ESP8266 SDIO

SPI Compatibility Mode

User Guide

Version 0.1

Espressif Systems IOT Team

http://bbs.espressif.com

Table of Contents

1. Overview 1

2. DEMO solution 2
   - 2.1. Introduction 2
   - 2.2. ESP8266 software compiling and downloading 2
   - 2.3. ESP8266 FLASH software downloading 3
   - 2.4. ESP8266 FLASH software downloading 3

3. ESP8266 software instruction 4
   - 3.1. Protocol principle: SDIO line breakage and SDIO status register 4
   - 3.2. Instructions on the read/write buffer and the registration linked list 5
   - 3.3. API functions in the ESP8266 DEMO 6
     - 3.3.1. void sdio_slave_init(void) 6
     - 3.3.2. void sdio_slave_isr(void *para) 6
     - 3.3.3. void rx_buff_load_done(uint16 rx_len) 6
     - 3.3.4. void tx_buff_handle_done(void) 6
     - 3.3.5. void rx_buff_read_done(void) 7
     - 3.3.6. void tx_buff_write_done(void) 7
     - 3.3.7. TRIG_TOHOST_INT() 7
     - 3.3.8. Other functions 7

4. Instruction on STM32 software 8
   - 4.1. Important functions 8
     - 4.1.1. void SdioRW (void *pvParameters) 8
     - 4.1.2. int esp_sdio_probe (void) 8
     - 4.1.3. int sif_spi_write_bytes (u32 addr, u8*src,u16 count,u8 func) 8
     - 4.1.4. interrupt sif_spi_read_bytes (u32 addr,u8* dst,u16 count,u8 func) 9
     - 4.1.5. int sif_spi_write_blocks (u32 addr, u8 * src, u16 count,u16 block_size) 9
     - 4.1.6. int sif_spi_read_blocks (u32 addr, u8 *dst, u16 count,u16 block_size) 10
     - 4.1.7. void EXTI9_5_IRQHandler (void) 11

1. Overview

This protocol uses the SDIO mode of the ESP8266 to communicate with other processor's SPI hosts. The electrical interface is connected through signal line No.4, including the SCLK, MOSI, MISO and interrupt signal No. 1 in the SPI protocol (note: no CS signal).

Downloading the ESP8266 SDIO can be different from downloading other programs. When the ESP8266 starts, the system reads the pin shared by the SPI interface and the SDIO interface by default. Therefore, the SDIO module communication protocol should be used. The ESP8266 should start in the SDIO mode, and then, the host will start the chip in the ESP8266 RAM through the SDIO downloaded programs. The majority of the programs that directly use CPU CACHE to call FLASH can be burnt to the FLASH chip connected to the HSPI interface beforehand.

Data received or sent by the ESP8266 SDIO is processed directly by the DMA module that supports linked list index.

The ESP8266 can receive and send the SDIO packets efficiently without using the CPU. It does so through the address of the memory map linked list.

2. DEMO solution

2.1. Introduction

The host is the Red Dragon demo board with STM32F103ZET6 as its core.The software is the FreeRTOS system developed by the IAR platform. The slave is the ESP_IOT reference board, which is based on the v0.9.3 SDK development.

2.2. ESP8266 software compiling and downloading

In the *SDIO communication demo \esp_iot_sdk_v0.9.3_sdio_demo\ap*, use the compiler to compile and generate the bin documents for downloading in order to complete the ESP8266 DEMO work.

ibmain.a in *SDIO communication demo \esp_iot_sdk_v0.9.3_sdio_demo\lib* is different from the version released in v0.9.3. When you use the released version of the SDK, use ***libmain.a*** in the DEMO to replace the original one. The new ***libmain.a*** will start the chip, and exchange the SPI module that reads FLASH and the HSPI mapping pin. Then, you can use DEMO to compile and generate.

Copy ***eagle.app.v6.irom0text.bin*** in *SDIO communication demo \esp_iot_sdk_v0.9.3_sdio_demo\bin* to *SDIO communication demoboard \XTCOM_UTIL*. ***eagle.app.v6.irom0text.bin*** is all the functions of FLASH chip read directly through the SPI by CPU CACHE in the ESP8266 program.

Run *BinToArray.exe* in *SDIO communication demo*. Transfer ***eagle.app.v6.flash.bin*** in *SDIO communication demo\ esp_iot_sdk_v0.9.3_sdio_demo\ bin* to ANSI C format array. The new array will be saved in *D:\*. The target route of *BinToArray.exe* must be *D:\*. If there is not a *D:\*, you can (1) use a virtual machine with a *D:\*; (2) connect the device to a U disk named D:\; or (3) search online for a tool that can transfer bin to array.

If there is a *D:\*, name ***hexarray.c*** in *D:\* as ***eagle_fw.h***, and define the array name as const unsigned char eagle_fw[] =....... Replace ***eagle_fw.h*** in *SDIO communication demo\STM32\ Eagle_Wifi_Driver\ egl_drv_simulation* (you can copy the array name and document name in the old ***eagle_fw.h***, rename the ***hexarray.c*** and use it to replace the old ***eagle_fw.h.***). Before starting the chip, write ***eagle.app.v6.flash.bin*** into the ESP8266 memory. ***eagle.app.v6.flash.bin*** should be transferred to array, and be written into the ESP8266 through STM32.

Use the IAR platform to open ***EglWB.ewp.eww*** in *SDIO communication demo\STM32\IAR\* to compile the programs.

2.3. ESP8266 FLASH software downloading

Use the serial line to connect the ESP_IOT reference board and the computer, and connect them with a 5V power supply. Connect J67 to the 2 pins on the right (enable the FLASH chip in the HSPI interface), and J66 to the 2 pins on the left (disable the FLASH chip in the SPI interface). Set MTD0, GPIO0 and GPIO2 to the UART mode 0, 0,1 (up, up, down).

Double-click *XTCOM_UTIL.exe* in *SDIO communication demo\XTCOM_UTIL*. Click Tools -> Config Device, and choose Com interface. Baud Rate: 115200. Click Open, and you will see open Success. Click Connect, and push the H Flash board power, you will see the connection is completed.

Click API TEST(A)->(5) HSpiFlash Image Download, and choose *eagle.app.v6.irom0text.bin* in *SDIO communication demo\XTCOM_UTIL*. Offset: 0x40000. Click Download, and the downloading will be completed.

2.4. ESP8266 FLASH software downloading

Use the pin header to connect the ESP_IOT reference board and the Red Dragon demo board. The details are shown below:

In the Red Dragon demo board JP1:

J62 pin headers in the ESP_IOT reference board (bottom-up)

GND	->	1	VSS/GND
SPI_CLK	->	4	SDIO_CLK
SPI_MOSI	->	5	SDIO_CMD
SPI_MISO	->	3	SDIO_DAT0
IRQ	->	2	SDIO_DAT1

The ESP_IOT reference board: change the jumper MTD0 to 1 (short the 2 pins below), GPIO0, GPIO2 random (1, x, x is the SDIO starting mode), CHIP_PD:ON (flip the switch downward). Keep jumper J66 connected to the 2 pins on the left, and jumper J67 connected to the 2 pins on the right.

Connect the 5V power adapter to the ESP_IOT reference board and the Red Dragon demo board. Turn on the demoboard power, download the compiled programs mentioned in Section 2.2 to STM32 in the IAR environment. Start the STM32 program, and turn on the ESP_IOT reference board power. the STM32 will write the starting program into the ESP8266, and after several seconds, it will automatically run the SDIO to return to the testing program.

3. ESP8266 software instruction

3.1. Protocol principle: SDIO line breakage and SDIO status register

In the SDIO SPI compatibility mode, pin SD_DATA1 of the ESP8266 is used as the interrupt line to send signals to the SPI host, and the signals are active low. When the ESP8266 SDIO status register is upgraded by software, the interrupt line will change from active high to active low. The host should write in data to resume the active high through SDIO. (to be specific, the host should write 1 into register with the address 0x30 through CMD53 or CMD52 command in order to resume the active high of the interrupt line.)

the SDIO status register is 32 bits, it is revised by ESP8266 software, and it can be read by the host through CMD53 or CMD52 command. The address is 0x20-0x23. The data structure is shown as below:

```
struct sdio_slave_status_element
{
                u32 wr_busy:1;
                u32 rd_empty :1;
                u32 comm_cnt :3;
                u32 intr_no :3;
                u32 rx_length:16;
                32 res:8;
};
```

To be specific:

(1) `wr_busy`, bit 0: 1, write buffer of the slave is full, and the ESP8266 is processing data from the host; 0, write buffer is empty, users can write data into the buffer.

(2) `rd_empty`, bit 1: 1, read buffer of the slave is empty, no data has been updated; 0, there is new data in the buffer for the host to read.

(3) `comm_cnt`, bit 2-4: count the read/write communication. Each time the ESP8266 SDIO module finishes an effective packet-reading/packet-writing, the count will increase by 1. Therefore, the host can judge whether a read/write communication has been effectively responded by the ESP8266.

(4) `intr_no`, bit 5-7: the protocol does not use this variable; reserved.

(5) `rx_length`, bit 8-23: actual length of the packets prepared in the read buffer.

(6) `res`, bit 24-31: reserved.

The communication procedures of the host are shown as below:

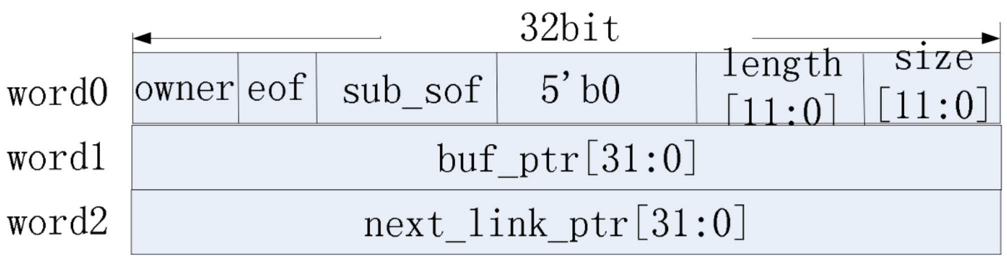
(1) upon receiving the interrupt request, the host reads the SDIO status register, and then clears the interuption, and reads/writes the packets according to the status register;

(2) it checks the SDIO status register regularly, and reads/writes the packets according to the status register.

3.2. Instructions on the read/write buffer and the registration linked list

DMA will directly send packets received and sent by the ESP8266 SDIO to corresponding memories. The ESP8266 software will define the linked list registration structure (or array), and buffer(s). In this example, only one buffer is used, and there is only one element in the linked list. Write the first address of the buffer into the linked list registration structure, and write in other information. When you write the first address of the linked list structure into the corresponding hardware register in the ESP8266, the DMA can automatically process the SDIO and the buffer.

The linked list registration structure is shown as below:



(1) `owner`: 1'b0: operator of the current link buffer is SW; operator of the current link buffer. MAC does not use this bit. 1'b1: operator of the current link buffer is HW.

(2) `eof`: flag of the end of the frame (for the end of AMPDU sub-frames, this flag is not needed). When the MAC sends the frames, it is used to mark the end of the frames. For links in eof, buffer_length[11:0] must be equal to the length of the remaining part of the frame. Otherwise, the MAC will report an error. When the MAC receives frames, it is used to indicate that the reception has been completed, and the value is set by hardware.

(3) `sub_sof`: the flag of the start of the sub-frame. It is used to distinguish different AMPDU sub-frames. It is only used when the MAC is sending packets.

(4) `length[11:0]`: actual size of the buffer.

(5) `size[11:0]` : total size of the buffer.

(6) `buf_ptr[31:0]`: starting address of the buffer.

(7) `next_link_ptr[31:0]`: starting address of the next discripter. When the MAC is receiving frames, the value is 0, indicating that there is no empty buffer to receive the frames.

3.3. API functions in the ESP8266 DEMO

3.3.1. void sdio_slave_init(void)

Function
Initialise the SDIO module, including initialising the status register, initialising the Rx and Tx registration linked list, configuring the communication interrupt line mode, configuring packet-sending/receiving interruption, and registering the interrupt service routine, etc.

3.3.2. void sdio_slave_isr(void *para)

Function and trigger condition
The SDIO interrupt processing function; this function will be triggered when the SDIO successfully receives or sends a packet. in DEMO, all the ESP8266 testing procedures are completed in the interrupt processing function. All the processing procedures of the registration linked lists, status registers and data during the communication process can be found in this function.

3.3.3. void rx_buff_load_done(uint16 rx_len)

Function
When `rx_buffer` receives new packets, this function should be called to change the status of the new packets to "to be read". This function contains related operations of the software/hardware of the registration linked list, and the status register. In DEMO, this function will be called in the interrupt service routine.

parameter
`rx_len`: actual length of the new packet (unit: byte).

3.3.4. void tx_buff_handle_done(void)

Function
When data in `tx_buffer` has been processed, this function should be called to change the SDIO status to "sent" in order to receive the next packet. This function contains related operations of the software/hardware of the registration linked list, and the status register. In DEMO, this function will be called in the interrupt service routine.

3.3.5. void rx_buff_read_done(void)

Function

When data in `rx_buffe` has been read, this function should be called to change the SDIO status to "non-readable". This function contains related operations of the status register, and should be called at the beginning of the `RX_EOF` interrupt service.

3.3.6. void tx_buff_write_done(void)

Function

When `tx_buffer` receives new packets, this function should be called to change the SDIO status to "non-writable". This function contains related operations of the status register, and should be called at the beginning of the `TX_EOF` interrupt service.

3.3.7. TRIG_TOHOST_INT()

Function

Macro, pull low the communication interrupt line, inform the host.

3.3.8. Other functions

Other functions are called during the testing procedure.

4. Instruction on STM32 software

4.1. Important functions

4.1.1. void SdioRW (void *pvParameters)

Function

SDIO testing thread, it contains all the read/write procedures.

Location

egl_thread.c. Registered by `SPITest()` of the same file in *egl_thread.c*.

4.1.2. int esp_sdio_probe (void)

Function

Enable related programs in the ESP8266.

Location

esp_main_sim.c. Called by `SPITest()` in *egl_thread.c*.

4.1.3. int sif_spi_write_bytes (u32 addr, u8*src,u16 count,u8 func)

Function

Write the SDIO byte mode into the API; encapsulate the write-in function of the CMD53 byte mode. It can process the register and the packets. According to the SDIO protocol, the maximum data length is 512 bytes.

Location

port_spi.c. Called by `SdioRW` in *egl_thread.c*.

Parameters

`src`: starting address of the packet to be sent.

`count`: length of the packet to be sent, (unit: byte)

`func`: function number. It is 0 for communication of `block_size` in the block mode used to revise the SDIO CMD53, and 1 for all other communications.

`addr`: starting address of the data to be written in. If you want to process the register, input the corresponding address, for example, 0x30, interrupt line clearance register, 0x110, revise block_size (`func`=0). If you want to process the packets, input a value that equals to 0x1f800 - `tx_length`, and 0x1f800 - `tx_length` should equal to count. If count > `tx_lengt`, the SPI host will send packets of count length. But data between `tx_length` + 1 and count will be discarded by the ESP8266 SDIO module. Therefore, when sending packets, `addr` is related to the actual length of the effective data.

4.1.4. interrupt sif_spi_read_bytes (u32 addr,u8* dst,u16 count,u8 func)

Function

The SDIO byte mode reads the API; encapsulate the read function of the CMD53 byte mode. It can process the register or the packets. According to the SDIO protocol, the maximum data length is 512 bytes.

Location

port_spi.c. Called by `SdioRW` in *egl_thread.c*.

Parameters

`dst`: starting address of the receiving buffer

`count`: length of the packet to be received (unit: byte)

`func`: function number. It is 0 for communication of `block_size` in the block mode used to read the SDIO CMD53, and 1 for all other communications.

`addr`: starting address of the data to be read. If you want to operate the register, input the corresponding address. For example, 0x20, the SDIO status register. If you want to operate the packets, input a value that equals 0x1f800 - `tx_length`, and 0x1f800 - `tx_length` equals count. If count > `tx_lengt`, the SPI host will send packets of count length. But data between `tx_length` + 1 and count will be discarded by the ESP8266 SDIO module. Therefore, when sending packets, `addr` is related to the actual length of the effective data.

4.1.5. int sif_spi_write_blocks (u32 addr, u8 * src, u16 count,u16 block_size)

Function

Write the SDIO block mode into the API; encapsulate the write-in function of the CMD53 byte mode. It can only transport the packets, According to the SDIO protocol, the maximum data length is 512 blocks.

Location

port_spi.c. Called by `dioRW` in `egl_thread.c and sif_io_sync` used by the program downloader in *es-p_main_sim.c*.

Parameters

`src`: starting address of the packet to be sent.

`count`: length of the packet to be sent (unit: block)

`block_size`: the number of bytes in 1 block. It should be equal to the 16 bit value whose `func`=0, and whose `addr`=0x110-111. In general, when initialising the SDIO, block_size of the ESP8266 SDIO should be configured. The starting value of DEMO is 512. During the operation, it is configured to be 1024. block_size should be an integer multiple of 4.

`addr`: starting address of the data to be written in. Input a value that equals 0x1f800 - `tx_length` (the same as the byte mode), and the `tx_length` should equal to count.

4.1.6. int sif_spi_read_blocks (u32 addr, u8 *dst, u16 count,u16 block_size)

Function

Write the SDIO block mode into the API; encapsulate the write-in function of the CMD53 byte mode. It can only transport the packets, According to the SDIO protocol, the maximum data length is 512 blocks.

Location

port_spi.c. Called by `dioRW` in `egl_thread.c and sif_io_sync` used by the program downloader in ***es-p_main_sim.c***.

Parameters

`src`: starting address of the receiving buffer

`count`: length of the packet to be received (unit: block)

`block_size`: the number of bytes in 1 block. It should be equal to the 16 bit value whose `func` = 0, and whose `addr`=0x110-111. In general, when initialising the SDIO, block_size of the ESP8266 SDIO should be configured. The starting value of DEMO is 512. During the operation, it is configured to be 1024. `block_size` should be an integer multiple of 4.

`addr`: starting address of the data to be read. Input a value that equals 0x1f800 - `tx_length` (the same as the byte mode), and the `tx_length` should equal to count.

4.1.7. void EXTI9_5_IRQHandler (void)

Function

The communication interrupt processing function offers enable signal for `egl_arch_sem_wait` (& BusIrqReadSem,1000) in thread function `SdioRW`, so that `SdioRW` thread can exit the wait state, and read the SDIO status register.

Location

spi_cfg.c.