ESP8266

SPI WiFi Passthrough 2-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. 6 is used to implement this protocol. Apart from signal line No.4 needed for standard SPI, signal line No.2 is also needed to inform the master of information of the slave receive and send buffer status, so as to control the data flow.

2. ESP8266 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, 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 is similar to that of the master, it should be command + address + read/write data. 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 host 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_W8` and `SPI_W15` are sent to the master through MISO.

Note: other values are used to read/write the SPI slave status register `SPI_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.

3. Instruction on the data flow control line

The ESP8266 uses 2 GPIOs to output the slave receive buffer status and send buffer status.

3.1. GPIO0 MOSI buffer status

When GPIO0 enters the slave receive interrupt, the interrupt program will resume the SPI slave to communicable status in order to prepare for the next communication. Then, GPIO0 will be written to be low level, data received will be processed, and GPIO0 will be written to be high level to exit the interrupt program. Therefore:

(1) Between the master enables an SPI write communication to GPIO0 generates a falling edge, if users enable any other SPIs, communication errors will occur.

(2) When GPIO0 is at low level, if the master enables any SPI to write (0x02 command), SPI_W0 to SPI_W7 in the slave receive register will be covered. But if there is effective data in the slave send register (refer to GPIO2 instructions), when GPIO0 is at low level, master can be started to read (0x03 command) data between `SPI_W8` to `SPI_W15` in the slave send register.

(3) If GPIO0 shifts from low level to high level, it means the slave has processed data from `SPI_W0` to `SPI_W7` in the receive register, and the master can start another write operation (0x02 command).

3.2. GPIO2 master receives the slave send buffer status

GPIO2 activities are slightly different from those of GPIO0. In the slave send interrupt, the interrupt program will resume the SPI slave to communicable status in order to prepare for the next communication. Then, GPIO0 will be written to be low level, and quit the interrupt program. After that, if data is sent to the ESP8266 through WiFi and is required to be forwarded by SPI, ESP8266 software will be written into `SPI_W8` to `SPI_W15`, and GPIO2 will be set to be high level. Therefore:

(1) Between the master enables an SPI read communication to GPIO2 generates a falling edge, if users enable any other SPIs, communication errors will occur.

(2) When GPIO2 is at low level, if the master enables any SPI to read (0x03 command), it can only read data the same as the previous data, or incomplete data. But if data in the slave receive register has been processed (refer to GPIO2 instructions), when GPIO2 is at low level, master can be started to write (0x02 command).

(3) If GPIO2 shifts from low level to high level, it means the slave has updated data from SPI_W8 to SPI_W15 in the send register, and the master can start the another read operation (0x03 command).

3.3. Master communication logic implementation

Incomplete C code is used to briefly introduce the communication logic:

```
//wr_rdy: ready to conduct the next SPI write operation
//rd_rdy: ready to conduct the next SPI read operation
unsigned char wr_rdy=1,rd_rdy=0;
void spi_read_func(....)
{
      // before starting the read operation, check if there is new data for the
      slave to read (rd_rdy is non-0);
      // also, check if the previous write operation is completed; write
      operationcompleted and processing data (signal GPIO0 is 0), or new data can be
      written into the slave (wr_rdy is non-0)
      if(rd_rdy&&((GPIO0= =0)||wr_rdy)){
      rd_rdy=0;    //rd_rdy set to be 0
      spi_transmit(0x03,0,*read_buff);// start the SPI transmission, command 3 +
      address 0 + 32 bytes of data
      ….
      }
}
void spi_write_func(...)
{
      // before starting the write operation, check if there is new data for the
      slave to receive (rd_rdy is non-0);
      // also, check if the previous read operation is completed; completed, no new
      data to be read (signal GPIO2 is 0), or new data to be read (rd_rdy is non-0)
      if(wr_rdy&&((GPIO2= =0)||rd_rdy)){
      wr_rdy=0;    //wr_rdy set to be 0
      spi_transmit(0x02,0,*write_buff);// start the SPI transmission, command 2 +
      address 0 + 32 bytes of data
      ...
      }
}

GPIO0_Raising_Edge_ISR() // rising edge interrupt program connected to the ESP8266
GPIO0
{
      wr_rdy=1;     // data sent by the master has been processed, ready for the
      next write operation
}
```

```
GPIO2_Raising_Edge_ISR() // rising edge interrupt program connected to the ESP8266
GPIO2
{
      rd_rdy=1; // the slave updates the send buffer, the master is ready to read
```

3.4. ESP8266 SPI slave API functions

3.1. 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.

3.2. spi_slave_isr_handler(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 complete the communication. The code is shown below.

Code

```
uint32 regvalue;
static uint32 t1 =0;
static uint32 t2 =0;
t1=system_get_time();

if(READ_PERI_REG(0x3ff00020)&BIT4){      //bit4: SPI interrupt
CLEAR_PERI_REG_MASK(SPI_SLAVE(SPI), 0x3ff);
}else if(READ_PERI_REG(0x3ff00020)&BIT7){ //bit7: HSPI interrupt,
regvalue=READ_PERI_REG(SPI_SLAVE(HSPI));   // record the interrupt type
     // turn off the SPI interrupt enable
```

```
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);
     // resume the SPI slave to communicable status, in order to prepare for
     the next communication
SET_PERI_REG_MASK(SPI_SLAVE(HSPI), SPI_SYNC_RESET);
     // clear the interrupt flag
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);
     // turn on the SPI interrupt enable
     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);
     //MISO processing program
     if(regvalue&SPI_SLV_WR_BUF_DONE){
     GPIO_OUTPUT_SET(0, 0);   //GPIO0 set to be 0
     idx=0;
     //read the data received
          while(idx<8){
          recv_data=READ_PERI_REG(SPI_W0(HSPI)+4*idx);
          //os_printf("rcv data : 0x%x \n\r",recv_data);
          spi_data[4*idx+0] = recv_data&0xff;
          spi_data[4*idx+1] = (recv_data>>8)&0xff;
          spi_data[4*idx+2] = (recv_data>>16)&0xff;
          spi_data[4*idx+3] = (recv_data>>24)&0xff;
          idx++;
          }
     system_os_post(USER_TASK_PRIO_1,MOSI,0);// send the reception completed
     message
```

```
            GPIO_OUTPUT_SET(0, 1);                          //GPIO0 set to be 1
            SET_PERI_REG_MASK(SPI_SLAVE(HSPI),  SPI_SLV_WR_BUF_DONE_EN);
            //master reads, slave sends the processing program
      if(regvalue&SPI_SLV_RD_BUF_DONE){
            GPIO_OUTPUT_SET(2, 0);   //GPIO2 set to be 0
      }
}else if(READ_PERI_REG(0x3ff00020)&BIT9){ //bit7: I2S interrupt

}
```