泰芯 TXW830x AH-SDK 开发指南

泰芯半导体
TaiXin Semiconductor

珠海泰芯半导体有限公司
TaiXin Semiconductor Co., Limited

修订记录

日期	版本	描　　述	修订人
2023-1-11	V2.1	增加 IIC 和软件 timer 的描述；	CWY/WY
2022-12-7	V2.0	增加外设和内部功能的描述； 修改文档名称为开发指南；	CWY/WY
2022-8-8	V1.2.1	修改 logo；	LSQ
2022-8-5	V1.2	增加 CDK 的 FTP 链接；	WY
2021-7-1	V1.1	增加 CKLink 介绍；	WY
2021-6-30	V1.0	初始版本；	WY

目录

1. 概述

本文介绍了泰芯 AH 的 SDK 开发环境和开发说明。

2. 开发环境

2.1. CKLink 介绍

CKlink lite 是泰芯 AH 的 SDK 的调试工具,可以在平头哥的淘宝官方店购买。

在淘宝首页搜索 CLLINK 即可搜到，购买链接为：

https://item.taobao.com/item.htm?spm=a230r.1.14.16.7e7b1b4c0aMKXb&id=526225414550&ns=1&abbucket=3#detail

图 2-1 CKLink lite

图 2-1 中的转接小板和调试线可以找我司 FAE 获取。

如果修改点很小，也可以不用在线调试，直接用串口升级固件看运行结果，就可以不用购买 CKLink。

2.2. CDK 安装说明

CDK 是平头哥 CPU 的集成开发环境，可以从泰芯 FTP 服务器下载。

下载地址：183.47.14.74；端口：21；账户 ：txguest；密码：txguest。

也可以从官方网站进行下载安装。下载地址为：
https://occ.t-head.cn/community/download?id=575997419775328256
如果下载不了请联系我司 FAE。

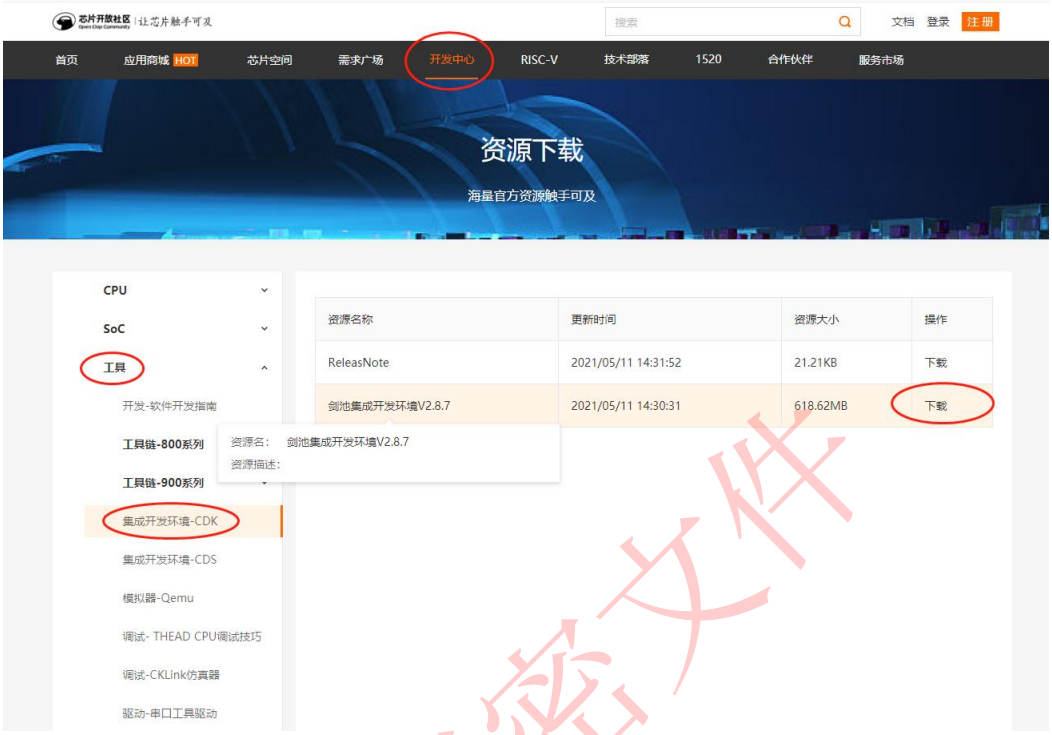


图 3-1 CDK 下载

另外，安装时请先把 CKLink 插在电脑上，CDK 安装过程中会自动安装 CKLink 的驱动。

2.3. CDK 编译&调试

CDK 安装完成后，打开 hgSDK 解压后的 project/txw4002a.cdkws 文件，hgSDK 的工程编译界面如下图所示，点击编译按钮或按 F7 进行编译。

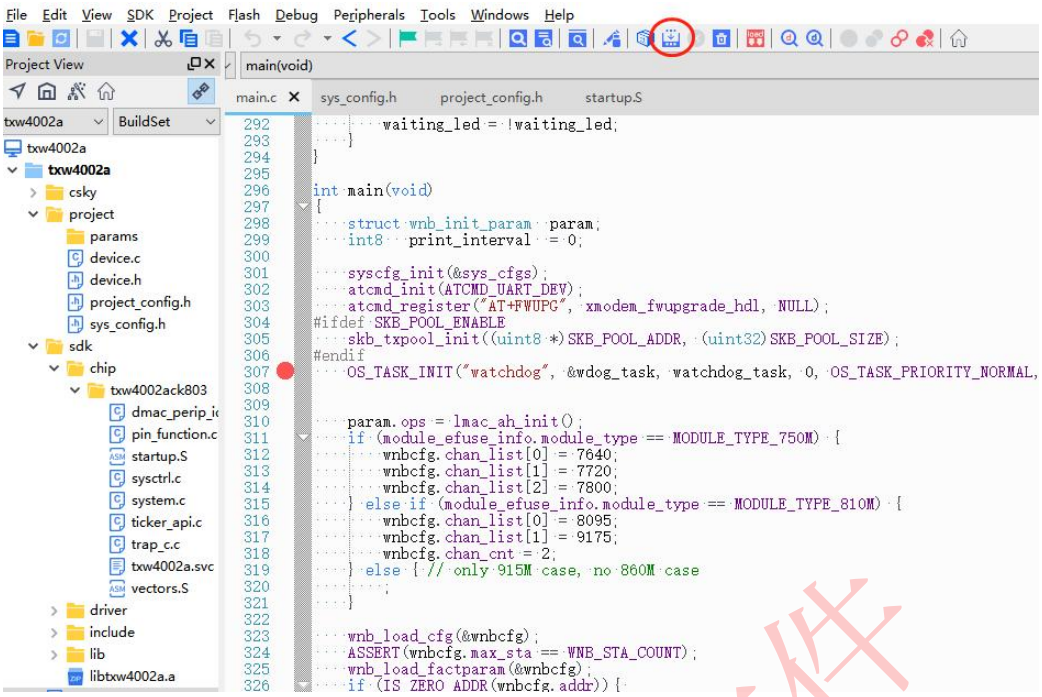


图 4-1 hgSDK 界面

```
size of target:
   text    data     bss     dec     hex filename
 286052    6784   23824  316660   4d4f4 ./Obj/txw4002a.elf
checksum value of target:0x9E726897 (454,656)
Executing Post Build commands ...

D:\work\hgSDK-v1.3.2.5-12097\project>cd /d D:\work\hgSDK-v1.3.2.5-12097\project\

D:\work\hgSDK-v1.3.2.5-12097\project>copy .\Obj\txw4002a.ihex project.hex
ÒÑ¸´Ö€         1 ¸öÎÄ¼þ¡£

D:\work\hgSDK-v1.3.2.5-12097\project>copy ..\..\..\..\tools\makecode\BinScript.exe BinScript.exe
Ïµí³ÕÒ²»µ½Ö¸¶¨µÄ·¾¶¡£

D:\work\hgSDK-v1.3.2.5-12097\project>copy ..\..\..\..\tools\makecode\crc.exe crc.exe
Ïµí³ÕÒ²»µ½Ö¸¶¨µÄ·¾¶¡£

D:\work\hgSDK-v1.3.2.5-12097\project>copy ..\..\..\..\tools\makecode\makecode.exe makecode.exe
Ïµí³ÕÒ²»µ½Ö¸¶¨µÄ·¾¶¡£

D:\work\hgSDK-v1.3.2.5-12097\project>BinScript.exe BinScript.BinScript
BinScript.exe v1.0.3
Successfully output a BIN file: huge-ic-ah.bin
Successfully output a BIN file: param.bin

D:\work\hgSDK-v1.3.2.5-12097\project>makecode.exe
makecode.exe v1.1.1
successfully maked huge-ic-ah_v1.3.2.5-12097_2021.6.23_.bin

Done
====0 errors, 0 warnings, total time : 19s916ms====
```

图 4-2 编译生成固件

固件编译生成后，如果需要在线调试，可以使用 CKLink 下载运行调试。

调试前需要进行一些设置。

需要将 debug level 修改为 Default(-g)，然后重新编译固件，如图 4-3 所示。

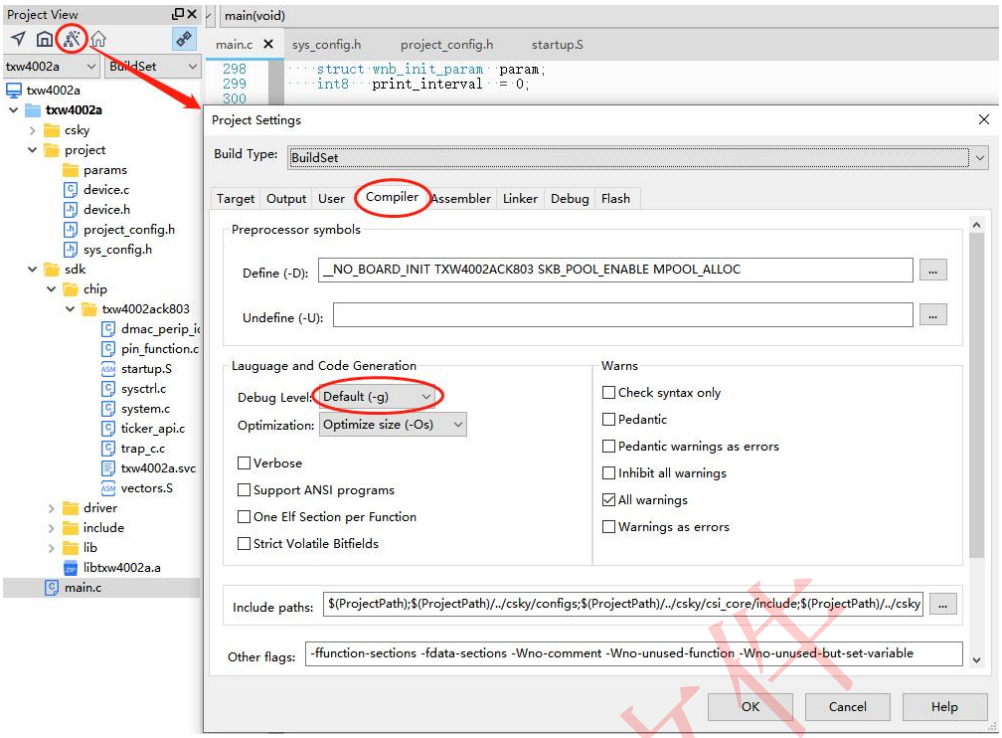


图 4-3 Compiler 页面修改 debug level

需要设置 ICE Clock，如图 4-4 所示。

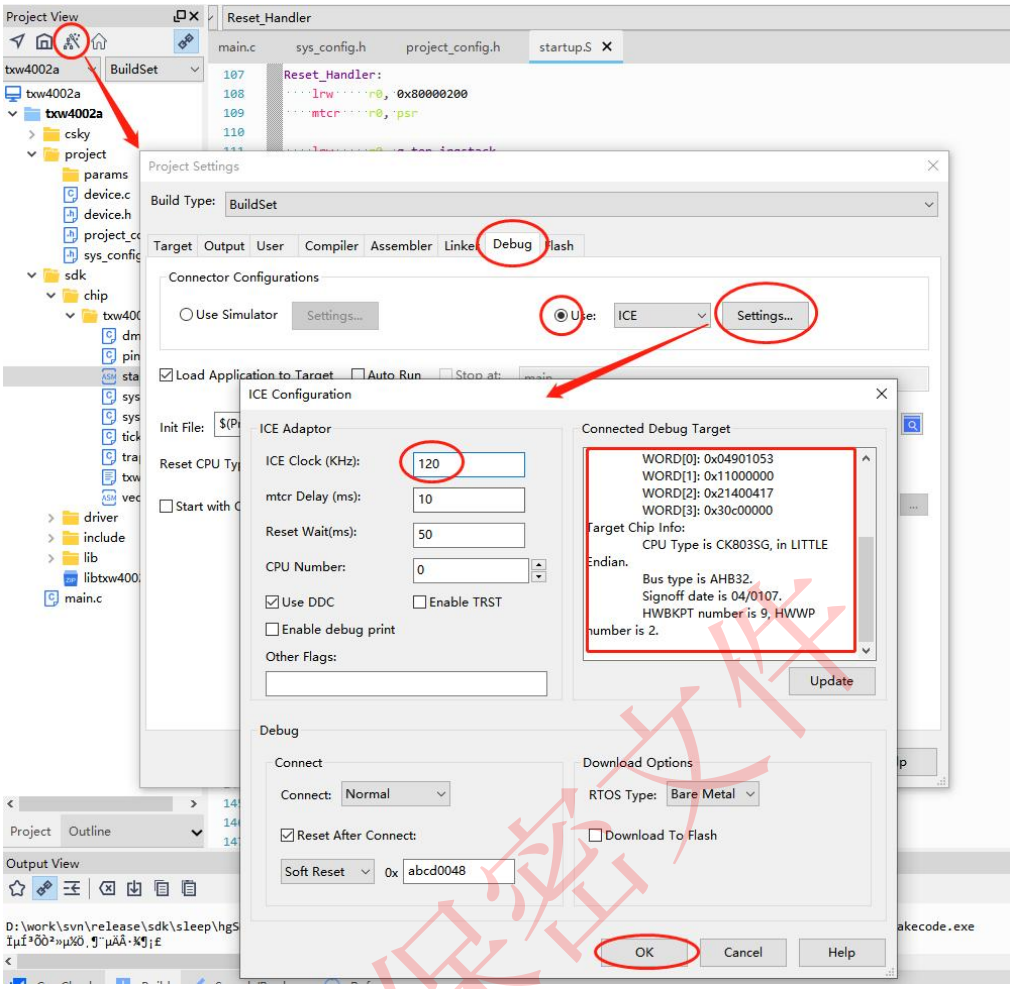


图 4-4 debug 页面设置 ICE Clock

（1）进入 project setting / debug，选中 Use ICE；

（2）进入 setting，如果调试板可以正确被连上，在 Connected Debug Target 框里就可以读到 Target chip Info；

（3）设置 ICE Clock 为 120Khz（因为由于 CDK 的问题，调试器设置频率更高时调试会有问题；虽然设置为 120khz 下载速度会比较慢，但是调试时没有问题），然后点击 OK 保存。

设置好后，将 CKLink 的调试线连到板子的调试口，点击 CDK 的调试按钮或 CTRL+F5，CDK 会开始下载固件。如图 4-5 所示。

图 4-5 调试按钮

固件下载完成后，CDK 会暂停在 Reset_handler 入口位置等待按 F5 运行。

另外，固件默认开启了 watchdog，在调试过程中停到断点处容易触发 watchdog 复位。可以在 main 函数开启 watchdog 位置设置断点，然后右键跳转到下一行代码运行，使得 watchdog enable 无效，如图 4-6 所示。

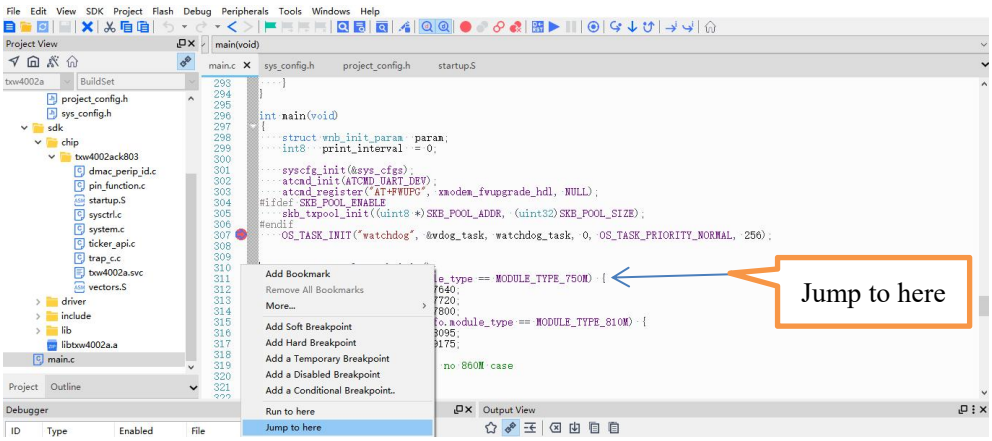


图 4-6 跳过 watchdog 开启的 task

2.4. 固件烧录

编译完成后提示生成固件，可以使用烧录工具(CSKYFlashProgrammer)进行烧录；如果板子已有固件，可以通过串口升级（at+fwupg）。升级的具体方法请咨询我司 FAE。

烧录的工具包为：CSKY-FlashProgrammer-windows-V1.0.6-20200115-1549，解压后如图 5-1 所示。

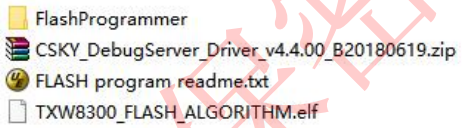


图 5-1 烧录工具包

先安装 CSKY_DebugServer_Driver，然后进入 FlashProgrammer 文件夹，双击运行”CSKYFlashProgrammer.exe”。

选中 Advance 子页，设置 Program Algorithm File，如图 5-2 所示。在上级目录里找到 TXW8300_FLASH_ALGORITHM.elf 。

图 5-2 选中烧录算法

回到主页面，增加 bin 文件烧录

图 5-3 增加 bin 烧录选项

选择要烧录的固件，选中 program 和 verify，点击 start 进行烧录，如图 5-4 所示。

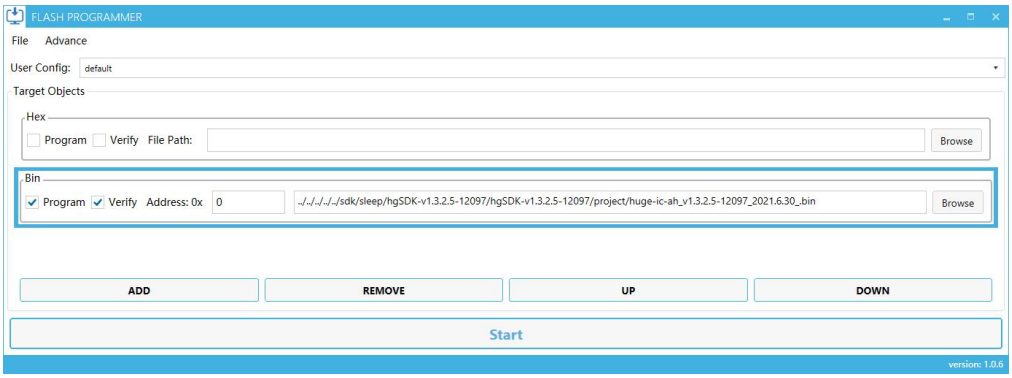


图 5-4 烧录固件

3. 基本开发

3.1. 外设说明

3.1.1. GPIO

- 设置 GPIO 输出模式，输出 1

```
gpio_set_dir(PA_6, GPIO_DIR_OUTPUT);
gpio_set_val(PA_6, 1);
```

- 设置 GPIO 输出模式，输出 0

```
gpio_set_dir(PA_6, GPIO_DIR_OUTPUT);
gpio_set_val(PA_6, 0);
```

- 设置 GPIO 输入模式，下拉 100k

```
gpio_set_dir(PA_6, GPIO_DIR_INPUT);
gpio_set_mode(PA_6, GPIO_PULL_DOWN, GPIO_PULL_LEVEL_100K);
```

- 设置 GPIO 输入模式，上拉 100k

```
gpio_set_dir(PA_6, GPIO_DIR_INPUT);
gpio_set_mode(PA_6, GPIO_PULL_UP, GPIO_PULL_LEVEL_100K);
```

- 设置 GPIO 输入模式，下拉 10k，IRQ

```
static void my_gpio_irq(int32 data, enum gpio_irq_event event)
{
    switch (event) {
        case GPIO_IRQ_EVENT_RISE:
            // Do something
            break;
        case GPIO_IRQ_EVENT_FALL:
            break;
        default:
            break;
    }
```

```
}
```

3.1.2. ADC

ADC 功能实际为 PA12 的特殊功能。

- 设置 GPIO 输入模拟功能，ADC 采样

```
gpio_set_dir(PA_12, GPIO_DIR_INPUT);
gpio_set_mode(PA_12, GPIO_PULL_UP, GPIO_PULL_LEVEL_NONE);
gpio_anolog(PA_12, 1);
// 获取 PA12 的 ADC 采样值，单位 V，浮点数
printf("\r\nio_input=%f\r\n",io_input_meas());
// 获取 PA12 的 ADC 采样值，单位 mV，整数
printf("\r\nio_input=%d\r\n",io_input_meas_mv());
```

3.1.3. 硬件 TIMER

当对定时的精度要求比较高（100uS 级别），并且计时不太长的时候（不超过 85mS），可以使用硬件 timer（TIMER1 和 TIMER3）。以 TIMER1 为例说明：

- 定义 TIMER1_ENABLE_MS，设置为 5ms 中断（project\project_config.h）：

```
#define TIMER1_ENABLE_MS        5
```

范围 1~85mS；如果超过需要用软件 timer；

- TIMER1 的初始化函数（project\main.c）

```
timer1_init(uint32 tmo_ms, enum timer_type type, uint32 cb_data);
//tmo_ms：TIMER 超时中断时间
//type：TIMER 的类型（单次中断或者循环中断）
//cb_data：TIMER 中断处理函数的参数
```

- TIMER1 中断处理函数（project\main.c）

```
timer1_cb(uint32 args);
//args：初始化时设置的中断处理函数参数
```

- TIMER1 停止函数

```
timer_device_stop((struct timer_device *)dev_get(HG_TIMER1_DEVID));
```

3.1.4. 软件 TIMER

当对定时的精度要求不太高（mS 级别），并且计时比较长的时候（超过 85mS），可以使用软件 timer。

- TIMER 的初始化函数

```
int os_timer_init(struct os_timer *timer, os_timer_func_t func,
                  enum OS_TIMER_MODE mode, void *arg)
//timer：TIMER 结构体
//func：定时器中断处理函数
//mode：TIMER 的类型（单次中断或者循环中断）
//arg：TIMER 中断处理函数的参数
```

- TIMER 启动函数

```
int os_timer_start(struct os_timer *timer, unsigned long expires)
//timer：TIMER 结构体
//expires：定时器超时中断时间，单位为 ms
```

- TIMER 中断处理函数（project\main.c）

```
static void test_timer_cb(void *args)
{
    os_printf("test_timer_cb\r\n");
}
//args：初始化时设置的中断处理函数参数
```

- TIMER 停止函数

```
int os_timer_stop(struct os_timer *timer)
```

- 示例代码：

```
struct os_timer test_tm;

//定时器中断处理函数
static void test_timer_cb(void *args)
{
```

```
    os_printf("test_timer_cb\r\n");
}

//定时器初始化
os_timer_init(&test_tm, test_timer_cb, OS_TIMER_MODE_PERIODIC, 0);

//启动定时器，单位为 ms
os_timer_start(&test_tm, 1000);
```

3.1.5. UART

可以用 UART0 来与外设通信。注意，UART1 固定用来打印调试，不用于二次开发的通信使用。

硬件连接：IOA10—RX，IOA11—TX

- 打开 IOT_DEV_UART 宏定义（project\project_config.h），设置串口参数、超时发送时间：

```
#define IOT_DEV_UART
#define UART_IOT_BAUDRATE       115200
#define UART_IOT_PARITY         UART_PARIRY_NONE
#define UART_IOT_DATABITS       UART_DATA_BIT_8
#define UART_IOT_STOPBITS       UART_STOP_BIT_1
#define UART_IOT_TIMER_MS       50
```

- 发送函数：

```
void uart_send(uint8 *data, uint32 len);
```

- 接收函数（project\main.c）：

```
void uart_recive(uint8 *data, int32 len)
{
    //os_printf("uart recv data:%s\r\n", data);
}
```

3.1.6. SPI master

硬件连接：

PIN_SPI1_CS —— PA6,
PIN_SPI1_CLK—— PA7,
PIN_SPI1_IO0_SDO ——PA8,
PIN_SPI1_IO1_SDI —— PA9,
PIN_SPI1_IO2 —— PA10,
PIN_SPI1_IO3 —— PA11,

- 打开 IOT_DEV_SPI 宏定义（project\project_config.h），设置时钟频率，时钟模式

```
#define IOT_DEV_SPI
#define SPI_IOT_CLK                1000000
#define SPI_IOT_CLK_MODE        SPI_CPOL_0_CPHA_0
```

- 发送函数

```
void spibus_master_write(uint8 *buff, int32 len);
```

- 接收函数（project\main.c）：

```
void spi_receive(uint8 *data)
{
    os_printf("spi recv data:%s\r\n", data);
}
```

3.1.7. I2C master

这里的 I2C 是用软件模拟的，GPIO 可以选择任意闲置的 IO。

- 打开 IOT_DEV_I2C 宏定义（project\project_config.h）

```
#define IOT_DEV_I2C
```

- 引脚自定义（sdk\lib\mac_bus\i2c_iot.c）：

```
#define I2C_GPIO_SCL                PA_7 //以 PA_7 和 PA_8 举例
#define I2C_GPIO_SDA                PA_8
#define I2C_DELAY                   (3) //时钟周期控制，单位 um
#define SLAVE_ADDRESS               0x3C
```

- 写函数（sdk\lib\mac_bus\i2c_iot.c）

```
//写寄存器
int32 i2c_gpio_write_reg(uint8 slave_addr, uint8 reg, uint8 *data, uint32 size)
//slave_addr:从机地址
//reg:寄存器地址
//data:数据
//size:数据大小

//写数据
int32 i2c_gpio_write_data(uint8 slave_addr, uint8 *data, uint32 size)
```

- 读函数（sdk\lib\mac_bus\i2c_iot.c）：

```
//读寄存器
int32 i2c_gpio_read_reg(uint8 slave_addr, uint8 reg, uint8 *data, uint32 size)
//slave_addr:从机地址
//reg:寄存器地址
//data:数据
//size:数据大小

//读数据
int32 i2c_gpio_read_data(uint8 slave_addr, uint8 *data, uint32 size)
```


3.2. 模拟 RTC

模组利用 AP 发送的 Beacon 包中时间戳实现 RTC 功能。

- lmac_set_rtc(ops, rtc)设置 ms 单位的 24 小时内时间
- lmac_get_rtc(ops)返回 ms 单位的 24 小时内时间

```
struct wireless_nb *wnb = sysvar(SYSVAR_ID_WIRELESS_NB);
uint32 rtc_ms;
lmac_set_rtc(wnb->ops, 0);
rtc_ms = lmac_get_rtc(wnb->ops);
```

3.3. 自定义驱动数据接口

驱动尽管已经包含大部分模组常用功能，但用户仍可在 SDK 中进行二次开发，自定义所需的驱动数据，如：IO 状态，外设状态等...

- __weak int32 customer_driver_data_proc(uint8 *data, uint32 len); 自定义驱动数据处理接口
- int32 customer_driver_data_send(uint8 *data, uint32 len); 自定义驱动数据发送接口

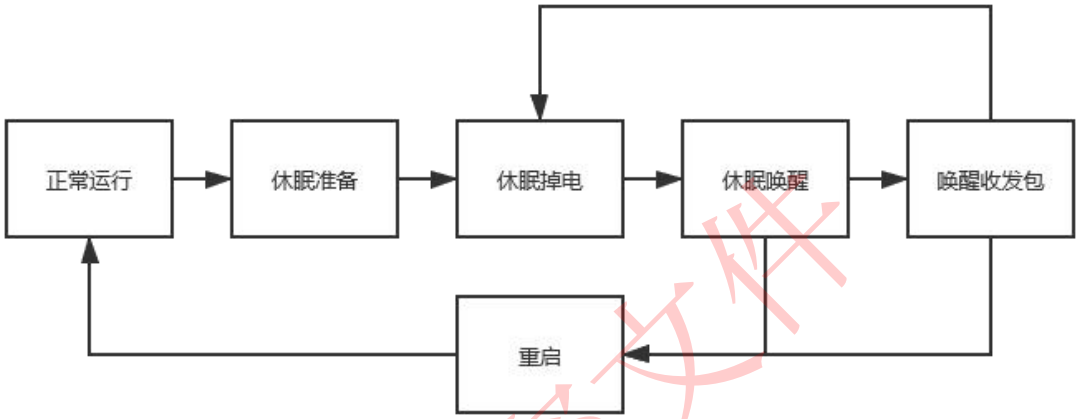
用户在驱动中通过命令 HGIC_CMD_SET_CUST_DRIVER_DATA 或者通过读写/proc/hgic/cust_driverdata 文件发送自定义驱动数据到模组

模组中通过 customer_driver_data_proc 获取到所发送的数据，用户可自定义数据结构内容，根据不同需要在模组处获取信息，最后通过 customer_driver_data_send 将获取的信息内容发回主控驱动，驱动在自行解析。

4. 休眠唤醒相关

4.1. 功能介绍

4.1.1. 休眠的不同阶段



休眠状态

正常运行 ：RF 收发及各外设正常工作，功耗最高
休眠准备 ： 逐步关闭 RF 收发和外设，功耗降低，可插入执行用户代码
休眠掉电 ：RF 收发及外设关闭，功耗最低，等待定时唤醒或 IO 唤醒
休眠唤醒 ： 休眠掉电期间可以通过设定的唯一唤醒 IO 或者设定的定时时间唤醒，唤醒后可插入执行用户代码
唤醒收发包 ：RF 收发及外设短时间恢复，功耗提高，用于接收 Beacon 包或者发送保活心跳包，接收数据包后可插入执行用户代码
重启 ：IO 唤醒或者收到有效 DTIM 包后，需要恢复所有模块功能，需要经过重启

4.1.2. 包含头文件

- #include "lib/dsleepdata.h" 休眠数据管理头文件
- #include "lib/sleep_api.h" 休眠函数定义头文件

4.1.3. 可能需重写接口

- __weak void system_sleep_enter_hook(void); 休眠准备阶段钩子
- __weak void system_sleep_wakeup_hook(void); 休眠唤醒阶段钩子
- __weak int32 system_sleep_rxdata_hook(uint8 *data, uint32 len); 唤醒收发包阶段钩子

用户可以通过重写以上__weak 函数实现在休眠前、唤醒后、接收数据后执行用户代码

> 休眠准备阶段钩子: 因为此时模组仍未掉电，此时能够不受限制执行绝大部分代码
> 休眠唤醒阶段钩子 和 唤醒收发包阶段钩子: 因为此时模组已经掉过电，此时只能够执行有限代码，下节将展示用户能够使用的常见接口

4.1.4. 可使用接口

在休眠后只能执行有限区域函数，因此在 IO 控制和打印调试上与休眠前使用有所区别，
以下列出用户比较需要使用的能在进入休眠后使用的接口

- void system_sleep_config(uint32 cmd, uint32 param1, uint32 param2); 休眠相关设置
- void system_sleep_reset(void); 休眠复位重启
- void dsleep_gpio_set_val(uint32 pin, uint32 val); 设置 IO 输出值
- uint32 dsleep_gpio_get_val(uint32 pin); 读取 IO 输入值
- void dsleep_itoa(int32 n); 打印整数
- void dsleep_print(char *buff); 打印字符串
- void delay_us(uint32 n); 延时 us
- uint32 io_input_meas_mv(void); 读取 IO 上的电压（以 mV 为单位）

4.1.5. 重点注意事项

由于 sleep 掉电后，只保留部分代码段，因此用户重写的函数需要设置函数属性__at_section(".dsleep_text")将代码存放在非掉电区域

如果需要使用非局部变量时，也需要设置变量属性__at_section(".dsleep_data")将变量存放在非掉电区域

需要打印字符串时，若直接传参字符串，字符串数据会作为常量存放在程序所在文件中的常量区，不在非掉电区域，因此需要使用定义全局变量并赋予变量属性

字符串字符数量不超过 4 个可以使用局部变量（32 位机 4 个字符可以用 32 位的立即数表示）

打印输出串口 IO 默认使用了 PA13 引脚，如果用户有需要使用到 PA13 引脚，则不能使用打印相关函数

模组掉电过后，只能执行非掉电区域代码，因此不能够调用 C 标准库的函数，因此不能使用浮点数运算，ADC 读取电压值也特意提供了整数返回值接口

默认休眠掉电前，所有 GPIO（除 Flash 引脚 PA0~PA3、唤醒 IO、PA 控制 IO 外）会被复位至模拟引脚降低功耗。主控可以通过命令设置或休眠配置函数保留休眠时的 GPIO 状态不被复位，并且在休眠掉电阶段保持输出引脚的输出状态（即能够输出高电平）

经历重启阶段时所有 IO 会有短暂的掉电过程。

4.2. 示例程序

4.2.1. 休眠相关设置

system_sleep_config 函数常用设置

- SLEEP_SETCFG_GPIOA_RESV 设置 PA_12 在休眠中保留原有 IO 设置

```
system_sleep_config(SLEEP_SETCFG_GPIOA_RESV, BIT(12), 0);
```

- SLEEP_SETCFG_GPIOB_RESV 设置 PB_4 在休眠中保留原有 IO 设置

```
system_sleep_config(SLEEP_SETCFG_GPIOB_RESV, BIT(4), 0);
```

- SLEEP_SETCFG_WKSRC_DETECT 设置 PB_4 为多个唤醒源的 IO 唤醒检测 IO，检测电平为高电平，即用于区分 IO 和 PIR 唤醒，唤醒后查询唤醒原因可以得知

```
system_sleep_config(SLEEP_SETCFG_WKSRC_DETECT, PB_4, 1);
```

4.2.2. 休眠准备阶段钩子

此阶段模组仍未掉电，此时能够使用 SDK 中的 API 进行配置，如：使用 SDK 中 GPIO 的 HAL API 来设置 IO 模式，直接传参打印字符串，使用浮点数运算等...

```
__at_section(".dsleep_text") void system_sleep_enter_hook(void)
{
    float a = 1.23;
    uint32 b = 0;
    b = (uint32)(a * 10);
    dsleep_print("enter:");
    dsleep_itoa(b);
    system_sleep_config(SLEEP_SETCFG_GPIOA_RESV, BIT(12), 0);
    system_sleep_config(SLEEP_SETCFG_GPIOB_RESV, BIT(4) | BIT(6), 0);
    gpio_set_dir(PB_6, GPIO_DIR_OUTPUT);
    gpio_set_dir(PB_4, GPIO_DIR_OUTPUT);
    gpio_set_dir(PA_12, GPIO_DIR_INPUT);
    gpio_set_val(PB_6, 1);
}
```

gpio_set_val

static int32 gpio_set_val(uint32 pin, int32 value)

GPIO set ouput logic value

Definition: gpio.h:211

GPIO_DIR_OUTPUT

@ GPIO_DIR_OUTPUT

Definition: gpio.h:30

GPIO_DIR_INPUT

@ GPIO_DIR_INPUT

Definition: gpio.h:27

gpio_set_dir

static int32 gpio_set_dir(uint32 pin, enum gpio_pin_direction direction)

GPIO set direction

Definition: gpio.h:193

4.2.3. 休眠唤醒阶段钩子

此阶段模组已经掉电，只能使用列出 可使用接口 中的接口函数或做简单的逻辑操作，打印字符串需要设置变量属性（直接定义在函数内的局部字符串变量也是属于文件常量区，不属于非掉电区域）

可使用 system_sleepdata_request 申请 sleepdata 空间

```
struct user_sleep_data {
    uint32 voltage;
};
__at_section(".dsleep_data") struct user_sleep_data *my_data;
// 需要在初始化函数时，为数据区申请一片 sleepdata 空间，该空间在休眠唤醒时不会被清除
// my_data = (struct user_sleep_data *)system_sleepdata_request(SYSTEM_SLEEPDATA_ID_USER0, sizeof(struct user_sleep_data));
__at_section(".dsleep_data") const uint8 test_string[] = "over4";
__at_section(".dsleep_text") void system_sleep_wakeup_hook(void)
{
    uint32 io_voltage;
    dsleep_gpio_set_val(PB_4, 0);
    dsleep_gpio_set_val(PB_4, 1);
    delay_us(10);
    dsleep_gpio_set_val(PB_4, 0);
    dsleep_gpio_set_val(PB_4, 1);
    dsleep_print((char *)test_string);
    adkey_init();
    io_voltage = io_input_meas_mv();
    // 注意使用确保申请正确的空间，以防野指针
    my_data->voltage = io_voltage;
    dsleep_itoa(io_voltage);
    if (io_voltage < 100) {
        system_sleep_reset();
    }
}
```

4.2.4. 唤醒收发包阶段钩子

此阶段模组已经掉电，和 休眠唤醒阶段钩子 使用方式一致，可对收到发给休眠模组 MAC 地址的数据包，进行自定义的额外检查（模组自带对已设置保活心跳包和已设置唤醒包的检测）

data 格式为 IEEE Std 802.11 中定义，如下图所示，数据处于 Frame Body 子域中，具体数据起始位置需要根据 Frame Control 子域判断，详细定义请参阅 IEEE Std 802.11 相关文档

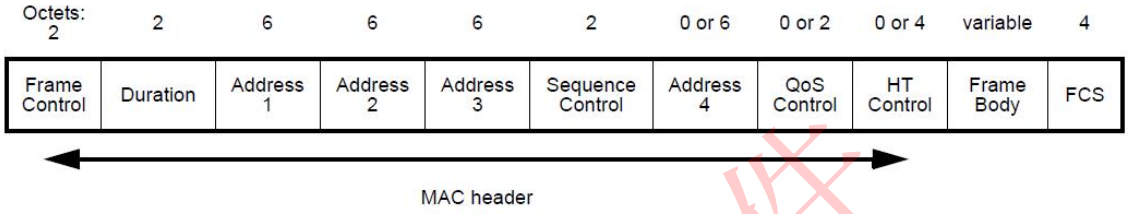


Figure 9-53—Data frame

Data Frame

```
__at_section(".dsleep_text") int32 system_sleep_rxdata_hook(uint8 *data, uint32
        len)
{
    int32 idx = 24;
    char str[] = "eth";
    /* Doing check rx data */
    if ((data[0] & 0x8f) == 0x88) { // QoS
        idx += 2;
        if (data[1] & 0x80) { // HTC
            idx += 2;
        }
    }
    /* Skip eth src */
    idx += 6;
    if (data[idx] == 0x08 && data[idx] == 0x00) {
        dsleep_print(str);
    }
    return 0;
}
```

4.3. 常见使用方式

4.3.1. 使用按键和 PIR 合并连接到硬件唤醒 IO 上

模组上只有一个硬件唤醒 IO，对于需要实时唤醒源，例如按键等可以直接接到硬件唤醒 IO 上，实现唤醒功能

如有两个唤醒源，可以通过两个唤醒源并在一起接入硬件唤醒 IO，并且配置另外一个检测 IO，在唤醒后通过检测 IO 判断唤醒源来区分，一般用于按键唤醒和 PIR 唤醒共用

1.先在休眠前设置检测 IO，假设设置 PB6 作为检测 IO，检测电平为高电平，检测 IO 应该接到按键唤醒源上，设置检测 IO 后，休眠唤醒后会检测 IO 电平，如果电平状态符合则判断为 IO 唤醒，否则判断为 PIR 唤醒

```
__at_section(".dsleep_text") void system_sleep_enter_hook(void)
{
    system_sleep_config(SLEEP_SETCFG_WKSRC_DETECT, PB_6, 1);
}
```

4.3.2. 按键单独接到硬件唤醒 IO 上，PIR 软件检测

在按键与 PIR 不方便接到一起的情况下，可以使用软件检测 PIR 电平

1.先在休眠前设置检测 IO，假设设置 PB6 作为输入引脚，用于检测 PIR 输出

```
__at_section(".dsleep_text") void system_sleep_enter_hook(void)
{
    gpio_set_dir(PB_6, GPIO_DIR_INPUT);
}
```

2.在休眠唤醒中检测 PIR 输出，检测到高电平时复位

```
__at_section(".dsleep_data") struct user_sleep_data *my_data;
// 需要在初始化函数时，为数据区申请一片 sleepdata 空间，该空间在休眠唤醒时不会被清除
// my_data = (struct user_sleep_data *)system_sleepdata_request(SYSTEM_SLEEPDATA_ID_USER0, sizeof(struct user_sleep_data));
__at_section(".dsleep_text") void system_sleep_wakeup_hook(void)
```

```
{
    if (dsleep_gpio_get_val(PB_6) == 1) {
        // 使用自定义结构体标识唤醒功能，以便唤醒后检测变量获知唤醒
        原因
        my_data->flag = 1;
        system_sleep_reset();
    }
}
```

5. 注意事项

WNB_STA_COUNT 宏定义的值不可修改