泰芯半导体
TaiXin Semiconductor

TaiXin TXW830x AH-SDK Development Guide

珠海泰芯半导体有限公司
Taixin Semiconductor Co.,Limited
珠海市高新区港湾一号科创园港 11 栋 3 楼
3rd Floor,Gang 11 Building,1st Jin Tang Road,Hi-tech Zone,Zhuhai China.

Liability and Copyright

Limitation of Liability

THIS DOCUMENT IS INTENDED FOR REFERENCE ONLY. Zhuhai Taixin Semiconductor Co,. Ltd (hereinafter referred to as "Taixin") does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Taixin reserves the right to make corrections, enhancements, and other changes to this document without notice.

Taixin assumes no liability for applications assistance or the design of customers' products. **Customers are solely responsible for the design, validation, and testing of its applications as well as for compliance with all legal, regulatory, and safety-related requirements concerning its applications.**

Taixin shall not be responsible for any damages, costs, losses, and/or liabilities arising out of customer's non-compliance with this section；concurrently, Customer will fully indemnify Taixin against any damages, costs, losses, and/or liabilities arising out of customer's non-compliance with this section.

Copyright Notice

Without the the written consent of Taixin, no party shall modify, adapt, alter, translate, or create derivative works from this document for commercial purposes.

Without the written consent of Taixin, no party shall disclose or distribute any or parts of the source code, SDK, binaries and object code mentioned in this document to any third party.

No party shall modify, reverse engineer, disassemble, decompile or otherwise attempt to discover the source code of any non-source code parts of the SDK including, but not limited to pre-compiled binaries and object code.

Furthermore, any other actions that may infringe upon Taixin's rights or other intellectual property owners are strictly prohibited.

For the subject who commits the above infringement, Taixin has the right to take necessary legal measure in accordance with the laws of the People's Republic of China or other applicable laws and international treaties, including but not limited to filing a lawsuit or arbitration against the infringer, or applying for legal compulsory measures.

CKLink and CDK are software and hardware product developed by T-HEAD Semiconductor.

Zhuhai Taixin Semiconductor Co., Ltd.

September 24,2024

Revision History

Date	Version	Revision Notes	Reviser
2024-01-16	V2.3	Added code for UART1 when using ADKEY.	WY
2024-01-02	V2.2	Added definitions for Pin functions.	WY
2023-01-11	V2.1	Added descriptions for IIC and software timer.	CWY/WY
2022-12-07	V2.0	Added descriptions for peripherals and internal functions. Changed the document title to "Development Guide".	CWY/WY
2022-08-08	V1.2.1	Updated the logo.	LSQ
2022-08-05	V1.2	Added FTP link for CDK.	WY
2021-07-01	V1.1	Added introduction to CKLink.	WY
2021-06-30	V1.0	Initial version.	WY

Table of Contents

1. Overview

This document introduces the SDK development environment and development instructions for TaiXin AH.

2. Development Environment

2.1. CKLink Introduction

CKLink-Lite is the debugging tool for the TaiXin AH SDK and can be purchased from T-Head's official Taobao store. Search for "CKLINK" on the Taobao homepage to find it. The purchase link is:

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

Figure 2-1 CKLink-Lite

The adapter board and debugging cables in Figure 2-1 can be obtained from our FAE.

If the changes are minimal, you can skip online debugging and directly use the serial port to upgrade the firmware and observe the running results, eliminating the need to purchase CKLink.

2.2. CDK Installation Instructions

CDK is the integrated development environment for T-Head CPUs and can be downloaded from TaiXin's FTP server.

- **Download link:** 183.47.14.74
- **Port:** 21
- **Account:** txguest
- **Password:** txguest

If you encounter issues downloading, please contact our FAE.

Additionally, please insert the CKLink into your computer before installation, as the CDK installation process will automatically install the CKLink driver.

2.3. CDK Compilation & Debugging

After installing CDK, open the hgSDK project file txw4002a.cdkws in the project directory. The hgSDK project compilation interface is shown below. Click the compile button or press F7 to compile.

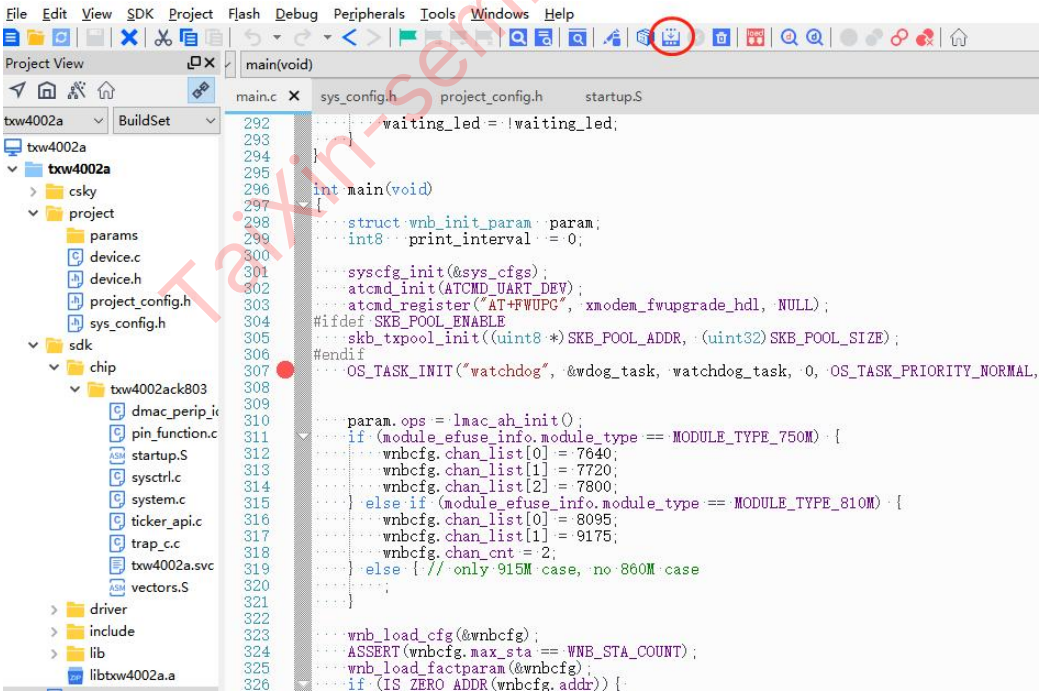


Figure 2-2 hgSDK Interface

```
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¸¶¨µÄÂ·¾¶¡£

D:\work\hgSDK-v1.3.2.5-12097\project>copy ..\..\..\..\tools\makecode\crc.exe crc.exe
ÏµÍ³ÕÒ²»µ½Ö¸¶¨µÄÂ·¾¶¡£

D:\work\hgSDK-v1.3.2.5-12097\project>copy ..\..\..\..\tools\makecode\makecode.exe makecode.exe
ÏµÍ³ÕÒ²»µ½Ö¸¶¨µÄÂ·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====
```

Figure 2-3 Firmware Compilation

Once the firmware compilation is complete, if online debugging is needed, use CKLink to download and run the debugging.

Before debugging, some settings are required: Change the debug level to Default(-g), then recompile the firmware as shown in Figure 2-4.

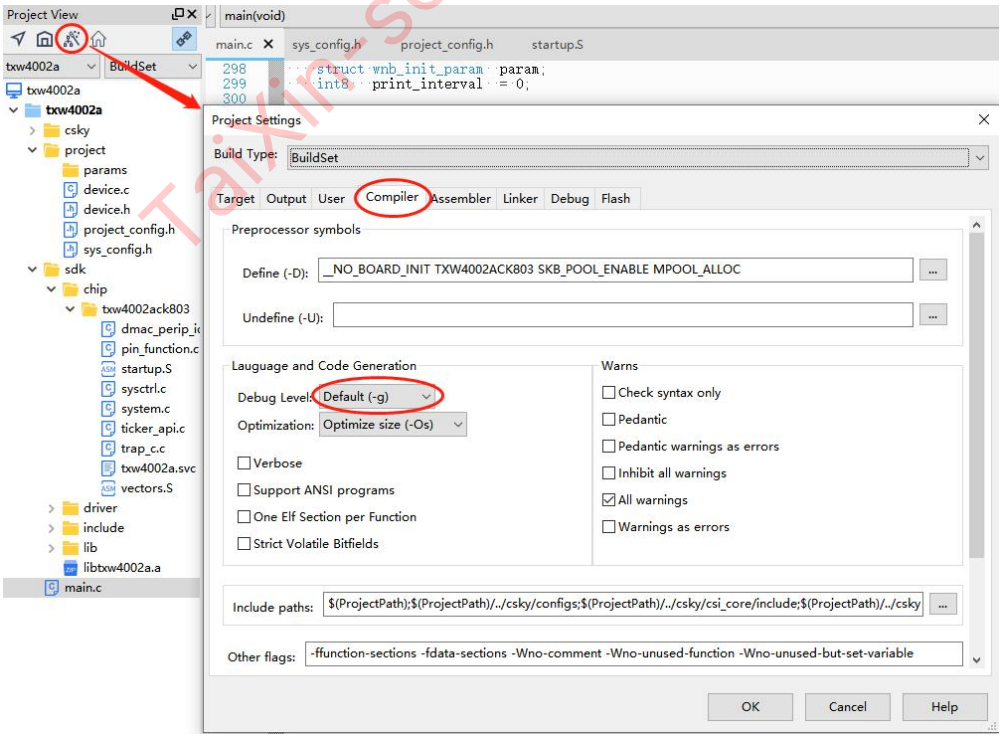


Figure 2-4 Modifying the Debug Level on the Compiler Page

Set the ICE Clock as shown in Figure 2-5.

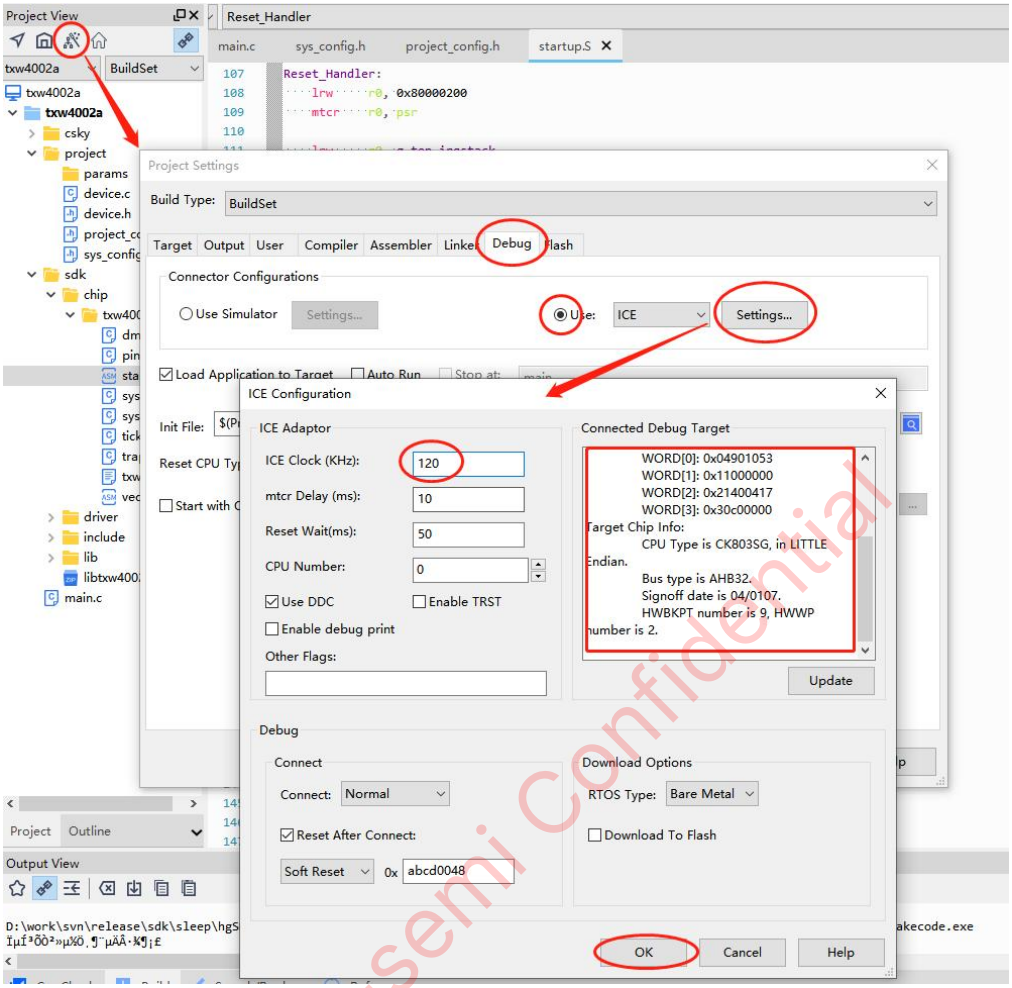


Figure 2-5 Debug Page ICE Clock Setting

1) Enter project settings/debug, select Use ICE;
2) Enter settings. If the debug board is correctly connected, you can read the Target chip Info in the Connected Debug Target box;
3) Set the ICE Clock to 120KHz. Due to a CDK issue, setting a higher frequency may cause debugging problems. Although setting it to 120KHz slows down download speed, it ensures stable debugging. Click OK to save.

After the settings, connect the CKLink debug cable to the board's debug port. Click the CDK debug button or press CTRL+F5, and CDK will start downloading the firmware, as shown in Figure 2-6.

Figure 2-6 Debug Button

Once the firmware download is complete, CDK will pause at the Reset_handler entry point, waiting for you to press F5 to run.

Additionally, the firmware default enables the watchdog. During debugging, if you hit a breakpoint, the watchdog may trigger a reset. You can set a breakpoint at the main function's watchdog enable position, right-click to skip to the next line of code, rendering the watchdog enable ineffective, as shown in Figure 2-7.

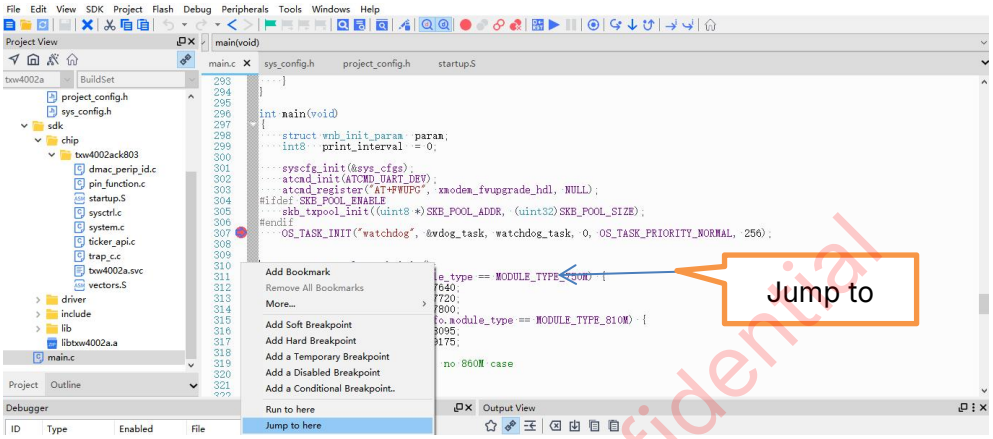


Figure 2-7 Skipping Watchdog Enable Task

2.4. Firmware Programming

After compiling the firmware, you can use the CSKYFlashProgrammer tool to program it. If the board already has firmware, you can upgrade it via the serial port using the command at+fwupg. For specific upgrade methods, please consult our FAE.

The programming tool package is named CSKY-FlashProgrammer-windows-V1.0.6-20200115-1549. After extracting the package, it appears as shown in Figure 2-8.

Figure 2-8 Programming Tool Package

First, install the CSKY_DebugServer_Driver, then enter the FlashProgrammer folder and double-click to run CSKYFlashProgrammer.exe.

Select the Advance tab and set the Program Algorithm File as shown in Figure 2-9. Locate TXW8300_FLASH_ALGORITHM.elf in the parent directory.

Figure 2-9 Selecting Programming Algorithm

Return to the main page and add the bin file for programming.

Figure 2-10 Adding Bin Programming Option

Select the firmware to be programmed, check program and verify, and click start to begin programming, as shown in Figure 2-11.

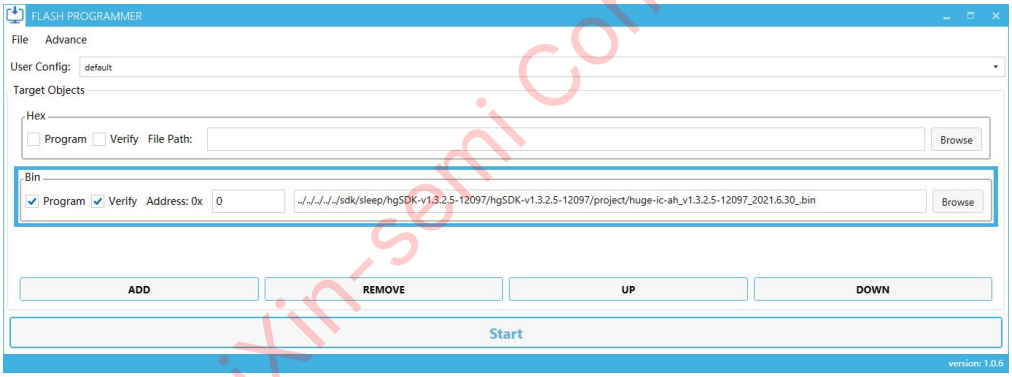


Figure 2-11 Programming Firmware

3. Basic Development

3.1. Pin function Definitions

pin voltage	Package Name	swd	IO	select	Function(0)	IO	Function(1)	IO	Function(2)	IO
fixed 3.3v	PA0				spi0_cs	O	uart1_rx	I		
fixed 3.3v	PA1				spi0_clk	O	uart1_tx	O		
fixed 3.3v	PA2				spi0_io0	B				
fixed 3.3v	PA3				spi0_io1	B				
SDVCCIO,1.8/3.3	PA6				sd_clk	I	spi3_slave_clk	I		
SDVCCIO,1.8/3.3	PA7				sd_cmd	B	spi3_slave_mosi	I		
SDVCCIO,1.8/3.3	PA8				sd_dat0	B	spi3_slave_miso	O	spi1_io1	B
SDVCCIO,1.8/3.3	PA9				sd_dat1	B	spi3_slave_cs	I		
SDVCCIO,1.8/3.3	PA10	swd_io	B	jtag_map2	sd_dat2	B				
SDVCCIO,1.8/3.3	PA11	swd_clk	I	jtag_map2	sd_dat3	B	spi3_slave_cs	I		
fixed 3.3v	PA12_DM	swd_io	B	jtag_map3	uart0_rx	I	iic0_scl	B		
fixed 3.3v	PA13_DP	swd_clk	I	jtag_map3	uart0_tx	O	iic0_sda	B		
fixed 3.3v	PA30	swd_io	B	jtag_map0	spi0_io2	B				
fixed 3.3v	PA31	swd_clk	I	jtag_map0	spi0_io3	B				
fixed 3.3v	PB0								rmii_ref_clkin	I
fixed 3.3v	PB1								rmii_rx_er	I
fixed 3.3v	PB2								rmii_rxd0	I
fixed 3.3v	PB3								rmii_rxd1	I
fixed 3.3v	PB4								rmii_txd0	O
fixed 3.3v	PB5								rmii_txd1	O
fixed 3.3v	PB6								rmii_crs_dv	I
fixed 3.3v	PB7								rmii_tx_en	O
fixed 3.3v	PB10								rmii_mdio	B
fixed 3.3v	PB11								rmii_mdc	O

pin voltage	Package Name	Function(3)	IO	Function(4)	IO	Function(6)	IO	Function(9)	IO	Function(10)	IO
fixed 3.3v	PA0	spi3_slave_cs	I	iic0_scl	B	spi0_clk	O				
fixed 3.3v	PA1	spi3_slave_clk	I	iic0_sda	B	spi0_cs	O				
fixed 3.3v	PA2	spi3_slave_mosi	I	uart0_rx	I	spi0_io1	B				
fixed 3.3v	PA3	spi3_slave_miso	O	uart0_tx	O	spi0_io0	B			uart1_tx	O
SDVCCIO,1.8/3.3	PA6	spi1_cs	O							uart1_tx	O
SDVCCIO,1.8/3.3	PA7	spi1_clk	O							uart1_rx	I
SDVCCIO,1.8/3.3	PA8	spi1_io0	B								
SDVCCIO,1.8/3.3	PA9	spi1_io1	B								
SDVCCIO,1.8/3.3	PA10	spi1_io2	B	uart0_rx	I						
SDVCCIO,1.8/3.3	PA11	spi1_io3	B	uart0_tx	O						
fixed 3.3v	PA12_DM	uart1_rx	I							uart1_tx	O
fixed 3.3v	PA13_DP	uart1_tx	O							uart1_rx	I
fixed 3.3v	PA30							uart0_tx	O	uart1_tx	O
fixed 3.3v	PA31							uart0_tx	O	uart1_tx	O
fixed 3.3v	PB0										
fixed 3.3v	PB1										
fixed 3.3v	PB2										
fixed 3.3v	PB3										
fixed 3.3v	PB4										
fixed 3.3v	PB5										
fixed 3.3v	PB6										
fixed 3.3v	PB7										
fixed 3.3v	PB10										
fixed 3.3v	PB11										

3.2. Peripheral Descriptions

3.2.1. GPIO

- Set GPIO to output mode, output 1

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

- Set GPIO to output mode, output 0

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

- Set GPIO to input mode, pull-down 100k

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

- Set GPIO to input mode, pull-up 100k

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

- Set GPIO to input mode, pull-down 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;
    }
}
```

3.2.2. ADKEY

The ADKEY (a low-speed ADC) function is actually a special function of PA12.

- Set GPIO to input analog function, ADC sampling

```
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);
// Get the ADKEY sampling value of PA12, in volts, floating point
printf("\r\nio_input=%f\r\n",io_input_meas());
// Get the ADKEY sampling value of PA12, in millivolts, integer
printf("\r\nio_input=%d\r\n",io_input_meas_mv());
```

If PA12 is used as ADKEY, then the debug serial port UART1 can be switched to PA31 for printing, and PA13 for at+ input. The corresponding code in pin_function.c is as follows:

```
case HG_UART1_DEVID:
```

```
        if (request) {
            jtag_map_set(0);
            gpio_set_altnt_func(PA_13, 10);//uart1-rx
            gpio_set_mode(PA_13, GPIO_PULL_UP, GPIO_PULL_LEVEL_100K);
            gpio_set_altnt_func(PA_31, 10);//uart1-tx
            gpio_set_mode(PA_31, GPIO_PULL_UP, GPIO_PULL_LEVEL_NONE);
        } else {
            gpio_set_dir(PA_13, GPIO_DIR_INPUT);
            gpio_set_dir(PA_31, GPIO_DIR_INPUT);
    }
```

Ensure not to set PA13/PA31 elsewhere.

3.2.3. Hardware TIMER

When high timing accuracy (on the order of 100 microseconds) is required and the timing duration is not very long (not exceeding 85mS), a hardware timer (TIMER1 and TIMER3) can be used. Here is an example using TIMER1:

- Define TIMER1_ENABLE_MS, and set it to 5ms interrupt (located in project\project_config.h):

```
#define TIMER1_ENABLE_MS        5
```

The range is 1~85ms; if it exceeds this, a software timer should be used.

- TIMER1 initialization function (located in project\main.c)

```
timer1_init(uint32 tmo_ms, enum timer_type type, uint32 cb_data);
//tmo_ms: TIMER timeout interrupt time
//type: Type of TIMER (single interrupt or cyclic interrupt)
//cb_data: Parameter for the TIMER interrupt handler
```

- TIMER1 interrupt handler (located in project\main.c)

```
timer1_cb(uint32 args);
//args: Parameter set during the initialization for the interrupt handler
```

- TIMER1 stop function

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

3.2.4. Software TIMER

When timing accuracy requirements are not very high (millisecond level) and the timing duration is relatively long (exceeding 85 milliseconds).The software timer can be

configured as follows:

- TIMER initialization function

```
int os_timer_init(struct os_timer *timer, os_timer_func_t func,
                  enum OS_TIMER_MODE mode, void *arg)
//timer: TIMER structure
//func: Timer interrupt handler function
//mode: Type of TIMER (single interrupt or cyclic interrupt)
//arg: Parameter for the TIMER interrupt handler
```

- TIMER start function

```
int os_timer_start(struct os_timer *timer, unsigned long expires)
//timer: TIMER structure
//expires: TIMER timeout interrupt time in milliseconds
```

- TIMER interrupt handler function (located in project\main.c)

```
static void test_timer_cb(void *args)
{
    os_printf("test_timer_cb\r\n");
}
//args: Parameter set during the initialization for the interrupt handler
```

- TIMER stop function

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

- Example code:

```
struct os_timer test_tm;

//Timer interrupt handler function
static void test_timer_cb(void *args)
{
    os_printf("test_timer_cb\r\n");
}

// Timer initialization
os_timer_init(&test_tm, test_timer_cb, OS_TIMER_MODE_PERIODIC, 0);

//Start the timer, time unit is milliseconds
os_timer_start(&test_tm, 1000);
```

3.2.5. UART

UART0 can be used for communication with peripherals. Note that UART1 is dedicated for printing debug information and should not be used for other communication purpose in development.

Hardware Connection: IOA10—RX,IOA11—TX

- Enable IOT_DEV_UART Macro Definition (project\project_config.h), and set the serial port parameters and timeout duration:

```
#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
```

- Send Function:

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

- Receive Function (project\main.c)::

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

3.2.6. SPI master

Hardware Connection:

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,

- Enable IOT_DEV_SPI Macro Definition(project\project_config.h), and set clock frequency and mode:

```
#define IOT_DEV_SPI
#define SPI_IOT_CLK             1000000
```

```
#define SPI_IOT_CLK_MODE        SPI_CPOL_0_CPHA_0
```

- Send Function:

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

- Receive Function (project\main.c):

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

3.2.7. I2C Master

The I2C here is software simulated, and GPIO pins can be chosen from any available IOs.

- Enable IOT_DEV_I2C Macro Definition: (project\project_config.h)

```
#define IOT_DEV_I2C
```

- Pin Customization (sdk\lib\mac_bus\i2c_iot.c):

```
#define I2C_GPIO_SCL                PA_7 //example with PA_7 and PA_8
#define I2C_GPIO_SDA                PA_8
#define I2C_DELAY                   (3) //clock period control, unit:um
#define SLAVE_ADDRESS               0x3C
```

- Write Function (sdk\lib\mac_bus\i2c_iot.c)

```
//Write to register
int32 i2c_gpio_write_reg(uint8 slave_addr, uint8 reg, uint8 *data, uint32 size)

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

- Read Function (sdk\lib\mac_bus\i2c_iot.c):

```
// Read from register
int32 i2c_gpio_read_reg(uint8 slave_addr, uint8 reg, uint8 *data, uint32 size)

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

3.3. Simulated RTC

The module uses the timestamp in the Beacon packets sent by the AP to implement the RTC

function.

- lmac_set_rtc(ops, rtc): Set the time within 24 hours in milliseconds.
- lmac_get_rtc(ops): Return the time within 24 hours in milliseconds.

```
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);
```

3.4. Custom Driver Data Interface

Although the driver already includes most common module functions, users can still perform secondary development in the SDK to customize the required driver data, such as IO status, peripheral status, etc.

- __weak int32 customer_driver_data_proc(uint8 *data, uint32 len); Custom driver data processing interface.
- int32 customer_driver_data_send(uint8 *data, uint32 len); Custom driver data sending interface.

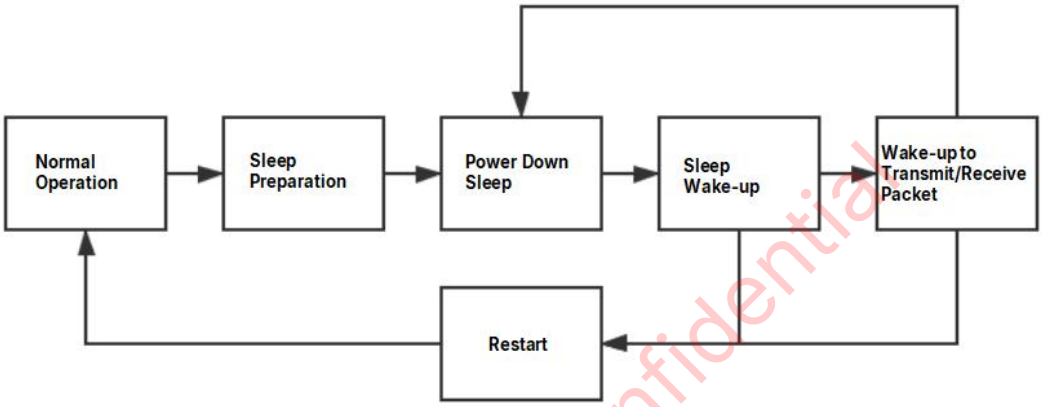
Users can send custom driver data to the module using the command HGIC_CMD_SET_CUST_DRIVER_DATA or by reading/writing the /proc/hgic/cust_driverdata file.

The module receives the sent data through customer_driver_data_proc, allowing users to define the data structure content and retrieve the necessary information from the module. Finally, the information can be sent back to the main driver using customer_driver_data_send, where the driver will parse it independently.

4. Sleep and Wake-up Related

4.1. Function Introduction

4.1.1. Different Stages of Sleep



- **Normal Operation:** RF transceivers and peripherals operate normally, with the highest power consumption.
- **Sleep Preparation:** Gradually shut down RF transceivers and peripherals, reducing power consumption, allowing the execution of user code.
- **Power Down Sleep:** RF transceivers and peripherals are turned off, with the lowest power consumption, waiting for timed wake-up or IO wake-up.
- **Sleep Wake-up:** During power down sleep, the system can be woken up by a designated wake-up IO or a set time interval. After waking up, user code can be executed.
- **Wake-up to Transmit/Receive Packet:** RF transceivers and peripherals temporarily resume operation, increasing power consumption, used for receiving Beacon packets or sending keep-alive heartbeat packets. After receiving data packets, user code can be executed.
- **Restart:** When woken up by IO or after receiving a valid DTIM packet, all module functions need to be restored, going through a restart.

4.1.2.Header Files Included

- #include "lib/dsleepdata.h" // Sleep data management header file
- #include "lib/sleep_api.h" // Sleep function definition header file

4.1.3.Possible Interface Rewriting

- __weak void system_sleep_enter_hook(void); Sleep Preparation Stage Hook
- __weak void system_sleep_wakeup_hook(void); Sleep Wake-up Stage Hook
- __weak int32 system_sleep_rxdata_hook(uint8 *data, uint32 len); Wake-up to Transmit/Receive Packet Stage Hook

Users can overwrite the above __weak functions to execute user code before sleep, after wake-up, or after receiving data.

Sleep Preparation Stage Hook: Most of the code can be executed without restrictions at this stage because the module is still powered on.

Sleep Wake-up Stage Hook and Wake-up to Transmit/Receive Packet Stage Hook: Only limited code can be executed at this stage because the module has been powered off. The next section will demonstrate common interfaces available for users.

4.1.4.Usable Interfaces

After sleep, only limited area functions can be executed, so there is a difference in IO control and print debugging from the use before sleep.

The interfaces that the user may need to be able to use after entering sleep as follows:

- void system_sleep_config(uint32 cmd, uint32 param1, uint32 param2); Sleep-related settings
- void system_sleep_reset(void); Sleep reset and restart
- void dsleep_gpio_set_val(uint32 pin, uint32 val); Set IO output value
- uint32 dsleep_gpio_get_val(uint32 pin); Read IO input value
- void dsleep_itoa(int32 n); Print integer
- void dsleep_print(char *buff); Print string
- void delay_us(uint32 n); Delay in microseconds
- uint32 io_input_meas_mv(void); Read the voltage on IO (in mV)

4.1.5.Key Considerations

After entering the sleep mode, only certain code segments are retained. Therefore, functions that the user rewrites must set the function attribute __at_section(".dsleep_text") to place the code in the non-power-down area.

If you need to use non-local variables, set the variable attribute __at_section(".dsleep_data") to store the variables in the non-power-down area.

When printing strings, if you pass the string directly as a parameter, the string data will be stored in the constant area of the file where the program resides, not in the non-power-down area. Therefore, you need to define global variables and assign them attributes.

For strings with up to 4 characters, you can use local variables (on a 32-bit machine, 4 characters can be represented by a 32-bit immediate value).

The default pin for UART output is PA13. If the user needs to use the PA13 pin, they cannot use the print-related functions.

After the module powers down, it can only execute code in the non-power-down area, so it cannot call C standard library functions. Thus, floating-point operations are not allowed. For ADC voltage readings, an integer return value interface is specifically provided.

Before default sleep power-down, all GPIOs (except Flash pins PA0~PA3, wake-up IO, and PA control IO) will be reset to analog pins to reduce power consumption. The main control can retain the GPIO state during sleep by setting commands or sleep configuration functions, keeping the output state of output pins during the power-down phase (i.e., it can output a high level).

During the restart phase, all IOs will experience a brief power-down period.

4.2. Example Programs

4.2.1. Sleep-related Settings

Common settings for the system_sleep_config function:

- SLEEP_SETCFG_GPIOA_RESV: Set PA_12 to retain its IO settings during sleep.

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

- SLEEP_SETCFG_GPIOB_RESV: Set PB_4 to retain its IO settings during sleep.

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

- SLEEP_SETCFG_WKSRC_DETECT: Set PB_4 as the IO wake-up detection pin for multiple wake-up sources, with a high-level detection. This is used to distinguish between IO and PIR wake-ups, and after waking up, the wake-up reason can be queried.

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

4.2.2. Sleep Preparation Stage Hook

At this stage, the module has not yet powered down, and you can use the API in the SDK for configuration. For example, you can set IO modes using the GPIO HAL API in the SDK, directly pass strings for printing, and use floating-point operations, etc.

```
__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. Sleep Wake-up Stage Hook

At this stage, the module has already powered down, and you can only use the listed

available interfaces or perform simple logical operations. Printing strings requires setting variable attributes (local string variables defined within functions are also part of the file's constant area, not the non-power-down area).

You can request sleepdata space using system_sleepdata_request.

```
struct user_sleep_data {
    uint32 voltage;
};
__at_section(".dsleep_data") struct user_sleep_data *my_data;
/* It's needed to request a section of sleepdata space for the data area during the
    initialization function, which will not be cleared during sleep wake-up. */
/* 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();
    // Ensure that you request the correct space to prevent wild pointers.
    my_data->voltage = io_voltage;
    dsleep_itoa(io_voltage);
    if (io_voltage < 100) {
        system_sleep_reset();
    }
}
```


4.2.4. Packet Transmission and Reception During Wake-up Stage Hook

At this stage, the module has already powered down, and the usage is the same as the sleep wake-up stage hook. You can perform custom additional checks on the data packets sent to the module's MAC address (the module itself detects the keep-alive heartbeat packets and wake-up packets that have been set).

The data format is defined in IEEE Std 802.11. As shown in the figure below, the data is in the Frame Body subfield, and the specific starting position of the data needs to be determined based on the Frame Control subfield. For detailed definitions, please refer to

the relevant IEEE Std 802.11 documents.

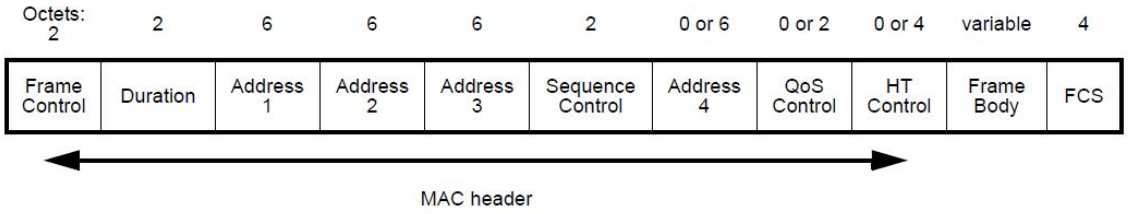


Figure 4-1 Data Frame

```
__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. Common Usage Methods

4.3.1. Using Buttons and PIR Combined with Hardware Wake-up IO

The module has only one hardware wake-up IO. For real-time wake-up sources, such as buttons, they can be directly connected to the hardware wake-up IO to achieve wake-up functionality.

If there are two wake-up sources, they can be combined and connected to the hardware wake-up IO, and another detection IO can be configured. After waking up, the detection IO is used to determine the wake-up source. This is generally used for shared button and PIR wake-up.

1) Set the detection IO before sleep. Suppose PB6 is set as the detection IO with a

high-level detection. The detection IO should be connected to the button wake-up source. After setting the detection IO, it will check the IO level after waking up. If the level state matches, it is determined as an IO wake-up; otherwise, it is determined as a PIR wake-up.

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

4.3.2. Button Connected to Hardware Wake-up IO, PIR Software Detection

When it is inconvenient to connect the button and PIR together, software detection can be used for the PIR level.

1) Set the detection IO before sleep. Suppose PB6 is set as an input pin for detecting PIR output.

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

2) Detect the PIR output during sleep wake-up, and reset when a high level is detected.

```
__at_section(".dsleep_data") struct user_sleep_data *my_data;
/* You need to request a section of sleepdata space for the data area during the
    initialization function, which will not be cleared during sleep wake-up. */
/* 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) {
    // Use custom structures to identify the wake-up function so that the wake-up
    reason can be detected after waking up.
        my_data->flag = 1;
        system_sleep_reset();
    }
}
```

5. Notes

The value defined by the WNB_STA_COUNT macro can not be modified.