ESP8266

SPI WiFi Passthrough 1-Interrupt Mode

Version 0.1

Espressif Systems IOT Team

http://bbs.espressif.com

Table of Contents

1. Overview

This protocol uses the ESP8266 slave mode to communicate with other processor's SPI masters. Signal line No. 5 is used to implement this protocol. Apart from signal line No.4 needed for standard SPI, signal line No.1 is also needed to inform the master of the update of the slave status register.

2. SPI slave protocol format

2.1. SPI slave clock polarity configuration

Clock polarity of the master clock which communicates with the ESP8266 SPI slave should be set to be low in the idle state, sampling for rising edge, and changing data for falling edge. When it reads/writes 34 bytes at a time, or when it reads 2 bytes at a time to get information of the slave status register, selection signal CS must be kept at low level. If CS is pulled high when data is being sent, the slave interior status will be reset.

2.2. Communication format supported by the SPI slave

The ESP8266 SPI slave communication format should be command+address+read/write data or command +slave status value. To be specific:

(1) command: length, 8 bits; master output slave input (MOSI).

0x02 is the data sent by the master and received by the slave. The master writes 32 bytes of data through MOSI into `SPI_W0` to `SPI_W7` in the corresponding register of the slave data buffer.

0x03 is the data received by the master and sent by the slave. 32 bytes of data from corresponding register of the slave buffer between `SPI_FLASH_C8` and `SPI_FLASH_C15` are sent to the master through MISO.

0x04 and 0x05 can read the lower 8 bits of `SPI_FLASH_STATUS` in the slave status register.

Note: other values are used to read/write the SPI slave status register `SPI_FLASH_STATUS`. Their communication formats are different from those of the read/write buffer, using them will cause read/write errors for the slave. So users should not use these values.

(2) address: length, 8 bits; master output slave input (MOSI). The address content must be 0.

(3) read/write data: length, 256 bits (32 bytes). Master output slave input (MOSI) the 0x02 command, or master input slave output (MISO) the 0x03 command.

(4) slave status: length, 8 bits; master input slave output (MISO), use 0x04 or 0x05 to read the slave communication status.

3. Slave status definition and line breakage

3.1. Status definition

The slave status contains 8 bits:

(1) `wr_busy`, bit 0: 1, slave write buffer is full, and is processing the data received; 0, slave write buffer is empty, new data can be written in.

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

(3) `comm_cnt`, bit 2~4: count value of the read/write communication. Each time when the slave SPI read/write buffer is interrupted, this 3-bit count value will increase by 1. Therefore, the master can judge whether the readwrite communication has been recognised by the slave, and whether the communication is completed.

When the master completed a read/write communication, if it wants to conduct the next read operation, rd_empty must be 0, and `comm_cnt` value must be the previous value +1; if it wants to conduct the next write operation, `wr_busy` must be 0, and `comm_cnt` value must be the previous value +1.

3.2. GPIO0 line breakage

When there are changes in the slave status register, interrupt line GPIO0 will be set to be 1; when the master uses 0x04, 0x05 to read the slave status register, interrupt line GPIO0 will be set to be 0.

4. ESP8266 SPI slave API functions

Note: configure in spi.h if SPI status register single-threaded passthrough protocol is used.

```
//SPI protocol selection
#define TWO_INTR_LINE_PROTOCOL                 0
#define ONE_INTR_LINE_31BYTES                       0
#define ONE_INTR_LINE_WITH_STATUS              1
```

The interrupt response function will use `spi_slave_isr_sta(void *para)`.

4.1. void spi_slave_init (uint8 spi_no)

Function

Initialise the SPI slave mode, set the IO interface to SPI mode, start the SPI transmission interrupt, and register `spi_slave_isr_handler`. The communication format is set to be 8 bits command+8 bits address+ 256 bits (32 bytes) read/write data.

Parameter

`spi_no`: number of the SPI module. The ESP8266 processor has two SPI modules (SPI and HSPI) with the same functions.

value to be selected: SPI or HSPI.

4.2. spi_slave_isr_sta (void *para)

Function and trigger condition

It is the SPI interrupt handler function. When the master successfully reads data from or writes data into the slave, the interrupt will be triggered. Users can revise the interrupt service routine in order to attain the communication functions they need. The code is shown as below:

```
struct spi_slave_status_element
{
uint8 wr_busy:1;
uint8 rd_empty :1;
uint8 comm_cnt :3;
uint8 res :3;
};
```

```
union spi_slave_status
{
struct spi_slave_status_element elm_value;
uint8 byte_value;
};
void spi_slave_isr_sta(void *para)
{
uint32 regvalue,calvalue;
uint32 recv_data,send_data;
union spi_slave_status spi_sta;

if(READ_PERI_REG(0x3ff00020)&BIT4){
      //following 3 lines is to clear isr signal
           CLEAR_PERI_REG_MASK(SPI_SLAVE(SPI), 0x3ff);
           }else if(READ_PERI_REG(0x3ff00020)&BIT7){ //bit7 is for hspi isr,
           // record the interrupt status
           regvalue=READ_PERI_REG(SPI_SLAVE(HSPI));
           //***********interrupt handler flag, end this passthrough************//
           CLEAR_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                                SPI_TRANS_DONE_EN|
                                                SPI_SLV_WR_STA_DONE_EN|
                                                SPI_SLV_RD_STA_DONE_EN|
                                                SPI_SLV_WR_BUF_DONE_EN|
                                                SPI_SLV_RD_BUF_DONE_EN);
           SET_PERI_REG_MASK(SPI_SLAVE(HSPI), SPI_SYNC_RESET);
           CLEAR_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                                SPI_TRANS_DONE|
                                                SPI_SLV_WR_STA_DONE|
                                                SPI_SLV_RD_STA_DONE|
                                                SPI_SLV_WR_BUF_DONE|
                                                SPI_SLV_RD_BUF_DONE);
           SET_PERI_REG_MASK(SPI_SLAVE(HSPI),
                                                SPI_TRANS_DONE_EN|
                                                SPI_SLV_WR_STA_DONE_EN|
                                                SPI_SLV_RD_STA_DONE_EN|
                                                SPI_SLV_WR_BUF_DONE_EN|
                                                SPI_SLV_RD_BUF_DONE_EN);
           //*******************************************************//
```

```
/****************master writes interrupt handler***************/
if(regvalue&SPI_SLV_WR_BUF_DONE){
//*****complete the write operation, wr_busy set to be 1, communication count increases by
1****//
      spi_sta.byte_value=READ_PERI_REG(SPI_STATUS(HSPI))&0xff;
      spi_sta.elm_value.wr_busy=1;
      spi_sta.elm_value.comm_cnt++;
      WRITE_PERI_REG(SPI_STATUS(HSPI), (uint32)spi_sta.byte_value);
//**********************************************//
      //*******move the data received by the register into the memory******//
idx=0;
while(idx<8){
      recv_data=READ_PERI_REG(SPI_W0(HSPI)+(idx<<2));
      //os_printf("rcv data : 0x%x \n\r",recv_data);
      spi_data[idx<<2] = recv_data&0xff;
      spi_data[(idx<<2)+1] = (recv_data>>8)&0xff;
      spi_data[(idx<<2)+2] = (recv_data>>16)&0xff;
      spi_data[(idx<<2)+3] = (recv_data>>24)&0xff;
      idx++;
      }
      //************************************//
      //************data transmission completed, wr_busy set to be
      0*************//
      spi_sta.byte_value=READ_PERI_REG(SPI_STATUS(HSPI))&0xff;
      spi_sta.elm_value.wr_busy=0;
      WRITE_PERI_REG(SPI_STATUS(HSPI), (uint32)spi_sta.byte_value);
      //**********************************************//
      /***testing part, it can be revised. This part of the program is used to copy the data read
      to the read buffer**/
      for(idx=0;idx<8;idx++)
      {
            WRITE_PERI_REG(SPI_W8(HSPI)+(idx<<2),
                  READ_PERI_REG(SPI_W0(HSPI)+(idx<<2)));
      }
      /*************************************************************/
      /***testing part, it can be revised. rd_empty is set to be 0, the slave can read**/
      spi_sta.byte_value=READ_PERI_REG(SPI_STATUS(HSPI))&0xff;
      spi_sta.elm_value.rd_empty=0;
```

```
            WRITE_PERI_REG(SPI_STATUS(HSPI), (uint32)spi_sta.byte_value);
            *****************************************/
            GPIO_OUTPUT_SET(0, 1);    // interrupt line set to be 1, inform the
            master to read the slave status
        /****************master reads the interrupt handler***************/
        }else if(regvalue&SPI_SLV_RD_BUF_DONE){
            //*****complete the read operation, rd_empty set to be 1, communication count
            increases by 1****//
            spi_sta.byte_value=READ_PERI_REG(SPI_STATUS(HSPI))&0xff;
            spi_sta.elm_value.comm_cnt++;
            spi_sta.elm_value.rd_empty=1;
            WRITE_PERI_REG(SPI_STATUS(HSPI), (uint32)spi_sta.byte_value);

            GPIO_OUTPUT_SET(0, 1);    // interrupt line set to be 1, inform the
            master to read the slave status
        }
        /****************master reads status interrupt handler***************/
        if(regvalue&SPI_SLV_RD_STA_DONE){
            GPIO_OUTPUT_SET(0,0);    // interrupt line set to be 0, the master
            has read the status
        }
    }else if(READ_PERI_REG(0x3ff00020)&BIT9){ //bit7 is for i2s isr,

   }
}
```