG2 寄存器中选择转换序列中的顺序通道及其顺序。

常规组中的通道方向必须写入 ADC_CFG2[0]位，该位命名为 scandir。当扫描设置为 1 时，通道从 0 切换到 9。例如，可以按照以下顺序进行转换：Ch3、Ch4、Ch5、Ch6、Ch7、Ch8、Ch9 或顺序：Ch3，Ch5、Ch6、Ch9，

将 seq_vect 从第 0 位扫描到第 9 位，如果设置了该位，则相应的通道将激活。如果 scandir 设置为 0，则方向相反。与值为 0x3FC 的 seq_vect 相同，通道的工作顺序为 Ch9、Ch8…Ch3。

2. 这种紊乱由一个接一个的转化组成。无序通道及其在转换序列中的顺序必须在 ADC_CFG2 寄存器中的相同位置进行选择，您可以随时通过设置 seq_vect 和 scandir 来配置通道，新配置将在当前转换完成后立即进行。

例如，可以按照以下顺序进行转换：Ch3、Ch6、Ch4、Ch7、Ch5、Ch8、Ch9，寄存器扫描器的功能与顺序模式完全相同

一旦启动 sfsm，只要 FSM 不处于空闲状态，无论是 IDLE_PD 还是 IDLE_SUS，通道选择都将完全自动。一旦一个通道完成转换，ADC 将根据 seq_vect 和 scandir 生成下一个工作通道和 gpadc_sel，并输出到模拟。

9.1.6.温度传感器/VREFINT 内部通道

温度传感器连接到通道 ADC_CH0，内部参考电压 VREFINT 连接到 ADC_CH1。这两个内部通道可以选择并转换为有序或无序通道。

9.1.7.单转换模式

在单转换模式下，ADC 进行一次转换。当 ADC_CFG1 寄存器的 trig_res 位为 0 时，通过设置 ADC_CFG0 寄存器中的 adc_start 位（仅用于常规通道）或外部触发器（用于常规通道或注入通道）启动此模式。另一种方法是将 sw_trigger_true 位设置为 1。

一旦所选频道的转换完成：

相同的作用并导致有序和无序转换：

（1）校准数据存储在 16 位 ch_0_datal 寄存器（s，7，8）中

（2）设置 EOC（转换结束）标志

（3）如果设置了 EOC_0_MSK，则生成中断。

然后停止 ADC。

9.1.8.连续转换模式

在连续转换模式下，一旦完成一次转换，ADC 就开始另一次转换。当 seq_life 位为 0 时，此模式通过外部触发器或通过设置 ADC_CFG0 寄存器中的 adc_start 位来启动。

每次转换后：

如果订单通道已转换：

（1）–转换后的数据存储在 16 位 ch_N_data 寄存器中

（2）–设置 EOC（转换结束）标志

（3）–如果设置了 EOC_N_MSK，则会生成中断。

（4）–设置 OVR（超限行驶）标志

（5）–如果设置了 OVR_MSK，则会生成中断。

如果仅转换了一个频道：

（1）–转换后的数据存储在 16 位 ch_N_data 寄存器中

（2）–设置 EOC（转换结束）标志

（3）–如果设置了 EOC_N_MSK，则会生成中断。

9.1.8.1. 扫描模式

此模式用于扫描一组模拟通道。

可以通过设置 ADC_CFG2 寄存器中的扫描位来选择扫描模式。一旦设置了该位，ADC

将扫描 seq_vect 寄存器中选择的所有通道。对该组的每个通道执行单个转换。每次转换结束后，组的下一个频道将自动转换。如果 seq_life 位设置为 0，则转换不会在最后选择的组信道处停止，而是从第一个选择的组通道再次继续。如果设置了 DMA 位，则直接存储器访问控制器用于在每个 EOC 之后将有序组信道的转换数据传输到 SRAM。

9.1.9.不连续模式

通过设置 ADC_CFG1 寄存器中的 AUTO_PD2 位启用此模式。它可用于决定转换后的状态，该转换是 SEQ_VECT 寄存器中选择的转换序列的一部分。值为 1 意味着在采样后 sfsm 将转换为 PD，您需要重复启动流程以进行另一次转换。0 表示挂起等待触发器或等待特定时间，该时间配置得足够长，以便有足够的时间完成信道转换。写入 dly_ cfg 寄存器中的 cfg_。

当外部触发器发生时，它开始 SEQ_VECT 寄存器中选择的下 n 个转换，直到序列中的所有转换完成。总序列长度由 ADC_CFG2 寄存器中的 seq_life 位定义。

例子：

Seq_life=3，要转换的通道=0、1、2、3、6、7、9、10，Seq_vect=0x2CF。

将对所有序列进行三次定时器，每个 EOC 事件在相应的通道采样完成后生成，也将在每次 seq_life_cnt 更改后生成 EOA

注 1：当序列转换为 finish 时，内部 seq_life_cnt 将为负 1，并且将 seq_life 设置为 0 意味着 seq_life 无限，则不会发生 finish。

注 2：wait_cnt_timeout 工作的 clk 是 apb_clk 除以 digi_clk_freq[7:1].

9.1.9.1. 逐通道可编程采样时间

ADC 在多个 ADC_CLK 周期内对输入电压进行采样，这些周期可以使用 ADC_CFG1 寄存器的 SMP_OUT_CTRL 和 avg_cfg 位中的 auto_compen 位进行修改。每个通道可以用不同的采样时间进行采样。

转换可以由外部事件触发（例如定时器捕获、EXTI 行）。如果设置了 EXTTRIG 控制位，则外部事件能够触发转换。trig_hw_sel[3:0]控制位允许应用程序选择决定 12 个可能事件中的哪一个可以触发常规组和注入组的转换。

注：当选择外部触发器进行 ADC 转换时，ADC_CFG1 寄存器中的 trig_mode 位可以选择信号的正边沿或负边沿可以开始转换。

9.1.9.2. DMA 请求

由于转换后的常规信道值存储在唯一的数据寄存器中，因此有必要使用 DMA 来转换

多个常规信道。这避免了已经存储在 ch_dma_data 寄存器中的数据丢失。

只有常规信道的转换结束才会生成 DMA 请求，这允许将其转换后的数据从 ch_DMA_data 寄存器传输到用户选择的目标位置。

9.1.9.3. ADC 中断

可以在常规组和注入组的转换结束时以及在设置相应的掩码位时产生中断。单独的中断使能位可用于灵活性。

注意：ADC 中断是 w1c，INTR 寄存器是原始中断状态，一位输出 ADC_INTR 是按位或掩码后所有原始中断的状态。

Interrupt Event	Event Flag	MASK Control Bit
转换结束	EOC	EOC_MASK
全部转换结束	EOA	EOA_MASK
转换通道之间的超越	OVR	OVR_MASK

表 9.3 ADC 中断

9.2. API 介绍

9.2.1.ADC 寄存器结构

```
typedef struct
{
    __IO uint32_t INTR;                    // offset: 4*0x000 = 0x00
    __IO uint32_t INTR_MSK;                // offset: 4*0x001 = 0x04
    __IO uint32_t DLY_CFG;                 // offset: 4*0x002 = 0x08
       uint32_t RESERVED1;                 // offset: 4*0x003
    __IO uint32_t ADC_CFG0;                // offset: 4*0x004 = 0x10
    __IO uint32_t ADC_CFG1;                // offset: 4*0x005 = 0x14
    __IO uint32_t ADC_CFG2;                // offset: 4*0x006 = 0x18
       uint32_t RESERVED2;                 // offset: 4*0x007
    __IO uint32_t ADC_SW_TRIGGER;          // offset: 4*0x008 = 0x20
       uint32_t RESERVED3;                 // offset: 4*(0x009~~0x000B)
    __IO uint32_t CH_0_CFG;                // offset: 4*0x00C = 0x30
    __IO uint32_t CH_1_CFG;                // offset: 4*0x00D = 0x34
    __IO uint32_t CH_2_CFG;                // offset: 4*0x00E = 0x38
    __IO uint32_t CH_3_CFG;                // offset: 4*0x00F = 0x3c
    __IO uint32_t CH_4_CFG;                // offset: 4*0x010 = 0x40
    __IO uint32_t CH_5_CFG;                // offset: 4*0x011 = 0x44
    __IO uint32_t CH_6_CFG;                // offset: 4*0x012 = 0x48
    __IO uint32_t CH_7_CFG;                // offset: 4*0x013 = 0x4c
```

```
    __IO uint32_t CH_8_CFG;                 // offset: 4*0x014 = 0x50
    __IO uint32_t CH_9_CFG;                 // offset: 4*0x015 = 0x54
    __IO uint32_t VOS_TEMP;                 // offset: 4*0x018 = 0x60
     __IO uint32_t CH_0_DATA;               // offset: 4*0x01A = 0x68
    __IO uint32_t CH_1_DATA;                // offset: 4*0x01B = 0x6c
    __IO uint32_t CH_2_DATA;                // offset: 4*0x01C = 0x70
    __IO uint32_t CH_3_DATA;                // offset: 4*0x01D = 0x74
    __IO uint32_t CH_4_DATA;                // offset: 4*0x01E = 0x78
    __IO uint32_t CH_5_DATA;                // offset: 4*0x01F = 0x7c
    __IO uint32_t CH_6_DATA;                // offset: 4*0x020 = 0x80
    __IO uint32_t CH_7_DATA;                // offset: 4*0x021 = 0x84
    __IO uint32_t CH_8_DATA;                // offset: 4*0x022 = 0x88
    __IO uint32_t CH_9_DATA;                // offset: 4*0x023 = 0x8c
         uint32_t RESERVED5[8];             // offset: 4*(0x024~~0x2B)
    __IO uint32_t CH_DMA_DATA;              // offset: 4*0x02C = 0xB0
    __IO uint32_t DMA_CNS;                  // offset: 4*0x02D = 0xB4
} HS_GPADC_Type;
```

Offset	Name	Description
0x0000	INTR	原始中断写入 1 清除
0x0004	INTR_MASK	中断掩码
0x0008	DLY_CFG	dly 计数器寄存器
0x0010	ADC_CFG0	Adc 状态寄存器
0x0014	ADC_CFG1	Adc 工作模式寄存器
0x0018	ADC_CFG2	通道寄存器
0x0020	ADC_SW_TRIG	开关触发寄存器
0x0030	CH_0_CFG	通道 0 寄存器
0x0034	CH_1_CFG	通道 1 寄存器
0x0038	CH_2_CFG	通道 2 寄存器
0x003c	CH_3_CFG	通道 3 寄存器
0x0040	CH_4_CFG	通道 4 寄存器
0x0044	CH_5_CFG	通道 5 寄存器
0x0048	CH_6_CFG	通道 6 寄存器
0x004c	CH_7_CFG	通道 7 寄存器
0x0050	CH_8_CFG	通道 8 寄存器
0x0054	CH_9_CFG	通道 9 寄存器
0x0060	VOS_TEMP_REG	Vos_temp 寄存器
0x0068	CH_0_DATA	通道 0 输出数据
0x006c	CH_1_DATA	通道 1 输出数据
0x0070	CH_2_DATA	通道 2 输出数据
0x0074	CH_3_DATA	通道 3 输出数据
0x0078	CH_4_DATA	通道 4 输出数据

0x007c	CH_5_DATA	通道 5 输出数据
0x0080	CH_6_DATA	通道 6 输出数据
0x0084	CH_7_DATA	通道 7 输出数据
0x0088	CH_8_DATA	通道 8 输出数据
0x008c	CH_9_DATA	通道 9 输出数据
0x00B0	CH DMA DATA	DMA 模式下的输出数据
0x00B4	DMA CNS	Dma 控制和状态

INTR address offset: 0x0000

Bit	R/W	Reset	Name	Description
31:13	RW	0x0	N/A	reserved
12	RW	0x0	OVR	中断 ovr 状态，1 个 ovr_intr 并写入 1 清除 中断过采样状态 读取 1:数据无效 读取 0:数据有效 写入 1:清除
11	RW	0x0	EOA	中断 eoa 状态 读取 1:所有转换的结束 写入 1:清除
10	RW	0x0	EOS	中断 eos 状态 读取 1:单个序列转换的结束 写入 1:清除
9	RW	0x0	EOC_9	中断 eoc_9 状态 读取 1:通道 9 转换的结束 写入 1:清除
8	RW	0x0	EOC_8	中断 eoc_8 状态 读取 1:通道 8 转换的结束 写入 1:清除
7	RW	0x0	EOC_7	中断 eoc_7 状态 读取 1:通道 7 转换的结束 写入 1:清除
6	RW	0x0	EOC_6	中断 eoc_6 状态 读取 1:通道 6 转换的结束 写入 1:清除中
5	RW	0x0	EOC_5	中断 eoc_5 状态 读取 1:通道 5 转换的结束 写入 1:清除
4	RW	0x0	EOC_4	中断 eoc_4 状态 读取 1:通道 4 转换的结束 写入 1:清除
3	RW	0x0	EOC_3	中断 eoc_3 状态 读取 1:通道 3 转换的结束 写入 1:清除

2	RW	0x0	EOC_2	中断 eoc_2 状态 读取 1:通道 2 转换的结束 写入 1:清除
1	RW	0x0	EOC_1	中断 eoc_1 状态 读取 1:通道 1 转换的结束 写入 1:清除
0	RW	0x0	EOC_0	中断 eoc_0 状态 读取 1:通道 0 转换的结束 写入 1:清除

INTR_MASK address offset: 0x0004

Bit	R/W	Reset	Name	Description
31:13	RW	0x0	INTR RESERVED REG	reserved
12	RW	0x0	OVR_MSK	中断 ovr 掩码 0:掩码 1:启用
11	RW	0x0	EOA_MSK	中断 eoa 掩码 0:掩码 1:启用
10	RW	0x0	EOS_MSK	中断 eos 掩码 0:掩码 1:启用
9	RW	0x0	EOC_9_MSK	中断 eoc_9 掩码 0:掩码 1:启用
8	RW	0x0	EOC_8_MSK	中断 eoc_8 掩码 0:掩码 1:启用
7	RW	0x0	EOC_7_MSK	中断 eoc_7 掩码 0:掩码 1:启用
6	RW	0x0	EOC_6_MSK	中断 eoc_6 掩码 0:掩码 1:启用
5	RW	0x0	EOC_5_MSK	中断 eoc_5 掩码 0:掩码 1:启用
4	RW	0x0	EOC_4_MSK	中断 eoc_4 掩码 0:掩码 1:启用
3	RW	0x0	EOC_3_MSK	中断 eoc_3 掩码 0:掩码 1:启用

2	RW	0x0	EOC_2_MSK	中断 eoc_2 掩码 0:掩码 1:启用
1	RW	0x0	EOC_1_MSK	中断 eoc_1 掩码 0:掩码 1:启用
0	RW	0x0	EOC_0_MSK	中断 eoc_0 掩码 0:掩码 1:启用

DLY_CFG address offset: 0x0008

Bit	R/W	Reset	Name	Description
31:24	N/A	0x0	N/A	reserved
23:16	RW	0x14	ORB_DLY	从一个通道到另一个通道的过阻塞延迟
15:8	RW	0x50	PD_DLY_MAX	从 pd 模式到正常工作模式的延时时间
7:0	RW	0x50	CFG_CHG_DLY	从暂停模式延迟到正常工作模式的时间

ADC_CFG0 address offset: 0x0010

Bit	R/W	Reset	Name	Description
31:13	R	0x0	N/A	reserved
12	R	0x0	ADC_SUSPEND	读取暂停模式状态： 0:其他状态 1:挂起模式
4	W	0x0	ADC_STOP	停止 adc 并返回到 IDLE_PD 写 1：清除 adc_start 状态
3:2	RW	0x0	N/A	Reserved
1	R	0x0	ADC_MODE	adc 模式 0:auadc 1:gpadc
0	RW	0x0	ADC_START	启动 adc sfsm 0:禁止 1:开始

ADC_CFG1 address offset: 0x0014

Bit	R/W	Reset	Name	Description
31:24	RW	0x20	DIGI_CLK_FREQ	等待计数器的 div clk_gate
23	RW	0x0	N/A	reserved
22:20	RW	0x4	AVG_CFG	1 的数量<<avg_cfg gpadc_out 求和然后求平均
19:15	RW	0x0	N/A	reserved
14:13	RW	0x0	GP_BIT_SEL	gpadc 采样周期 01:32 个周期 10:64 个周期

				11:128 个周期 注：00 无定义
12	RW	0x0	DMA_EN	DMA 使能控制 0:禁止 1:启用
11:8	RW	0x0	TRIG_HW_SEL	选择触发源 4'b0000:ext_trigger = timer1_toggle[0] 4'b0001:ext_trigger = timer1_toggle[1] 4'b0010:ext_trigger = timer1_toggle[2] 4'b0011:ext_trigger = timer1_toggle[3] 4'b0100:ext_trigger = timer2_toggle 4'b1000:ext_trigger = timer3_toggle[0] 4'b1001:ext_trigger = timer3_toggle[1] 4'b1010:ext_trigger = timer3_toggle[2] 4'b1011:ext_trigger = timer3_toggle[3] 4'b1100:ext_trigger = timer1_ext 4'b1101:ext_trigger = timer2_ext 4'b1110:ext_trigger = timer3_ext Default:ext_trigger = 1'b0
7	RW	0x0	N/A	reserved
6	RW	0x0	TRIG_RES	触发器分辨率 0:每个通道 1:每个序列
5:4	RW	0x0	TRIG MODE	选择触发源 01:ext_trigger pedge 10:ext_trigger nedge 11:ext_trigger pedge || ext_trigger nedge Default:adc_sw_trig_wr || sw_trigger_true
3	RW	0x0	N/A	reserved
2	RW	0x0	AUTO_PD2	整个序列后，返回以下状态： 0:暂停模式 1:断电模式
1	RW	0x0	AUTO_DELAY	ch 就绪且未获取数据，将在其之后阻塞 ch 0:禁止 1:使能
0	RW	0x0	AUTO_PD1	触发后，每个通道断电 0:通电 1:掉电

ADC_CFG2 address offset: 0x0018

Bit	R/W	Reset	Name	Description
31:16	RW	0x1	SEQ_LIFE	序列号
15:11	RW	0x0	N/A	reserved
10:1	RW	0x10	SEQ_VECT	通道启用

				0:禁用 1:启用 序列向量[*]：CH_*_启用
0	RW	0x0	SCANDIR	ch 序列顺序 0:decr 模式从 10 到 0 1:从 0 到 10 的增量模式

ADC_SW_TRIG address offset: 0x0020

Bit	R/W	Reset	Name	Description
31	RW	0x0	SW_TRIGGER_TRUE	触发器始终为真
30:0	N/A	0x0	N/A	reserved

CH_0_CFG address offset: 0x0030

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	01:1.25V 注：00/10/11 无定义

CH_1_CFG address offset: 0x0034

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	和 CH_0_CFG 相同

CH_3_CFG address offset: 0x003c

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	和 CH_0_CFG 相同

CH_4_CFG address offset: 0x0040

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	和 CH_0_CFG 相同

CH_5_CFG address offset: 0x0044

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	和 CH_0_CFG 相同

CH_6_CFG address offset: 0x0048

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	和 CH_0_CFG 相同

CH_7_CFG address offset: 0x004c

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	和 CH_0_CFG 相同

CH_8_CFG address offset: 0x0050

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	和 CH_0_CFG 相同

CH_9_CFG address offset: 0x0054

Bit	R/W	Reset	Name	Description
31:2	N/A	0x0	N/A	reserved
1:0	RW	0x0	SEL_GP_VERF	和 CH_0_CFG 相同

VOS_TEMP_REG address offset: 0x0060

Bit	R/W	Reset	Name	Description
31:28	N/A	0x0	N/A	reserved
27	RW	0x1	PD_GP_TEMP_GEN	温度检测电源控制 0:通电 1:断电
26:24	RW	0x5	TEMP_SEL	0x0:20℃ 0x1:21℃ 0x2:22℃ 0x3:23℃ 0x4:24℃ 0x5:25℃ 0x6:26℃ 0x7:27℃
23:0	RW	0x0	N/A	reserved

CH_0_DATA address offset: 0x0068

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_0_DATAL	ch0 中的 adc 输出数据

CH_1_ DATA address offset: 0x006C

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_1_DATAL	ch1 中的 adc 输出数据

CH_2_ DATA address offset: 0x0070

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_2_DATAL	ch2 中的 adc 输出数据

CH_3_ DATA address offset: 0x0074

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_3_DATAL	ch3 中的 adc 输出数据

CH_4_ DATA address offset: 0x0078

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_4_DATAL	ch4 中的 adc 输出数据

CH5_ DATA address offset: 0x007C

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_5_DATAL	ch5 中的 adc 输出数据

CH_6_ DATA address offset: 0x0080

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_6_DATAL	ch6 中的 adc 输出数据

CH_7_ DATA address offset: 0x0084

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_7_DATAL	ch7 中的 adc 输出数据

CH_8_ DATA address offset: 0x0088

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_8_DATAL	ch8 中的 adc 输出数据

CH_9_ DATA address offset: 0x008C

Bit	R/W	Reset	Name	Description
31:16	N/A	0x0	N/A	reserved
15:0	RW	0x0	CH_9_DATAL	ch9 中的 adc 输出数据

CH_DMA_ DATA address offset: 0x00B0

Bit	R/W	Reset	Name	Description
31:0	RW	0x0	CH_DMA_DATA	dma 模式下的 adc 输出数据

DMA_CNS address offset: 0x00B4

Bit	R/W	Reset	Name	Description
31:8	N/A	0x0	N/A	reserved
7:4	RW	0x0	CH_ID_DMA	dma 模式下的活动信道
3:2	N/A	0x0	N/A	reserved
1:0	R	0x0	DFSM	dma fsm 仅读取用于调试

9.2.2.变量参数

9.2.2.1. adc_channel_t

```
typedef enum
{
    ADC_CH_0 = 0, // b0000 channel temperature
    ADC_CH_1 = 1, // b0001 channel vbat
    ADC_CH_2 = 2, // b0010 channel gpio<13>
    ADC_CH_3 = 3, // b0011 channel gpio<8>
    ADC_CH_4 = 4, // b0100 channel gpio<9>
    ADC_CH_5 = 5, // b0101 channel gpio<10>
    ADC_CH_6 = 6, // b0110 channel gpio<11>
    ADC_CH_7 = 7, // b0111 channel gpio<12>
    ADC_CH_8 = 8, // b1000 channel gpio<2>
    ADC_CH_9 = 9, // b1001 channel gpio<3>
    ADC_CH_MAX,
} adc_channel_t;
```

ADC 各个通道参数。

9.2.2.2. adc_sampling_cycles_t

```
typedef enum
{
    ADC_SAMPLING_CYCLES_32 = 1,
    ADC_SAMPLING_CYCLES_64 = 2,
    ADC_SAMPLING_CYCLES_128 = 3,
    ADC_SAMPLING_CYCLES_MAX,
} adc_sampling_cycles_t;
```

ADC 采样周期。

9.2.2.3. adc_temper_refer_t

```
typedef enum {
    ADC_TEMPER_20 = 0, // temperature 20C
    ADC_TEMPER_21 = 1, // temperature 21C
    ADC_TEMPER_22 = 2, // temperature 22C
    ADC_TEMPER_23 = 3, // temperature 23C
    ADC_TEMPER_24 = 4, // temperature 24C
    ADC_TEMPER_25 = 5, // temperature 25C
    ADC_TEMPER_26 = 6, // temperature 26C
    ADC_TEMPER_27 = 7, // temperature 27C
    ADC_TEMPER_MAX,
} adc_temper_refer_t;
```

ADC 的参考温度档位。

9.2.2.4. adc_callback_t

```
typedef void (*adc_callback_t)(uint32_t event, int adc_data);
```

ADC 事件回调。

9.2.2.5. adc_dma_config_t

```
typedef struct
{
    /// DMA fifo enable
    bool              use_fifo;
    /// Fifo buffer
    void              *buffer;
    /// Fifo buffer length
    uint32_t          buffer_len;
    /// DMA block link list item
    dma_llip_t      *block_llip;
    /// DMA block link list item number
    uint32_t          block_num;
    /// Event callback
    dma_callback_t callback;
}adc_dma_config_t;
```

ADC 的 DMA 配置参数。

use_fifo：DMA 的 fifo 使能项；

buffer：fifo 的指针；

buffer_len：fifo 的长度；

block_llip：DMA 块链表项；

block_num：DMA 区块链表项号；

callback：事件回调。

9.2.2.6. adc_calib_sel_t

```
typedef struct
{
   ADC_CALIB_CP = 0,
   ADC_CALIB_FT = 1,
}adc_calib_sel_t;
```

ADC 校准参数保存选项。

ADC_CALIB_CP：保存至 efuse；

ADC_CALIB_FT：保存至 flash。

9.2.3.库函数

9.2.3.1. adc_open

函数名	adc_open
函数原型	void adc_open(void)
功能描述	ADC 开启时钟
输入参数	无
输出参数	无
返回值	无

例：开启 ADC 时钟，开始初始化。

```
adc_open();
```

9.2.3.2. adc_close

函数名	adc_close
函数原型	adc_close (void)
功能描述	ADC 关闭时钟
输入参数	无
输出参数	无
返回值	无

例：关闭 ADC 时钟。

```
 adc_close ();
```

9.2.3.3. adc_calib_comm

函数名	adc_calib_comm
函数原型	static uint16_t adc_calib_comm(adc_channel_t channel, adc_sampling_cycles_t sampling_cycles, bool compen);
功能描述	输出 adc 采样的 code 值或者实际值
输入参数 1	channel：ADC 通道
输入参数 2	sampling_cycles：ADC 采样周期项
输入参数 3	compen：0：输出 code 值；1：输出实际值
返回值	adc 采样的 code 值或者实际值

例：选择 128 的采样周期，设置 ch8 输出实际电压值。

```
uint16_t ret = adc_calib_comm(ADC_CH_8, ADC_SAMPLING_CYCLES_128,1);
log_debug("ch8 output voltage = %d\n", ret);
```

9.2.3.4. adc_set_calibarate_param

函数名	adc_set_calibarate_param
函数原型	void adc_set_calibarate_param(cpft_gpadc_calib_t *config32, cpft_gpadc_calib_t *config64, cpft_gpadc_calib_t *config128);
功能描述	设置 ADC 校准数据
输入参数 1	config32：指向结构 cpft_gpadc_calib_t 的指针，config32 中包含 ADC 相关校准数据
输入参数 2	config64：指向结构 cpft_gpadc_calib_t 的指针，config64 中包含 ADC 相关校准数据
输入参数 3	config128：指向结构 cpft_gpadc_calib_t 的指针，config128 中包含 ADC 相关校准数据
输出参数	无
返回值	无

9.2.3.5. adc_voltage_cal_by_sw

函数名	adc_voltage_cal_by_sw
函数原型	float adc_voltage_cal_by_sw(int out, adc_sampling_cycles_t sampling_cycles);
功能描述	校准输入的 code 值转换为实际值
输入参数 1	ADC 采样的 code 值
输入参数 2	sampling_cycles：ADC 采样周期项
输出参数	无
返回值	ADC 采样的实际值

例：选择 128 的采样周期，设置 ch8 输出 code 值，再带入输出实际值。

```
int ret_raw = adc_calib_comm(ADC_CH_8, ADC_SAMPLING_CYCLES_128,1);
```

```
float ret = adc_voltage_cal_by_sw(ret_raw, ADC_SAMPLING_CYCLES_128);
log_debug("ch8 output voltage = %f\n", ret);
```

9.2.3.6. adc_voltage_read_by_channel

函数名	adc_voltage_read_by_channel
函数原型	uint16_t adc_voltage_read_by_channel(adc_channel_t channel, adc_sampling_cycles_t sampling_cycles);
功能描述	输出对应 ADC 通道的实际值
输入参数 1	channel：ADC 通道
输入参数 2	sampling_cycles：ADC 采样周期项
输出参数	无
返回值	对应 ADC 通道的实际值

例：选择 128 的采样周期，设置 ch8 的实际值。

```
int ret = adc_voltage_read_by_channel(ADC_CH_8, ADC_SAMPLING_CYCLES_128);
log_debug("ch8 output voltage = %d\n", ret);
```

9.2.3.7. adc_temperature_read_normal

函数名	adc_temperature_read_normal
函数原型	int adc_temperature_read_normal(adc_sampling_cycles_t sampling_cycles);
功能描述	输出对应 ADC 温度通道的实际值
输入参数	sampling_cycles：ADC 采样周期
输出参数	无
返回值	对应 ADC 温度通道的实际值

例：选择 128 的采样周期，读取当前温度的实际值。

```
int ret = adc_temperature_read_normal (ADC_SAMPLING_CYCLES_128);
log_debug("chip temperature = %d\n", ret);
```

9.2.3.8. adc_temperature_read

函数名	adc_temperature_read
函数原型	int adc_temperature_read(void);
功能描述	输出采样周期为 128 的 ADC 温度通道的实际值
输入参数	无
输出参数	无
返回值	ADC 温度通道的实际值

例：读取当前温度的实际值。

```
int ret = adc_temperature_read ();
```

```
log_debug("chip temperature = %d\n", ret);
```

9.2.3.9. adc_calib_first_step

函数名	adc_calib_first_step
函数原型	bool adc_calib_first_step(adc_channel_t channel)
功能描述	ADC 第一步校准
输入参数	channel：ADC 通道
输出参数	无
返回值	false: fai,true: succ

9.2.3.10. adc_calib_second_step

函数名	adc_calib_second_step
函数原型	bool adc_calib_second_step(adc_channel_t channel, adc_temper_refer_t temper, cpft_gpadc_calib_t *pCalibDataSave);
功能描述	ADC 第二步校准
输入参数 1	channel：ADC 通道
输入参数 2	temper：参考温度档
输入参数 3	calib_sel：ADC 校准参数保存选项
输入参数 4	pCalibDataSave：指向结构 cpft_gpadc_calib_t 的指针，pCalibDataSave 中包含相关校准数据
输出参数	无
返回值	ADC 转换的初始数据

9.2.3.11. adc_calib_third_step

函数名	adc_calib_third_step
函数原型	bool adc_calib_third_step(adc_channel_t channel, int inputVoltValue)
功能描述	启动 ADC 以获得输出电压值并判断结果
输入参数 1	channel：ADC 通道
输入参数 2	inputVoltValue：以 V 为单位输入电压
输出参数	无
返回值	false: fai,true: succ

9.2.3.12. adc_voltage_read_single_begin_nostop

函数名	adc_voltage_read_single_begin_nostop
函数原型	void adc_voltage_read_single_begin_nostop(adc_channel_t channel, adc_sampling_cycles_t sampling_cycles);
功能描述	ADC 单通道连续扫描模式配置设置
输入参数 1	channel：ADC 通道

输入参数 2	sampling_cycles：ADC 采样周期项
输出参数	无
返回值	无

例：选择 128 的采样周期，选择 ch8 配置连续采样模式。

```
adc_voltage_read_single_begin_nostop(ADC_CH_8,ADC_SAMPLING_CYCLES_128);
```

9.2.3.13. adc_voltage_read_single_nostop

函数名	adc_voltage_read_single_nostop
函数原型	uint16_t adc_voltage_read_single_nostop(adc_channel_t channel, adc_sampling_cycles_t sampling_cycles);
功能描述	读取 ADC 单通道连续扫描设置下对应的通道实际值
输入参数 1	channel：ADC 通道
输入参数 2	sampling_cycles：ADC 采样周期项
输出参数	无
返回值	单通道连续扫描下的实际值

例：选择 128 的采样周期，选择 ch8 配置连续采样模式，读出当前的实际值。

```
adc_voltage_read_single_begin_nostop(ADC_CH_8, ADC_SAMPLING_CYCLES_128);
uint16_t ret = adc_voltage_read_single_nostop(ADC_CH_8, ADC_SAMPLING_CYCLES_128);
log_debug("ch in nostop = %d\n",ret);
```

9.2.3.14. adc_voltage_read_multi_begin_nostop

函数名	adc_voltage_read_multi_begin_nostop
函数原型	void adc_voltage_read_multi_begin_nostop(uint16_t chBmp, adc_sampling_cycles_t sampling_cycles);
功能描述	ADC 多通道连续扫描设置
输入参数 1	chBmp：都通道设置项
输入参数 2	sampling_cycles：ADC 采样周期项
输出参数	无
返回值	无

例：选择 128 的采样周期，选择 ch2~ch9 配置连续采样模式。

```
adc_voltage_read_multi_begin_nostop(0x7FC, ADC_SAMPLING_CYCLES_128);
```

9.2.3.15. adc_voltage_read_multi_nostop

函数名	adc_voltage_read_multi_nostop
函数原型	void adc_voltage_read_multi_nostop(uint16_t chBmp, float *cha_val, adc_sampling_cycles_t sampling_cycles);
功能描述	读取 ADC 多通道连续扫描设置下对应的通道实际值

输入参数 1	chBmp：都通道设置项
输入参数 2	cha_val：对应通道输出的实际值
输入参数 3	sampling_cycles：ADC 采样周期项
输出参数	多通道连续扫描下的实际值
返回值	无

例：选择 128 的采样周期，选择 ch2~ch9 配置连续采样模式，读出当前的实际值。

```
float multi_buf[9];
adc_voltage_read_multi_begin_nostop(0x7FC, ADC_SAMPLING_CYCLES_128);

adc_voltage_read_multi_nostop(0x7FC, multi_buf, ADC_SAMPLING_CYCLES_128);

for (uint32_t i = 0; i < sizeof(multi_buf) / sizeof(multi_buf[0]); i++)
{
    log_debug("example_adc multi channel voltage[%d] =%f\n", i, multi_buf[i]);
}
```

9.2.3.16. adc_temperature_read_begin_nostop

函数名	adc_temperature_read_begin_nostop
函数原型	void adc_temperature_read_begin_nostop(adc_sampling_cycles_t sampling_cycles)
功能描述	ADC 连续方式配置温度
输入参数	sampling_cycles：ADC 采样周期项
输出参数	无
返回值	无

例：选择 128 的采样周期，选择温度配置连续采样模式。

```
adc_temperature_read_begin_nostop(ADC_SAMPLING_CYCLES_128);
```

9.2.3.17. adc_temperature_read_nostop

函数名	adc_temperature_read_nostop
函数原型	int adc_temperature_read_nostop(adc_sampling_cycles_t sampling_cycles)
功能描述	ADC 连续方式读取温度
输入参数	sampling_cycles：ADC 采样周期项
输出参数	无
返回值	连续扫描模式下的温度值

例：选择 128 的采样周期，选择温度通道配置连续采样模式，读出当前的实际值。

```
adc_temperature_read_begin_nostop(ADC_SAMPLING_CYCLES_128);

int ret = adc_temperature_read_nostop(ADC_SAMPLING_CYCLES_128);
log_debug(“temperature in nostop = %d\n”,ret);
```

9.2.3.18. adc_dma_config

函数名	adc_dma_config
函数原型	HS_DMA_CH_Type *adc_dma_config(HS_DMA_CH_Type *dma, const adc_dma_config_t *config, uint32_t *src_addr);
功能描述	ADC 的 DMA 配置
输入参数 1	dma：指向结构 HS_DMA_CH_Type 的指针，包含 DMA 控制器相关信息
输入参数 2	config：指向结构 adc_dma_config_t 的指针，包含 ADC 的 DMA 相关配置的信息
输入参数 3	src_addr：ADC 输出地址
输出参数	无
返回值	选择的 DMA 项

9.2.3.19. adc_voltage_read_by_channel_irq

函数名	adc_voltage_read_by_channel_irq
函数原型	bool adc_voltage_read_by_channel_irq(adc_channel_t channel, adc_sampling_cycles_t sampling_cycles, adc_callback_t cb);
功能描述	ADC 读取中断模式相应通道输入的电压值
输入参数 1	channel：ADC 通道
输入参数 2	sampling_cycles：ADC 采样周期项
输入参数 3	cb：指向结构 adc_callback_t 的指针，cb 中包含 ADC 事件回调
输出参数	对应通道的电压值
返回值	false: fai,true: succ

例：中断模式下读取 ch8 通道的输入电压

```
void adc_test_handler(uint32_t event, int adc_data)
{
    log_debug("adc_data = %d\n", adc_data);
    float ret = adc_voltage_cal_by_sw(adc_data, ADC_SAMPLING_CYCLES_128);
    log_debug("ret = %f\n", ret);
}

adc_voltage_read_by_channel_irq(ADC_CH_8, ADC_SAMPLING_CYCLES_128, adc_test_handler);
```

9.2.3.20. adc_temperature_read_irq

函数名	adc_temperature_read_irq
函数原型	bool adc_temperature_read_irq(adc_sampling_cycles_t sampling_cycles, adc_callback_t cb);
功能描述	ADC 温度通道中断方式读取
输入参数 1	sampling_cycles：ADC 采样周期项
输入参数 2	cb：指向结构 adc_callback_t 的指针，cb 中包含 ADC 事件回调
输出参数	无
返回值	false: fai,true: succ

例：温度通道在中断方式下读取

```
void adc_test_handler(uint32_t event, int adc_data)
{
    log_debug("adc_data = %d\n", adc_data);
    float ret = adc_voltage_cal_by_sw(adc_data, ADC_SAMPLING_CYCLES_128);
    log_debug("ret = %f\n", ret);
}

adc_temperature_read_irq(ADC_SAMPLING_CYCLES_128, adc_test_handler);
```

9.3. ADC 应用例程

9.3.1.单端模式通道使用

下面介绍校准后的单端电压通道的使用。

ADC 通路校准之后，我们要想使用，只需要调用 ADC 单端读取电压函数，输入相应的参数即可。目前函数名为 adc_voltage_read_by_channel。

软件实现：

```
void example_adc(void)
{
    pmu_pin_mode_set(BIT_MASK(8), PMU_PIN_MODE_FLOAT);
    pinmux_config(8, PINMUX_ANALOG_CH4_PIN8_CFG);

    int16_t ret = 0;

    ret = adc_voltage_read_by_channel(ADC_CH_8, ADC_SAMPLING_CYCLES_128);

   log_debug("GPIO2 ch8 = %d\n",ret);
}
```

此代码为单端模式下，选择通道 8（GPIO2）作为输入端，单端的输出结果可有串口打印查看。

9.3.2.VBAT 通道使用

ADC 通路校准之后，我们要想使用，只需要调用 ADC 的 vbat 读取电压函数，就可以读取到 vbat 的值。目前函数名为 adc_voltage_read_by_channel。

软件实现：

```
void example_adc(void)
{
    int16_t ret = 0;

    ret = adc_voltage_read_by_channel(ADC_CH_1, ADC_SAMPLING_CYCLES_128);

    log_debug("vbat = %d\n", ret);
}
```

选择通道 1（vbat）作为输入端，只需调用即可在串口上看到当前 vbat 电压值的打印。

9.3.3.温度通道使用

ADC 通路校准之后，我们要想使用，只需要调用 ADC 读取温度函数，即可得到当前温度值。目前函数名为 adc_temperature_read。

软件实现：

```
void example_adc(void)
{
    int16_t ret = 0;
    ret = adc_temperature_read_normal(ADC_SAMPLING_CYCLES_128);

    log_debug("tem=%d\n", ret);
}
```

调用的初始化以及配置和温度的通道以及封装到函数中，只需入参再选择参考电压，增益和采样周期设置，即可在串口上看到当前温度值的打印。

9.3.4.连续扫描方式的读取

9.3.4.1. 单通道

软件实现：

```
void example_adc(void)
{
    pmu_pin_mode_set(BIT_MASK(8), PMU_PIN_MODE_FLOAT);
    pinmux_config(8, PINMUX_ANALOG_CH4_PIN8_CFG);

    adc_voltage_read_single_begin_nostop(ADC_CH_8, ADC_SAMPLING_CYCLES_
128);

    float ret = adc_voltage_read_single_nostop(ADC_CH_8, ADC_SAMPLING_CYCL
ES_128);

    log_debug("example_adc single channel voltage =%f\n", ret);
}
```

此代码为连续扫描模式下单通道电压值的读取（函数返回电压值，单位：mV），其中，adc_voltage_read_single_begin_nostop()函数内包含 adc 的初始化，启动 adc 的相关配置。adc_voltage_read_by_channel()函数会一直读取通道 ADC_CH_8 (GPIO2)的电压值，因此，单端连续扫描的输出结果可以看此函数的返回值。

9.3.4.2. 多通道

软件实现：

```
#define NOSTOP_CH_ALL 0x3FC
#define VOLTAGE_CH_COUNT 8

float multi_buf[VOLTAGE_CH_COUNT];
void example_adc(void)
{
    adc_voltage_read_multi_begin_nostop(NOSTOP_CH_ALL, ADC_SAMPLING_CY
CLES_128);

    adc_voltage_read_multi_nostop(NOSTOP_CH_ALL, multi_buf, ADC_SAMPLING
_CYCLES_128);

    for (uint32_t i = 0; i < sizeof(multi_buf) / sizeof(multi_buf[0]); i++)
    {
        log_debug("example_adc multi channel voltage[%d] =%f\n", i, multi_buf[i]);
```

```
    }
}
```

此代码为连续扫描模式下多通道的读取（函数返回电压值，单位：mV），adc_voltage_read_multi_begin_nostop ()中的 0x3FC 代表前三个 ADC 采集的通道 ch2~ch9，分别通过 GPIO13、GPIOO8、GPIO9、GPIO10、GPIO11、GPIO12、GPIO2、GPIO3 采集电压。此函数内包含 adc 的初始化，启动 adc 的相关配置。adc_voltage_read_multi_nostop()函数会一直读取通道 ch2~ch9 的电压值，函数的返回值是一个数组 multi_buf，其中，multi_buf[0]存放的是 CH0 的电压值，multi_buf[1]存放的是 CH0 的电压值，multi_buf[2]存放的是 CH0 的电压值。因此，单端多通道的连续扫描的输出结果可以看此函数的返回值。

9.3.5. 中断方式读取

ADC 通路校准之后，我们要想使用中断回调方式读取 ADC 值，只需要调用 ADC 中断读取电压函数，输入相应的参数即可。目前函数名为 adc_voltage_read_by_channel_irq、adc_temperature_read_irq(温度通道中断方式)。

软件实现：

```
void adc_test_handler(uint32_t event, int adc_data)
{
    log_debug("[%s][%d], %d\n", __FUNCTION__, __LINE__, adc_data);
}

void example_adc(void)
{
    adc_voltage_read_by_channel_irq(ADC_CH_8, ADC_SAMPLING_CYCLES_128, adc_test_handler);
}
```

选择通道 8（GPIO2）作为输入端，只需调用以上测试代码例子，即可在串口上看到当前电压值的打印。

9.3.6. ADC 的 DMA 搬运方式读取

ADC 通路校准之后，我们要想使用 DMA 方式读取 ADC 值，只需要调用以下 ADC 的 DMA 搬运方式 demo，输入相应的参数即可。

软件实现：

```
#define BLOCK_SIZE 256
#define ADC_DMA_BUFFER_SIZE 1024
#define BLOCK_NUM (ADC_DMA_BUFFER_SIZE * 4 / BLOCK_SIZE)
DMA_DECLARE_LLIP(adc_dma_block_llip, BLOCK_NUM);
#define ADC_VREF_1P25V 1
```

```
#define ADC_PGA_GAIN_0P125 0xC

static uint32_t adc_dma_buffer[ADC_DMA_BUFFER_SIZE];
static float adc_dma_buffer_float[ADC_DMA_BUFFER_SIZE];
static HS_DMA_CH_Type *adc_dma = NULL;

static void dma_cb(dma_status_t status, uint32_t cur_src_addr, uint32_t cur_dst_addr, uint32_t xfer_size)
{
    log_debug("\r\n");
    if (status == DMA_STATUS_BLOCK_OK)
    {
        log_debug("DMA block transfer succeed, block info: ");
    }
    else if (status == DMA_STATUS_ABORT)
    {
        log_debug("DMA block transfer aborted, current info: ");
    }
    else
    {
        log_debug("DMA block transfer error, current info: ");
    }
    log_debug("status = 0x%08x, src_addr = 0x%08x, dst_addr = 0x%08x, xfer_size = %d\r\n",status, cur_src_addr, cur_dst_addr, xfer_size);
    // fflush(stdout);
}

void adc_dma_test()
{
  adc_channel_t channel = 8;
  uint32_t src_addr;
  const adc_dma_config_t adc_dma_cfg = {
      .use_fifo = false,
      .buffer = adc_dma_buffer,
      .buffer_len = ADC_DMA_BUFFER_SIZE * 4,
      .block_llip = adc_dma_block_llip,
      .block_num = BLOCK_NUM,
      .callback = dma_cb, // NULL,
    };

  memset(adc_dma_buffer, 0, ADC_DMA_BUFFER_SIZE * 4);

  log_debug("adc dma test start\n");
```

```
    dma_init();

    /* open adc clk */
    adc_open();

        // 0x400A2008
        REGW(&HS_GPADC->DLY_CFG, MASK_3REG(GPADC_CFG_CHG_DLY, 0x50,
    GPADC_PD_DLY_MAX, 0x50, GPADC_ORB_DLY, 0x14));

        // 0x400A2014
        REGW(&HS_GPADC->ADC_CFG1, MASK_5REG(GPADC_AUTO_PD1, 0x1, G
PADC_AUTO_PD2, 0x1, GPADC_TRIG_RES, 0x1, GPADC_AVG_CFG, 0x4, GPAD
C_DIGI_CLK_FREQ, 0x1F));

        // 0x400A2018  set the data direction , vector to be some channel,life to be
endless
        REGW(&HS_GPADC->ADC_CFG2, MASK_3REG(GPADC_SCANDIR, 0x0, GPA
DC_SEQ_VECT, 1 << channel, GPADC_SEQ_LIFE, 0x0));

        // 0x400A2020
        REGW(&HS_GPADC->ADC_SW_TRIGGER, MASK_2REG(GPADC_SW_TRIGG
ER_TRUE, 0x1, GPADC_SW_TRIGGER, 0x1));

        /* set adc pga gain */
        REGW(&HS_AUDIO->CODEC_ANA_CTRL_1, MASK_1REG(AU_AU_PGA_GAI
N, ADC_PGA_GAIN_0P125));

        /* set analog sampling cycles */
        REGW(&HS_GPADC->ADC_CFG1, MASK_1REG(GPADC_GP_BIT_SEL, ADC_
SAMPLING_CYCLES_128));

        // adc channel cfg
        REGW(((uint32_t *)&HS_GPADC->CH_0_CFG) + channel, MASK_1REG(GPAD
C_MODE_VERF, ADC_VREF_1P25V));

        // 0x400A205C
        REGW(&HS_GPADC->SMP_OUT_CTL, MASK_1REG(GPADC_AUTO_COMPE
N, 0));

        adc_dma = adc_dma_config((HS_DMA_CH_Type *)NULL, &adc_dma_cfg, &src
_addr);
        dma_start(adc_dma, src_addr, (uint32_t)adc_dma_buffer, ADC_DMA_BUFFER_
SIZE);
```

```
    // 0x400A2010
    REGW(&HS_GPADC->ADC_CFG0, MASK_1REG(GPADC_ADC_START, 0x1));

    // wait dma stop
    dma_wait_stop(adc_dma);

    // stop adc convert
    HS_GPADC->ADC_CFG0 = GPADC_ADC_STOP_MASK;

    // close clk of adc
    adc_close();

    // stop adc dma
    dma_stop(adc_dma);

    log_debug("adcDmaData:\n");
    for (uint32_t i = 0; i < ADC_DMA_BUFFER_SIZE; ++i)
    {
        if ((i > 0) && (0 == (i % 8)))
        {
            log_debug("\n");
        }
        adc_dma_buffer_float[i] = adc_voltage_cal_by_sw(adc_dma_buffer[i], ADC_
SAMPLING_CYCLES_128);

        log_debug("%f ", adc_dma_buffer_float[i]);
    }
    log_debug("\n");
}
```

此部分代码为 DMA 方式下 ADC 通道 CH8 电压值的读取（函数返回电压值，单位：mV），其中， adc_dma_test 函数内包含 ADC 的初始化，启动 ADC 的相关配置，而且会一直读取通道 ADC_CH_8 (GPIO2)的电压值，可在串口中看大 DMA 搬运打印的值。

9.4. ADC 使用注意事项

- 使用 GPADC 采样时，VBAT 供电的大小要大于或等于输入信号的大小，否则得到的会不准确；
- 不同 sampling cycles 情况下的输入信号范围都是 0.1V 到 3.5V；
- 不同 sampling cycles 情况下，精度不同：32 个 cycles 为 8bit，64 个 cycles 为 10bit；128 个 cycles 为 11bit。
- GPADC 的阻抗为无穷大。
- 使用 GPADC 时，如果板子上 VDD_AUDIO 连接电容，AU_EN_LDO28_OFFCHIPCAP

这个寄存器需要置 1，否则置 0。

- 当 VDD_AUDIO 连接电容时，由于电容的充放电，VDD_AUDIO 的启动时间也会发生变化，因此当电压波动比较大时，是因为 VDD_AUDIO 还没完全启动，就开始采集电压，此时需要在 VDD_AUDIO 开启之后加延时，根据电容的大小，延时会不同，可以根据示波器抓取 VDD_AUDIO 的启动时间，来设置延时时间。
- GPADC 和 AUDIO ADC 共用同一电路，因此两个外设功能不能同时开启。

10. OTP

11. RANDOM

11.1. 简介

RANDOM 模块是用来产生硬件随机数的电路。RANDOM 时钟与 apb_clk 同频。

RANDOM 对应的基地址为：

模块	基地址
RANDOM	0x40004000

注意：RANDOM 在代码中被宏定义为 HS_RANDOM_BASE。

11.2. API 介绍

11.2.1. RANDOM 寄存器结构

```
typedef struct
{
    __IO uint32_t RANDOM;
} HS_RANDOM_Type;
```

Offset	寄存器	描述
0x00	RNG_RND	硬件随机数寄存器

RNG_RND address offset: 0x0000

Bit	R/W	Reset	Name	Description
31:0	R	0x0	RNG_RND	硬件随机数值

11.3. RANDOM 应用例程

11.3.1. 例程 1：产生随机数

软件实现：

```
/**
 ********************************************************************************************
 * @brief Function to initialize the random seed.
 * @param[in] seed The seed number to use to generate the random sequence.
 ********************************************************************************************
```

```
 */
void random_init(uint32_t seed)
{
    srand(seed);
}

/**
 ****************************************************************************************
 * @brief Function to get an 8 bit random number.
 * @return Random byte value.
 ****************************************************************************************
 */
uint8_t rand_byte(void)
{
    return (uint8_t)(rand() & 0xFF);
}

/**
 ****************************************************************************************
 * @brief Function to get an 16 bit random number.
 * @return Random half word value.
 ****************************************************************************************
 */
uint16_t rand_hword(void)
{
    return (uint16_t)(rand() & 0xFFFF);
}

/**
 ****************************************************************************************
 * @brief Function to get an 32 bit random number.
 * @return Random word value.
 ****************************************************************************************
 */
uint32_t rand_word(void)
{
    return (uint32_t)rand();
}

void random_data_get(void)
{
    uint8_t randDataByte[6];
    uint16_t randDataShort[6];
    uint32_t randDataWord[6];
```

```
    uint8_t i = 0;

    for( i=0; i<6; i++)
    {
        randDataByte[i] = rand_byte();
        randDataShort[i] = rand_hword();
        randDataWord[i] = rand_word();
    }

    log_debug_array_ex("random byte data is\r\n",randDataByte, sizeof(randDataByte));
    log_debug_array_ex("random hword data is\r\n",randDataShort,
sizeof(randDataShort));
    log_debug_array_ex("random word data is\r\n",randDataWord,
sizeof(randDataWord));
}

/*****************************************************************
 * EXTERN FUNCTIONS
 */
 /**
  ****************************************************************************************
  * @brief Test case according to the test number
  ****************************************************************************************
  */
void rand_test_case(uint8_t* pCmd)
{
    uint32_t case_num = atoi((const char *)pCmd);

    switch (case_num)
    {
        case RAND_TEST_CASE_330100:
            log_debug("RAND_TEST_CASE_330100\n");

            random_data_get();

            log_debug("RAND_TEST_CASE_330100_end\n");
            break;

        default:
            log_debug("[%s][%d][The input test number is not support!]\n",
__FUNCTION__, __LINE__);

            break;
    }
```

```
}

/*****************************************************************
 * MAIN FUNCTIONS
 */
int main(void)
{
    uint8_t         cmd[SHELL_FIFO_SIZE];
    uint8_t*        pCmd = &cmd[0];
    volatile bool ret = false;

    // Enble all IRQ quikly.
    __set_PRIMASK(0);

    // Disable WDT.
    wdt_enable(0);

    // Init interactive UART.
    uart_interaction_init();

    shell_init(HS_UART1);

    uint32_t topClk = cpm_get_clock(CPM_TOP_CLK);
    uint32_t cpuClk = cpm_get_clock(CPM_CPU_CLK);

    log_debug("BL1824X_Chip Test, topClk =%d, cpuClk=%d \n",topClk,cpuClk);

        random_init(HS_RANDOM->RANDOM);

    while (1)
    {
        ret = shell_get_cmd(&pCmd);

        if (ret == true)
        {
            log_debug("\r\ncmd : %s.\n", cmd);

            rand_test_case(pCmd);
        }
    }

    shell_uninit();

    return 0;
```

```
}
```

rand()函数是 C 语言中用来产生一个 0-0x7fff 的随机数的函数。srand()函数是随机数发生器的初始化函数。rand()函数每次调用前都会查询是否调用过 srand(seed)，是否给 seed 设定了一个值，如果有那么它会自动调用 srand(seed)一次来初始化它的起始值。若之前没有调用 srand（seed），那么系统会自动给 seed 赋初始值，即 srand（1）自动调用它一次。

srand()函数需要提供一个种子，如果 seed 种子相同，rand()产生的随机数将是相同的。因此，为了使随机数随机化，我们可以使用系统时间作为种子，然后再传给 srand 函数；也可以使用硬件随机数寄存器 RANDOM 中的硬件随机数作为种子传进去。本应用例程使用了后者。

11.4. RANDOM 使用注意事项

- RANDOM 寄存器产生随机数的频率为 32KHz，通常不直接使用寄存器中的值作为随机数，而是作为随机数种子，配合 srand()与 rand()使用；
- rand()函数每次调用前都要初始化随机数种子 srand(seed)；

12. AES

12.1. 简介

AES 算法是当前最流行的对称加密算法，也是一种分组加密算法，分组密码就是把明文分为固定长度的一组一组，每次加密一组数据，直到加密完整个明文数据。AES 算法根据分组长度可以分为 AES-128，AES-192，AES-256，其所要求的秘钥长度和加密轮数也各不相同。但是这三种模式的算法在本质上没有区别，我们的库支持 AES-128。

12.2. AES 应用例程

12.2.1. 例程 1： AES 加密

软件实现：

```
#define AES_TEST_NUM 16

/*****************************************************************
 * LOCAL FUNCTIONS
 */
static uint8_t aes_data_in[AES_TEST_NUM] =
{0x00,0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88,0x99,0xaa,0xbb,0xcc,0xdd,0xee,0xff
};
static uint8_t aes_key[AES_TEST_NUM] =
{0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0
f};
static uint8_t aes_data_out_ideal[AES_TEST_NUM] =
{0x69,0xc4,0xe0,0xd8,0x6a,0x7b,0x04,0x30,0xd8,0xcd,0xb7,0x80,0x70,0xb4,0xc5,0x5
a};
static uint8_t aes_data_out_real[AES_TEST_NUM];

void aes_128_test(void)
{
    bool status_fail = false;
    uint8_t i = 0;

    aes_cipher(aes_key,aes_data_in,aes_data_out_real);
```

```
    for( i=0; i<AES_TEST_NUM; i++ )
    {
        if( aes_data_out_real[i] != aes_data_out_ideal[i] )
        {
            status_fail = true;
            break;
        }
    }

    if( status_fail )
        log_debug("Fail!!!!!! aes_128_test\r\n");
    else
        log_debug("Pass****** aes_128_test\r\n");
}

/*****************************************************************
 * PUBLIC FUNCTIONS
 */
int main(void)
{
    hardware_init();
    ble_stack_config();

    om_ble_enable(NULL);

    log_debug("running %d\n", pmu_reboot_reason());

    log_debug("AES_TEST_CASE_330200\n");

    aes_128_test();

    log_debug("AES_TEST_CASE_330200_end\n");

    while(1);

    // Enter main loop.
    co_sche();

    return 0;
}
```

在这个例程中，最核心的函数是 aes_cipher()，这是利用 aes 来加密的加密函数，而它的入参有三个：第一个是 aes_key，也就是密钥，用来加密明文的密码，在对称加密算法中，加密与解密的密钥是相同的；第二个是 aes_data_in，也就是明文，没有经过加密的数据；

第三个是 aes_data_out_real，也就是密文，经加密函数处理后的数据。

12.3. AES 使用注意事项

- AES 加密功能需要在 om_ble_enable(NULL)之后使用；

13. Sleep

13.1. 简介

BL1824X 有 4 种电源管理模式：Active mode（主动模式）,Idle mode（空闲模式）, Sleep mode（睡眠模式）, Deep sleep mode（深度睡眠模式）。

在有源模式和空闲模式下，数字模块(Timer, UART, SPI, PWM…)的时钟可以独立使能或关闭。具有独立电源 dom 的模拟模块的功率也可以通过应用程序来开启或关闭。在空闲模式下，处理器的时钟是门控的，所有的中断都可以唤醒系统。在休眠模式下，GPIO 中断、休眠定时器中断和 BLE 中断可以唤醒系统。在深度睡眠模式下，只有 GPIO 的中断才能唤醒系统。BLE 栈在深度睡眠模式下不工作。唤醒将触发系统重新启动。

BL1824X4 种模式的区别如下表所示：

Mode	CPU Clock	CPU Power	Wakeup Source	Have Power Modules
Active	ON	ON	N/A	All Modules (**)
Idle	OFF	ON	All Interrupts	All Modules (**)
Sleep	OFF	OFF	GPIO, Sleep-Timer	SRAM, GPIO, 32K, BASEBAND
Deep Sleep	OFF	OFF	GPIO	SRAM, GPIO, 32K

表 13.1 BL1824X 四种模式的区别说明

13.2. 功能描述

13.2.1. Deep_sleep 配置

13.2.1.1. 广播后睡眠配置

1. 打开睡眠

co_power_sleep_enable(true);

2. 关掉定时器

co_timer_set(&simple_timer, 1000, TIMER_REPEAT, simple_timer_handler, NULL);

3. 设置 timeout(广播时间)

```
const static struct adv_param app_adv_map[] = {
/*adv data    adv dta len    scan_data     scan_data_len intervel    timeout    type*/
{
```

```
    app_adv_data, sizeof(app_adv_data), NULL, 0, 0x80, 0x60,
    BLE_GAP_ADV_TYPE_ADV_IND},
};
```

如上设置完毕后，广播 60S 后进入 Deep_sleep。

13.2.1.2. 直接进入睡眠

1. 打开睡眠

```
co_power_sleep_enable(true);
```

2. 关掉定时器

```
co_timer_set(&simple_timer, 1000, TIMER_REPEAT, simple_timer_handler, NULL);
```

3. 关掉 BLE

```
//      const static ble_gap_addr_t gap_addr = {
//      BLE_GAP_ADDR_TYPE_RANDOM_STATIC, { 0x0d, 0xBF, 0x66, 0x21,
0xD0, 0xdd}
//      };
//      app_adv_restart(&gap_addr);
```

如上设置后，CPU 直接进入 Deep_sleep。

13.2.2. Sleep 配置

13.2.2.1. 不开 BLE 进入 sleep

1. 打开睡眠

```
co_power_sleep_enable(true);
```

2. 打开定时器设置定时器时间为 2 秒

```
co_timer_set(&simple_timer, 2000, TIMER_REPEAT, simple_timer_handler, NULL);
```

3. 关掉 BLE

```
//      const static ble_gap_addr_t gap_addr = {
//      BLE_GAP_ADDR_TYPE_RANDOM_STATIC, {0x0d, 0xBF, 0x66, 0x21,
0xD0, 0xdd}
//      };
//      app_adv_restart(&gap_addr);
```

配置后，在定时器唤醒 2S 之间进入 Sleep。

13.2.2.2. 开 BLE 进入睡眠

1. 打开睡眠

co_power_sleep_enable(true);

2. 关掉定时器

co_timer_set(&simple_timer, 1000, TIMER_REPEAT, simple_timer_handler, NULL);

3. 设置 intervel(广播间隔)

```
const static struct adv_param app_adv_map[] = {
/*adv data   adv dta len   scan_data   scan_data_len intervel   timeout   type*/
{ app_adv_data, sizeof(app_adv_data), NULL, 0, 0xC80, 0x00,
BLE_GAP_ADV_TYPE_ADV_IND},
};
```

配置后，在广播唤醒之间为 Sleep。

13.3. 使用注意事项

13.3.1. 外设最小功率

在 Sleep 和 Deep_sleep 模式下，各个外设需要最小功率大于 Sleep 和 Deep_sleep 的功率，会导致各个外设失效，下表是各个模块最小功率对应的电源模块(GPIO 模块由两部分组成，一部分位于 PMU 模块中，另一部分不属于 PMU，不在休眠模式下工作)。

Modules	Minimum Requested Power Mode
DMA	Active
ADC	Idle
CPM	Idle
SF	Idle
SPI	Idle
Timer	Idle
UART	Idle
WDT	Idle
Pinmux	Idle
Baseband	Sleep
GPIO (**)	Idle
GPIO(in PMU)	Deep Sleep
PMU	Deep Sleep

表 13.2 不同睡眠模式下外设模块状态

13.3.2. 外设寄存器

当系统从休眠模式唤醒时，部分外设的寄存器设置全部丢失，需要重新配置外设。这些重新配置的外设包括:GPIO、Timer、UART、SPI、WDT、DMA、ADC。

13.3.3. 代码调试

当系统进入 Sleep 模式时，CPU 下电。因此，调试接口(SWD)将被断开，KEIL 将失去连接。函数 co_power_sleep_enable()可以禁用 Sleep 模式，以便更好地调试。