

TXW80X SDK 快速入门手册




注意

由于产品版本升级或者其他原因，本文档会不定期更新。除非另行约定，本文档仅作为使用指导，不做任何担保。

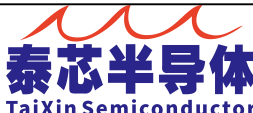
保密等级	A	TXW80X SDK 快速入门手册	文件编号	TX-0000
发行日期	2022-07-26		文件版本	V1.1


修订记录

日期	版本	描 述	修订人
2022-07-26	V1.1	1、增加 CDK 下载代码和量产烧录描述 2、增加 XIP 方案代码效率调试和优化描述	TX
2022-05-25	V1.0	初始版本	TX

 泰芯半导体 TaiXin Semiconductor	珠海泰芯半导体有限公司 Zhuhai Taixin Semiconductor Co., Ltd	珠海市高新区港湾一号科创园港 11 栋 3 楼
---	---	-------------------------

版权所有 侵权必究 Copyright © 2022 by Tai Xin All rights reserved
--

保密等级	A	TXW80X SDK 快速入门手册	文件编号	TX-0000
发行日期	2022-07-26		文件版本	V1.1
目录				
TXW80X SDK 快速入门手册..... 1				
1. 概述..... 1				
2. 集成开发环境..... 1				
2.1. CKLink 调试器..... 1				
2.2. CDK 开发环境..... 2				
2.3. 编译和调试..... 2				
2.3.1. 编译..... 2				
2.3.2. 调试..... 2				
2.3.3. 中途插入调试..... 3				
2.4. 固件烧录..... 5				
2.4.1. CDK 下载代码..... 5				
2.4.2. CKLink 烧写..... 5				
2.4.3. UART 烧写..... 6				
2.4.4. 量产烧录..... 6				
3. 硬件开发板..... 7				
3.1. 音视频开发板..... 7				
3.1.1. 音视频开发板接口介绍..... 7				
3.2. 无线网卡开发板..... 8				
3.3. 低功耗开发板..... 8				
3.4. IOT 开发板..... 8				
3.5. 网桥开发板..... 8				
3.6. 触摸开发板..... 8				
4. SDK 概述..... 9				
4.1. SDK Feature..... 9				
4.2. SDK 框架图..... 9				
4.3. 目录结构..... 10				
4.4. SDK 启动流程..... 11				
4.5. 系统参数..... 12				
4.6. FLASH 布局..... 14				
4.7. Memory 布局..... 15				
4.8. 方案开发基本规则..... 16				
4.9. 驱动 API 使用说明..... 18				
4.10. OSAL API 使用说明..... 18				
4.10.1. Task..... 18				
4.10.2. Mutex..... 21				
4.10.3. Semaphore..... 21				
4.10.4. Msgqueue..... 22				
		珠海泰芯半导体有限公司 Zhuhai Taixin Semiconductor Co.,Ltd	珠海市高新区港湾一号科创园港 11 栋 3 楼	
版权所有 侵权必究 Copyright © 2022 by Tai Xin All rights reserved				

保密等级	A	TXW80X SDK 快速入门手册	文件编号	TX-0000
发行日期	2022-05-25		文件版本	V1.0
<div>4.10.5. SoftTimer.....24</div> <div>4.11. 调试信息.....25</div> <div>4.11.1. WiFi 协议栈调试信息输出.....25</div> <div>4.11.2. 系统 Heap 使用情况.....26</div> <div>4.11.3. CPU 使用率.....26</div> <div>4.11.4. XIP 取指效率.....27</div> <div>5. 其他学习资源.....28</div>				
		珠海泰芯半导体有限公司 Zhuhai Taixin Semiconductor Co.,Ltd	珠海市高新区港湾一号科创园港 11 栋 3 楼	
版权所有 侵权必究 Copyright © 2022 by Tai Xin All rights reserved				

1. 概述

本文为使用方案软件设计开发人员而写，目的帮助您快速入门 SDK。

本文档主要适用于以下工程师：

- 技术支持工程师
- 方案软件开发工程师

本文档适用的产品范围：

型号	封装	包装
TXW806		
TXW803		
TXW802		
TXW801	QFN24	

2. 集成开发环境

2.1. CKLink 调试器

CKLink Lite 是 TXW80X 芯片/模组的 SDK 的调试工具，可以在平头哥的淘宝官方店购买。



图 2.1.1 - 调试器

2.2. CDK 开发环境

CDK 是平头哥 CPU 的集成开发环境，可以从平头哥官方网站进行下载安装。本 SDK 基于 CDK v2.8.8 版本研发，建议使用此版本或者更高版本进行开发。

如果下载不了请联系我司 FAE。

2.3. 编译和调试

2.3.1. 编译

CDK 安装好后，到 SDK project 路径下双击打开工程 *.cdkproj，在 CDK 菜单点击编译或者快捷键 F7 执行编译。

2.3.2. 调试

调试时建议设置 ICE Clock 120Khz，并且勾选 reset after connect，使用 softreset。断点调试时请关闭代码中的看门狗。具体操作步骤如下（如图 2.3.2.1 所示）：

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（如果需要调试器设置频率更高，请参考 FAQ），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 会启动调试。

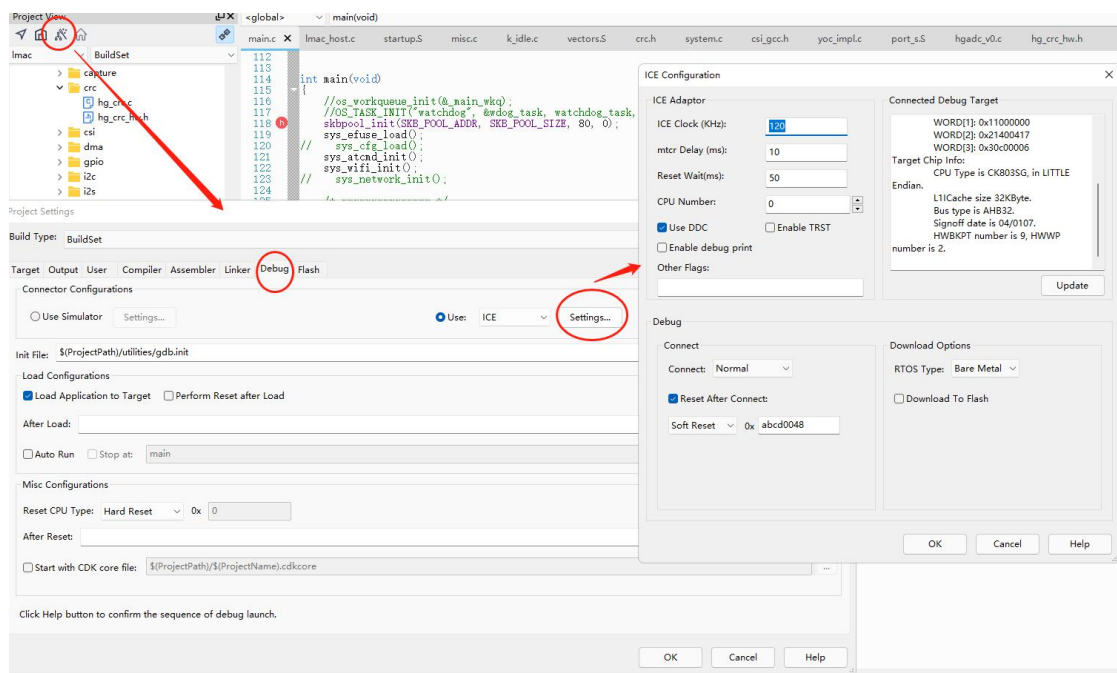
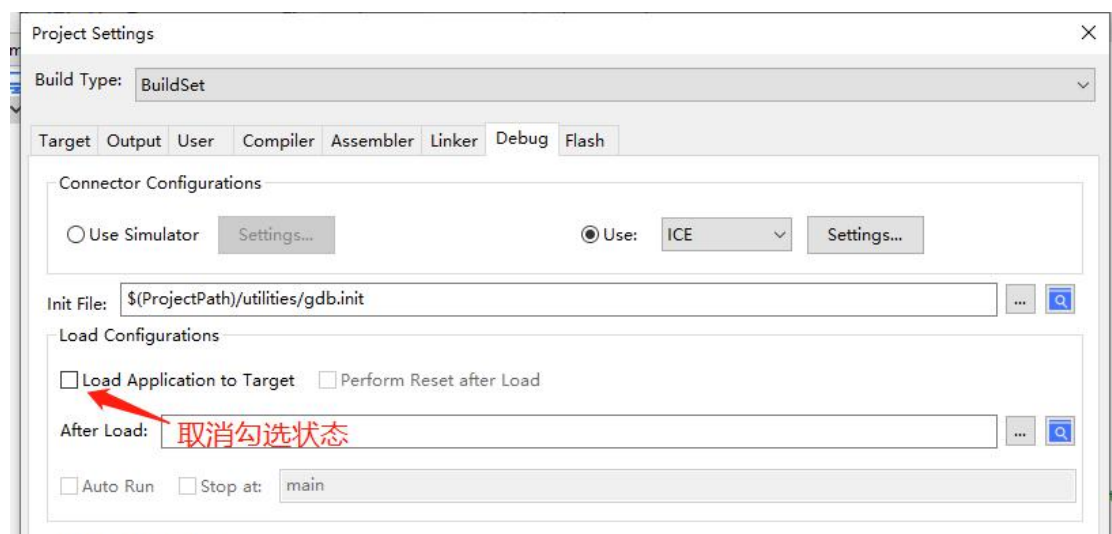


图 2.3.2.1 - 调试设置

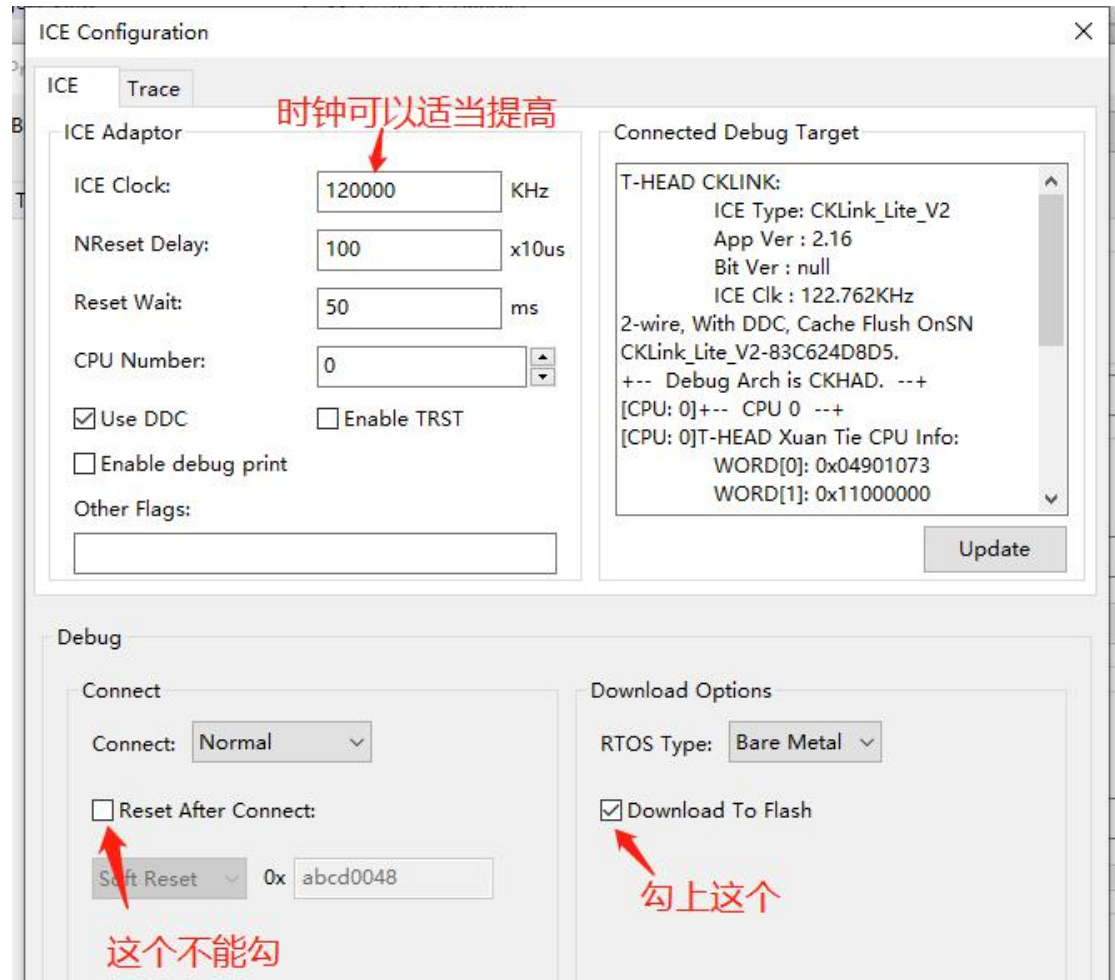
2.3.3. 中途插入调试

有时候需要中途插入调试器进行调试。此时不能按照普通的调试方式进行操作（会复位，破坏现场）。需要如下配置 SDK，然后再插入调试器调试：

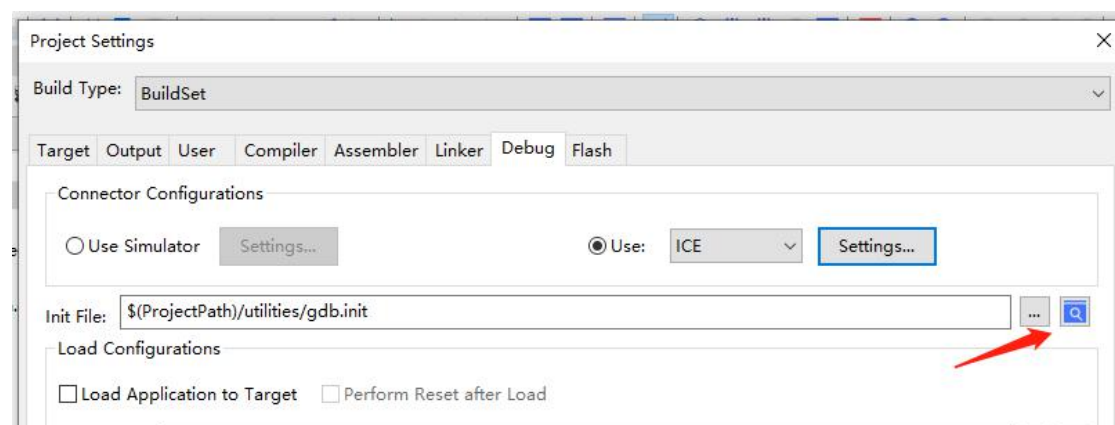
1. 在 Project Settings 窗口中（可以使用 ALT+F7 快捷键打开），取消 “Load Application to Target” 的勾选。



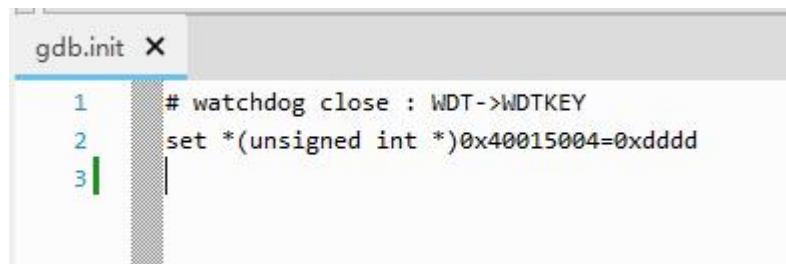
2. 继续在 Project Settings 窗口中，点击 ICE 左边的 Settings，在弹出的 ICE Configuration 窗口中，勾选“Download to Flash”，去除“Reset After Connect”的勾选状态。另外可以根据系统时钟的频率，设置一下 ICE Clock 的值（只要不高于 1/2 系统时钟即可）。最后按 OK 退出窗口。




3. 回到 Project Settings 窗口，点击 Init File 最右边的放大镜图案，修改 gdb.init 内容。此时可以按 OK，退出 Project Settings 窗口。



4. 根据实际修改 gdb.init。如果要调试的程序没有开看门狗，则可以将 gdb.init 里面的内容全部删除；如果开了看门狗，则 gdb.init 需要留下如下 2 条语句。



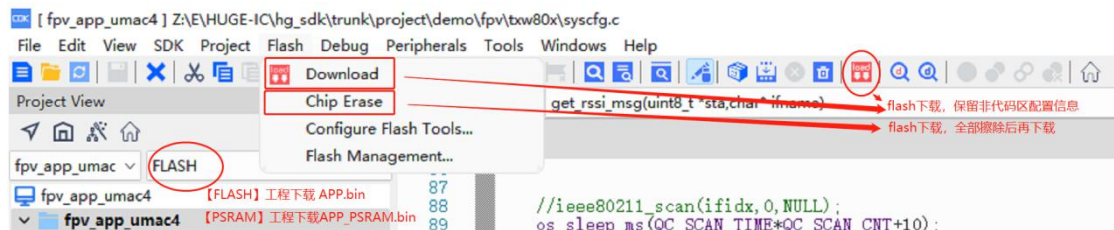
```
1 # watchdog close : WDT->WDTKEY
2 set *(unsigned int *)0x40015004=0xdddd
3
```

上述步骤完成后，可以插入调试器，点击进入调试模式。

2.4. 固件烧录

2.4.1. CDK 下载代码

CDK 代码编译成功后可以通过菜单“Flash”->“Download”进行下载，此方式下载会保留非代码区的配置信息；如果需求清除配置信息，可以选择“Chip Erase”方式进行下载。

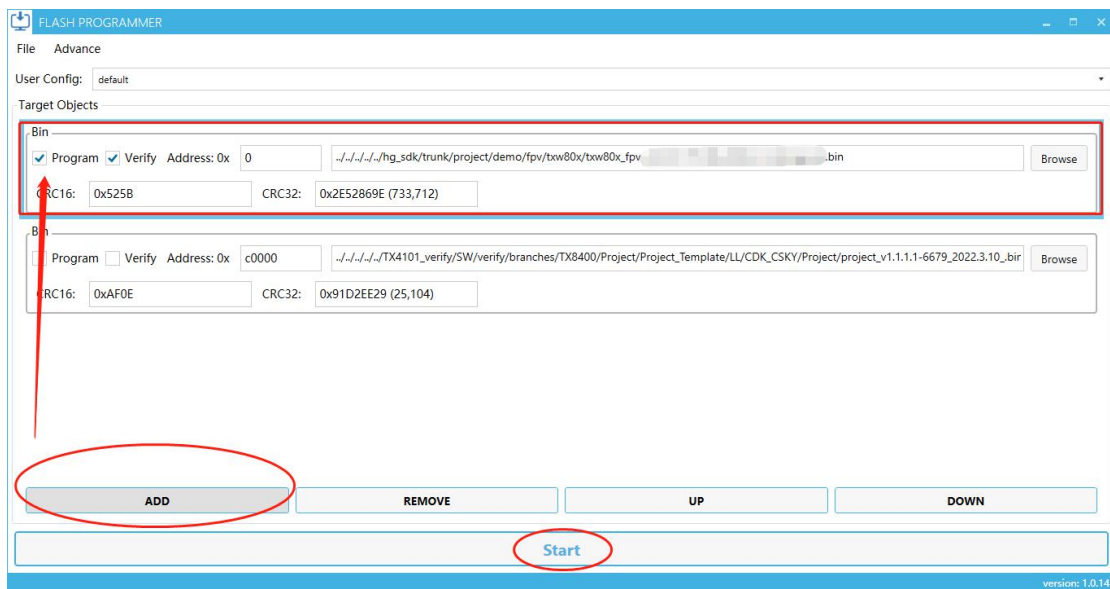
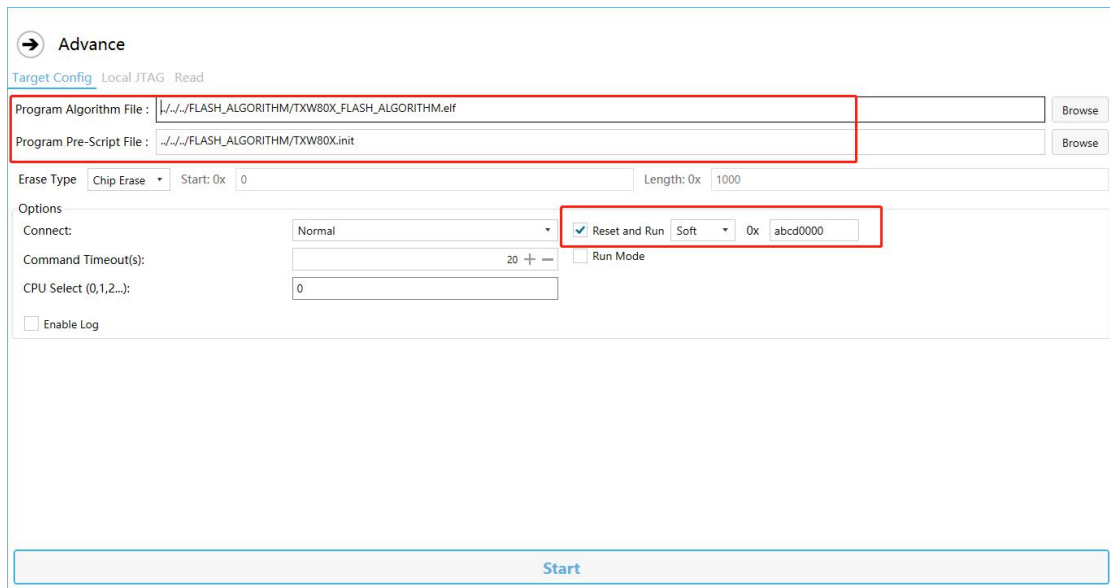


2.4.2. CKLink 烧写

烧录工具为：CSKY-FlashProgrammer-windows，平头哥官方网站进行下载安装。建议使用 V1.0.14 及以上版本。具体烧写步骤为：

- 1、执行 CSKYFlashProgrammer.exe
- 2、在 advance TargetConfig 中设置 Program Algorithm File 和 Script
- 3、根据需要勾选是否下载完立即运行代码
- 4、在 advance LocalJTAG 中设置 ICE CLK：12Mhz

5、回到主界面添加下载文件并点击 start 按钮下载。



2.4.3. UART 烧写

SDK 集成了串口升级功能，如果已经打开，可以通过串口命令“TX+OTA”进行升级。具体方法请咨询我司 FAE。

2.4.4. 量产烧录

量产烧录详见《TXW80X 量产和烧录指南.pdf》

3. 硬件开发板

为了快速入门和方案评估，我们提供各种应用场景的开发板。

3.1. 音视频开发板

3.1.1. 音视频开发板接口介绍

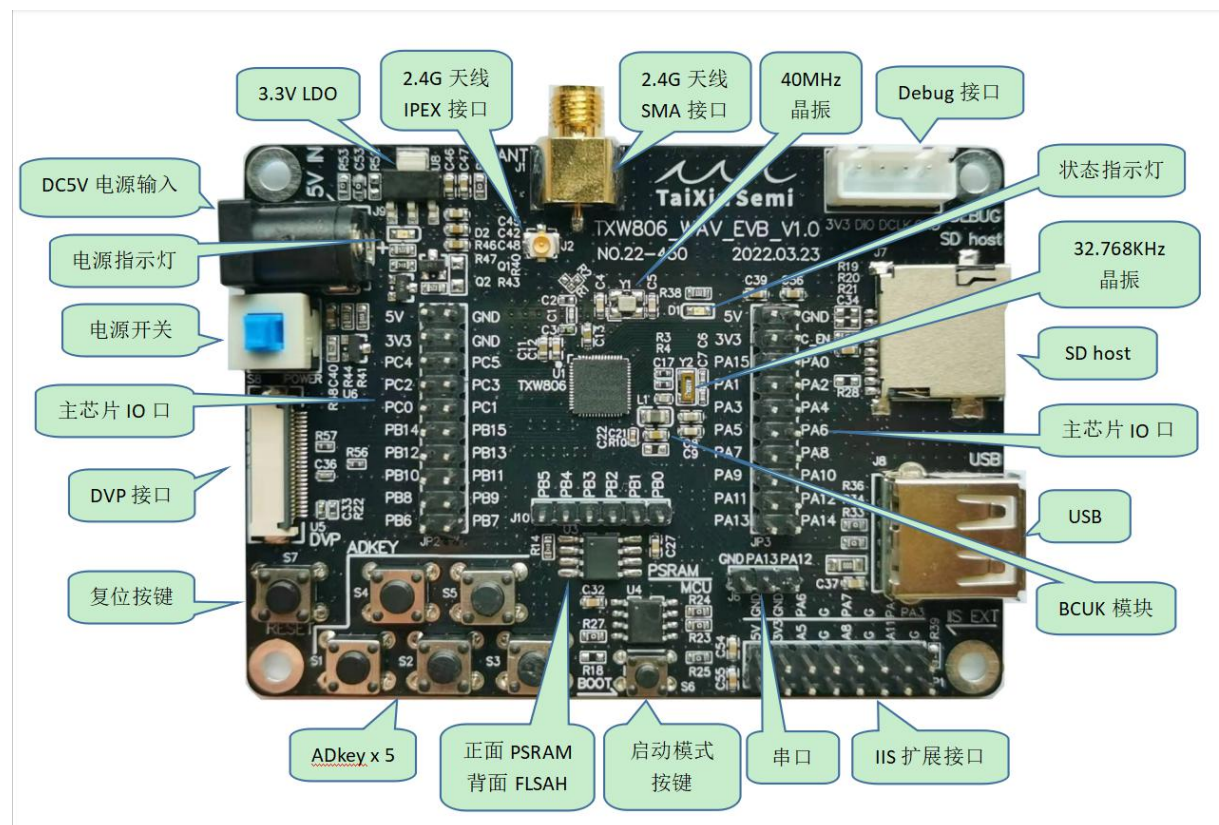


图 3.1.1.1 - 音视频开发板主视图

特殊说明

模式启动按键：此按键可以一键拯救系统，在芯片上电即跑死，cklink 烧写和其他升级都失效的情况下使用。**注意按此按键时不能连接 Debug 接口。**

IIS 扩展接口：此接口注意是为了扩展音频子板。

3.2. 无线网卡开发板

3.3. 低功耗开发板

3.4. IOT 开发板

3.5. 网桥开发板

3.6. 触摸开发板

4. SDK 概述

4.1. SDK Feature

◆ 应用场景

- VGA/720P 耳镜、航拍
- miniDV
- 可视门铃
- Wi-Fi 触控家电

◆ 操作系统

- AliOS

◆ 工作角色

- AP
- STA

◆ 网络协议栈

- LWIP
- MQTT
- HTTP

◆ 文件系统

- (ex)FAT

◆ 抽象数据总线

- USB device、SDIO device、SPI slave、UART、I2C

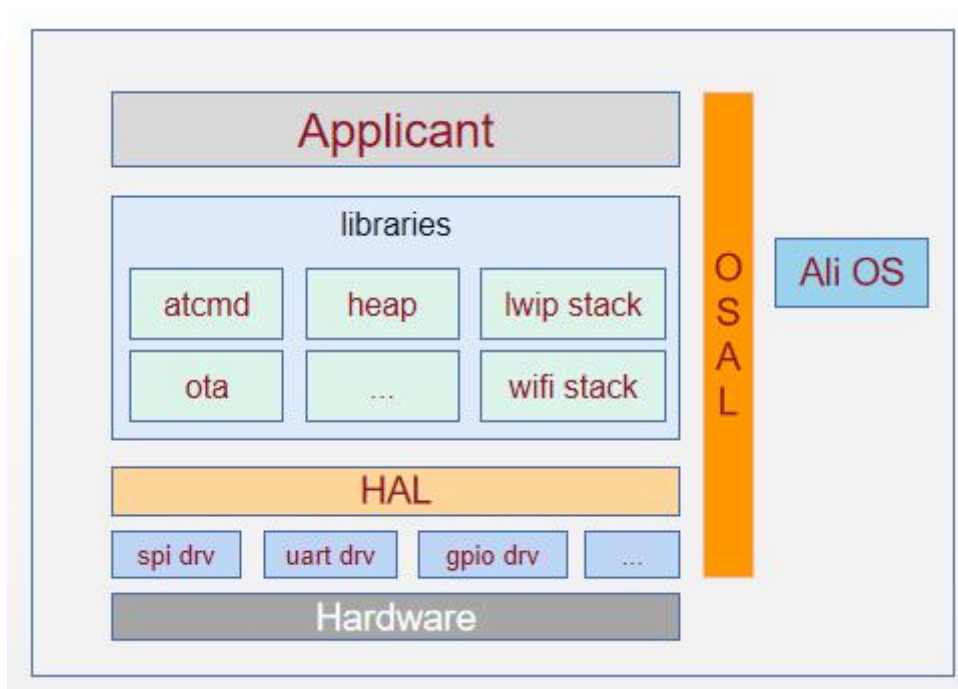
◆ 抽象外设

- GPIO、ADC、DMA、DVP、MJPEG、IIS、PDM、USB、SDIO 、SPI、UART、I2C、LED、PWM、UART、WDT、TIMER、TOUCH、AES、CRC

4.2. SDK 框架图

SDK 基本框架如下图所示，SDK 定义了 OSAL 和 HAL 2 个抽象层；OSAL 抽象层定义了与 OS 相关操作的 API，HAL 定义了硬件设备访问 API。

在使用 SDK 进行二次开发代码时，涉及到与 OSAL 和 HAL 相关的 API 调用时，请使用 OSAL 和 HAL 提供的 API



4.3. 目录结构

TXW80x SDK 解压后的目录结构如下图所示。

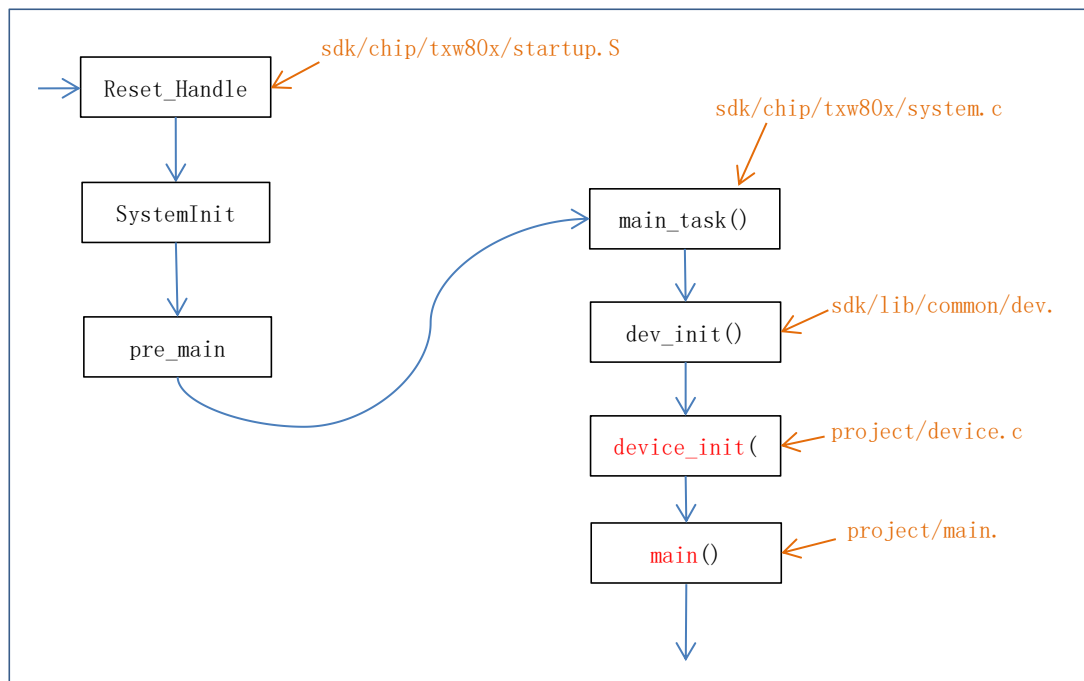


- csky: CSKY RTOS 代码目录，通常不需要修改该部分代码。
- project: CDK 工程应用代码目录，main.c 位于该目录。
- sdk/chip: 芯片定义及启动代码目录
- sdk/driver: 外设驱动代码目录
- include: SDK 头文件目录
- lib: 常用 library，包括 lwip 协议栈，以太网 PHY 驱动等。用户自行移植的代码库可以放在此目录下。

4.4. SDK 启动流程

SDK 的启动代码在 `sdk/chip/txw80x` 目录下，大致的启动流程如下图所示，其中 `device_init` 和 `main` 函数可以进行相应修改，其它启动流程代码请勿修改。

- **device_init** 函数：系统设备初始化函数，完成了驱动初始化和设备注册；所有设备只有注册后才可以使⤵用。在该函数中可以删除未使用的设备的初始化代码，以减少代码体积。
- **main** 函数：应用程序入口函数。



4.5. 系统参数

SDK 的系统参数定义在 project/sys_cfg.h 文件中，如下所示：

```
11: struct sys_config {
12:     ...uint16 magic_num, crc;
13:     ...uint16 size, rev1, rev2, rev3;
14:     ...////////////////////////
15:
16:     ...uint8 wifi_mode, bss_bw, tx_mcs, channel;
17:     ...uint8 bssid[6], mac[6];
18:     ...uint8 ssid[32];
19:     ...uint8 key[32];
20:     ...uint16 bss_max_idle, beacon_int;
21:     ...uint16 ack_tmo, dtim_period;
22:     ...uint32 key_mgmt;|
23:
24:     ...uint32 dhcpc_en:1, dhcpd_en:1, xxxxxx:30;
25:     ...uint32 ipaddr, netmask, gw_ip;
26:     ...uint32 dhcpd_startip, dhcpd_endip, dhcpd_lease_time;
27:     ...
28: };
29:
```

- 其中前面 12 个 byte: magic_num, crc, size 是核心字段，不能修改，应用代码也不能使用这些字段；rev1, rev2, rev3 是 SDK 预留扩展字段，不建议使用。系统参数在 project/sys_cfg.c 中配置示例如下：

```
24:
25: struct sys_config sys_cfgs = {
26:     ...wifi_mode = WIFI_MODE_AP,
27:     ...channel = CHANNEL_DEFAULT,
28:     ...beacon_int = 100,
29:     ...dtim_period = 1,
30:     ...bss_max_idle = 300,
31:     ...key_mgmt = WPA_KEY_MGMT_PSK,
32:     ...ipaddr = 0x0101A8C0, //192.168.1.1
33:     ...netmask = 0x00FFFFFF, //255.255.255.0
34:     ...gw_ip = 0x0101A8C0, //192.168.1.1
35:     ...dhcpd_startip = 0x0201A8C0, //192.168.1.2
36:     ...dhcpd_endip = 0xFE01A8C0, //192.168.1.254
37:     ...dhcpd_lease_time = 3600,
38:     ...dhcpd_en = 1,
39:     ...dhcpc_en = 1,
40: };
41:
```

- **wifi_mode**: 字段是 Wi-Fi 工作模式，支持 STA、AP、AP+STA。
- **channel**: 2.4G channel 选择，0 表示自动选择，非 0 则强制 channel（STA 不能强制 channel）。
- **beacon_int**: beacon interval (ms)。
- **dtim_period**: dtim interval (beacon 个数)。
- **bss_max_idle**: bss 范围内最大的不活动时间，AP 会超时后查询 STA 是否还在。

- **key_mgmt**: 加密模式选择
- **Ipaddr**: IP 地址
- **Netmask**: 网络掩码
- **gw_ip**: 网关
- **dhcpd_startip**: DHCP 服务器分配 IP 地址范围的开始
- **dhcpd_endip**: DHCP 服务器分配 IP 地址范围的结束
- **dhcpd_lease_time**: DHCP IP 租赁时间
- **dhcpd_en**: DHCP 服务器使能
- **dhcpc_en**: DHCP 客户端使能
- **参数区存储位置**: 参数区存储位置的指定在 device.c 里面的 **syscfg_info_get** 函数，该函数指定了每个参数区所在的 flash 设备，参数区的地址及大小。SDK 默认使用了 Flash 最后 2 个 sector（2 个参数区所占用的 sector 不能重叠）。

```
int32 syscfg_info_get(struct syscfg_info *pinfo)
{
#ifdef SYSCFG_ENABLE
    pinfo->flash1 = &flash0;
    pinfo->flash2 = &flash0;
    pinfo->size = pinfo->flash1->sector_size;
    pinfo->addr1 = pinfo->flash1->size - (2 * pinfo->size);
    pinfo->addr2 = pinfo->flash1->size - pinfo->size;
    ASSERT((pinfo->addr1 & ~(pinfo->flash1->sector_size - 1)) == pinfo->addr1);
    ASSERT((pinfo->addr2 & ~(pinfo->flash2->sector_size - 1)) == pinfo->addr2);
    ASSERT((pinfo->size >= sizeof(struct sys_config) &&
            (pinfo->size == (pinfo->size & ~(pinfo->flash1->sector_size - 1))));
    return 0;
#else
    return -1;
#endif
}
```

- **参数 API:**

参数读取/存储的 API 定义在 sdk/include/lib/syscfg.h 文件中。SDK 定义了全局变量 **sys_cfgs**，该变量作为整个 SDK 参数的引用，可以在其它模块直接访问该变量。

```
#ifndef _HGIC_SYSCFG_H_
#define _HGIC_SYSCFG_H_

#define syscfg_dbg(fmt, ...) //os_printf("%s:%d:"fmt, __FUNCTION__, __LINE__, ##__VA_ARGS__)
#define syscfg_err(fmt, ...) os_printf("%s:%d:"fmt, __FUNCTION__, __LINE__, ##__VA_ARGS__)

#define SYSCFG_MAGIC (0x1234)

struct syscfg_info {
    struct spi_nor_flash *flash1, *flash2;
    uint32 addr1, addr2;
    uint32 size;
};

int32 syscfg_init(struct sys_config *cfg);
int32 syscfg_read(struct sys_config *cfg);
int32 syscfg_write(struct sys_config *cfg);
int32 syscfg_info_get(struct syscfg_info *pinfo);
void syscfg_loaddef(void);

#endif
```

其中:

- **syscfg_init**: syscfg 模块初始化，并加载 flash 参数。该函数通常在 main 函数

启动时执行，并读取 flash 参数。

- **syscfg_read:** 读取参数。会从 2 个参数区中选择有效的参数区进行读取。
- **syscfg_write:** 保存参数，交替使用 2 个参数区。
- **syscfg_info_get:** 指定参数区的存储位置信息。在 device.c 中根据实际 flash 信息进行指定。
- **syscfg_loaddef:** 参数区 load default，会设置 2 个参数区都为无效区。

系统时钟 SDK 默认配置为 DEFAULT_SYS_CLK = 180Mhz，方案需要修改系统时钟时需要重写函数：system_clock_init， system_clock_init 源码如下：

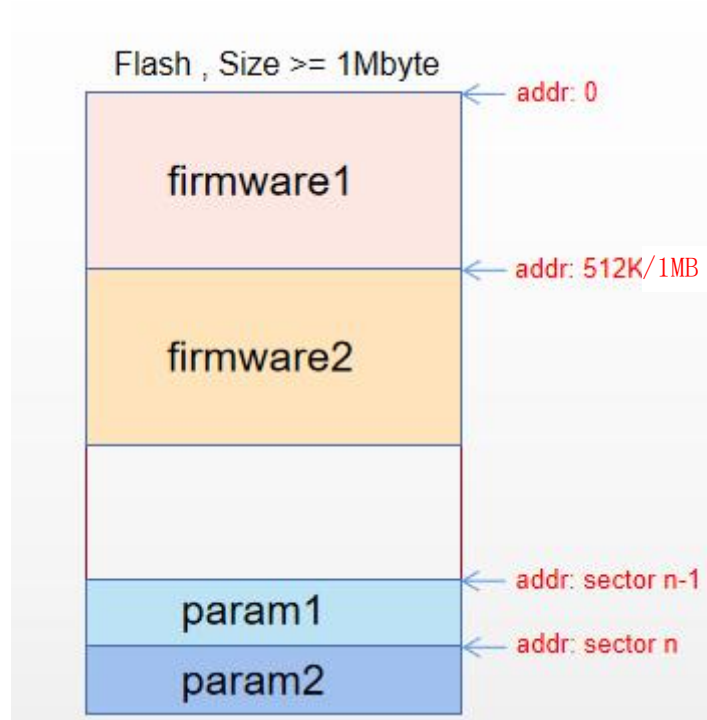
```
#define DEFAULT_SYS_CLK    180000000UL

__weak void system_clock_init(void)
{
    if (!sysctrl_cmu_sysclk_set(DEFAULT_SYS_CLK, 0)) {
        while (1);
    }
}
```

4.6. FLASH 布局

SDK 设计了双固件区和双参数区，其中 2 个固件区地址固定为 0 和 512KB/1MB 的位置。参数区默认使用 flash 最后 2 个 sector，大小为 1 个 sector。

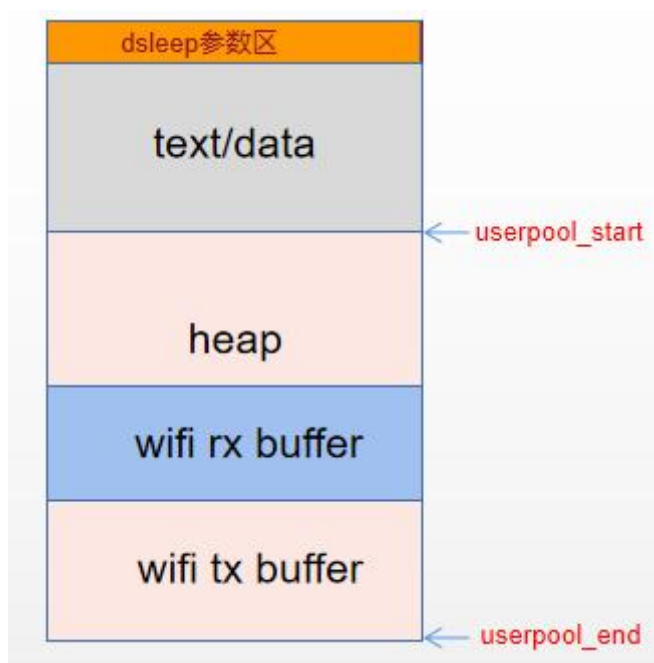
默认 Flash 分区如下：



参数区可以根据实际情况进行调整。

4.7. Memory 布局

TXW80X 芯片的 SRAM Size 为 288KByte，整体分布如下：



- dsleep 参数区：最前面为 Deep Sleep 功能参数区。
- text/data：程序的代码段和 data 段。
- heap：程序可使用的 heap 区间。heap 区间大小可根据实际应用需求进行设定。

- WiFi rx buffer: Wi-Fi rx buffer 区间
- WiFi tx buffer: Wi-Fi tx buffer 区间

可以根据 tx/rx 数据量的大小来调整 Wi-Fi tx/rx buffer 的大小。例如实际通信 tx 数据多，rx 数据少，则可以调大 tx buffer，减少 rx buffer。

以上各个 buffer 区间的设置在 sys_config.h 中进行定义。

```

27:
28: #ifndef SYS_HEAP_SIZE
29: #define SYS_HEAP_SIZE (108*1024)           Heap 默认 size
30: #endif
31:
32: #ifndef WIFI_RX_BUFF_SIZE
33: #define WIFI_RX_BUFF_SIZE (17*1024)       Wi-Fi rx buffer size
34: #endif
35:
36: #define SYS_HEAP_START (USER_POOL_START)   Heap buffer 地址
37: #define WIFI_RX_BUFF_ADDR (SYS_HEAP_START+SYS_HEAP_SIZE) Wi-Fi rx buffer 地址
38: #define SKB_POOL_ADDR (WIFI_RX_BUFF_ADDR+WIFI_RX_BUFF_SIZE) Wi-Fi tx buffer 地址
39: #define SKB_POOL_SIZE (USER_POOL_START+USER_POOL_SIZE-SKB_POOL_ADDR)
40:
41:

```

4.8. 方案开发基本规则

方案开发需要遵循的一些基本准则，包括头文件引用、内存分配、延时、打印和数据格式转换等；另外方案开发常见配置主要看 project_config.h 和 sys_config.h，里面包含了系统参数、存储空间、外设功能和参数等配置。

- 头文件引用

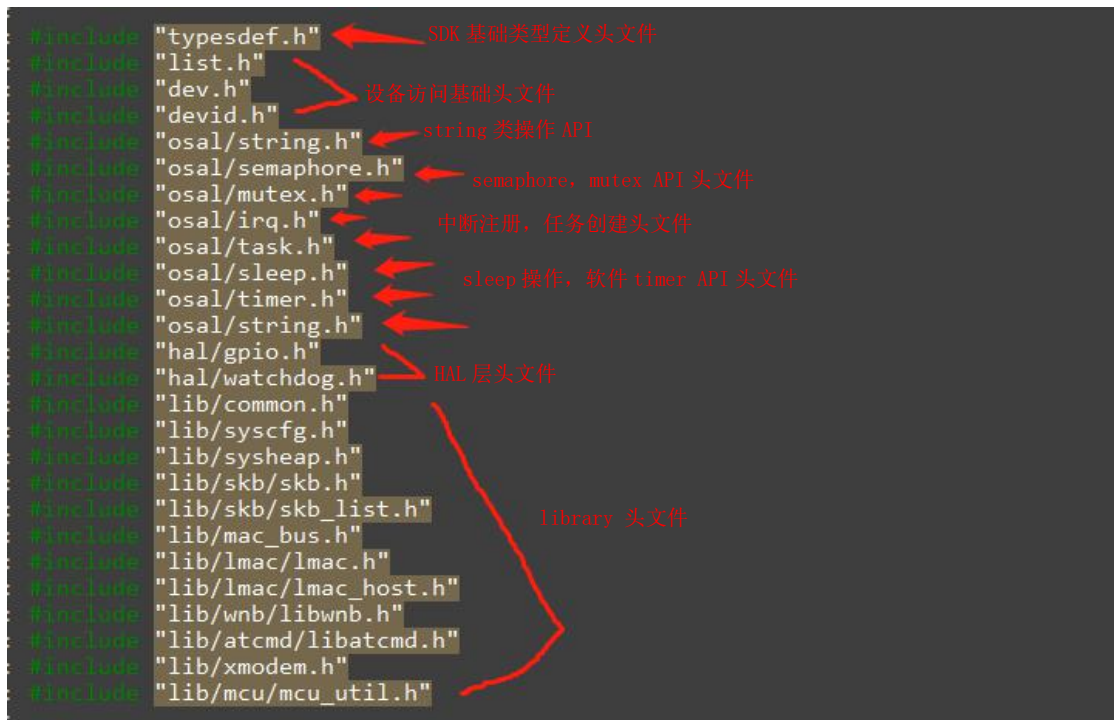
使用 SDK 进行二次开发代码时，头文件引用请使用 SDK 定义的头文件。头文件引用基本顺序如下图所示。

typedef.h 是基础类型头文件，所有源码文件只引用该头文件即可，不要直接引用编译器的头文件。

list.h, dev.h, devid.h 是设备访问基础头文件

设备 HAL api 在 include/hal 目录，根据实际访问的设备，添加引用对应的头文件。

osal/string.h, osal/semaphore.h, osal/xxx 是 OSAL API，头文件在 include/osal 目录。需要使用这类 API 时，引用 osal/xxx.h 即可。



● 常用 API

调试打印

- ◆ `os_printf`: 带时间戳的打印 API
- ◆ `dump_hex`: 以十六进制格式打印数据

内存分配

- ◆ `os_malloc`: 申请内存
- ◆ `os_free`: 释放内存
- ◆ `os_zalloc`: 申请内存, 并清理。
- ◆ `os_realloc`: 申请内存, 复制原内存的数据, 释放原内存空间。

Sleep 函数

- ◆ `os_sleep`: task sleep API, 单位秒
- ◆ `os_sleep_ms`: task sleep API, 单位毫秒

格式转换

- ◆ `hex2int`: 十六进制字符串转换成 int 类型数据
- ◆ `hex2char`: 十六进制字符串转换成 char 类型数据
- ◆ `hex2bin`: 十六进制字符串转换成 char 数组
- ◆ `str2mac`: MAC 地址字符串转换成 char 数组

4.9. 驱动 API 使用说明

SDK 设计了硬件抽象层 HAL，应用程序访问外设请使用 HAL API。HAL api 会执行具体 driver 的 API。

应用程序访问外设基本步骤：

1. 获取需要访问的 device：使用 `dev_get` API 获取需要访问的外设 device。该 API 通过指定的 device id 获取已经初始化的 device。device id 定义在 `include/devid.h` 文件中。

2. 使用 HAL API 访问 device。

基本示例如下：

```
static void watchdog_task(void *args)
{
    struct watchdog_device *wd = (struct watchdog_device *)dev_get(HG_WDTV1_DEVID);
    ASSERT(wd);
    watchdog_device_open(wd, 0);
    watchdog_device_start(wd, 5);
    while (1) {
        watchdog_device_feed(wd);
        os_sleep(2);
    }
}
```



4.10. OSAL API 使用说明

SDK 设计了 OS 抽象层，提供了常用的 semaphore/mutex/timer/irq/msg queue/task 等功能的 OSAL API，代码开发时请使用 OSAL 提供的 API。

4.10.1. Task

Task OSAL API 定义在 `include/osal/task.h` 文件中。


```

1: typedef enum {
2:     OS_TASK_PRIORITY_IDLE      = 0,
3:     OS_TASK_PRIORITY_LOW      = 0x10,
4:     OS_TASK_PRIORITY_BELOW_NORMAL = 0x20,
5:     OS_TASK_PRIORITY_NORMAL    = 0x30,
6:     OS_TASK_PRIORITY_ABOVE_NORMAL = 0x40,
7:     OS_TASK_PRIORITY_HIGH      = 0x50,
8:     OS_TASK_PRIORITY_REALTIME   = 0x60,
9:     OS_TASK_PRIORITY_ISR       = 0xFF,
10: } OS_TASK_PRIORITY;
11:
12: #define OS_TASK_INIT(name, task, func, data, prio, stksize) do { \
13:     os_task_init((const uint8 *)name, task, (os_task_func_t)func, (uint32)data); \
14:     os_task_set_stacksize(task, stksize); \
15:     os_task_set_priority(task, prio); \
16:     os_task_run(task); \
17: }while(0)
18:
19: struct os_task;
20:
21: struct os_task_time {
22:     uint32 id;
23:     const uint8 *name;
24:     uint32 time;
25: };
26:
27: typedef void (*os_task_func_t)(void *arg);
28:
29: int32 os_task_init(const uint8 *name, struct os_task *task, os_task_func_t func, uint32 data);
30: int32 os_task_priority(struct os_task *task);
31: int32 os_task_stacksize(struct os_task *task);
32: int32 os_task_set_priority(struct os_task *task, uint8 priority);
33: int32 os_task_set_stacksize(struct os_task *task, int32 stack_size);
34: int32 os_task_run(struct os_task *task);
35: int32 os_task_stop(struct os_task *task);
36: int32 os_task_del(struct os_task *task);
37: int32 os_task_runtime(struct os_task_time *task_times, int32 count);
38: struct os_task os_task_current(void);

```

● OS_TASK_INIT

该宏定义执行了 task 的初始化，堆栈大小/优先级设置，并启动运行 task。使用该宏定义可以简化 task 创建。

具体示例可以参考 watchdog task:

```

1: int main(void)
2: {
3:     int8 print_interval = 0;
4:
5:     OS_TASK_INIT("watchdog", &wdog_task, watchdog_task, 0, OS_TASK_PRIORITY_NORMAL, 256);
6:     syscfg_init(&sys_cfgs);
7:     sys_atcmd_init();
8:     sys_wifi_init();
9:     sys_keyled_init();
10: }

```

Diagram labels for the OS_TASK_INIT call:

- task name: "watchdog"
- task object: &wdog_task
- task function: watchdog_task
- task data: 0
- task priority: OS_TASK_PRIORITY_NORMAL
- task stack: 256

● os_task_init: task 初始化

int32 os_task_init(const uint8 *name, struct os_task *task, os_task_func_t func, uint32 data)

参数:

- name: task 名称
- task: 具体 task 对象
- func: task 需要执行的 function
- data: 执行 function 需要的参数

返回值:

- RET_OK: 初始化成功
- RET_ERR: 初始化失败

- **os_task_priority:** 查看 task 的优先级
`int32 os_task_priority(struct os_task *task)`
 参数:
 - task: 需要查看的 task 对象
 返回值:
 - task 的优先级
- **os_task_stacksize:** 查看 task 的 stack size
`int32 os_task_stacksize(struct os_task *task)`
 参数:
 - task: 需要查看的 task 对象
 返回值:
 - task 的堆栈 size
- **os_task_set_priority:** 设置 task 的优先级 (**task 运行之前设置有效**)。
`int32 os_task_set_priority(struct os_task *task, uint8 priority)`
 参数:
 - task: 需要设置的 task 对象
 - priority: task 的优先级。优先级定义: OS_TASK_PRIORITY
 返回值:
 - RET_OK: 设置成功
 - RET_ERR: 设置失败
- **os_task_set_stacksize:** 设置 task 的堆栈 size (**task 运行之前设置有效**)。
`int32 os_task_set_stacksize(struct os_task *task, int32 stack_size)`
 参数:
 - task: 需要设置的 task 对象
 - stack_size: task 的堆栈 size
 返回值:
 - RET_OK: 设置成功
 - RET_ERR: 设置失败
- **os_task_run:** 启动运行 task
`int32 os_task_run(struct os_task *task)`
 参数:
 - task: 需要启动运行的 task 对象
 返回值:
 - RET_OK: task 启动运行成功 (**不代表 task 立即被执行**)。
 - RET_ERR: task 启动运行失败。

4.10.2. Mutex

Mutex 功能 OSAL API 定义在 include/osal/mutex.h, 如下图所示:

```
#ifndef _OS_MUTEX_H_
#define _OS_MUTEX_H_    include/osal/mutex.h

#ifdef RTOS_ENABLE
#ifdef __cplusplus
extern "C" {
#endif

struct os_mutex;

int32 os_mutex_init(struct os_mutex *mutex);
int32 os_mutex_lock(struct os_mutex *mutex, int32 tmo);
int32 os_mutex_unlock(struct os_mutex *mutex);
int32 os_mutex_del(struct os_mutex *mutex);

#ifdef __cplusplus
}
#endif

#ifdef __MBED__
#include "osal/mbed/mutex.h"
#endif

#ifdef __CSKY__
#include "osal/csky/mutex.h"
#endif
#else
struct os_mutex { uint32 val; };
#define os_mutex_init(m) (0)
#define os_mutex_lock(m, t) (0)
#define os_mutex_unlock(m) (0)
#define os_mutex_del(m) (0)
#endif
#endif
```

超时时间, 单位: 毫秒

4.10.3. Semaphore

Semaphore 功能 OSAL API 定义在 include/osal/semaphore.h

```

#ifndef __OS_SEMAPHORE_H
#define __OS_SEMAPHORE_H           include/osal/semaphore.h

#ifdef __cplusplus
extern "C" {
#endif

struct os_semaphore;

int32 os_sema_init(struct os_semaphore *sem, int32 val);
int32 os_sema_del(struct os_semaphore *sem);
int32 os_sema_down(struct os_semaphore *sem, int32 tmo_ms);
int32 os_sema_up(struct os_semaphore *sem);

#ifdef __cplusplus
}
#endif

#ifdef MBED
#include "osal/mbed/semaphore.h"
#endif

#ifdef CSKY
#include "osal/csky/semaphore.h"
#endif

#ifndef osWaitForever
#define osWaitForever 0xFFFFFFFFu
#endif
#endif

```

sema 初始值，通常为 0

超时时间，单位：毫秒

wait forever 宏定义

4.10.4. Msgqueue

Message Queue 功能 OSAL API 定义在 include/osal/msgqueue.h 文件中。Mesasge queue 存储的是 uint32 类型数据。

```

#ifndef __OS_MSG_QUEUE_H_
#define __OS_MSG_QUEUE_H_

#ifdef __cplusplus
extern "C" {
#endif

struct os_msgqueue;

int32 os_msgq_init(struct os_msgqueue *msgq, int32 size);
uint32 os_msgq_get(struct os_msgqueue *msgq, int32 tmo_ms);
int32 os_msgq_put(struct os_msgqueue *msgq, uint32 data, int32 tmo_ms);
int32 os_msgq_del(struct os_msgqueue *msgq);
int32 os_msgq_cnt(struct os_msgqueue *msgq);

#ifdef __cplusplus
}
#endif

#ifdef __MBED__
#include "osal/mbed/msgqueue.h"
#endif

#ifdef __CSKY__
#include "osal/csky/msgqueue.h"
#endif

#ifndef OS_MSGQ_DEF
#define OS_MSGQ_DEF(name, size) struct os_msgqueue name;
#endif

#ifndef osWaitForever
#define osWaitForever 0xFFFFFFFF
#endif

#endif

```

include/osal/msgqueue.h

msg queue 的 size

超时时间，单位毫秒

超时时间，单位毫秒

wait forever

- **os_msgq_init:** 消息队列初始化，指定队列大小。
int32 os_msgq_init(struct os_msgqueue *msgq, int32 size);
参数：
 - msgq: 需要初始化的 msg queue
 - size: 指定 msg queue 的 size。初始化时会从 heap 申请 size*sizeof(uint32) 大小的 memory 空间。
返回值：
 - RET_OK: 初始化成功
 - RET_ERR: 初始化失败
- **os_msgq_get:** 从 msg queue 提取数据，可指定超时时间
uint32 os_msgq_get(struct os_msgqueue *msgq, int32 tmo_ms);
参数：
 - msgq: 需要提取数据的 msg queue
 - tmo_ms: get 操作超时时间，单位毫秒；或者永久等待：osWaitForever
返回值

- 从 msg queue 提供的数据
- **os_msgq_put:** 向 msg queue 中存入数据
`int32 os_msgq_put(struct os_msgqueue *msgq, uint32 data, int32 tmo_ms);`
 参数:
 - msgq: 需要存入数据的 msg queue
 - data: 需要存入的数据
 - tmo_ms: put 操作超时时间, 单位毫秒; 或者永久等待: `osWaitForever`
 返回值
 - RET_OK: 数据成功存入 msg queue
 - 其它值: 存入失败。
- **os_msgq_del:** 删除 msg queue
`int32 os_msgq_del(struct os_msgqueue *msgq);`
- **os_msgq_cnt:** 查看当前 msg queue 中存储的数据个数。
`int32 os_msgq_cnt(struct os_msgqueue *msgq);`

4.10.5. SoftTimer

Soft Timer 的 OSAL API 定义在 `osal/timer.h` 文件中。

Soft timer 是基于 OS tick 运行的, 所以 timer 的时间精度就是 os tick, SDK 默认 OS Tick 是 1ms。

使用 soft timer 的注意事项:

1. 尽量缩短 timer 的 callback 函数执行时间。所有的 soft timer 共用 1 个执行序列, 某个 timer 执行时间过长会影响其它 timer 的执行。
2. soft timer 有 2 种工作模式:
 - ONCE: timer 启动触发后, 需要再次启动才会运行。
 - PERIODIC: timer 只需要启动一次, 会根据指定的超时时间循环周期性触发。

```

#ifndef __OS_TIMER_H_
#define __OS_TIMER_H_

#ifdef __cplusplus
extern "C" {
#endif

enum OS_TIMER_MODE {
    OS_TIMER_MODE_ONCE,
    OS_TIMER_MODE_PERIODIC
};

struct os_timer;

typedef void (*os_timer_func_t)(void *arg);

int os_timer_init(struct os_timer *timer, os_timer_func_t func,
                  enum OS_TIMER_MODE mode, void *arg);
int os_timer_start(struct os_timer *timer, unsigned long expires);
int os_timer_stop(struct os_timer *timer);
int os_timer_del(struct os_timer *timer);

#ifdef __cplusplus
}
#endif

#ifdef __MBED__
#include "osal/mbed/timer.h"
#endif

#ifdef __CSKY__
#include "osal/csky/timer.h"
#endif

#endif

```

4.11. 调试信息

SDK 提供了基本的调试手段，用于开发过程中的问题排查。

4.11.1. WiFi 协议栈调试信息输出

WiFi 协议栈底层调试信息，主要是用于分析无线传输问题分析。在无线传输遇到问题，请抓取该打印信息进行分析。

```

-----
local:06:07:08:09:12:67
  chip-temperature: 39
  freq:2437, bg_rssi:-76
  gain_table:0
  tx: tx dma:3412, total tx:3410, retry:439, tx lost:5, tx err:0
  rx: rx irq:1263
  rx err: phy err:503, rx frm err:1383,

sta:06:07:08:09:12:22, aid:1, rssi:-37, evm:-24, tx frm type:*3, tx mcs:*5, freq offset:3042
-----

```

WiFi 协议栈调试信息

部分打印信息的含义如下:

local: 当前 MAC 地址。

chip-temperature: 当前芯片温度。

freq: 当前工作频点。

bg_rssi: 最近一次的背景噪声, 单位是 dB。

tx: 从上一次打印到现在的 tx 统计信息。其中 tx dma 代表总共 dma 的帧数; total tx 代表已经发送完成的帧数; retry 代表空口重传的次数; tx lost 代表发送失败的帧数; tx err 代表发送出错的次数。

rx: 从上一次打印到现在的 rx 统计信息。rx irq 代表进入 rx 接收帧中断的次数。

rx err: 从上一次打印到现在的 rx 接收出错的统计信息。由于 2.4G 干扰多且复杂, 会出现大量的 rx err。

sta: 代表已连接 sta 的信息。aid 是 sta 的 aid 信息; rssi 代表最近一次 sta 的信号强度; evm 代表最近一次接收帧的信号质量 (越小越好), 只有接收 OFDM 调制方式的帧才会更新; tx frm type 和 tx mcs 用于代表最近一次我们往 sta 发送的帧格式; freq offset 代表我们感知到的最近一次 sta 频偏信息。

注意: 可以通过 AT 命令 (AT+PRINT_PERIOD=__) 设置该打印的时间, 单位是 ms。时间设置为 0 时, 则关闭打印。

4.11.2. 系统 Heap 使用情况

调用 `sysheap_status()` 可以打印系统 Heap 使用情况, 该打印信息会输出当前 heap 剩余情况。可以根据该打印信息调整 heap size, 使用 heap size 保持够用即可, 使 tx/rx buffer 可以使用更多的 memory。

```

[6073]-----
[6076]sysheap free regions: 2
[6079]01: 0x20056ffc ~ 0x200571a4, size:424      [8]
[6083]02: 0x200579e4 ~ 0x2005b3d8, size:14836   [13]
[6088]total free size:15260
[6091]-----
[6095]total size:40960
[6098]
LMAC STATUS:

```

系统 heap 使用情况: heap size: 40K
还剩余 15260bytes, 分为 2 个内存片

4.11.3. CPU 使用率

调用 `cpu_loading_print()` 可以打印 CPU 使用率, 该打印信息显示了各个 Task 的 CPU

使用情况。从该信息可以分析代码运行是否存在异常。

```
[6073]-----
[6075]Task Runtime Statistic, interval:6073ms
[6079]-----
[6084] 1 idle_task      92% (5626)
[6087] 2 timer_task     0% (1)
[6090] 3 MAIN           1% (70)
[6093] 7 lmac tx        2% (159)
[6096] 8 lmac tx status 0% (24)
[6099] 9 lmac rx        0% (48)
[6102]10 lmac_test      0% (1)
[6105]13 wrb           2% (125)
[6108]-----
[6112]
```

各个 Task 的 CPU 使用率

4.11.4. XIP 取指效率

对于跑 XIP 应用的方案，需要关心代码取指效率，这个主要受 SPI FLASH 读模式、频率以及 Cache 命中率影响。其中 SPI FLASH 读模式、频率在工程 makecode.ini 中配置；Cache 命中率 = $1 - \text{cpfmtr}/\text{cpfatr}$ 。其中 Cache 丢失 (cpfmtr) 和命中 (cpfatr) 次数可以通过函数 `csi_cache_get_miss_time()` 和 `csi_cache_get_access_time()` 获取，当需要统一某段时间内的 cache 命中率时，建议统计前调用 `csi_cache_reset_profile()` 清除历史数据。

当方案代码运行吃力时可以考虑以下优化方向：

- 1、提高 SPI FLASH 访问效率（提高线模式、提高频率等）
- 2、优化代码提高 Cache 命中率（加强代码时间和空间局部性）
- 3、优化代码自身执行效率

5. 其他学习资源

【TXW80X 技术规格书】

该手册介绍了 TXW80X 芯片， 包含 TXW80X 产品概述（特性、功能框图、管脚定义和布局、管脚功能）、功能描述（CPU、存储和闪存、时钟、模拟和数字外设等）、电气参数（工作条件、直流和交流特性、功耗、可靠性、射频性能等）等信息。

【TXW80X 硬件设计指南】

该手册介绍了 TXW80X 硬件设计， 包含 TXW80X 主要电源、射频和高速模块设计和 layout 指南和注意事项、原理图等。

【TXW80X SDK API 参考手册】

该手册介绍了 TXW80X SDK 抽象层 API。

【TXW80X FAQ】

该手册介绍了 TXW80X 方案开发过程中的疑难答疑。

【TXW80X 视频开发指南】

该手册介绍了 TXW80X 视频方案开发，包括 DVP/MJPEG/USB 配置、DVP 镜头驱动、USB Sensor 驱动、以及视频和图片数据流。

【TXW80X USB 开发指南】

该手册介绍了 TXW80X USB 方案开发，包括 USB Host 和 USB device 初始化和工作流程等。

【TXW80X 蓝牙配网开发指南】

该手册介绍了 TXW80X 蓝牙配网开发。

【TXW80X 音频开发指南】

该手册介绍了 TXW80X 音频方案开发，包括音频子板介绍、 PDM/IIS 音频接口使用和音频数据流等。

【TXW80X IOT 开发指南】

该手册介绍了 TXW80X IOT 方案开发。

【TXW80X 低功耗开发指南】

该手册介绍了 TXW80X 低功耗方案开发。

【TXW80X 认证测试指南】

该手册介绍了 TXW80X 方案认证测试。

【TXW80X 天线匹配和测试指南】

该手册介绍了 TXW80X 方案认证测试。

【TXW80X 量产和烧录指南】

该手册详细解释了 TXW80X 量产烧录相关配置、工具和流程，包括在线烧录和离线烧录。