第一章 ESP8266 模组简介

1.1 ESP8266 芯片概述

ESP8266 系列模组是深圳市安信可科技有限公司开发的一系列基于乐鑫 ESP8266 的超低功耗的 UART-WiFi 模块的模组，可以方便地进行二次开发，接入云端服务，实现手机 3/4G 全球随时随地的控制，加速产品原型设计。

模块核心处理器 ESP8266 在较小尺寸封装中集成了业界领先的 Tensilica L106 超低功耗 32 位微型 MCU，带有 16 位精简模式，主频支持 80 MHz 和 160 MHz，支持 RTOS，集成 Wi-Fi MAC/ BB/RF/PA/LNA，板载天线。支持标准的 IEEE802.11 b/g/n 协议，完整的 TCP/IP 协议栈。用户可以使用该模块为现有的设备添加联网功能，也可以构建独立的网络控制器。

ESP8266 是高性能无线 SOC，以最低成本提供最大实用性，为 WiFi 功能嵌入其他系统提供无限可能。

特点

- 802.11 b/g/n
- 内置 Tensilica L106 超低功耗 32 位微型 MCU，主频支持 80 MHz 和 160 MHz，支持 RTOS
- 内置 10 bit 高精度 ADC
- 内置 TCP/IP 协议栈
- 内置 TR 开关、balun、LNA、功率放大器和匹配网络
- 内置 PLL、稳压器和电源管理组件，802.11b 模式下+20 dBm 的输出功率
- MPDU 、 A-MSDU 的聚合和 0.4 s 的保护间隔
- WiFi @ 2.4 GHz，支持 WPA/WPA2 安全模式
- 支持 AT 远程升级及云端 OTA 升级
- 支持 STA/AP/STA+AP 工作模式
- 支持 Smart Config 功能（包括 Android 和 iOS 设备）
- HSPI 、UART、I2C、I2S、IR Remote Control、PWM、GPIO
- 深度睡眠保持电流为 10 uA，关断电流小于 5 uA
- 2 ms 之内唤醒、连接并传递数据包
- 待机状态消耗功率小于 1.0 mW (DTIM3)
- 工作温度范围：-20℃- 85℃

1.2 ESP8266 芯片介绍

1.2.1 芯片引脚介绍

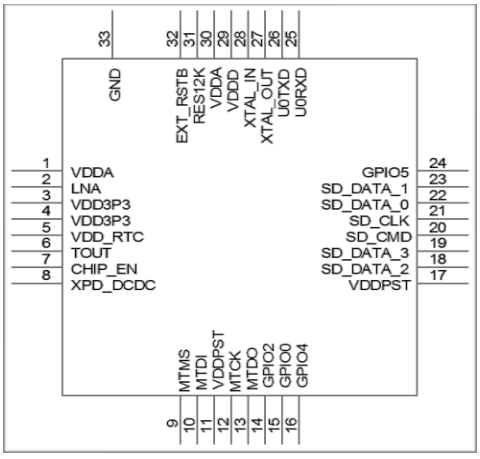


图 1.2.1 ESP6288 芯片引脚

1.2.2 芯片引脚功能定义

管脚	名称	类型	功能
1	VDD	P	模拟电源 3.0 V ~ 3.6 V
2	LNA	I/O	射频天线接口，芯片输出阻抗为 50Ω 无需对芯片进行匹配，但建议保留Π型匹配忘了对天线进行匹配
3	VDD3P3	P	功放电源 3.0V~3.6V
4	VDD3P3	P	功放电源 3.0V~3.6V
5	VDD_RTC	P	NC(1.1V)
6	TOUT	I	ADC 端口(注:芯片内部 ADC 端口),可用于检测 VDD3P3(Pin3,Pin4) 电源电压和 TOUT(Pin6)的输入电压。(二者不可同时使用)
7	CHIP_PU	I	芯片使能端 高电平：有效，芯片正常工作；低电平：芯片关闭，电流很小
8	XPD_DCDC	I/O	深度睡眠唤醒；GPIO16
9	MTMS	I/O	GPIO14;HSPI_CLK
10	MTDI	I/O	GPIO12;HSPI_MISO
11	VDDPST	P	数字 I/O 电源(1.8V~3.3V)
12	MTCK	I/O	GPIO13; HSPI_MOSI; UART0_CTS
13	MTDO	I/O	GPIO15; HSPI_CS; UART0_RTS
14	GPIO2	I/O	可用作烧写闪存时 UART1_TX；GPIO2
15	GPIO0	I/O	GPIO0; SPI_CS2
16	GPIO4	I/O	GPIO4
17	VDDPST	P	数字 I/O 电源(1.8V~3.3V)
18	SDIO_DATA_2	I/O	连接到 SD_D2(穿了 200Ω)；PIHD;HSPIHD;GPIO9
19	SDIO_DATA_3	I/O	连接到 SD_D3(穿了 200Ω)；SWIWP;HSPIWP;GPIO10
20	SDIO_CMD	I/O	连接到 SD_CMD(穿了 200Ω)；SPI_CS0;GPIO11
21	SDIO_CLK	I/O	连接到 SD_CLK(穿了 200Ω)；SPI_CLK;GPIO12
22	SDIO_DATA_0	I/O	连接到 SD_D0(穿了 200Ω)；SPI_MISO;GPIO13
23	SDIO_DATA_1	I/O	连接到 SD_D1(穿了 200Ω)；SPI_MOSI;GPIO14
24	GPIO5	I/O	GPIO5
25	U0RXD	I/O	可用作烧写闪存时 UART_TX；GPIO3
26	U0TXD	I/O	可用作烧写闪存时 UART_RX；GPIO1;SPI_CS1
27	XTAL_OUT	I/O	连接晶振输出端，也可用于 BT 的时钟输入
28	XTAL_IN	I/O	连接晶振输入端
29	VDDD	P	模拟电源 3.0 V ~ 3.6 V
30	VDDA	P	模拟电源 3.0 V ~ 3.6 V
31	RES12K	I	串联 12KΩ电阻到地
32	EXT_RSTB	I	外部重置信号(低电平有效)
33	GND	P	电源地(大焊盘)

1.3 ESP8266 模组介绍

1.3.1 模组引脚介绍

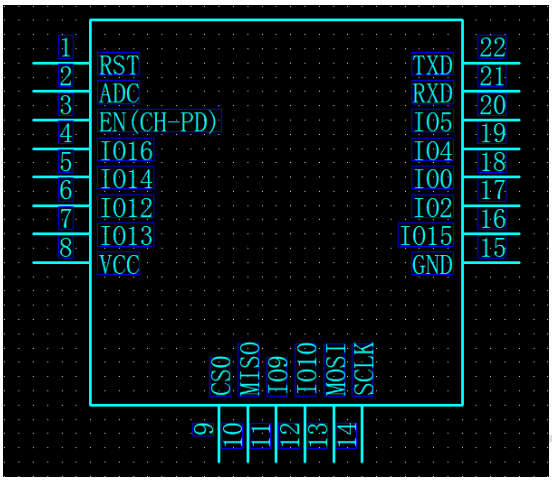
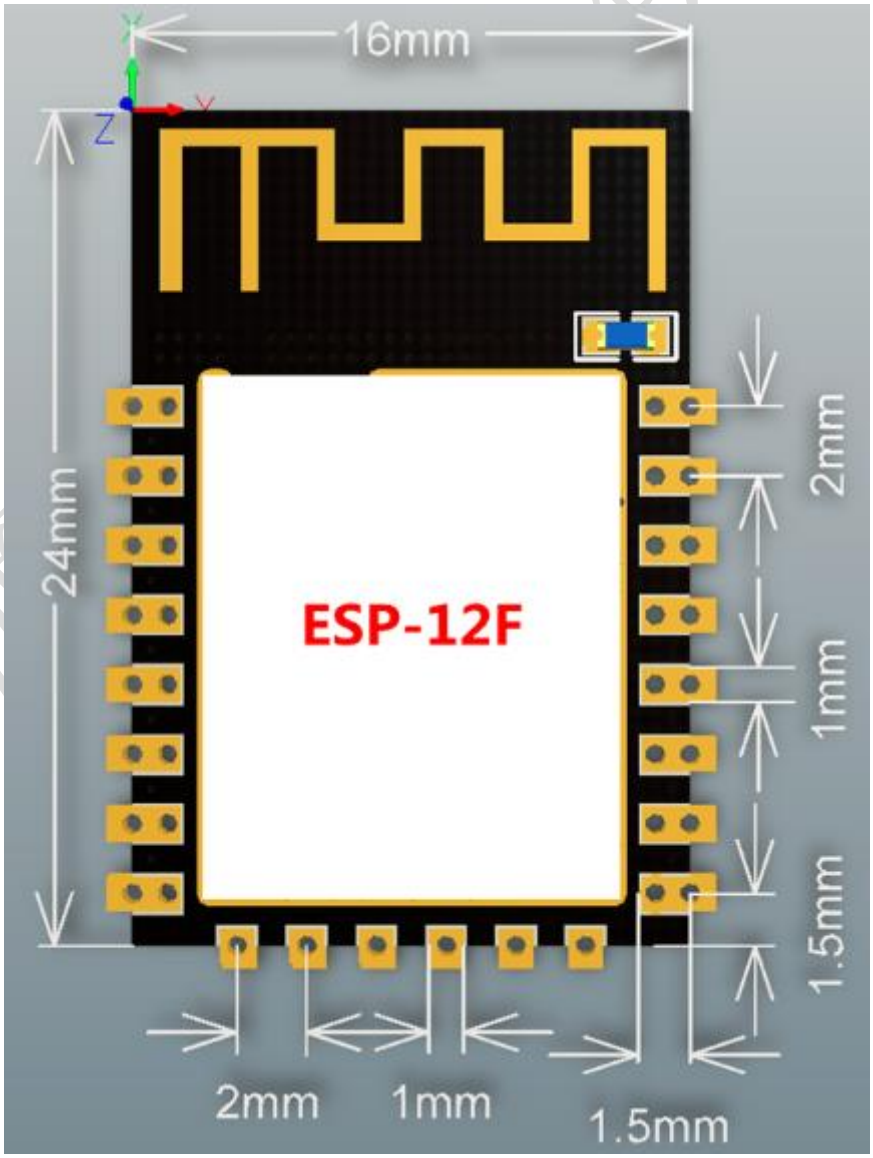


图 1.3.1 ESP8266 模组 IO 口定义

1.3.2 ESP8266 模组封装定义



第二章 ESP8266 环境搭建

2.1 软件下载及安装

该教程文档参考安信可官方文档教程，网址：http://wiki.ai-thinker.com/ai_ide_install

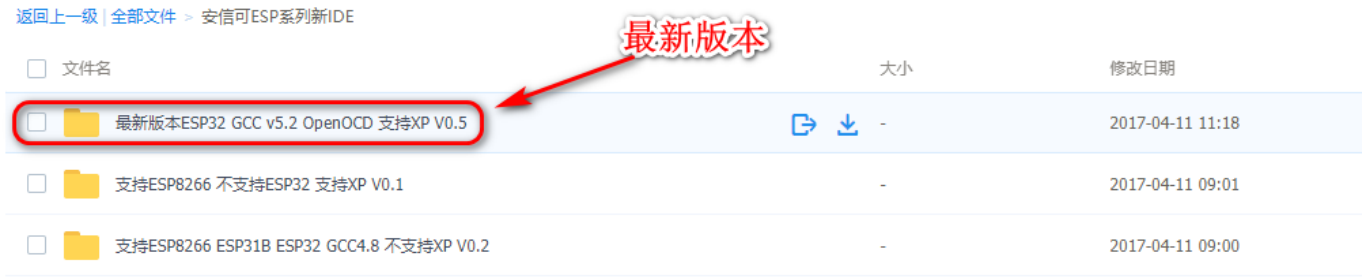
2.1.1 简介

安信可一体化开发环境有以下特点：

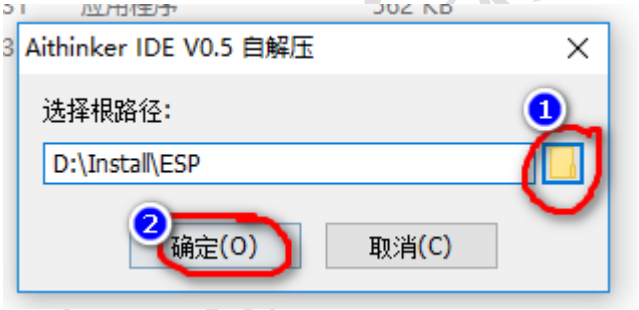
- 支持 ESP8266 NONOS 和 FreeRTOS 环境开发
- 支持 ESP31B/ESP32 FreeRTOS 环境开发
- 下载即用，无需另外配置环境
- 可直接编译所有乐鑫官方推出的 SDK 开发包

2.1.2 软件下载

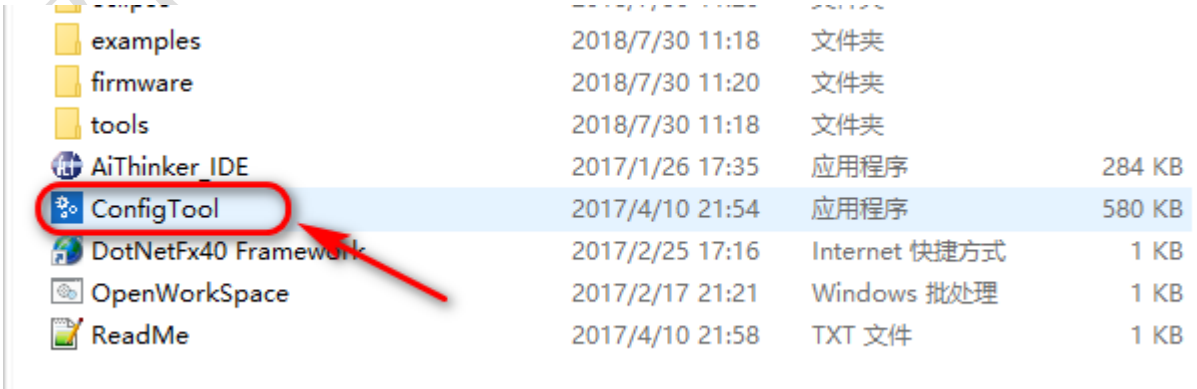
下载网址：https://pan.baidu.com/s/1skRvR1j#list/path=%2F&parentPath=%2F

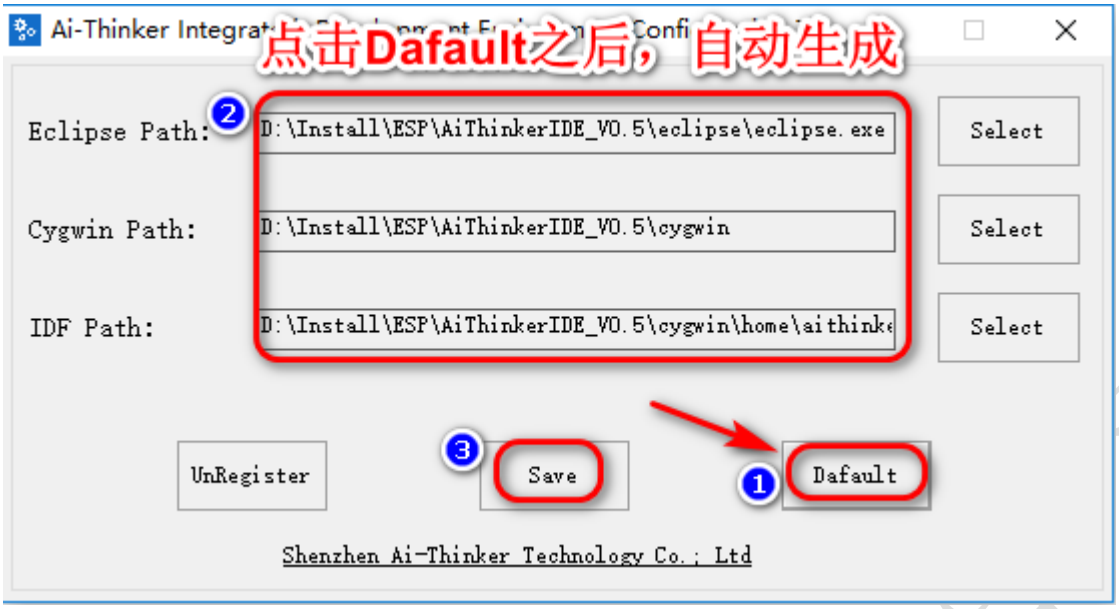


下载之后是一个自解压文件，双击文件即可安装(※安装时不要出现中文 空格等特殊符号※)

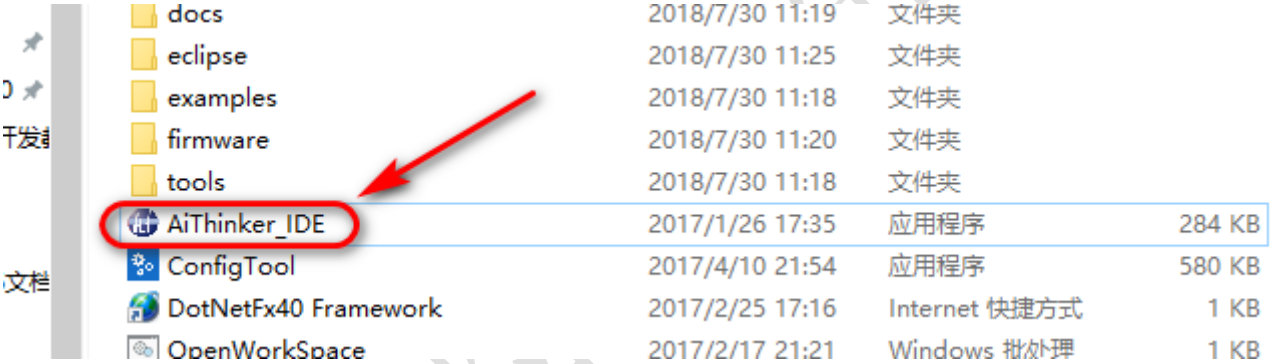


自解压之后，找到文件，先对文件进行配置(最好以管理员身份运行)

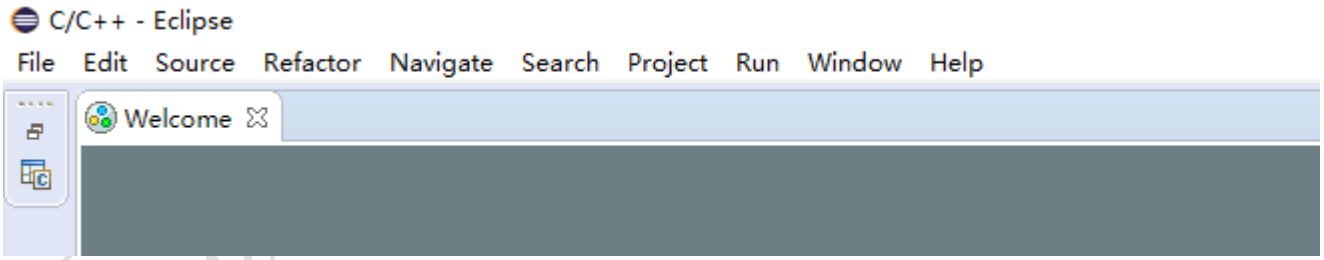




双击即可打开软件，软件打开时间相对比较长，耐心等待



软件工程打开



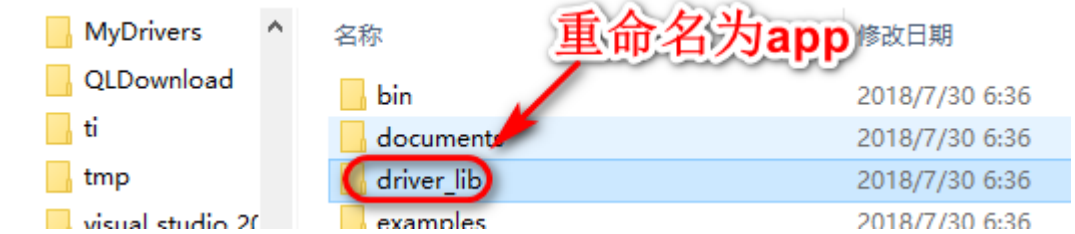
2.1.3 搭建环境

该教程文档参考安信可官方文档教程，网址：http://wiki.ai-thinker.com/ai_ide_use

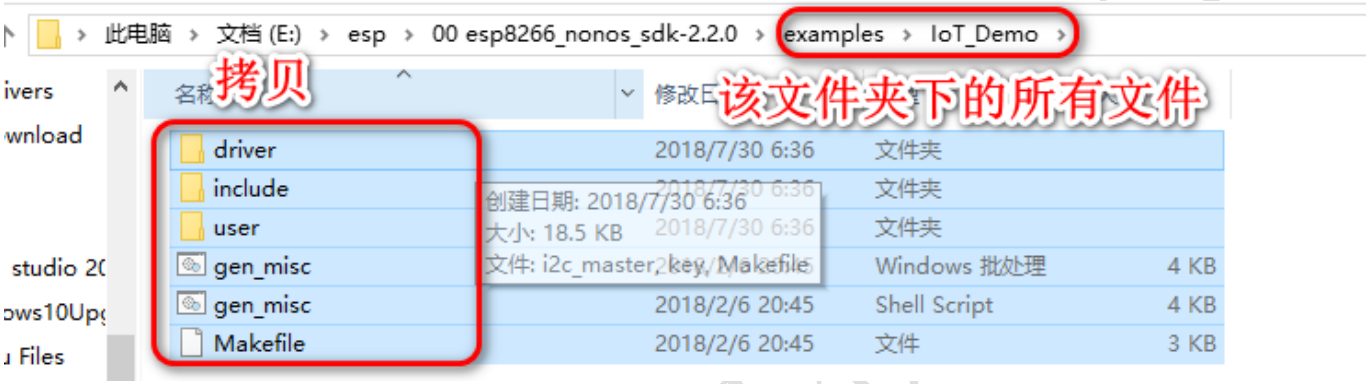
下载 esp8266_nonos_sdk-2.2.0.zip DEMO 模板，下载网址：
http://wiki.ai-thinker.com/_media/esp8266/sdk/esp8266_nonos_sdk-2.2.0.zip

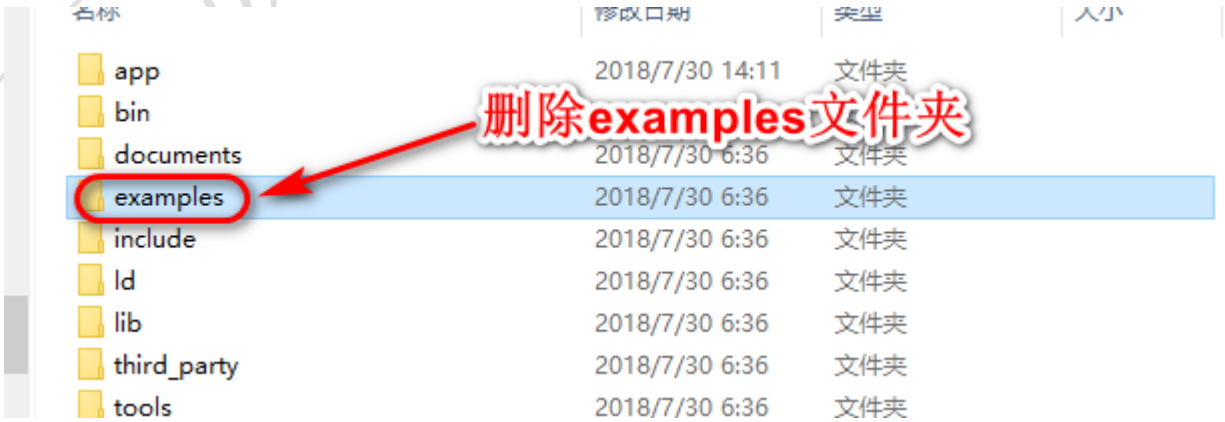
下载完成之后解压文件，之后对工程进行修改操作。

将 sdk 目录下的 driver_lib 重命名为 app

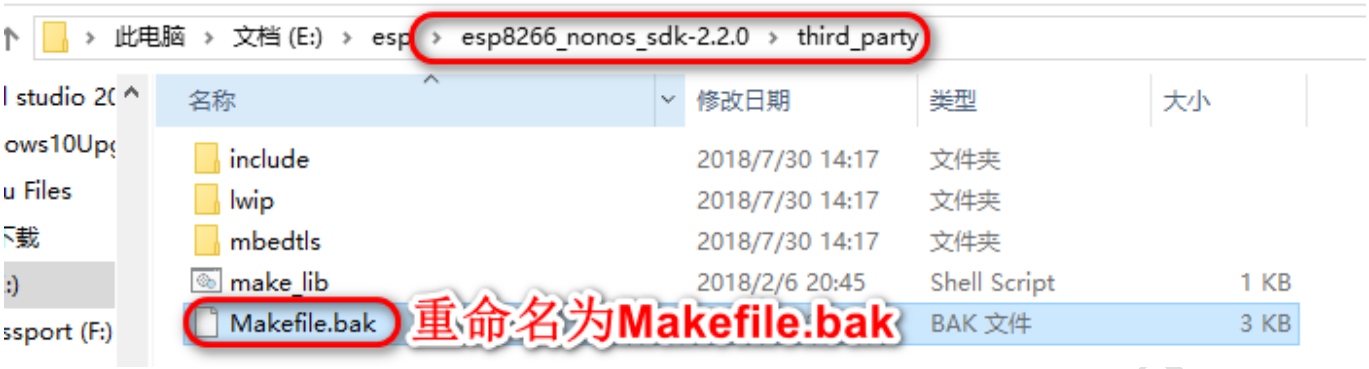


拷贝 \examples\IoT_Demo 下的所有文件到刚才的 app 目录（提示覆盖则确认）





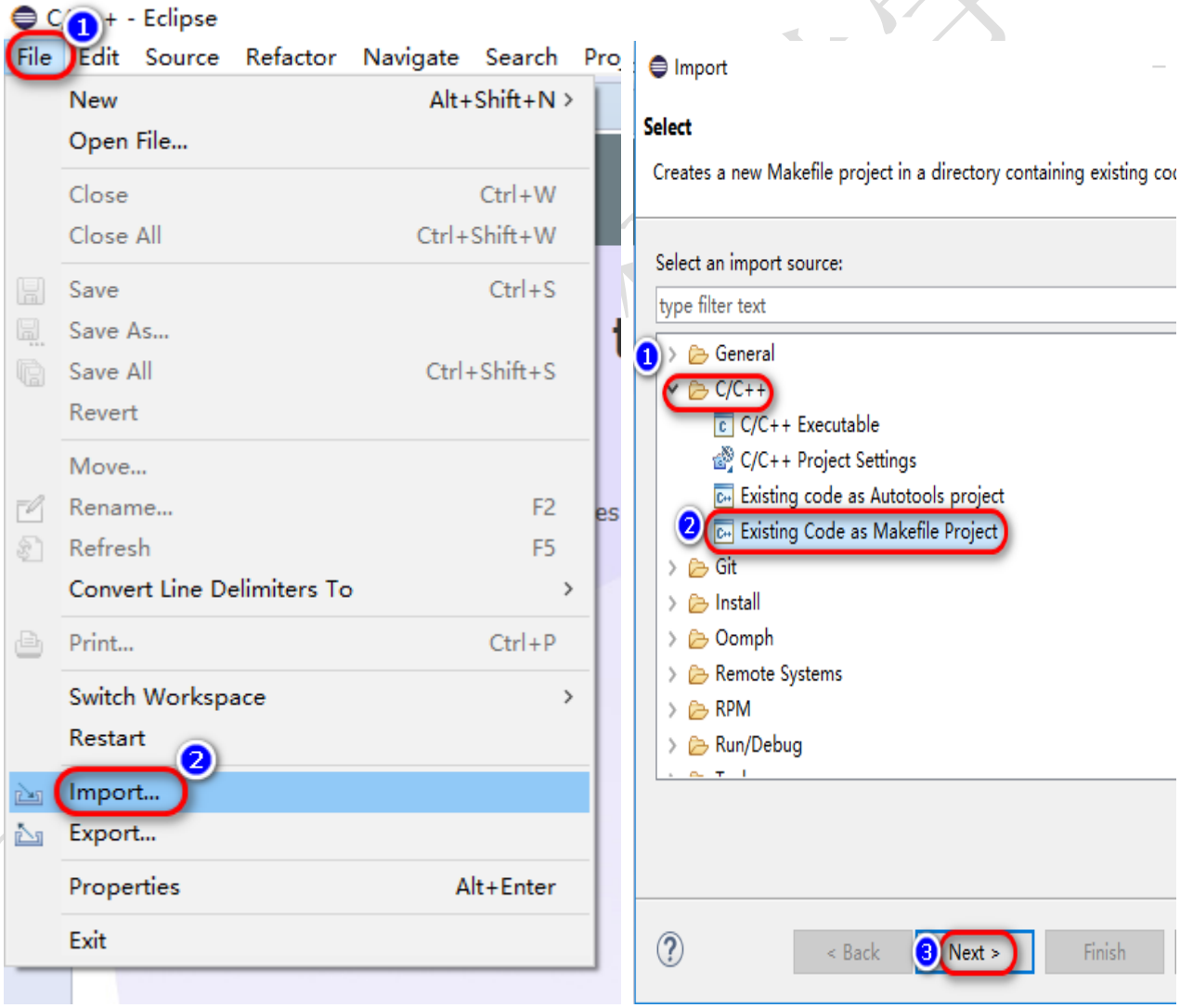
将 ESP8266_NONOS_SDK/third_party/makefile 重命名为 makefile.bak ，以防止编译时报错

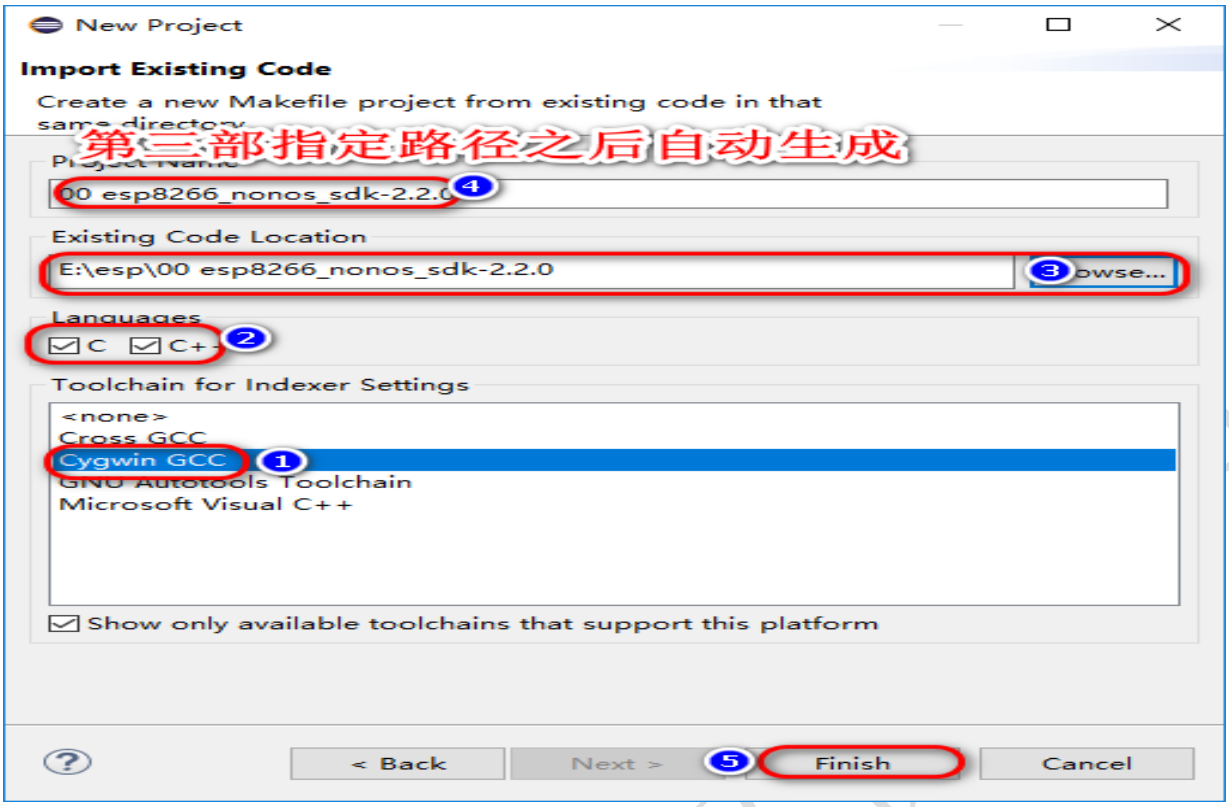


到此为止，工程修改完成。

==========

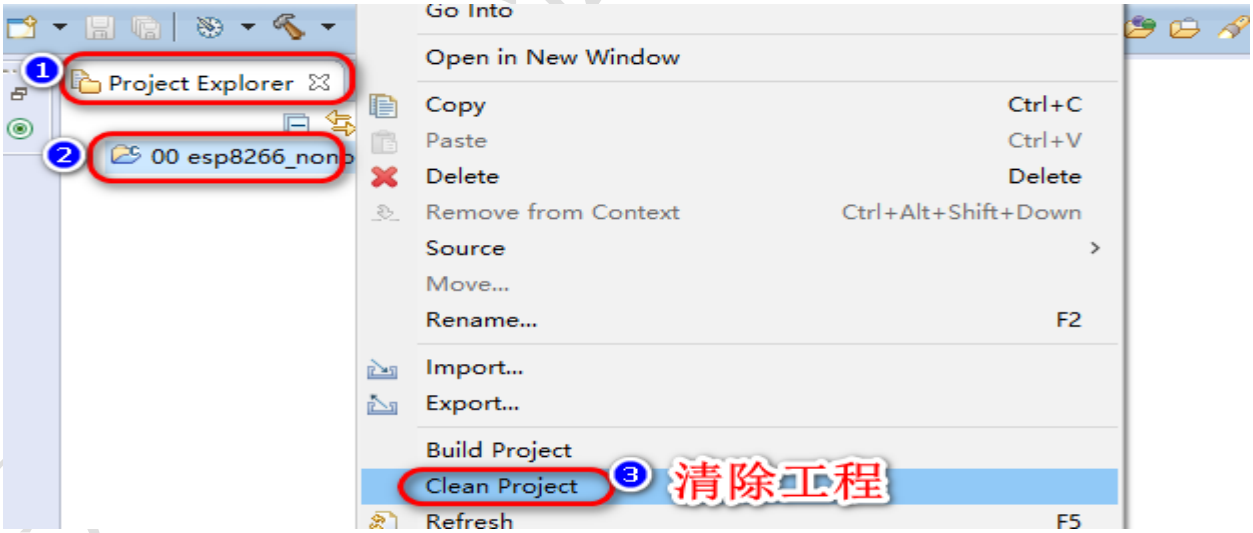
打开安信可 ESP 客户端，导入工程。

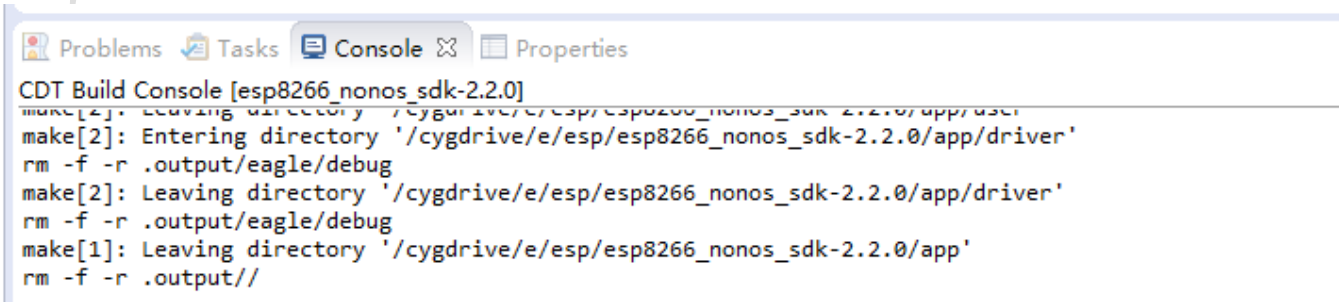




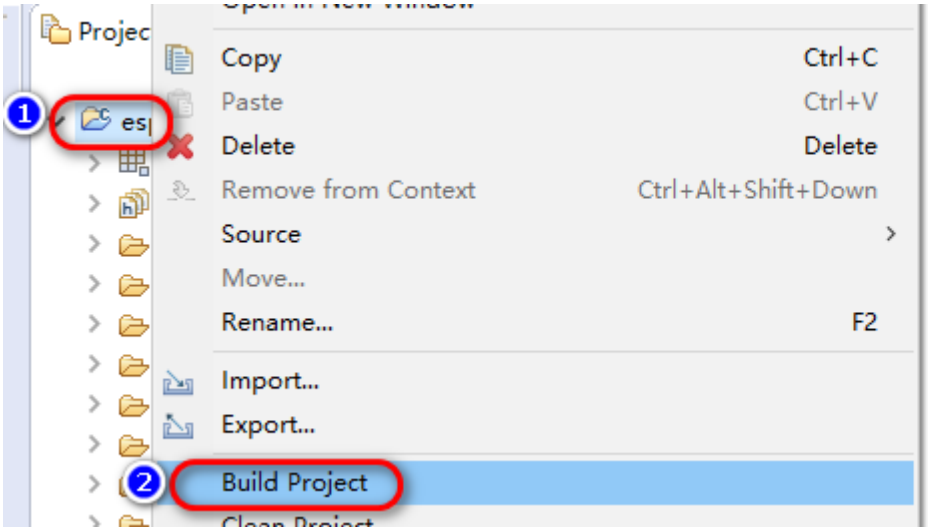
当工程导入成功之后，我们需要对其进行编译

注：代码编译之前一定要先对工程进行清除操作





接着编译过程



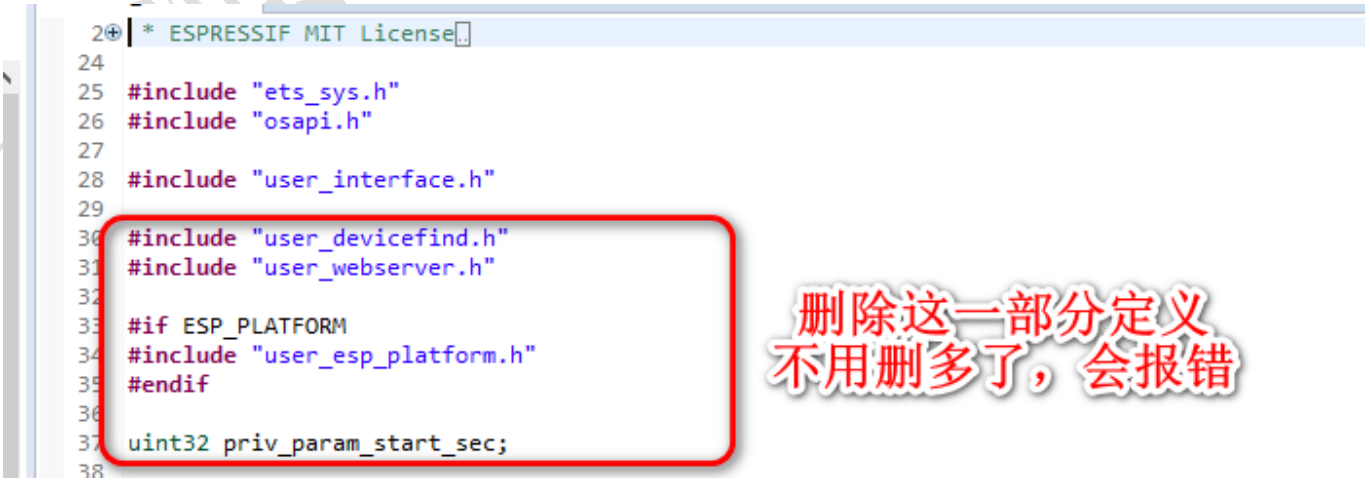
到此为止，环境搭建完成。

2.1.4 创建工程模板

创建新的工程模板是在安信可提供的 demo 模板上面进行调整性修改为我们自己的工程。

软件导入工程的教程不再进行说明，请自行参考 2.1.3 章节。

2.1.4.1 修改 user_main.c 文件



```
switch (size_map) {
    case FLASH_SIZE_4M_MAP_256_256:
        rf_cal_sec = 128 - 5;
        priv_param_start_sec = 0x3C;
        break;

    case FLASH_SIZE_8M_MAP_512_512:
        rf_cal_sec = 256 - 5;
        priv_param_start_sec = 0x7C;
        break;

    case FLASH_SIZE_16M_MAP_512_512:
        rf_cal_sec = 512 - 5;
        priv_param_start_sec = 0x7C;
        break;
    case FLASH_SIZE_16M_MAP_1024_1024:
        rf_cal_sec = 512 - 5;
        priv_param_start_sec = 0xFC;
        break;
```

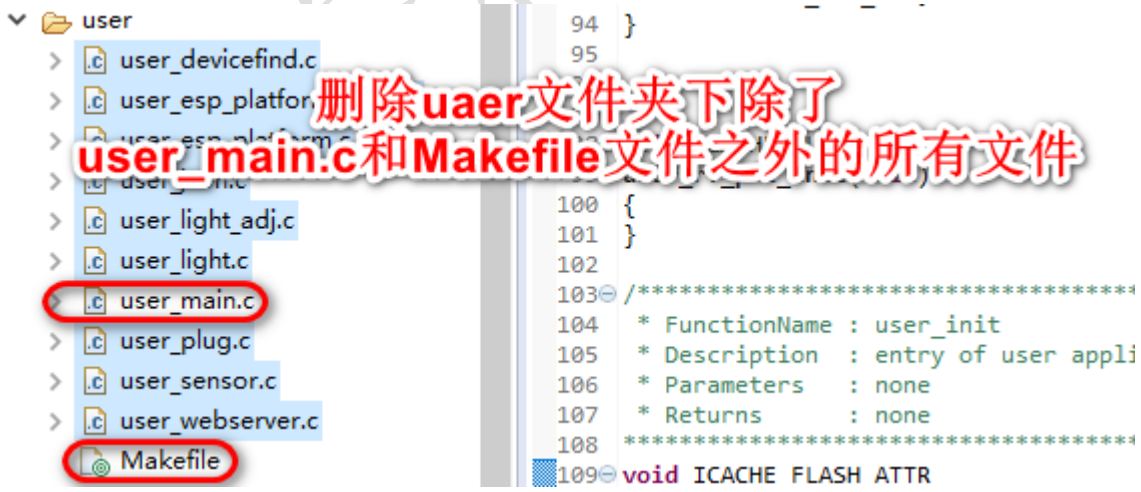
因为priv_param_start_sec变量在之前已经被删除了
使用把switch语句里面的对priv_param_start_sec
变量的赋值代码全部删除

截图太小，还有一部分没有接到图的对
priv_param_start_sec
变量赋值的代码也要删除

```
void ICACHE_FLASH_ATTR
user_init(void)
{
    os_printf("SDK version:%s\n", system_get_sdk_version());

#if ESP_PLATFORM
    /*Initialization of the peripheral drivers*/
    /*For light demo , it is user_light_init();*/
    /* Also check whether assigned ip addr by the router.If so, connect to ESP-server  */
    user_esp_platform_init();
#endif
    /*Establish a udp socket to receive local device detect info.*/
    /*Listen to the port 1025, as well as udp broadcast.
    /*If receive a string of device_find_request, it rely its IP address and MAC.*/
    user_devicefind_init();

    /*Establish a TCP server for http(with JSON) POST or GET command to communicate with the device.*/
    /*You can find the command in "2B-SDK-Espressif IoT Demo.pdf" to see the details.*/
    /*the JSON command for curl is like:*/
    /*3 Channel mode: curl -X POST -H "Content-Type:application/json" -d "{\"period\":1000,\"rgb\":{\"red\":16000,\"green\":16000,\"blue\":16000}}
    /*5 Channel mode: curl -X POST -H "Content-Type:application/json" -d "{\"period\":1000,\"rgb\":{\"red\":16000,\"green\":16000,\"blue\":16000,\
#ifdef SERVER_SSL_ENABLE
    user_webserver_init(SERVER_SSL_PORT);
#else
    user_webserver_init(SERVER_PORT);
#endif
}
```

将user_init函数里面的所有代码都删除

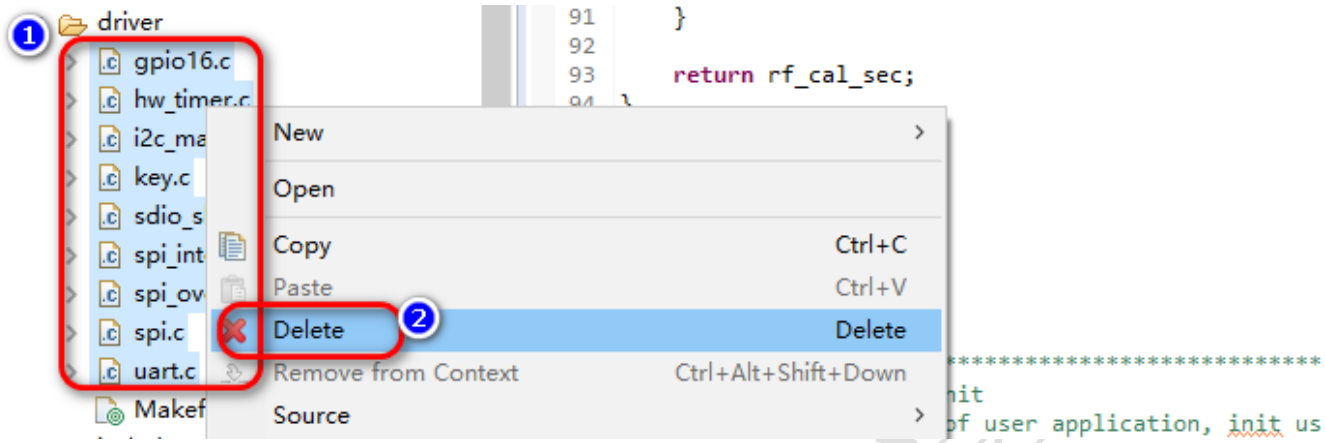
2.1.4.2 移除 user 文件夹下多余文件



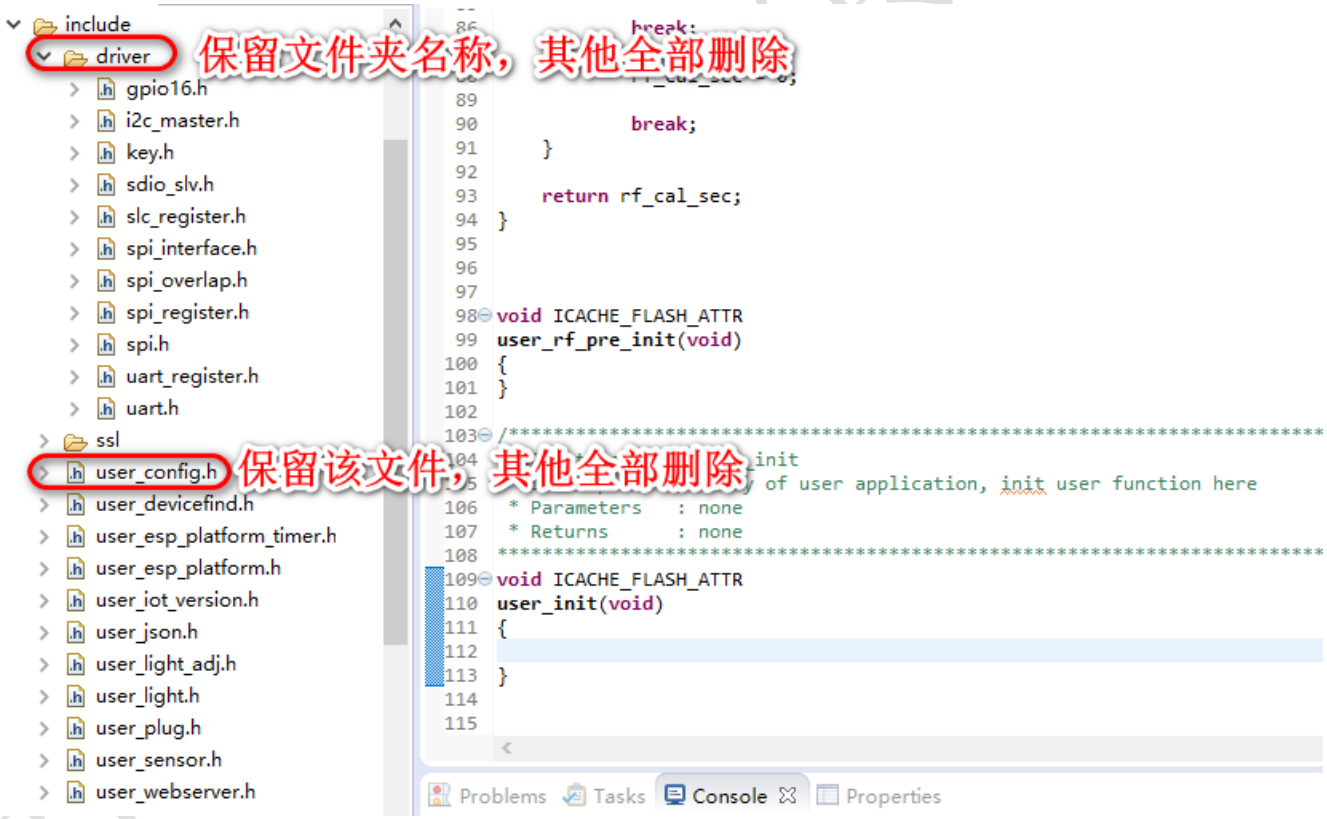
删除uaer文件夹下除了
user_main.c和Makefile文件之外的所有文件

2.1.4.3 移除 driver 文件夹下的多余文件

删除 driver 文件夹下的除了 makefile 文件之外的所有文件



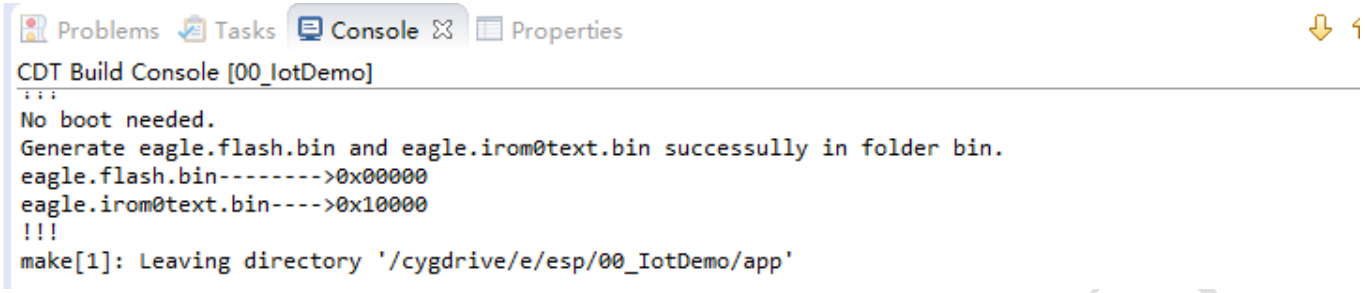
2.1.4.4 删除 driver 文件夹下的多余文件



2.1.4.5 修改 user_config.h 文件代码

2.1.4.6 编译检测

先清除工程，然后编译过程



无错误，工程模板创建完成。

2.2 二进制文件烧录教程

2.2.1 下载烧录

该教程文档参考安信可官方文档教程，网址：http://wiki.ai-thinker.com/esp_download

ESP FLASH TOOL 是 Espressif 官方开发的烧录工具，用户可根据实际的编译方式和 Flash 的容量，将 SDK 编译生成的多个 bin 文件一键烧录到 ESP8266/ESP32 的 SPI Flash 中。

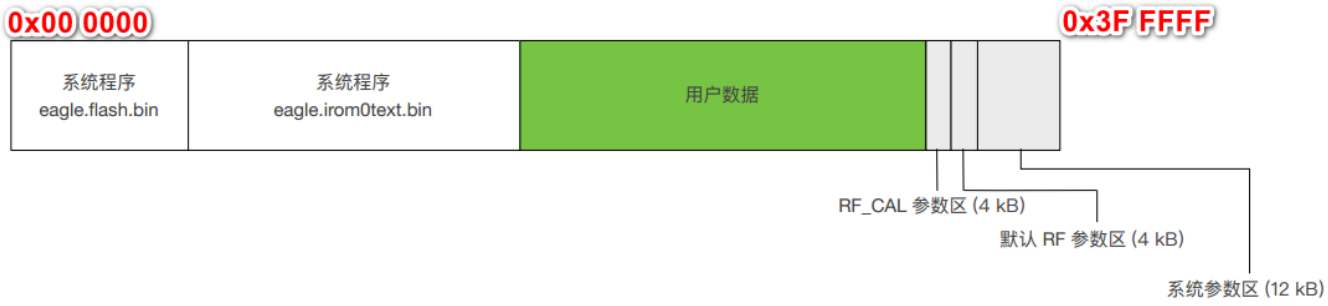
ESP FLASH TOOL 下载：http://wiki.ai-thinker.com/_media/tools/flash_download_tools_v2.4_150924.rar

必须声明：ESP12F 芯片 Flash 内存空间为 4MB 即 4096KB 即 32MBit.

2.2.1.1 Flash 空间布局

首先我们介绍一下 non-FOTA 固件在不同容量 Flash 中的布局

Flash 的起始地址空间为：0x00 0000—0x3F FFFF



分区作用：

- 系统程序 eagle.flash.bin：用于存放运行系统必要的固件。
- 系统程序 eagle.irom0text.bin：存放用户编写的工程代码。
- 用户数据：当有多余的 Flash 空间用于用户数据区时，空闲区域均可用于存放用户数据。用户可在其中意空闲位置设置用户参数区，建议至少为用户参数区预留 12 KB 空间。

- RF_CAL 参数：用于系统自动保存校准后的 RF 参数。
- 默认 RF 参数：将 esp_int_data_default.bin 下载至该区，用于保存默认的参数信息。
- 系统参数：用于保存系统参数信息。

2.2.1.2 下载地址

BIN	各个 Flash 容量对应的下载地址					
	512	1024	2048	4096	8192	16*1024
blank.bin	0x7B000	0xFB000	0x1FB000	0x3FB000	0x7FB000	0xFFB000
esp_init_data_default.bin	0x7C000	0xFC000	0x1FC000	0x3FC000	0x7FC000	0xFFC000
blank.bin	0x7E000	0xFE000	0x1FE000	0x3FE000	0x7FE000	0xFFE000
eagle.flash.bin 系统程序	0x00000					
eagle.irom0text.bin 用户程序	0x10000					

下载步骤：

1.硬件配置

GPIO15	GPIO0	GPIO2	
1	X	X	SDIO/SPI WIFI
0	0	1	USART DOWNLOAD
0	1	1	FLASH BOOT

2.软件配置

①.打开 ESP Flash Download Tool，选择 ESP8266 Download Tool;

②.选择 SPI Download

③.选择烧录文件及文件烧录的地址(地址按照从低到高排列)

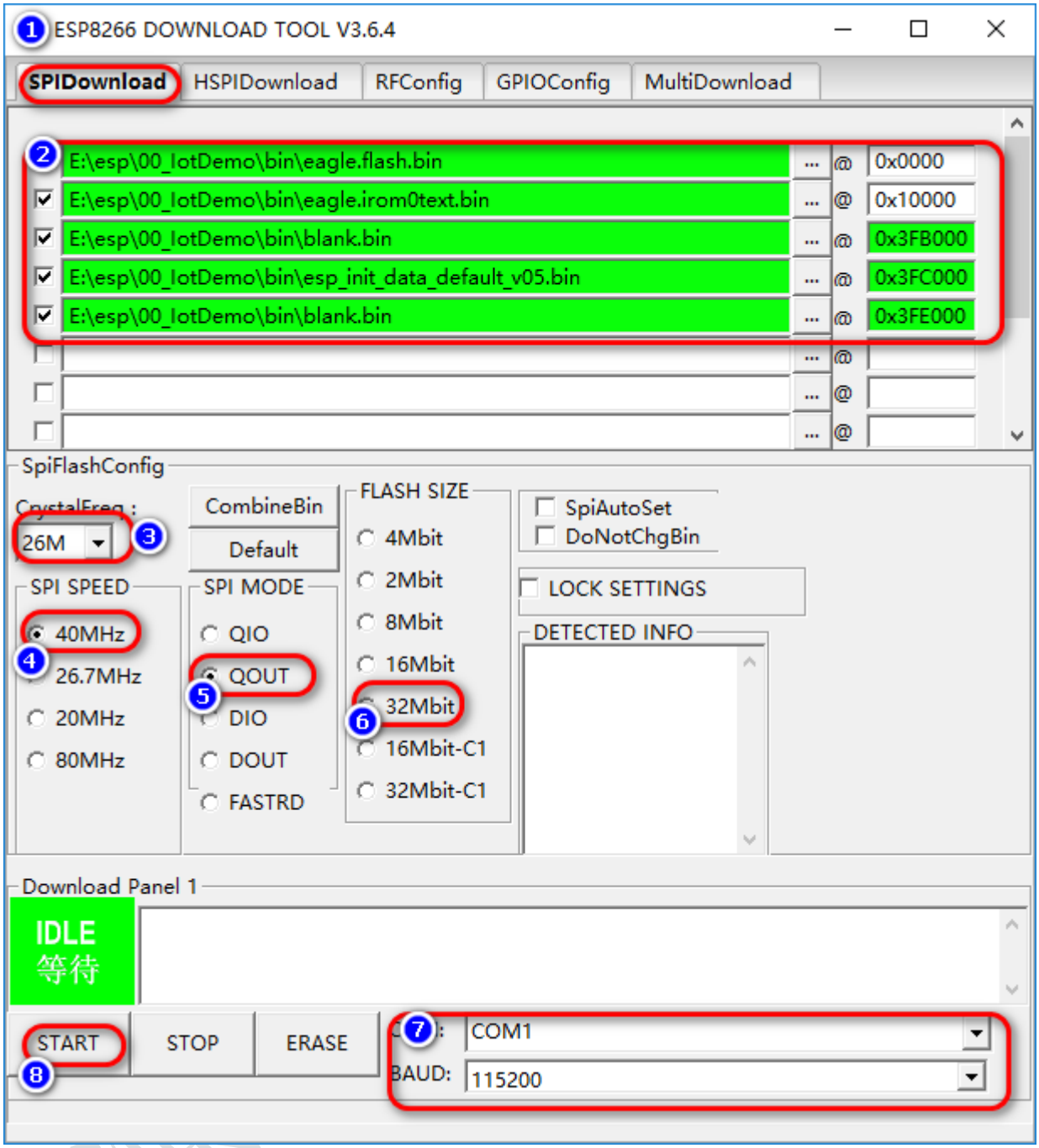
④.选择晶振(26MHz)

⑤.选择 SPI 下载速度(40MHz)

⑥.选择 SPI MODE(DOUT)

⑦.选择 FLASH SIZE(32MBit)

⑧.匹配串口和波特率，开始下载



下载工具配置说明：

配置项	配置说明
SPI FLASH CONFIG	
CrystalFreq	根据实际选用的晶振型号选择晶振频率。
CombineBin	将勾选的 BIN 文件合成一个 target.bin，下载地址为 0x0000。
Default	将 SPI Flash 的配置恢复到默认值。
SPI SPEED	选择 SPI Flash 的读写速度，最大值为 80 MHz。
SPI MODE	根据实际使用的 Flash 选择对应的模式。如果 Flash 采用 Dual SPI，选择

	DIO 或 DOUT；如果 Flash 采用 Quad SPI，选择 QIO 或 QOUT
FLASH SIZE	根据实际编译的配置对应选择的 Flash 大小。 16Mbit-C1 是 1024+1024 的情况； 32Mbit-C1 是 1024+1024 的情况。
SpiAutoSet	不建议勾选 SpiAutoSet，推荐用户根据实际情况对 Flash 进行手动配置。 用户如果了勾选 SpiAutoSet，下载工具将会按照默认的 Flash map 下载，16 Mbit 和 32 Mbit 的 Flash map 会被设置为 512 Kbyte + 512 Kbyte。
DoNotChgBin	• 用户可勾选 DoNotChgBin， Flash 的运行频率，方式和布局会以用户编译时的配置选项为准。 • 如果不勾选该选项， Flash 的运行频率，方式和布局会以下载工具最终的配置为准。

注：如果选择了 DoNotChgBin，下载工具会按照您在 Makefile 文件里面的设置进行烧录的

```
25 BOOT?=none
26 APP?=0
27 SPI_SPEED?=40
28 SPI_MODE?=QIO
29 SPI_SIZE_MAP?=0
```

在实际的烧录工程之中，我们只要不破坏 RF_CAL 参考区/默认 RF 参考区/系统参考区的空间，我们只需要烧录 eagle.flash.bin 和 eagle.irom0text.bin 文件即可。

我们的 eagle.irom0.text.bin 文件官方推荐的烧录地址是 0x10000，但是这个地址是可以修改的，如果想要修改这个系统文件的烧录地址，需要修改 ESP8266_NONOS_SDK/ld/eagle.app.v6.ld 文件里面的代码

```
3 MEMORY
4 {
5   dport0_0_seg :                  org = 0x3FF00000, len = 0x10
6   dram0_0_seg :                   org = 0x3FFE8000, len = 0x14000
7   iram1_0_seg :                   org = 0x40100000, len = 0x8000
8   irom0_0_seg :                   org = 0x40210000, len = 0x5C000
9 }
```

上面截图的代码来自于官方 dome 例程里面的文件，此处设置的 eagle.irom0.text.bin 文件烧录地址就是 0x10000，在 0x10000 之前的 0x00000 到 0x0FFFF 的空间里面可以存放 eagle.flash.bin，eagle.flash.bin 文件的 •最大内存就是 64KB=64 * 1024。虽然此地址可以修改，但是建议不要去修改这个路径。

第三章 ESP8266 的基本模块

3.1 ESP8266 工程框架与常用单片机框架的区别

3.1.1 常用单片机框架

此框架只是按照本人平时写代码的风格写的，仅供参考---勿喷

main.c 文件

```
#include <stdio.h>
#include "stm32f10x.h"
#include "led.h"
int main(void)
{
    //模块初始化代码
    while (1)
    {
        //代码主运行区域
    }
    return 0;
}
```

led.c

```
#include "stm32f10x.h"
void LedInitConfig(void)
{
    //初始化LED灯的IO口配置代码
}
```

led.h

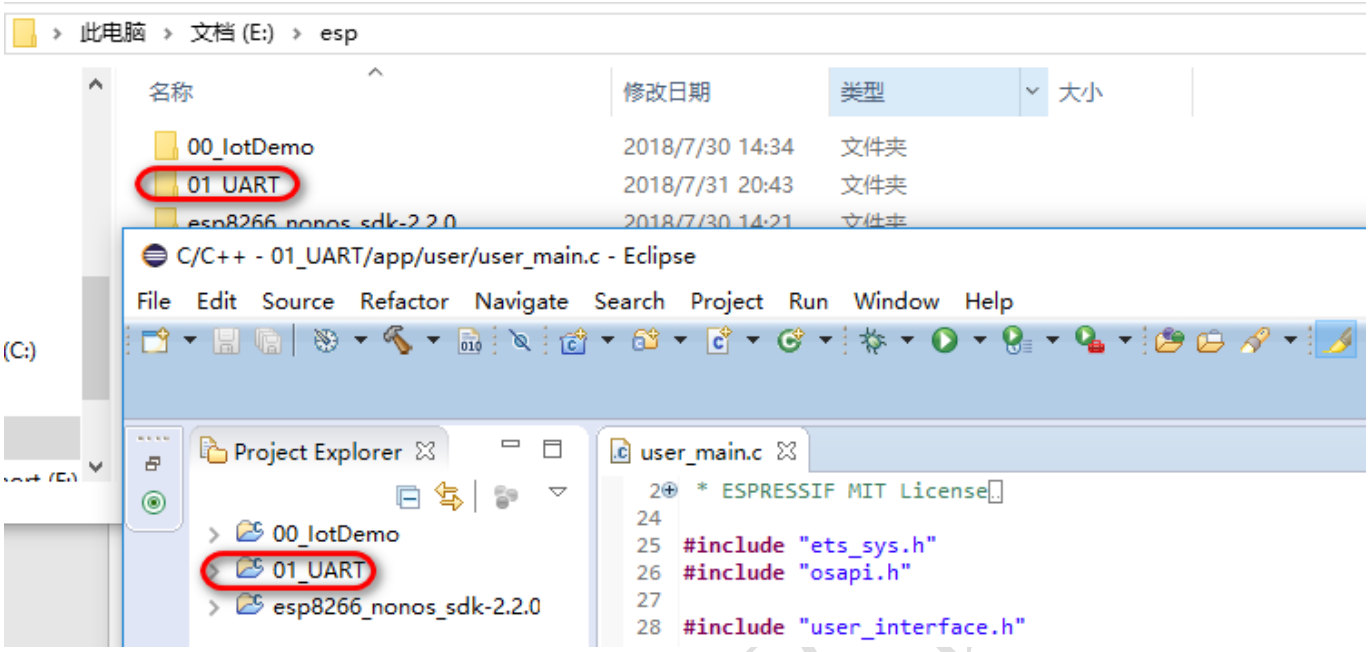
```
#ifndef _LED_H__
#define _LED_H__

void LedInitConfig(void);

#endif
```

3.1.2 ESP8266 代码框架

3.2 移植官方库文件实现串口打印

3.2.1 实现串口打印函数

1.在实现串口打印函数之前，我们应该把之前完成的工程模板复制一份修改名字，然后导入软件。



2.打开乐鑫官方的 ESP8266 API 文档(文档名称：2C-ESP8266__SDK__API Guide__CN_V1.5.4，文档下载路径：https://www.espressif.com/zh-hans/support/download/documents?keys=&field_type_tid%5B%5D=14)【吐槽一波：乐鑫的文档整理的真心不给力，PDF 文档都没有目录的】

如果想要 ESP8266 支持串口打印，我们需要调用系统的 API 函数。

① os_printf 函数

功能：

格式化输出，打印字符串。

注意：

本接口默认从 UART0 打印。IOT_Demo 中的 uart_init 可以设置波特率，将 os_printf 改为 UART1 打印：os_install_putc1((void *)uart1_write_char);

请勿调用本接口打印超过 125 字节的数据，或者频繁连续调用本接口打印，否则可能会对视部分待打印数据。

函数定义：

void os_printf(const char *s)

参数：

const char *s – 字符串

返回值：

无

示例：

os_printf("SDK version: %s \n", system_get_sdk_version());

② system_get_sdk_version 函数

功能：

查询 SDK 版本信息

函数定义：

const char* system_get_sdk_version(void)

参数：

无

返回：

SDK 版本信息

示例：

printf(“SDK version: %s \n”, system_get_sdk_version());

3.代码实现串口打印

1. 在 user_main.c 文件里添加支持 os_printf 函数和 system_get_sdk_version 函数的头文件


```
25 #include "ets_sys.h"
26 #include "osapi.h"
27
28 #include "user_interface.h"
29
```

2. 在 user_main.c 文件里 user_init(void) 添加打印语句代码【串口默认是使用 uart0 打印】

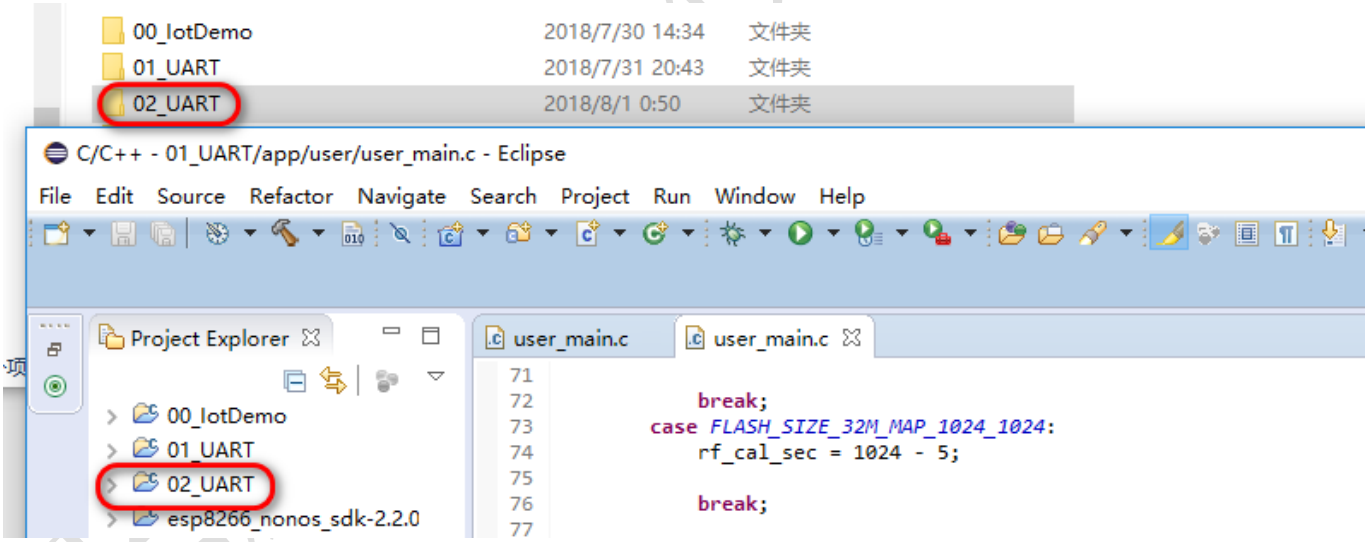
```
user_init(void)
{
    os_printf("\r\n=====================================\r\n");
    os_printf("SDK version: %s \n", system_get_sdk_version());
    os_printf("\r\n=====================================\r\n");
}
```

```
110 user_init(void)
111 {
112     os_printf("\r\n=====================================\r\n");
113     os_printf("SDK version: %s \n", system_get_sdk_version());
114     os_printf("\r\n=====================================\r\n");
115 }
116
```

3. 保存工程->清除工程->编译工程->烧录代码【这个操作请阅读上一章节，不加叙述】，代码烧录之后运行，打开串口软件，设置串口波特率【因为 ESP8266 外部晶振为 26MHz，所以选择 74880 的波特率，我在写本章文档时用的是其他的串口调试助手，没有 74880 的波特率选择，只能使用 76800，效果差不多】，完成之后，按下复位按键，就可以实现代码的功能了。

3.2.2 移植官方库文件实现打印函数

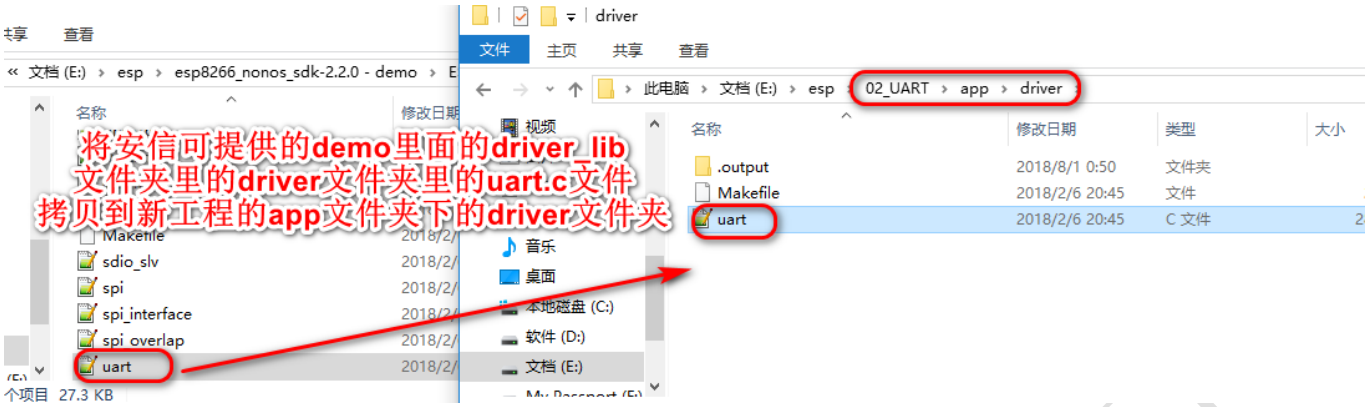
1. 在实现串口打印函数之前，我们应该把之前完成的工程模板复制一份修改名字，然后导入软件。



2. 移植串口所需的文件到指定文件

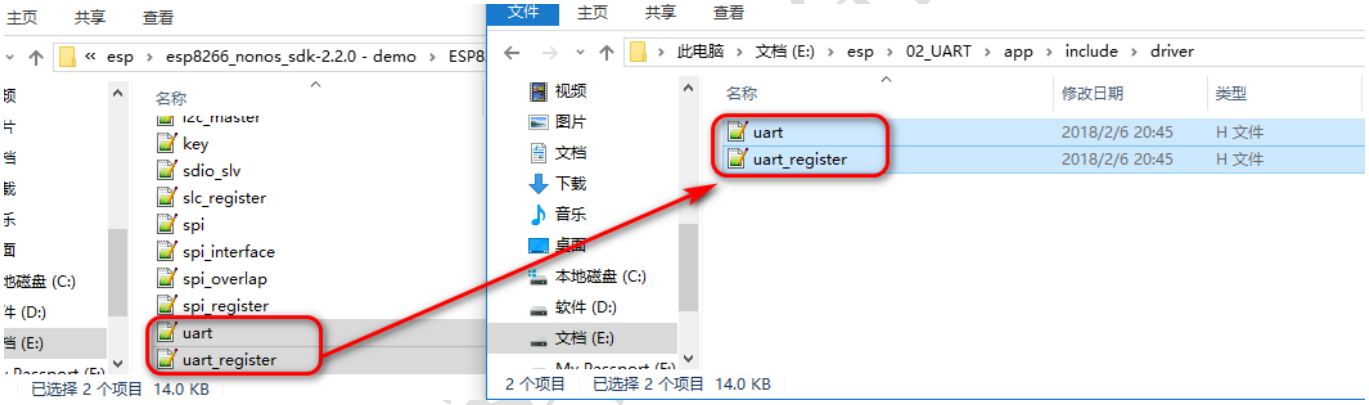
①移植 uart.c 文件

将安信可提供的 demo 里面的 driver_lib 文件夹里的 driver 文件夹里的 uart.c 文件拷贝到新工程的 app 文件夹下的 driver 文件夹【不要放错文件夹了，会出问题的】

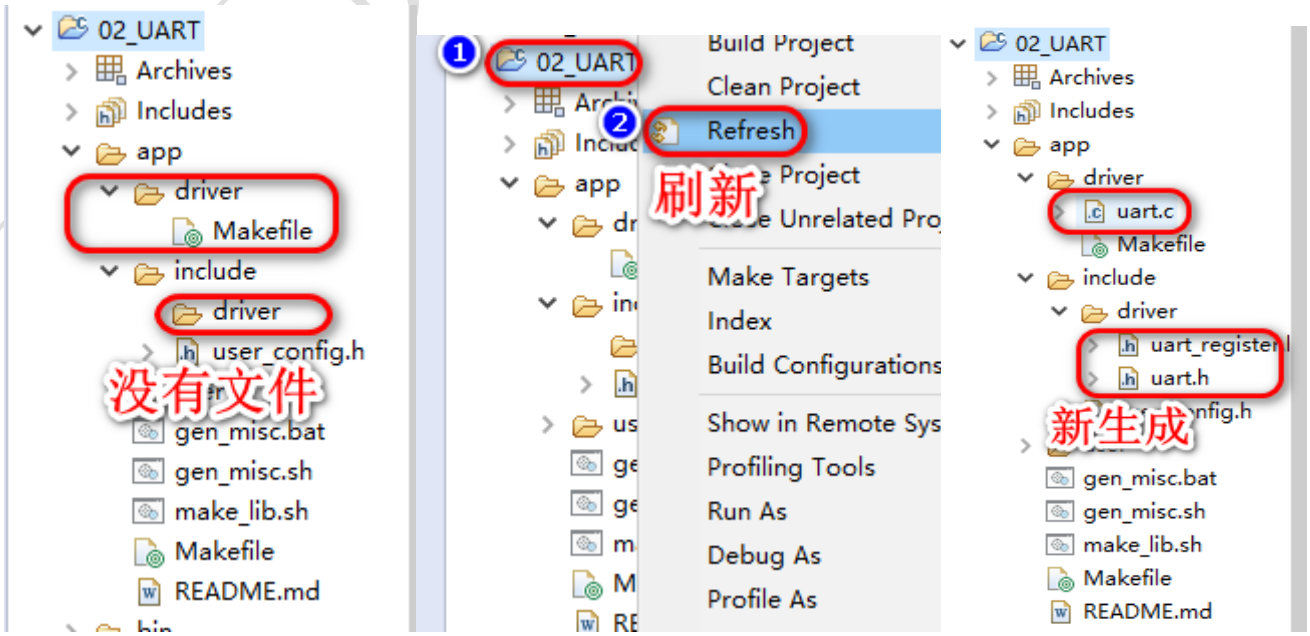


②移植 uart.h 和 uart_register.h 文件

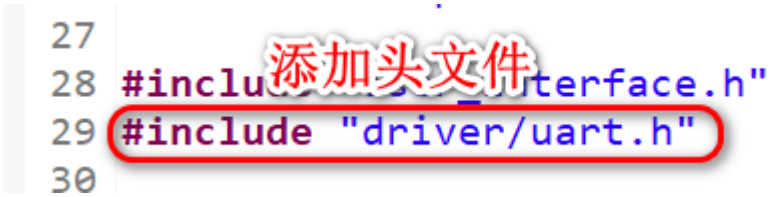
将安信可提供的 demo 里面的 driver_lib 文件夹里的 driver 文件夹下的 include 文件夹下的 driver 文件夹下的 uart.h 和 uart_register.h 文件拷贝到新工程的 app 文件夹下的 include 文件夹下的 driver 文件夹【不要放错文件夹了，会出问题的】



③刷新工程，编译文件



④添加头文件，添加 API 代码



常用 API 代码：

I void uart_init(UartBautRate uart0_br, UartBautRate uart1_br)函数

功能：

双 UART 模式，两个 UART 波特率初始化

函数定义：

void uart_init(UartBautRate uart0_br, UartBautRate uart1_br)

参数：

UartBautRate uart0_br ： usart 0 波特率

UartBautRate uart1_br ： usart 1 波特率

波特率：

```
typedef enum {
    BIT_RATE_9600 = 9600,
    BIT_RATE_19200 = 19200,
    BIT_RATE_38400 = 38400,
    BIT_RATE_57600 = 57600,
    BIT_RATE_74880 = 74880,//默认波特率
    BIT_RATE_115200 = 115200,
    BIT_RATE_230400 = 230400,
    BIT_RATE_460800 = 460800,
    BIT_RATE_921600 = 921600
} UartBautRate;
```

返回值：

无

II void uart0_sendStr(const char *str) 函数

功能:

UART 0 发送字符串

函数定义：

void uart0_sendStr(const char *str)

参数：

const char *str—>要发送的字符串

返回值：

无

⑤ 添加代码

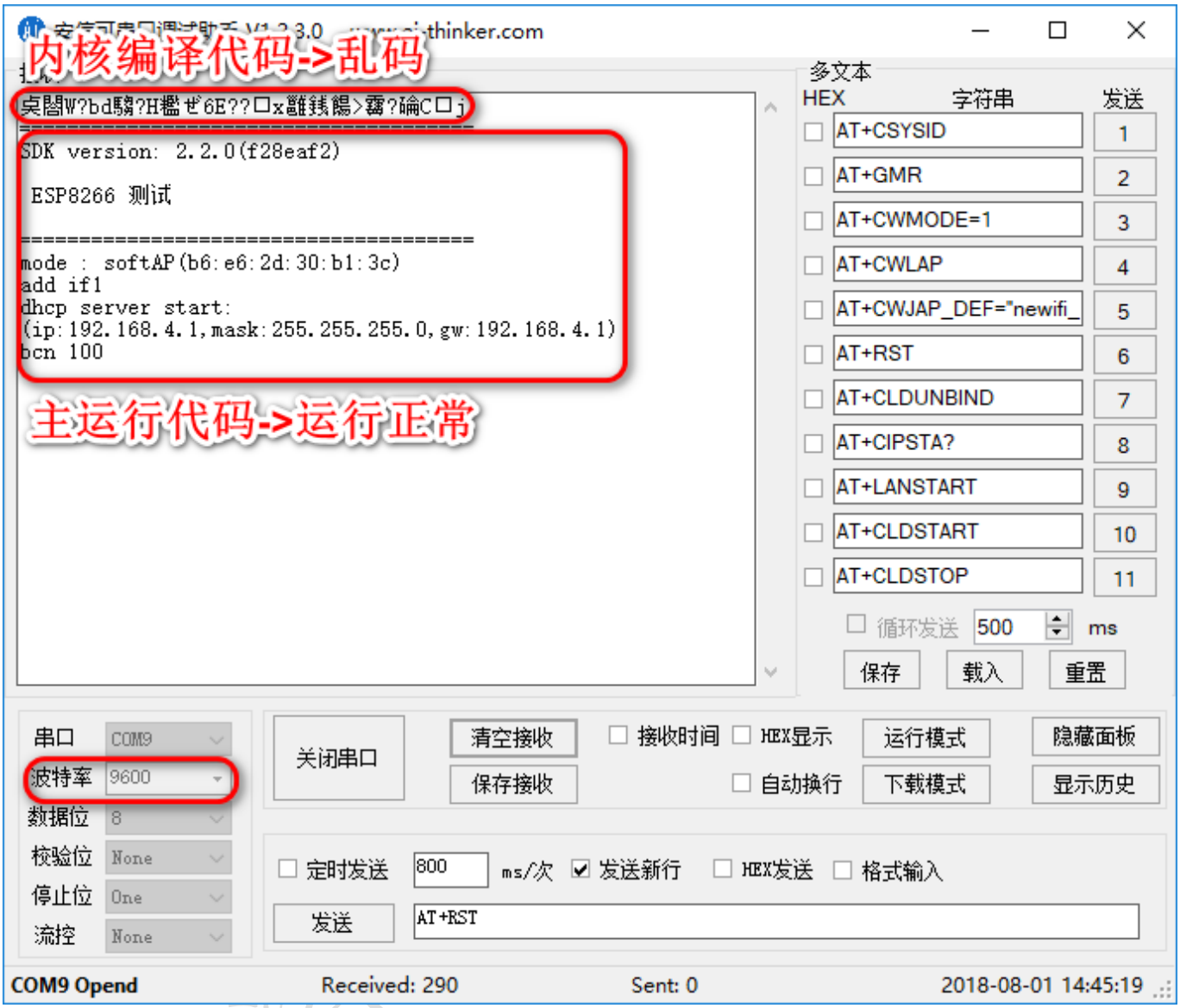
```
user_init(void)
```

```
{
    uart_init(9600, 9600);//设置串口波特率
    os_printf("\r\n=====================================\r\n");
    os_printf("SDK version: %s \n", system_get_sdk_version());
    uart0_sendStr("\r\n ESP8266 测试 \r\n");//串口0打印字符串
    os_printf("\r\n=====================================\r\n");
}
```

代码编译效果：

当波特率选择为 74880 时，

当波特率选择为 9600 时，



出现上述情况的原因为：内核代码在编译的过程中是以 26MHz【即 74880】的外部晶振当时钟源的，所以在串口打印输出时，波特率选择 74880 的情况下，内核代码打印是正常的；波特率选择 9600 的情况下，内核代码打印就是一堆乱码；当我们的代码运行到主运行代码时，这是由于我们在 user_init()函数里添加了修改波特率的代码，所以此时必须要按照我们设置的波特率才能正常打印。也就是说当波特率选择 9600 的情况下，user_init()里面的代码就可以支持打印了。

3.3 GPIO 输出控制

从本章节开始，我们就要开始学习 ESP8266 模块的 GPIO 操作了。对于大部分的 GPIO 口都有两种状态：输出和输入。

GPIO 的输出：ESP8266 的输出控制。在 ESP8266 引脚作为 GPIO 输出时，首先要配置 GPIO 为输出模式，这个和普通的 32 位单片机的 IO 输出是一样的，然后提供改变 IO 口的电平来控制输出模块的状态。在这里我用我自己手上的一款 ESP8266 的单片机为例(这个单片机采购于技新网 https://www.jixin.pro/product/3864.html)。

3.3.1 GPIO 的管脚定义

GPIO NO	PIN NO	Pin name
GPIO0	pin15	GPIO0_U
GPIO1	pin26	U0TXD_U
GPIO2	pin14	GPIO2_U
GPIO3	pin25	U0RXD_U
GPIO4	pin16	GPIO4_U
GPIO5	pin24	GPIO5_U
GPIO6	pin21	SD_CLK_U
GPIO7	pin22	SD_DATA0_U
GPIO8	pin23	SD_DATA1_U
GPIO9	pin18	SD_DATA2_U
GPIO10	pin19	SD_DATA3_U
GPIO11	pin20	SD_CMD_U
GPIO12	pin10	MTDI_U
GPIO13	pin12	MTCK_U
GPIO14	pin9	MTMS_U
GPIO15	pin13	MTDO_U

注：上表的解释说明(PGIO0 为例):原理图引脚和 PCB 的丝印命名为 GPIO0，它在 ESP8266 模块的第 15 个引脚，其功能引脚名称为 GPIO0_U。【功能引脚名称详见文档：0D-ESP8266_PIN_List_Release_15-11-2014.xlsx，文档下载地址：http://wiki.ai-thinker.com/_media/esp8266/docs/0d-esp8266_pin_list_release_15-11-2014.xlsx】

Inst Name	Side	Coordinate	X	Function1	Type	Function2	Type	Function3	Type	Function4	Type
MTDI_U				MTDI	I	I2SI_DATA	I/O/T	HSPIQ MISO	I/O/T	GPIO12	I/O/T
MTCK_U				MTCK	I	I2SI_BCK	I/O/T	HSPID MOSI	I/O/T	GPIO13	I/O/T
MTMS_U				MTMS	I	I2SI_WS	I/O/T	HSPICLK	I/O/T	GPIO14	I/O/T
MTDO_U				MTDO	O/T	I2SO_BCK	I/O/T	HSPICS	I/O/T	GPIO15	I/O/T
U0RXD_U				U0RXD	I	I2SO_DATA	I/O/T		O	GPIO3	I/O/T
U0TXD_U				U0TXD	O	SPICS1	I/O/T		O	GPIO1	I/O/T
SD_CLK_U				SD_CLK	I	SPICLK	I/O/T		O	GPIO6	I/O/T
SD_DATA0_U				SD_DATA0	I/O/T	SPIQ	I/O/T		O	GPIO7	I/O/T
SD_DATA1_U				SD_DATA1	I/O/T	SPID	I/O/T		O	GPIO8	I/O/T
SD_DATA2_U				SD_DATA2	I/O/T	SPIHD	I/O/T		O	GPIO9	I/O/T
SD_DATA3_U				SD_DATA3	I/O/T	SPIWP	I/O/T		O	GPIO10	I/O/T

3.3.2 GPIO 相关 API 函数

本教程参考《ESP8266 技术参考》和《ESP8266 SDK 编程手册》

GPIO 相关接口位于：ESP8266_NONOS_SDK/include/eagle_soc.h & gpio.h
使用示例可参考：ESP8266_NONOS_SDK/examples/IoT_Demo/user/user_plug.c

3.3.1.1 PIN_FUNC_SELECT()函数

函数原型：PIN_FUNC_SELECT(PIN_NAME, FUNC)
函数功能：将指定引脚设定为指定功能
形参：

PIN_NAME：指定引脚的引脚名称

形参可填参数为：3.3.1 GPIO 引脚定义章节的” PERIPHS_IO_MUX_”+pin name
例如：GPIO0_U 在这里就应该写为 PERIPHS_IO_MUX_GPIO0_U

FUNC：指定的功能

形参可填参数为：参考下表里面指定功能里面的参数(这些变量已经在 eagle_soc.h 文件里面宏定义)

指定引脚	指定功能				
	Function1(0)	**Function2(1)**	**Function3(2)**	**Function4(3)**	**Function5(4)**
MTDI_U	MTDI	I2SI_DATA	HSPIQ MISO	GPIO12	U0DTR
MTCK_U	MTCK	I2SI_BCK	HSPID MOSI	GPIO13	U0CTS
MTMS_U	MTMS	I2SI_WS	HSPICLK	GPIO14	U0DSR
MTDO_U	MTDO	I2SO_BCK	HSPICS	GPIO15	U0RTS
U0RXD_U	U0RXD	I2SO_DATA		GPIO3	CLK_XTAL
U0TXD_U	U0TXD	SPICS1		GPIO1	CLK_RTC
SD_CLK_U	SD_CLK	SPICLK		GPIO6	U1CTS
SD_DATA0_U	SD_DATA0	SPIQ		GPIO7	U1TXD
SD_DATA1_U	SD_DATA1	SPID		GPIO8	U1RXD
SD_DATA2_U	SD_DATA2	SPIHD		GPIO9	HSPIHD
SD_DATA3_U	SD_DATA3	SPIWP		GPIO10	HSPIWP
SD_CMD_U	SD_CMD	SPICS0		GPIO11	U1RTS
GPIO0_U	GPIO0	SPICS2			CLK_OUT
GPIO2_U	GPIO2	I2SO_WS	U1TXD		U0TXD
GPIO4_U	GPIO4	CLK_XTAL			
GPIO5_U	GPIO5	CLK_RTC			

返回值：无
使用示例：PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO0_U, FUNC_GPIO0);

3.3.1.2 PIN_PULLUP_EN()函数

函数名称：PIN_PULLUP_EN(PIN_NAME)
函数功能：引脚上拉使能
形参：PIN_NAME：” PERIPHS_IO_MUX_”+pin name
示例：PIN_PULLUP_EN(PERIPHS_IO_MUX_GPIO4_U);

3.3.1.3 PIN_PULLUP_DIS()函数

函数名称：PIN_PULLUP_DIS (PIN_NAME)

函数功能：引脚上拉失能

形参：PIN_NAME：” PERIPHS_IO_MUX_”+pin name

示例：PIN_PULLUP_ DIS (PERIPHS_IO_MUX_GPIO4_U);

3.3.1.4 gpio_output_set()函数

函数名称：void gpio_output_set(uint32 set_mask,uint32 clear_mask,uint32 enable_mask,uint32 disable_mask)

函数功能：设置 GPIO 属性

形参：

uint32 set_mask ：设置输出为高的位，对应位为 1，输出高；对应位为 0，不改变状态

uint32 clear_mask ：设置输出为低的位，对应位为 1，输出低；对应位为 0，不改变状态

uint32 enable_mask ：设置使能输出的位

uint32 disable_mask ：设置使能输入的位

返回值：无

示例：

```
gpio_output_set(BIT12, 0, BIT12, 0); //#define BIT12  0x00001000 设置 GPIO12 输出高电平
gpio_output_set(0, BIT12, BIT12, 0): //设置 GPIO12 输出低电平
gpio_output_set(BIT12, BIT13, BIT12|BIT13, 0)://设置 GPIO12 输出高电平, GPIO13 输出低电平
gpio_output_set(0, 0, 0, BIT12); //设置 GPIO12 为输入
```

3.3.3 GPIO 输入输出相关宏

3.3.2.1 GPIO_OUTPUT_SET(gpio_no, bit_value)

宏名称：GPIO_OUTPUT_SET(gpio_no, bit_value)

宏功能：设置指定引脚(gpio_no)为指定电平(bit_value)

形参：

gpio_no：指定的 GPIO 口

可写参数：GPIO_ID_PIN(n)//n 的取值范围为[15:0]，对应的是 GPIO[15:0]

bit_value：指定电平

可填参数：1-->高电平 0-->低电平

示例：GPIO_OUTPUT_SET(GPIO_ID_PIN(0),1);//设置 GPIO0 引脚为高电平

3.3.2.2 GPIO_DIS_OUTPUT(gpio_no)

宏名称：GPIO_DIS_OUTPUT(gpio_no)

宏功能：设置指定引脚(gpio_no)为输入模式

形参：

gpio_no：指定的 GPIO 口

可写参数：GPIO_ID_PIN(n)//n 的取值范围为[15:0]，对应的是 GPIO[15:0]

示例：GPIO_DIS_OUTPUT(GPIO_ID_PIN(0));//设置 GPIO0 为输入模式

3.3.2.3 GPIO_INPUT_GET(gpio_no)

宏名称：GPIO_INPUT_GET(gpio_no)
宏功能：获取指定引脚(gpio_no)的电平状态
形参：
gpio_no：指定的 GPIO 口
可写参数：GPIO_ID_PIN(n)//n 的取值范围为[15:0]，对应的是 GPIO[15:0]
示例：GPIO_INPUT_GET(GPIO_ID_PIN(0));//获取 GPIO0 的当前状态

3.3.4 GPIO 输出寄存器

3.3.3.1 输出使能寄存器 GPIO_ENABLE_W1TS

BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

注：R->只读 W->只写 RW->可读可写 (该协议以下文档内容全部通用)

BIT[15:0] 输出使能位(可读可写)：若对应的位被置 1，表示该 IO 的输出被使能。BIT[15:0]对应 16 个 GPIO 的输出使能位。

3.3.3.2 输出禁用寄存器 GPIO_ENABLE_W1TC

BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

BIT[15:0] 输出禁用位(可读可写)：若对应的位被置 1，表示该 IO 的输出被禁用。BIT[15:0]对应 16 个 GPIO 的输出禁用位。

3.3.3.3 输出使能状态寄存器 GPIO_ENABLE

BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

BIT[15：0] 输出使能状态位(可读可写)：该寄存器的 BIT[15:0]的值，反映的是对应的 PIN 脚的输出使能状态。

GPIO_ENABLE 的 BIT[15:0]通过给 GPIO_ENABLE_WITS 的 BIT[15:0]和 GPIO_ENABLE_W1TC 的 BIT[15:0]写值来控制。例如 GPIO_ENABLE_W1TS 的 BIT[0]置 1，则 GPIO_ENABLE 的 BIE[0] = 1；

3.3.3.4 输出低电平寄存器 GPIO_OUT_W1TC

BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

BIT[15:0]输出低电平位(只写寄存器)：若对应的位被置 1，表示该 IO 的输出为低电平(同时使能输出)。BIT[15:0]对应 16 个 GPIO 的输出状态。

注：如果需要将该 PIN 配置位高电平,需要配置 GPIO_OUT_W1TS 寄存器。

3.3.3.5 输出高电平寄存器

BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W

BIT[15:0]输出高电平位(只写寄存器)：若对应的位被置 1，表示该 IO 的输出为高电平(同时需要使能输出)。BIT[15:0]对应 16 个 GPIO 的输出状态。

注：如果需要将该 PIN 配置为低电平，需要配置 GPIO_OUR_W1TC 寄存器。

3.3.3.6 输出状态寄存器 GPIO_OUT

BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

BIT[15:0]输出状态位(读写寄存器)：该寄存器的[15:0]的值，反映的是对应的 PIN 脚出书的状态。

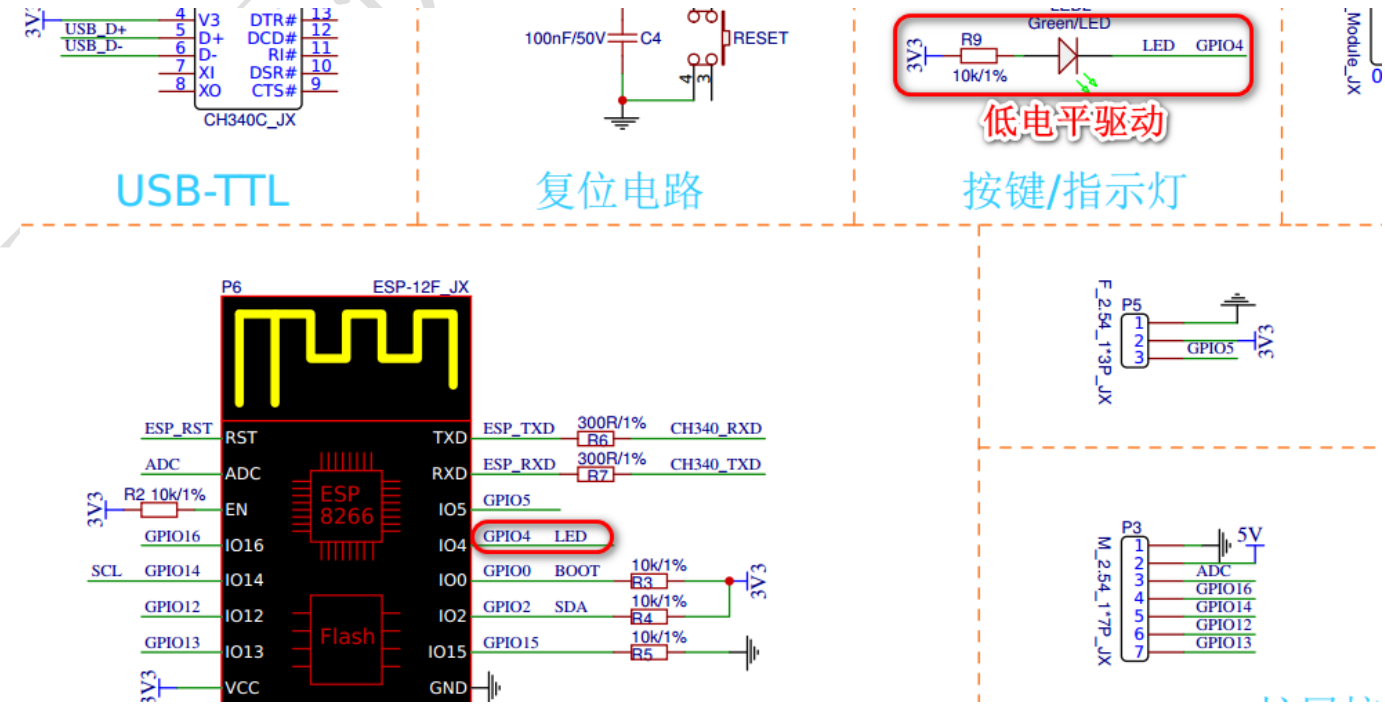
GPIO_OUT 的 BIT[15:0]是由 GPIO_OUT_W1TS 的 bit[15:0] 和 GPIO_OUT_W1TC 的 bit[15:0]共同决定的。例如：GPIO_OUT_W1TS 的 Bit[1]=1，那么 GPIO_OUT[1]=1。GPIO_OUT_W1TC 的 Bit[2]=1，那么 GPIO_OUT[2]=0。

3.3.5 ESP8266 输出控制举例

在这里我用我自己手上的一款 ESP8266 的单片机为例(这个单片机采购于技新网 https://www.jixin.pro/product/3864.html)。

物联网开发板-ESP8266,此模块以 ESP8266 模组为核心，配合稳压电路、USB-TTL 电路、串口下载电路，实现 ESP8266 的 SDK 开发，实现物联网功能。配合以 OLED、DHT11 模块，实现温度上报到云端、显示云端下发的消息等。【技小新官方点评】

3.3.4.1 硬件连接图



由原理图可知：LED 灯对连接的是 ESP8266 模块的 GPIO4 引脚，并且使用低电平驱动。

注：在学习所有单片机的时候都是从 LED 灯和按键控制开始的，原因是所有的单片机引脚只有输入和输出两种状态。LED 灯属于输出设备，是最简单的输出控制设备之一；按键属于输入设备，是最简单的输入控制设备之一；这两个模块是带大家入门单片机的敲门砖。所有高大上的时序都是 GPIO 的输入输出得到的。

3.3.4.2 初始化 LED 灯

这里的初始化代码我先把它放在 user_int()函数里面【我目前对 eclipse 这个软件操作不熟悉，后面再进行优化，敬请见谅】

① 添加头文件

GPIO 相关接口位于 /ESP8266_NONOS_SDK/include/eagle_soc.h & gpio.h。

使用示例可参考 /ESP8266_NONOS_SDK/examples/IoT_Demo/user/user_plug.c.

```
#include "user_interface.h"
#include "driver/uart.h"
#include "gpio.h"
#include "eagle_soc.h"
```

此处不需要指定include文件夹

② 配置 GPIO4 为 IO 模式

```
//1.设置LED灯所对应的GPIO4口为IO口模式 -- 详见文档3.3.1.1章节
PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO4_U, FUNC_GPIO4);     // GPIO4设为IO口
```

③ 配置 GPIO4 为低电平，点亮 LED 灯

```
//2.设置GPIO4初始化输出电平 -- 详见文档3.3.1.4和3.3.2.1章节
GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 0);//将GPIO4初始化为低点平状态-->灯亮
```

④ user_init() 源代码

```
void ICACHE_FLASH_ATTR user_init(void)
{
    uart_init(9600, 9600);//设置串口波特率
    os_printf("=============================================\r\n");
    os_printf("\t SDK version:\t%s", system_get_sdk_version());
    os_printf("\r\n嵌入式陈工个人编辑资料  未经本人同意请勿私自传播\r\n");
    os_printf("\r\nGPIO输出测试代码\r\n");
    os_printf("=============================================\r\n");
    //1.设置LED灯所对应的GPIO4口为IO口模式 -- 详见文档3.3.1.1章节
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO4_U, FUNC_GPIO4); // GPIO4设为IO口
    //2.设置GPIO4初始化输出电平 -- 详见文档3.3.1.4和3.3.2.1章节
    GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 0);//将GPIO4初始化为低点平状态→灯亮
}
```

3.4 有 bug 的延时控制

延时控制是使用延时函数让代码控制单片机或者模块保存当前状态。

之所以会出现 bug，是因为系统会对 ESP8266 模块自动复位。然后导致代码的延时函数的作用没有体现出来，这个问题可以暂时不去理会它，我们将在下一章节里面解决这个问题的。

3.4.1 通用延时函数

通用延时函数通常是使用循环语句来达到延时效果。因为我们 ESP8266 模块使用的是 26M 的外部晶振，所以模块的系统运行时间是 1/26M 秒每一行代码。(1/26M 秒=1/26K 毫秒=1/26 微秒)

```
/*
 * 函数名称：void DealyUs(unsigned int data)
 * 函数功能：微秒延时函数
 * 函数形参：unsigned int data：延时时间
 * */
void DelayUs(unsigned int data)
{
    int i,j;
    for(i = 0; i < data; i ++)
    {
        for(j = 0; j < 26; j ++)
        {
            //ESP8266模块外部系统晶振26MHz
            //所以当在外部系统晶振频率下，每次运行时间为1/26000000S = 1/26000MS = 1/26US
            ;
        }
    }
}

/*
 * 函数名称：void DelayMs(unsigned int data)
 * 函数功能：毫秒延时函数
 * 函数形参：unsigned int data：延时时间
 * */
void DelayMs(unsigned int data)
{
    int i,j;
    for(i = 0; i < data; i ++)
    {
        for(j = 0; j < 1000; j ++)
        {
            DelayUs(1);
        }
    }
}
```

3.4.2 系统延时函数

乐鑫官方为我们提供了一个 8266 系统微秒级的延时函数，这个延时函数可以为我们提供比较精准的延时操作。

3.4.2.1 系统延时函数 API

函数原型：void os_delay_us(uint16 us)
函数功能：延时函数。
函数形参：
uint16 us 延时时间
可填参数：0 -- 65535
返回值： 无

推荐使用由系统 API 函数生成的延时函数，经测试我自己写的有 for 语句组合的延时函数并不能很好的达到延时效果。

3.4.2.1 根据系统延时函数编写的毫秒延时函数

```
/*
 * 函数名称：void os_DelayMs(unsigned int data)
 * 函数功能：毫秒延时函数
 * 函数形参：unsigned int data：延时时间
 * */
void os_DelayMs(uint32_t data)
{
    //在系统里面已经对uint32进行了重命名了
    //typedef unsigned int        uint32_t;//c_types.h
    uint32_t i,j;
    for(i = 0; i < data; i ++)
    {
        for(j = 0; j < 1000; j ++)
        {
            os_delay_us(1);
        }
    }
}
```

3.4.3 测试代码

我将以上代码在主函数进行调用，然后烧录到 ESP8266 里面之后发现延时函数是可以起作用的，但是由于模块本身没有进行处理[下一节会讲如何处理]，模块一直在复位，效果一直不良好。

```
void ICACHE_FLASH_ATTR user_init(void)
{
    uart_init(9600,9600);//设置串口波特率
    os_printf("=============================================\r\n");
    os_printf("\t SDK version:\t%s", system_get_sdk_version());
    os_printf("\r\n嵌入式陈工个人编辑资料\r\n未经本人同意请勿私自传播\r\n");
```

```
    os_printf("\r\nGPIO输出测试代码\r\n");
    os_printf("\r\nLED闪烁\r\n");
    os_printf("==========================================\r\n");
    //1.设置LED灯所对应的GPIO4口为IO口模式 -- 详见文档3.3.1.1章节
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO4_U, FUNC_GPIO4); // GPIO4设为IO口
    //2.设置GPIO4初始化输出电平 -- 详见文档3.3.1.4和3.3.2.1章节
    GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 1);//将GPIO4初始化为高点平状态--》灯灭

    while(1)
    {
        GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 0);//点亮LED
//      DelayMs(500);
        os_DelayMs(500);
        GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 1);//熄灭LED
//      DelayMs(500);
        os_DelayMs(500);
    }
}
```

3.5 系统看门狗

在这里首先要理解看门狗的的作用，看门狗是单片机的一个内部模块，该模块的主要作用是防止系统跑偏，也就是说防止系统自动复位。在上一节的延时函数里面，将代码烧录到单片机里面之后之所以会出现延时效果不良好，然后模块一直出现复位情况的原因就是因为系统一直在进行复位。

在乐鑫提供的官方文档 2c-esp8266_non_os_sdk_api_reference_cn(下载链接：https://www.espressif.com/sites/default/files/documentation/2c-esp8266_non_os_sdk_api_reference_cn.pdf) 第二章里面已经说到

Non-OS SDK 适合于用户需要完全控制代码执行顺序的应用程序。由于没有操作系统，non-OS SDK 不支持任务调度，也不支持基于优先级的抢占。

Non-OS SDK 最适合于事件驱动的应用程序，由于没有操作系统，non-OS SDK 没有单个任务堆栈大小的限制或者执行时隙要求。

Non-OS SDK 不支持抢占任务或进程切换。因此开发者需要自行保证程序的正确执行，用户代码不能长期占有 CPU。否则会导致看门狗复位，ESP8266 重启。

如果某些特殊情况下，用户线程必须执行较长时间(比如大于 500ms)，建议经常调用 system_soft_wdt_feed() API 来喂软件看门狗，而不建议禁用软件看门狗。

注：这里需要说明的是我们模块的软件看门狗是你在开启单片机模块之后自动开启的。

3.5.1 系统看门狗相关 API 函数

3.5.1.1 system_soft_wdt_feed()

函数原型：void system_soft_wdt_feed(void)
函数功能：喂软件看门狗
函数形参：无
返回值： 无

注意：	仅支持在软件看门狗开启情况下，调用本接口。

3.5.1.2 system_soft_wdt_restart()

函数原型：	void system_soft_wdt_restart(void)
函数功能：	重启软件看门狗
函数形参：	无
返回值：	无
注意：	仅支持在软件看门狗关闭(system_soft_wdt_stop())情况下，调用本接口。

3.5.1.3 system_soft_wdt_stop()

函数原型：	void system_soft_wdt_stop(void)
函数功能：	关闭软件看门狗
函数形参：	无
返回值：	无
注意：	请勿将软件看门狗关闭太长时间(小于 5 秒)，否则将触发硬件看门狗复位

3.5.2 参考测试代码

```
void ICACHE_FLASH_ATTR user_init(void)
{
    system_soft_wdt_feed();//喂软件看门狗，防止程序跑偏
    uart_init(9600, 9600);//设置串口波特率
    os_printf("=========================================\r\n");
    os_printf("\t SDK version:\t%s", system_get_sdk_version());
    os_printf("\r\n嵌入式陈工个人编辑资料\r\n未经本人同意请勿私自传播\r\n");
    os_printf("\r\nGPIO输出测试代码\r\n");
    os_printf("\r\n带看门狗\r\n");
    os_printf("\r\nLED闪烁\r\n");
    os_printf("=========================================\r\n");
    //1.设置LED灯所对应的GPIO4口为IO口模式 -- 详见文档3.3.1.1章节
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO4_U, FUNC_GPIO4); // GPIO4设为IO口
                                                         //2.设置GPIO4初始化输出电平 -- 详见文档
3.3.1.4和3.3.2.1章节
    GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 1);//将GPIO4初始化为高点平状态--》灯灭

    while (1)
    {
        GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 0);//点亮LED
        os_DelayMs(500);
        GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 1);//熄灭LED
        os_DelayMs(500);
    }
}
```

3.6 GPIO 输入控制

3.6.1 GPIO 输入寄存器

GPIO 的输入相关的 API 函数请参考第 3.3.2 章节，GPIO 的输入相关宏定义请参考 3.3.3 章节。因为之前文档已经详细说明了，所以在这里就不再加以叙述。

3.6.1.1 GPIO 输入寄存器 GPIO_IN

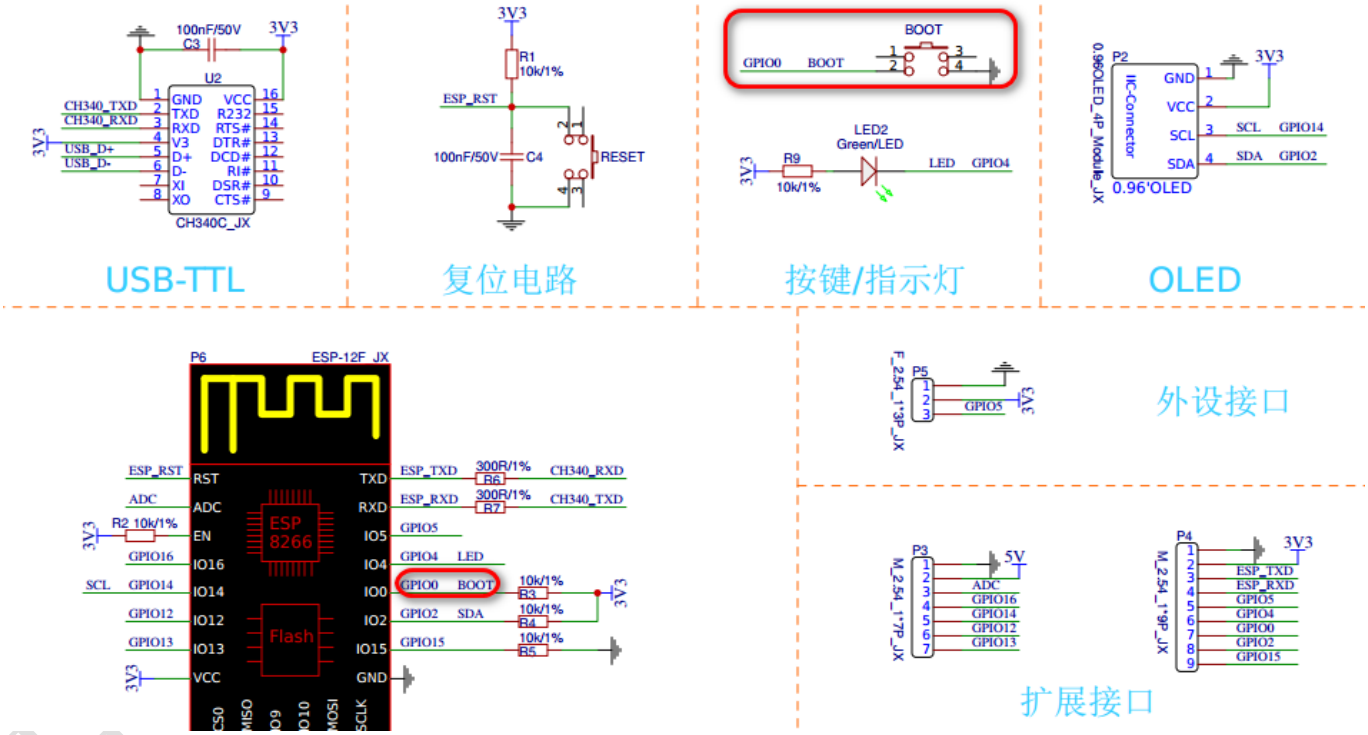
BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

BIT[15:0]输入状态位(可读可写)：若对应的位为 1，表示该 IO 的引脚状态为高电平；若对应的位为低电平，表示该 IO 的引脚状态为低电平。BIT[15:0]对应 16 个 GPIO 的输入状态位。

注：GPIO 的输入检测功能，是缺省设置，无需使能。

3.6.2 ESP8266 输入控制举例

3.6.2.1 硬件连接图



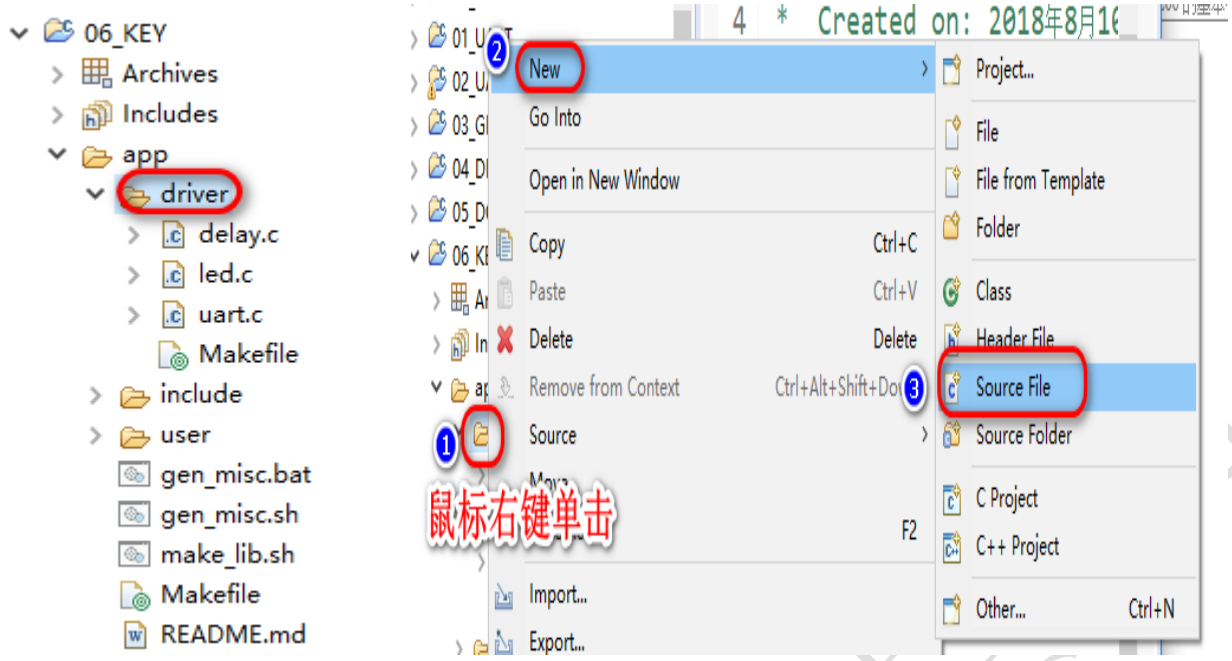
由原理图可知：按键 BOOT 接到 ESP8266 模块的 GPIO0 引脚。由于 GPIO0 通过 R3 进行电源上拉，所以 GPIO0 默认为高电平状态。当使能按键引脚之后，按下按键为低电平。

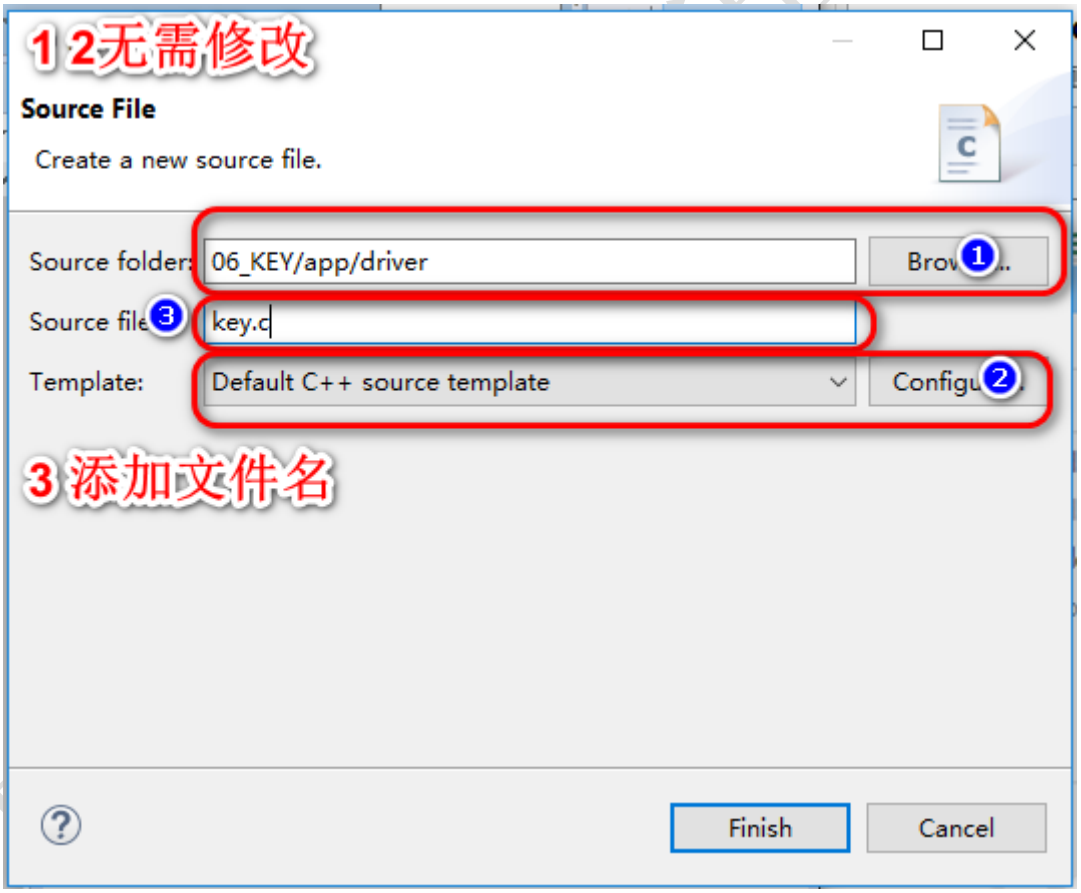
3.6.2.2 初始化按键

这里的按键初始化代码使用多文件编程。

1 新建驱动文件 key.c 和 key.h

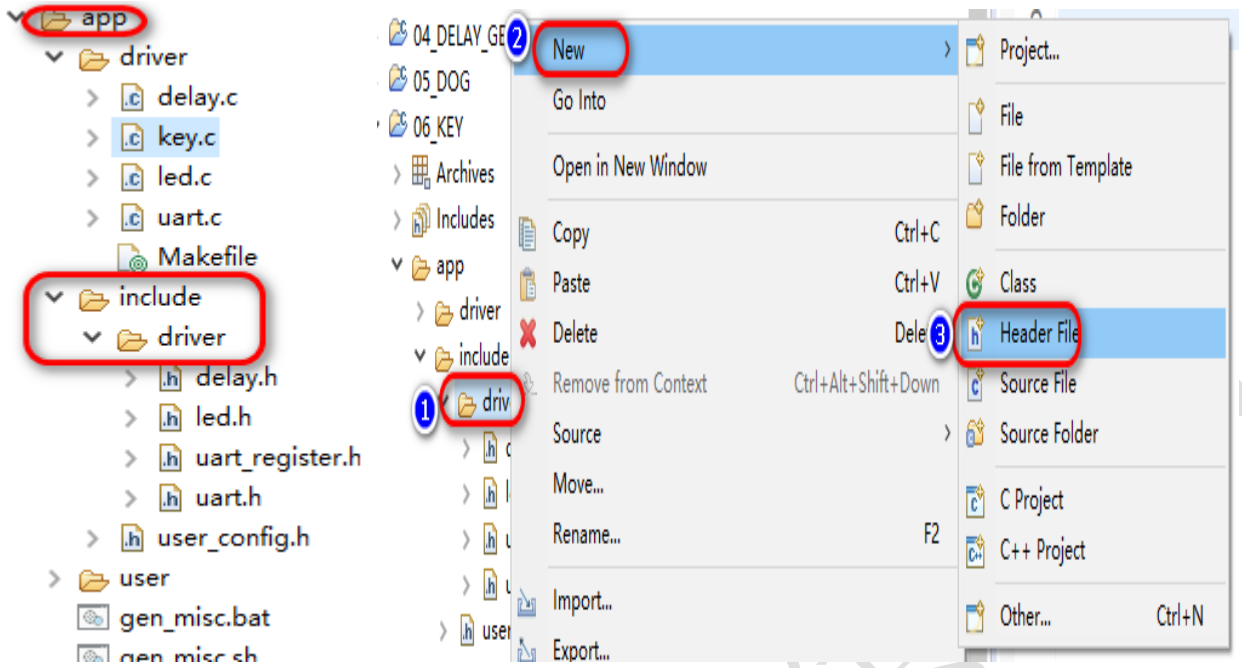
我们为了不去修改工程里面的 makefile 文件，所以我们直接将 key.c 放置在 app 文件夹下的 driver 文件里面

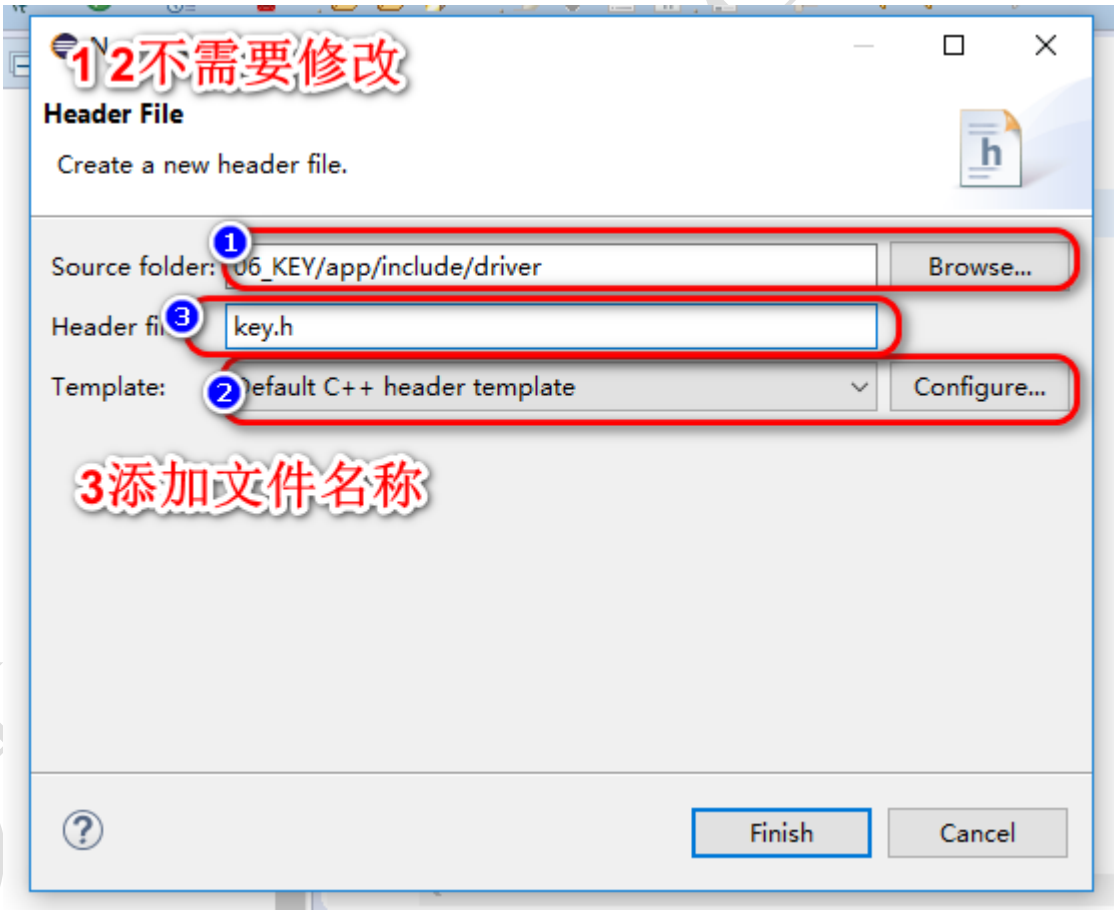

06_KEY
Archives
Includes
app
driver
delay.c
led.c
uart.c
Makefile
include
user
gen_misc.bat
gen_misc.sh
make_lib.sh
Makefile
README.md
New
Go Into
Open in New Window
Copy Ctrl+C
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Do
Source
F2
Import...
Export...
Project...
File
File from Template
Folder
Class
Header File
Source File
Source Folder
C Project
C++ Project
Other... Ctrl+N
鼠标右键单击


1 2无需修改
Source File
Create a new source file.
Source folder: 06_KEY/app/driver
Source file: key.c
Template: Default C++ source template
3 添加文件名
Finish
Cancel


app
driver
delay.c
key.c
led.c
uart.c
Makefile
include
user

将 key.h 文件放置在 app 文件夹下的 include 文件夹下的 driver 文件里面





2 添加头文件

GPIO 相关接口位于 /ESP8266_NONOS_SDK/include/eagle_soc.h & gpio.h。

使用示例可参考 /ESP8266_NONOS_SDK/examples/IoT_Demo/user/user_plug.c.

```
14 #include "driver/uart.h"
15 #include "gpio.h"
16 #include "eagle_soc.h"
17 #include "driver/delay.h"
18
19
```

3 配置 IO 口模式

```
10 void KeyInitConfig(void)
11 {
12     //1.设置按键所对应的GPIO0口为IO口模式 -- 详见文档3.3.1.1章节
13     PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO0_U, FUNC_GPIO0);     // GPIO0设为IO口
14     //2.由于GPIO0通过R3进行电源上拉，所以GPIO0默认为高电平状态，在这里要将内部拉高取消
15     PIN_PULLUP_DIS(PERIPHS_IO_MUX_GPIO0_U);
16     //3.设定GPIO0引脚为输入模式
17     GPIO_DIS_OUTPUT(GPIO_ID_PIN(0));
18 }
```

4 源代码

```
//key.c源代码
#include "driver/key.h"

void KeyInitConfig(void)
{
    //1.设置按键所对应的GPIO0口为IO口模式 -- 详见文档3.3.1.1章节
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO0_U, FUNC_GPIO0); // GPIO0设为IO口
    //2.由于GPIO0通过R3进行电源上拉，所以GPIO0默认为高电平状态，在这里要将内部拉高取消
    PIN_PULLUP_DIS(PERIPHS_IO_MUX_GPIO0_U);
    //3.设定GPIO0引脚为输入模式
    GPIO_DIS_OUTPUT(GPIO_ID_PIN(0));
}
```

```
//usrt_mian.c源代码
void ICACHE_FLASH_ATTR user_init(void)
{
    system_soft_wdt_feed();//喂软件看门狗，防止程序跑偏
    uart_init(9600, 9600);//设置串口波特率
    LedInitConfig();
    KeyInitConfig();
    os_printf("============================================\r\n");
    os_printf("\t SDK version:\t%s", system_get_sdk_version());
    os_printf("\r\n嵌入式陈工个人编辑资料\r\n未经本人同意请勿私自传播\r\n");
    os_printf("\r\n按键控制测试代码\r\n");
```

```
    os_printf("\r\n带看门狗\r\n");
    os_printf("==========================================\r\n");



    while (1)
    {
        if ((GPIO_INPUT_GET(GPIO_ID_PIN(0))) == 0)//检测按键是否按下
            GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 0);//点亮LED
        else
            GPIO_OUTPUT_SET(GPIO_ID_PIN(4), 1);//点亮LED
    }
}
```

3.7 GPIO 外部中断

3.7.1 GPIO 外部中断相关 API 函数

3.7.1.1 ETS_GPIO_INTR_ATTACH()函数

函数原型：ETS_GPIO_INTR_ATTACH(func, arg)

```
#define ETS_GPIO_INTR_ATTACH(func, arg)  ets_isr_attach(ETS_GPIO_INUM, (func), (void *)(arg))
void ets_isr_attach(int i, ets_isr_t func, void *arg);
```

函数功能：注册 GPIO 中断处理函数

函数形参：

func：中断服务函数名称

arg：

返回值：无

例子：ETS_GPIO_INTR_ATTACH(KeyExtiInterruptFunction, NULL);

3.7.1.2 ETS_GPIO_INTR_DISABLE()函数

函数原型：ETS_GPIO_INTR_DISABLE()

函数功能：关闭 GPIO 中断功能

函数形参：无

返回值： 无

3.7.1.3 ETS_GPIO_INTR_ENABLE()函数

函数原型：ETS_GPIO_INTR_ENABLE()

函数功能：打开 GPIO 中断功能

函数形参：无

返回值： 无

3.7.1.4 gpio_pin_intr_state_set()函数

函数原型：void gpio_pin_intr_state_set(uint32 i, GPIO_INT_TYPE intr_state)
函数功能：设置 GPIO 中断出发状态
函数形参：
uint32 i：GPIO 引脚的 ID

```
可填参数：GPIO_ID_PIN(x)   //x 表示要设置的引脚号
```

GPIO_INT_TYPE intr_state：中断触发方式；

```
GPIO_PIN_INTR_DISABLE = 0     //不触发中断
GPIO_PIN_INTR_POSEDGE = 1    //上升沿触发中断
GPIO_PIN_INTR_NEGEDGE = 2    //下降沿触发中断
GPIO_PIN_INTR_ANYEDGE = 3    //双边沿触发中断
GPIO_PIN_INTR_LOLEVEL = 4     //低电平触发中断
GPIO_PIN_INTR_HILEVEL = 5,     //高电平触发中断
以上形参定义在一枚举类型里面
```

返回值：无
例子：gpio_pin_intr_state_set(GPIO_ID_PIN(0), GPIO_PIN_INTR_NEGEDGE);//设置 GPIO0 为下降沿触发进入中断

3.7.1.5 GPIO_REG_READ()函数

函数原型：GPIO_REG_READ(reg)

```
以上函数为宏定义：
#define GPIO_REG_READ(reg)    READ_PERI_REG(PERIPHS_GPIO_BASEADDR + reg)
#define PERIPHS_GPIO_BASEADDR                    0x60000300
#define READ_PERI_REG(addr) (*((volatile uint32_t *)ETS_UNCACHED_ADDR(addr)))
#define ETS_UNCACHED_ADDR(addr) (addr)
```

函数功能：获取 GPIO 中断状态
函数形参：
reg：乐鑫官方文档提供的地址形参为 GPIO_STATUS_ADDRESS

```
#define GPIO_STATUS_ADDRESS              0x1c
```

返回值：所有 GPIO 口的中断状态
官方举例：

```
uint32 gpio_status;
gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);
```

注：官方标明该文件必须添加在 GPIO 中断处理函数内

3.7.1.6 GPIO_REG_WRITE()函数

函数原型：GPIO_REG_WRITE(reg, val)

```
以上函数为宏定义：
#define GPIO_REG_WRITE(reg, val) WRITE_PERI_REG(PERIPHS_GPIO_BASEADDR + reg, val)
#define PERIPHS_GPIO_BASEADDR                    0x60000300
#define WRITE_PERI_REG(addr, val) (*((volatile uint32_t *)ETS_UNCACHED_ADDR(addr))) = (uint32_t)(val)
```

```
#define ETS_UNCACHED_ADDR(addr) (addr)
```

函数功能：清除 GPIO 中断状态->退出中断

函数形参：

reg：乐鑫官方文档提供的地址形参为 GPIO_STATUS_ADDRESS

val：GPIO_REG_READ()函数的返回值

返回值：无

官方举例：

```
GPIO_REG_WRITE(GPIO_STATUS_W1TC_ADDRESS, gpio_status);
```

3.7.2 GPIO 中断寄存器

3.7.2.1 中断状态寄存器 GPIO_STATUS

BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

BIT[15:0](可读可写)：若对应的位被置 1，表示该 IO 中断发生。BIT[15:0]对应的 16 个 GPIO。

3.7.2.2 清中断寄存器 GPIO_STATUS_W1TC

BIT15	BIT14	BIT13	BIT12	BIT11	BIT10	BIT9	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW

BIT[15:0](可读可写)：向对应的位写 1，对应 GPIO 的中断状态就会被清除。

3.6.3 参考测试代码

//user_mian.c 源代码

```
void ICACHE_FLASH_ATTR user_init(void)
{
    system_soft_wdt_feed();//喂软件看门狗，防止程序跑偏
    uart_init(9600, 9600);//设置串口波特率
    LedInitConfig();//LED灯初始化函数
    KeyInitConfig();//按键初始化函数
    KeyExtiInitConfig();//配置按键外部中断
    os_printf("============================================\r\n");
    os_printf("\t SDK version:\t%s", system_get_sdk_version());
    os_printf("\r\n嵌入式陈工个人编辑资料\r\n未经本人同意请勿私自传播\r\n");
    os_printf("\r\n按键外部中断测试代码\r\n");
    os_printf("\r\n带看门狗\r\n");
    os_printf("============================================\r\n");
}
```

//key.c 源代码

```
/*
* key.c
*
*  Created on: 2018年8月10日
*      Author: 小良哥
*/

#include "driver/key.h"
#include "driver/led.h"
#include "ets_sys.h"            // 回调函数

void KeyInitConfig(void)
{
    //1.设置按键所对应的GPIO0口为IO口模式 -- 详见文档3.3.1.1章节
    PIN_FUNC_SELECT(PERIPHS_IO_MUX_GPIO0_U, FUNC_GPIO0); // GPIO0设为IO口
    //2.由于GPIO0通过R3进行电源上拉，所以GPIO0默认为高电平状态，在这里要将内部拉高取消
    PIN_PULLUP_DIS(PERIPHS_IO_MUX_GPIO0_U);
    //3.设定GPIO0引脚为输入模式
    GPIO_DIS_OUTPUT(GPIO_ID_PIN(0));
}

void KeyExtiInitConfig(void)
{
    //1.关闭GPIO中断
    ETS_GPIO_INTR_DISABLE();
    //2.注册中断回调函数
    ETS_GPIO_INTR_ATTACH((ets_isr_t)KeyExtiInterruptFunction, NULL);
    //3.设置外部中断IO口以及触发方式
    gpio_pin_intr_state_set(GPIO_ID_PIN(0), GPIO_PIN_INTR_NEGEDGE);    // GPIO_0下降沿中断
    //4.打开GPIO中断
    ETS_GPIO_INTR_ENABLE();
}

//按键外部中断服务函数--需要函数声明
void KeyExtiInterruptFunction(void)
{
    static i = 1;
    uint32 gpio_status;
    uint32 GpioFlag;
    //1.读取GPIO中断状态
    gpio_status = GPIO_REG_READ(GPIO_STATUS_ADDRESS);
    //2.清除中断状态位
    GPIO_REG_WRITE(GPIO_STATUS_W1TC_ADDRESS, gpio_status);
    //3.判断按键的中断状态
    GpioFlag = gpio_status & (0x01 << 0);
    os_printf("\r\n进入中断\r\n");
```

```
    if (GpioFlag == 1)
    {
        LED(i);
        i = !i;
        os_printf("\r\n按下按键\r\n");
    }
}
```

//key.h 源代码

```
/*
* key.h
*
*  Created on: 2018年8月10日
*      Author: 小良哥
*/

#ifndef APP_INCLUDE_DRIVER_KEY_H_
#define APP_INCLUDE_DRIVER_KEY_H_

#include "ets_sys.h"
#include "osapi.h"
#include "user_interface.h"
#include "driver/uart.h"
#include "gpio.h"
#include "eagle_soc.h"
#include "driver/delay.h"

void KeyInitConfig(void);
void KeyExtiInitConfig(void);
void KeyExtiInterruptFunction(void);
#define KEY (!!(GPIO_INPUT_GET(GPIO_ID_PIN(0))))

#endif /* APP_INCLUDE_DRIVER_KEY_H_ */
```

3.8 系统软件定时器

3.8.1 定时器相关注意事项

以下内容摘录于《ESP8266 Non-OS SDK API 参考》。

- 对于需要进行轮询的应用，建议使用系统定时器定期检查事件。

  --如果使用循环(while 和 for)，不仅效率低下，而且阻塞 CPU，不建议使用。

  --如果需要在定时器回调中执行 os_delay_us 或 while 或 for，请勿占用 CPU 超过 15ms。

- 请勿频繁电源定时器，建议频率不高于每 5ms 一次(微秒计时器则为 100us)。有关定时器使用的详细信息，请参阅 os_timer_arm()和相关的 API 说明。
- 微秒定时器不是很精确，请在回调中考虑 500us 的抖动。如需实现高精度的定时器，可以参考驱动程序(driver_lib)使用硬件定时器。请注意，PWM API 不能与硬件定时器同时使用。
- 请勿长时间关闭中断，ISR 执行事件也应当尽可能短(即微秒级)。

3.8.2 软件定时器相关 API 函数

以下软件定时器接口位于/ESP8266_NONOS_SDK/include/osapi.h。请注意，以下接口使用的定时器由软件实现，定时器的函数在任务中被执行。因为任务可能被中断，或者被其他高优先级的任务延迟，因此以下 os_timer 系列的接口并不能保证定时器精确执行。

如果需要精确的定时，例如，周期性操作某 GPIO，请使用硬件定时器，具体可参考 hw_timer.c，硬件定时器的执行函数在中断里被执行。

注意：

- 对于同一个 timer，os_timer_srm 或 os_timer_arm_us 不能重复调用，必须先 os_timer_disarm
- os_timer_setfn 必须在 timer 未使能的情况下调用，在 os_timer_arm 或 os_timer_arm_us 之前或 os_timer_disarm 之后

3.8.2.1 定时器数据类型

数据类型：os_timer_t
底层宏定义：#define os_timer_t ETSTimer
底层结构体重命名：

```
typedef struct _ETSTIMER_ {
    struct _ETSTIMER_    *timer_next;
    uint32_t             timer_expire;
    uint32_t             timer_period;
    ETSTimerFunc         *timer_func;
    void                 *timer_arg;
} ETSTimer;
```

例子：os_timer_t os_timer_500ms; //定义一个定时器名称

3.8.2.2 os_timer_arm()函数

函数原型：void os_timer_arm (os_timer_t *ptimer, uint32_t milliseconds, bool repeat_flag)
函数功能：使能毫秒级定时器
函数形参：
os_timer_t *ptimer：定时器结构—我们自己定义的定时器名称的地址
uint32_t milliseconds：定时事件，单位：毫秒
如未调用 system_timer_reinit，可支持范围 5 ~ 7067555ms
如未调用 system_timer_reinit，可支持范围 100 ~ 428496ms
bool repeat_flag：定时器是否重复 1：重复 0：不重复
返回值：无
例子：

os_timer_arm (&os_timer_500ms, 500, 1);//设置 os_timer_500ms 定时器为一个 500ms 可重复延时的定时器

3.8.2.3 os_timer_disarm()函数

函数原型：void os_timer_disarm (os_timer_t *ptimer)
函数功能：取消定时器定时
函数形参：
os_timer_t *ptimer：定时器名称地址
返回值：无
例子：os_timer_disarm (&os_timer_500ms);//取消 os_timer_500ms 定时器

3.8.2.4 os_timer_setfn()函数

函数原型：void os_timer_setfn(os_timer_t *ptimer, os_timer_func_t *pfunction, void *parg);

```
#define os_timer_setfn ets_timer_setfn
void ets_timer_setfn(os_timer_t *ptimer, os_timer_func_t *pfunction, void *parg);
```

函数功能：设置定时器回调函数。使用定时器，必须设置回调函数。
函数形参：
os_timer_t *ptimer：定时器名称地址

```
#define os_timer_t    ETSTimer
typedef struct _ETSTIMER_ {
    struct _ETSTIMER_      *timer_next;
    uint32_t               timer_expire;
    uint32_t               timer_period;
    ETSTimerFunc           *timer_func;
    void                   *timer_arg;
} ETSTimer;
```

os_timer_func_t *pfunction：定时器回调函数[需要进行类型转换]
void *parg：回调函数的参数
例子：os_timer_setfn(&os_timer_500ms,(os_timer_func_t*)TimerBackFunction,NULL);//设置 os_timer_500ms_back()为回调函数

回调函数的作用类似于中断服务函数，但是没有固定的回调函数的名称，需要自己定义。

3.8.2.5 system_timer_reinit()函数

函数原型：void system_timer_reinit (void)
函数功能：
重新初始化定时器，当需要使用微秒级定时器时调用
函数形参：无
返回值：无

注：1.同时定义 2. system_timer_reinit 在程序最开始调用，user_init 的第一句。

3.8.2.6 os_timer_arm_us()函数

函数原型：void os_timer_arm_us (os_timer_t *ptimer, uint32_t microseconds, bool repeat_flag);
函数功能：使能微秒级定时器
函数形参：
os_timer_t *ptimer：定时器地址
uint32_t microseconds：定时时间，单位：微秒，最小定时 0x64，最大可输入 0Xffffffff(100 - 268435455)
bool repeat_flag：定时器是否重复
返回值：无
例子：os_timer_arm_us(&os_timer_500ms,500000);//定时器 500ms 延时

3.8.3 定时器控制参考代码

//timer.c 源代码

```
/*
* timer.c
*
*  Created on: 2018年8月14日
*      Author: 小良哥
*/

#include "driver/delay.h"
#include "driver/led.h"
#include "driver/key.h"
#include "driver/uart.h"
#include "driver/timer.h"

/*
* 函数名称：void TimerInitConfig(uint32_t DelayMs,bool repeat_flag)
* 函数功能：定时器初始化函数
* 函数形参：
*       uint32_t DelayMs   延时时间
*       bool repeat_flag   是否重复
*            1：重复   0：不重复
* 返回值：
* */
os_timer_t os_timer_500ms;//定义一个定时器名称
void TimerInitConfig(uint32_t DelayMs, bool repeat_flag)
{
    os_timer_disarm(&os_timer_500ms);//关闭定时器
    os_timer_setfn(&os_timer_500ms, (os_timer_func_t *)TimerBackFunction, NULL);//设置定时器回调函数
    os_timer_arm(&os_timer_500ms, DelayMs, repeat_flag);
}

/*
* 函数名称：void TimerBackFunction(void)
```

```
* 函数功能：定时器回调函数
* 函数形参：无
* 返回值：    无
* */
int i = 1;
void TimerBackFunction(void)
{
    //  static int i = 1;//为什么静态变量不起作用？？？
    i = !i;
    LED(i);
    os_printf("\r\n定时器测试代码---LED灯翻转\r\n");
}
```

//timer.h 源代码

```
/*
* timer.h
*
*  Created on: 2018年8月14日
*      Author: 小良哥
*/

#ifndef APP_INCLUDE_DRIVER_TIMER_H_
#define APP_INCLUDE_DRIVER_TIMER_H_

#include "ets_sys.h"
#include "osapi.h"

#include "user_interface.h"
#include "gpio.h"
#include "eagle_soc.h"
#include "osapi.h"

extern os_timer_t os_timer_500ms;

void TimerInitConfig(uint32_t DelayMs, bool repeat_flag);
void TimerBackFunction(void);

#endif /* APP_INCLUDE_DRIVER_TIMER_H_ */
```

//user_main.c 源代码

//添加头文件

```
#include "ets_sys.h"
#include "osapi.h"

#include "user_interface.h"
```

```
#include "driver/uart.h"
#include "gpio.h"
#include "eagle_soc.h"
#include "driver/delay.h"
#include "driver/led.h"
#include "driver/key.h"

/******************************************************************************
* FunctionName : user_init
* Description  : entry of user application, init user function here
* Parameters   : none
* Returns      : none
*******************************************************************************/
void ICACHE_FLASH_ATTR user_init(void)
{
    system_soft_wdt_feed();//喂软件看门狗，防止程序跑偏
    uart_init(9600, 9600);//设置串口波特率
    LedInitConfig();//LED灯初始化函数
    KeyInitConfig();//按键初始化函数
    KeyExtiInitConfig();//配置按键外部中断
    TimerInitConfig(500, 1);
    os_printf("=========================================\r\n");
    os_printf("\t SDK version:\t%s", system_get_sdk_version());
    os_printf("\r\n嵌入式陈工个人编辑资料\r\n未经本人同意请勿私自传播\r\n");
    os_printf("\r\n定时器控制代码\r\n");
    os_printf("\r\n带看门狗\r\n");
    os_printf("=========================================\r\n");
}
```

3.9 硬件定时器

以下硬件中断定时器接口位于 /ESP8266_NONOS_SDK/examples/driver_lib/hw_timer.c。

硬件中断定时器就是利用中断进行的定时，而且比软件定时器更为的准确。以下内容摘录于《ESP8266 Non-OS SDK API 参考》。

3.9.1 硬件定时器相关 API 函数

3.9.1.1 hw_timer_init()函数

函数原型：void hw_timer_init (FRC1_TIMER_SOURCE_TYPE source_type,u8 req)
函数功能：初始化硬件 ISR 定时器
函数形参：

FRC1_TIMER_SOURCE_TYPE source_type：定时器的 ISR 源

可填参数：

FRC1_SOURCE： 使用 FRC1 中断源

NMI_SOURCE： 使用 NMI 中断源

u8 req：是否支持自动填装

可填参数：

0：不自动填装

1：自动填装

返回值：无

例子：hw_timer_init(FRC1_SOURCE,1);

3.9.1.2 hw_timer_arm()函数

函数原型：void hw_timer_arm (uint32 val)

函数功能：使能硬件中断定时器

函数形参：

uint32 val：定时时间

自动填装模式：

使用 FRC1 中断源 FRC1_SOURCE，取值范围：50 – 1677721us

使用 NMI 中断源 NMI_SOURCE，取值范围：100 -- 1677721us

非自动填装模式：取值范围：10 – 1677721us

返回值：无

例子：hw_timer_arm(100);

3.9.1.3 hw_timer_set_func()函数

函数原型：void hw_timer_set_func(void (* user_hw_timer_cb_set)(void))

函数功能：设置定时器回调函数。使用定时器，必须设置回调函数。

函数形参:

void (* user_hw_timer_cb_set)(void)：定时器回调函数，函数定义时请勿添加 ICACHE_FLASH_ATTR 宏。

返回值：无

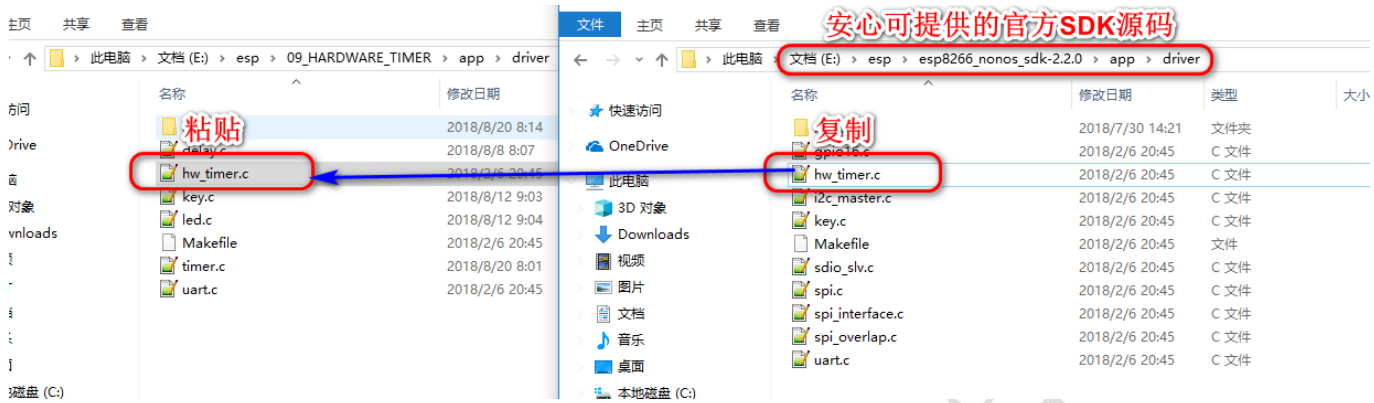
例子：hw_timer_set_func(hw_test_timer_cb);

3.9.2 使用注意事项

1. 如果使用 NMI 中断源，且为自动填装的定时器，调用 hw_timer_arm 时参数 val 必须大于 100。
2. 如果使用 NMI 中断源，那么该定时器将为最高优先级，可打断其他 ISR。
3. 如果使用 FRC1 中断源，那么该定时器无法打断其他 ISR。
4. hw_timer.c 的接口不能跟 PWM 驱动接口函数同时使用，因为二者共用了同一个硬件定时器。
5. 硬件中断定时器的回调函数定义，请勿添加 ICACHE_FLASH_ATTR 宏。
6. 使用 hw_timer.c 的接口，请勿调用 wifi_set_sleep_type(LIGT_SLEEP); 将自动睡眠模式设置为 Light-sleep。因为 Light-sleep 在睡眠期间会停 CPU，停 CPU 期间不能响应 NMI 中断。

3.9.3 参考测试代码

1.因为硬件定时器的 API 函数接口位于 /ESP8266_NONOS_SDK/examples/driver_lib/hw_timer.c 之中，所有我们要先移植 hw_timer.c 到我们的工程



2.编写源代码

```
/*
* HwTimer.c
*
*  Created on: 2018年9月10日
*      Author: 小良哥
*/
#include "driver/HwTimer.h"
#include "driver/led.h"
#include "driver/uart.h"

/*
* 函数名称：void HwTimerInterrupt(void)
* 函数功能：硬件定时器回调函数
* 函数形参：无
* 返  回  值：无
* */
void HwTimerInterrupt(void)
{
    static int LedFlag = 1;
    LedFlag = !LedFlag;
    LED(LedFlag);
    os_printf("硬件定时器中断!!!\r\n");
}

/*
*
* */
void HwTimerInitConfig(void)
{
    //  hw_timer_init(0,1);   //使用FRC1中断源  FRC1_SOURCE[枚举为0],自动填装[0:不自动填装  1:自动填装]
```

```
    hw_timer_init(1, 1);  //使用NMI 中断源  FRC1_SOURCE[枚举为0],自动填装[0:不自动填装  1:自动填装]
    hw_timer_set_func(HwTimerInterrupt);//设置HwTimerInterrupt()为定时器回调函数
    hw_timer_arm(1000000);//使能硬件定时器  1秒钟延时[单位：us  最大值:1677721us]
}
```

```
/*
* HwTimer.h
*
*  Created on: 2018年9月10日
*      Author: 小良哥
*/

#ifndef APP_INCLUDE_DRIVER_HWTIMER_H_
#define APP_INCLUDE_DRIVER_HWTIMER_H_

#include "ets_sys.h"
#include "osapi.h"
void HwTimerInterrupt(void);
void HwTimerInitConfig(void);

#endif /* APP_INCLUDE_DRIVER_HWTIMER_H_ */
```

```
    //user_main.c
/******************************************************************************
* FunctionName : user_init
* Description  : entry of user application, init user function here
* Parameters   : none
* Returns      : none
*******************************************************************************/
void ICACHE_FLASH_ATTR user_init(void)
{
    system_soft_wdt_feed();//喂软件看门狗，防止程序跑偏
    uart_init(115200, 115200);//设置串口波特率
    LedInitConfig();//LED灯初始化函数
    KeyInitConfig();//按键初始化函数
    KeyExtiInitConfig();//配置按键外部中断
//  TimerInitConfig(500,1);
    hw_timer_set_func();
    HwTimerInitConfig();
    os_printf("==========================================\r\n");
    os_printf("\t SDK version:\t%s", system_get_sdk_version());
    os_printf("\r\n嵌入式陈工个人编辑资料\r\n未经本人同意请勿私自传播\r\n");
    os_printf("\r\n硬件定时器控制代码\r\n");
    os_printf("==========================================\r\n");
}
```

3.10 系统任务

3.10.1 系统任务原理

以下资料来自《ESP8266 Non-OS SDK API 参考》第 2.2 章节与第 3.3 章节。

Non-OS SDK 不像基于 RTOS 的应用程序支持任务调度。 Non-OS SDK 使用四种类型的函数：

- 应用函数
- 回调函数
- 用户任务
- 中断服务函数

3.10.1.1 应用函数

应用函数类似于嵌入式 C 编程中常用的 C 函数。这些函数必须由另一个函数调用。应用函数在定义时建议添加 **ICACHE_FLASH_ATTR** 宏，相应程序将存放在 flash 中，被调用时才会加载到 cache 运行。而如果添加了 **IRAM_ATTR** 宏的函数，则会在上电启动时加载到 iRAM 中。

3.10.1.2 回调函数

回调函数是指不直接从用户程序调用的函数。而是当某系统事件发送时，相应的回调函数由 non-OS SDK 内核调用执行。这使得开发者能够在不使用 RTOS 或者轮询事件的情况下响应实时事件。

要编写回调函数，用户首先需要使用相应的 register_cb API 注册回调函数。回调函数的示例代码包括定时器回调函数和网络事件回调函数。

3.10.1.3 中断服务函数

中断服务函数(ISR)是一种特殊类型的回调函数。发生硬件中断时会调用这些函数。当时能中断时，必须注册相应的中断处理函数。请注意，**ISR 必须添加 IRAM_ATTR**。

3.10.1.4 用户任务

用户任务可以分为三个优先级：0、1、2.任务优先级为 2 > 1 > 0。即 Non-OS SDK 最多只支持 3 个用户任务，优先级分别为 0、1、2.

用户任务一般用于函数不能被调用的情况下。要创建用户任务，请参阅本文档中的 system_os_task()的 API 描述。例如 espconn_disconnect() API 不能直接在 espconn 的回调函数中调用，因此建议开发者在 espconn 回调中创建用户任务来执行 espconn_disconnect()。

3.10.2 硬件中断定时器相关 API 函数

系统接口位于 /ESP8266_NONOS_SDK/include/user_interface.h。

os_XXX 系列接口位于 /ESP8266_NONOS_SDK/include/osapi.h。

3.10.2.1 system_os_task ()函数

函数原型：bool system_os_task(os_task_t task,uint8 prio,os_event_t *queue,uint8 qlen)

函数功能：创建系统任务，最多支持创建 3 个任务，优先级分别为 0/1/2

函数形参：

os_task_t task：函数任务

uint8 prio：任务优先级，可分为 0/1/2；0 为最低优先级。这表示最多只支持建立 3 个任务。

可填形参：

```
enum {
        USER_TASK_PRIO_0 = 0,
        USER_TASK_PRIO_1,
        USER_TASK_PRIO_2,
        USER_TASK_PRIO_MAX
      };
```

os_event_t *queue：消息队列指针

uint8 qlen：消息队列深度

返回值：

ture ：成功

false：失败

官方参考示例代码：

```
#define  SIG_RX              0
#define  TEST_QUEUE_LEN      4
os_event_t   *testQueue;

void test_task(os_event_t *e)
{
    switch (e->sig)
    {
        case SIG_RX:
            os_printf(sig_rx %c / n, (char)e->par);
            break;
        default:
            break;
    }
}

void task_init(void)
{
    testQueue = (os_event_t*)os_malloc(sizeof(os_event_t)*TEST_QUEUE_LEN);
    system_os_task(test_task, USER_TASK_PRIO_0, testQueue, TEST_QUEUE_LEN);
}
```

3.10.2.2 system_os_post()函数

函数原型：bool system_os_post(uint8 prio,os_signal_t sig,os_param_t par)
函数功能：向任务发送消息
函数形参：
uint8 prio：任务优先级，与建立时的任务优先级对应
os_signal_t sig：消息类型
os_param_t par：消息参数
返回值：
ture ：成功
false：失败

3.10.3 参考源码

3.10.3.1 系统任务调用顺序

① 创建任务指针

② 分配任务指针空间

③ 创建任务函数

④ 创建任务

⑤ 给系统安排任务

⑥ 编写任务函数(根据消息类型/消息参数实现相应功能)

3.10.3.2 参考驱动源码

```
//user_main.c
#include "ets_sys.h"
#include "osapi.h"

#include "user_interface.h"
#include "driver/uart.h"
#include "gpio.h"
#include "eagle_soc.h"
#include "driver/delay.h"
#include "driver/led.h"
#include "driver/key.h"
#include "driver/HwTimer.h"
#include "osapi.h"
#include "mem.h"                // 内存申请等函数

/*
*   ① 创建任务指针
*   ② 分配任务指针空间
*   ③ 创建任务
*   ④ 创建任务函数
```

```
*    ⑤ 给系统安排任务
*    ⑥ 编写任务函数(根据消息类型/消息参数实现相应功能)
* */
#define  TEST_QUEUE_LEN         4    //消息队列深度
os_event_t    *testQueue;            //① 创建任务指针

                                     //④ 创建任务  形参必须为s_event_t *类型
void test_task(os_event_t * Task_message)
{
    //⑤ 给系统安排任务
    os_printf("消息类型=%d，消息参数=%c\r\n", Task_message->sig, Task_message->par);
}

void ICACHE_FLASH_ATTR user_init(void)
{
    int i;
    char data = 0;
    char ch = 'a';
    uart_init(115200, 115200);//设置串口波特率
    DelayMs(1000);              // 延时1秒
    //  LedInitConfig();//LED灯初始化函数
    //  KeyInitConfig();//按键初始化函数
    //  KeyExtiInitConfig();//配置按键外部中断
    //  TimerInitConfig(500,1);
    //  hw_timer_set_func();
    //  HwTimerInitConfig();
    os_printf("=========================================\r\n");
    os_printf("\t SDK version:\t%s", system_get_sdk_version());
    os_printf("\r\n嵌入式陈工个人编辑资料\r\n未经本人同意请勿私自传播\r\n");
    os_printf("\r\n系统任务调度代码\r\n");
    os_printf("\r\n带看门狗\r\n");
    os_printf("=========================================\r\n");

    //② 分配任务指针空间
    testQueue = (os_event_t*)os_malloc(sizeof(os_event_t)*TEST_QUEUE_LEN);
    //③ 创建任务函数
    system_os_task(test_task, USER_TASK_PRIO_0, testQueue, TEST_QUEUE_LEN);
    for (i = 0; i < 5; i++)
    {
        system_soft_wdt_feed();//喂软件看门狗，防止程序跑偏

        os_DelayMs(1000);          // 延时1秒  系统延时函数
        os_printf("安排任务：Task = %d\r\n", i);

        // 调用任务(参数1=任务等级 / 参数2=消息类型 / 参数3=消息参数)
        // 注意：参数3必须为无符号整数，否则需要强制类型转换
        //-------------------------------------------------------------
```

```
        system_os_post(USER_TASK_PRIO_0, data ++, ch ++);
    }
    os_printf("任务创建完成\r\n");
}
```