3.10 LCD display picture

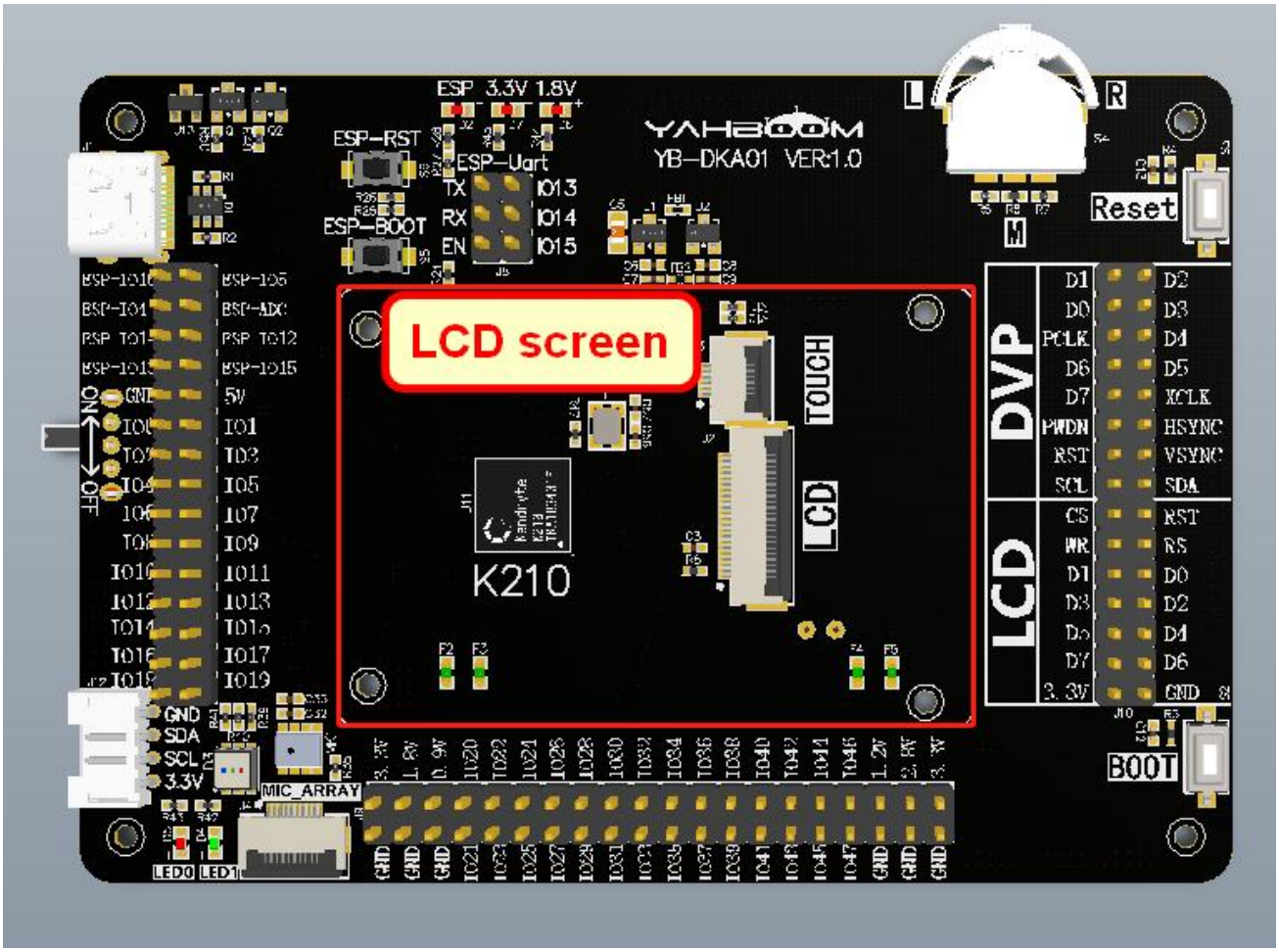
1. Experiment purpose

In this lesson, we mainly learns how to make LCD display pictures and string.

2.Experiment preparation

2.1 components

LCD



2.2 Component characteristics

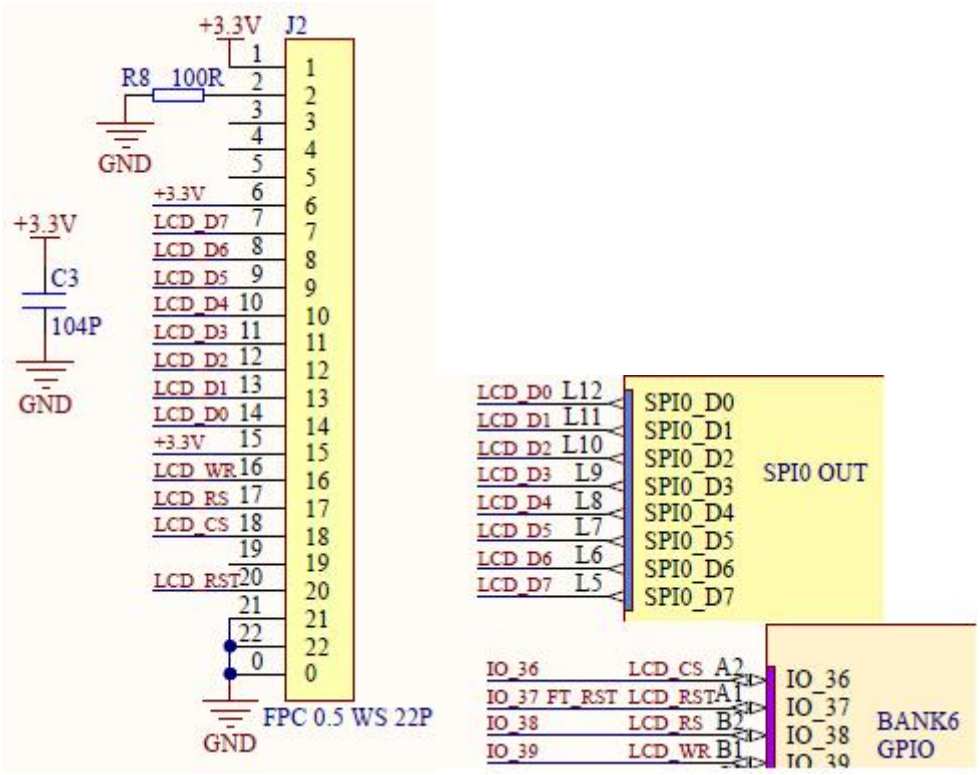
2.0-inch LCD, the resolution is 320*240, the driver chip is st7789, which is small in size, thin in thickness, and low in energy consumption.

Working voltage is 3.3V. Mterial is TFT.

2.3 Hardware connection

The K210 development board has an LCD display installed by default.

A total of eight pins of LCD_D0~D7 are connected to SPI0_D0~D7, LCD_CS is connected to IO36, LCD_RST is connected to IO37, LCD_RS is connected to IO38, and LCD_WR is connected to IO39.



2.4 SDK API function

The header file is spi.h

SPI is a high-speed, full-duplex, synchronous communication bus.

We will provide following interfaces to users.

• spi_init: Set the SPI working mode, multi-line mode and bit width.

• spi_init_non_standard: Set instruction length, address length, number of waiting clocks, instruction address transmission mode in multi-line mode.

• spi_send_data_standard: SPI standard mode transmits data.

• spi_send_data_standard_dma: Use DMA to transfer data in SPI standard mode.

• spi_receive_data_standard: Receive data in standard mode.

• spi_receive_data_standard_dma: Receive data through DMA in standard mode.

• spi_send_data_multiple: Send data in multi-wire mode.

• spi_send_data_multiple_dma: Using DMA to send data in multi-wire mode.

• spi_receive_data_multiple: Receive data in multi-wire mode.

• spi_receive_data_multiple_dma: Receive data through DMA in multi-wire mode.

• spi_fill_data_dma: The same data is always sent through DMA, which can be used to refresh data.

• spi_send_data_normal_dma: Send data through DMA, no need set instruction address.

• spi_set_clk_rate: Set the SPI clock frequency.

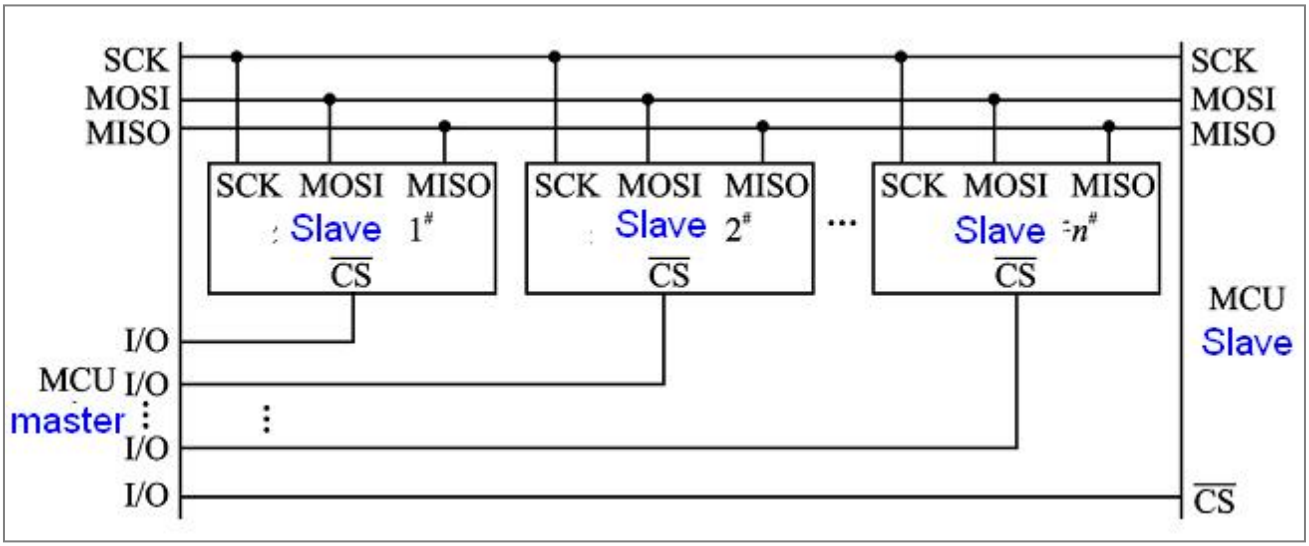
• spi_handle_data_dma: SPI transfers data through DMA.

3. Experimental principle

SPI is a high-speed, high-efficiency serial interface technology. It usually consists of a master module and one or more slave modules. The master module selects a slave module for synchronous communication.

SPI need 4 wires are required for communication, MISO (master device data input), MOSI (master device data output), SCLK (clock), CS (chip select).
(1) MISO-Master Input Slave Output, master device data input, slave device data output;
(2) MOSI-Master Output Slave Input, master device data output, slave device data input;
(3) SCLK-Serial Clock, clock signal, generated by the master device;
(4) CS-Chip Select, slave device enable signal, controlled by master device.

When there are multiple slave devices, each slave device has a chip select pin connected to the master device.
When master device need to communicates with a slave device. We need to pull the chip select pin level of the slave device low or high.



4. Experiment procedure

4.1 According to the above hardware connection pin diagram, K210 hardware pins and software functions use FPIOA mapping relationship.

```
/*****************************HARDWARE-PIN*********************************/
//Hardware IO port, corresponding Schematic
#define PIN_LCD_CS              (36)
#define PIN_LCD_RST             (37)
#define PIN_LCD_RS              (38)
#define PIN_LCD_WR              (39)

/*****************************SOFTWARE-GPIO********************************/
//Software GPIO port, corresponding program
#define LCD_RST_GPIONUM         (0)
#define LCD_RS_GPIONUM          (1)


/*****************************FUNC-GPIO************************************/
//Function of GPIO port, bound to hardware IO port
#define FUNC_LCD_CS             (FUNC_SPI0_SS3)
#define FUNC_LCD_RST            (FUNC_GPIOHS0 + LCD_RST_GPIONUM)
#define FUNC_LCD_RS             (FUNC_GPIOHS0 + LCD_RS_GPIONUM)
#define FUNC_LCD_WR             (FUNC_SPI0_SCLK)
```

```
void hardware_init(void)
{
    /**
    *PIN_LCD_CS     36
    *PIN_LCD_RST    37
    *PIN_LCD_RS     38
    *PIN_LCD_WR     39
    **/
    fpioa_set_function(PIN_LCD_CS,  FUNC_LCD_CS);
    fpioa_set_function(PIN_LCD_RST, FUNC_LCD_RST);
    fpioa_set_function(PIN_LCD_RS,  FUNC_LCD_RS);
    fpioa_set_function(PIN_LCD_WR,  FUNC_LCD_WR);

    /* Enable SPI0 and DVP data*/
    sysctl_set_spi0_dvp_data(1);

}
```

4.2 Set the IO port level voltage of the LCD to 1.8V

```
void io_set_power(void)
{
    sysctl_set_power_mode(SYSCTL_POWER_BANK6, SYSCTL_POWER_V18);
}
```

4.3 Initializing the LCD, the main function is to activate the LCD display, set the display direction and display color, and enable the display and clear the display.

```c
void lcd_init(void)
{
    uint8_t data = 0;
    /* Hardware initialization*/
    tft_hard_init();
    /*Reset LCD */
    tft_write_command(SOFTWARE_RESET);
    usleep(100000);
    /*Close sleep mode */
    tft_write_command(SLEEP_OFF);
    usleep(100000);
    /*Set the pixel format:65K, 16bit/pixel */
    tft_write_command(PIXEL_FORMAT_SET);
    data = 0x55;  /* 0101 0101*/
    tft_write_byte(&data, 1);
    /* Turn on display inversion */
    tft_write_command(INVERSION_DISPLAY_ON);
    /* set lce display direction */
    lcd_set_direction(DIR_YX_LRUD);

    /*Enable display*/
    tft_write_command(DISPLAY_ON);
    /*Clear display */
    lcd_clear(WHITE);
}
```

4.4 Display the picture, x1 and y1 are the starting point, width is the width of the picture (maximum 320), height is the height of the picture (maximum 240), and the ptr pointer points to the picture to be displayed.

```c
/* LCD draw picture*/
void lcd_draw_picture_half(uint16_t x1, uint16_t y1, uint16_t width, uint16_t height, uint16_t *ptr)
{
    lcd_set_area(x1, y1, x1 + width - 1, y1 + height - 1);
    tft_write_half(ptr, width * height);
}
```

4.5 Display string, where x and y are the starting coordinates, str pointer points to the string to be displayed, and color is the font color.

```c
/* LCD display string*/
void lcd_draw_string(uint16_t x, uint16_t y, char *str, uint16_t color)
{
    while (*str)
    {
        lcd_draw_char(x, y, *str, color);
        str++;
        x += 8;
    }
}
```

```c
/* LCD display characters*/
void lcd_draw_char(uint16_t x, uint16_t y, char c, uint16_t color)
{
    uint8_t i = 0;
    uint8_t j = 0;
    uint8_t data = 0;

    for (i = 0; i < 16; i++)
    {
        data = ascii0816[c * 16 + i];
        for (j = 0; j < 8; j++)
        {
            if (data & 0x80)
                lcd_draw_point(x + j, y, color);
            data <<= 1;
        }
        y++;
    }
}
```

```c
/* Set the color of a certain point*/
void lcd_draw_point(uint16_t x, uint16_t y, uint16_t color)
{
    lcd_set_area(x, y, x, y);
    tft_write_half(&color, 1);
}
```

4.6 ST7789 bottom SPI driver.
Start transmitting commands and data

```c
/* Start transmission command */
static void set_start_cmd(void)
{
    gpiohs_set_pin(LCD_RS_GPIONUM, GPIO_PV_LOW);
}

/* Start transferring data*/
static void set_start_data(void)
{
    gpiohs_set_pin(LCD_RS_GPIONUM, GPIO_PV_HIGH);
}
```

SPI write command, data

```c
/* SPI write command */
void tft_write_command(uint8_t cmd)
{
    set_start_cmd();
    spi_init(SPI_CHANNEL, SPI_WORK_MODE_0, SPI_FF_OCTAL, 8, 0);
    spi_init_non_standard(SPI_CHANNEL, 8 /*instrction length*/, 0 /*address length*/, 0 /*wait cycles*/,
                          SPI_AITM_AS_FRAME_FORMAT /*spi address trans mode*/);
    spi_send_data_normal_dma(DMAC_CHANNEL0, SPI_CHANNEL, SPI_SLAVE_SELECT, (uint8_t *)(&cmd), 1, SPI_TRANS_CHAR);
}

/*SPI write data (uint8_t type)*/
void tft_write_byte(uint8_t *data_buf, uint32_t length)
{
    set_start_data();
    spi_init(SPI_CHANNEL, SPI_WORK_MODE_0, SPI_FF_OCTAL, 8, 0);
    spi_init_non_standard(SPI_CHANNEL, 8 /*instrction length*/, 0 /*address length*/, 0 /*wait cycles*/,
                          SPI_AITM_AS_FRAME_FORMAT /*spi address trans mode*/);
    spi_send_data_normal_dma(DMAC_CHANNEL0, SPI_CHANNEL, SPI_SLAVE_SELECT, data_buf, length, SPI_TRANS_CHAR);
}
```

4.7 At the end of the main function is a while(1) loop.

```c
int main(void)
{
    /*Hardware pin initialization */
    hardware_init();

    /*Set IO port voltage */
    io_set_power();

    /*Initialize LCD */
    lcd_init();

    /*Display picture*/
    lcd_draw_picture_half(0, 0, 320, 240, gImage_logo);
    sleep(1);

    /*Display string*/
    lcd_draw_string(16, 40, "Hello Yahboom!", RED);
    lcd_draw_string(16, 60, "Nice to meet you!", BLUE);

    while (1);
    return 0;
}
```

4.8 Compile and debug, burn and run

Copy the lcd to the src directory in the SDK.

Then, enter the build directory and run the following command to compile.

cmake .. -DPROJ=lcd -G "MinGW Makefiles"

make

```
[ 93%] Linking C executable lcd
Generating .bin file ...
[100%] Built target lcd
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build>
```

After the compilation is complete, the **lcd.bin** file will be generated in the build folder.

We need to use the type-C data cable to connect the computer and the K210 development board.

Open kflash, select the corresponding device, and then burn the **lcd.bin** file to the K210 development board.

5. Experimental phenomenon

After the firmware is write,

The LCD screen will display the picture.

After one second, it will print "Hello Yahboom!" and "Nice to meet you!"

6. Experiment summary

6.1 The resolution of the LCD display is 320*240. Before displaying the picture, you need to convert the picture to a resolution of 320*240, and then use the picture conversion tool to convert the picture to a .c file, and then quote the picture variable.

6.2 LCD is based on SPI communication.

6.3 Before displaying, we need to configure the display direction and display format and other parameters.

Appendix -- API

Header file is spi.h

spi_init

Description: Set the SPI working mode, multi-line mode and bit width.

Function prototype: **void spi_init(spi_device_num_t spi_num, spi_work_mode_t work_mode, spi_frame_format_t frame_format, size_t data_bit_length, uint32_t endian)**

Parameter:

Parameter name	**Description**	**Input/Output**
spi_num	SPI number	Input
work_mode	Four modes of polar phase	Input
frame_format	Multi-line mode	Input
data_bit_length	Bit width of the data in a single transmission	Input
endian	Big end 0: little endian 1: Big endian	Input

Return value: No

spi_config_non_standard

Description: Set instruction length, address length, number of waiting clocks, instruction address transmission mode in multi-line mode.

Function prototype: **void spi_init_non_standard(spi_device_num_t spi_num, uint32_t instruction_length, uint32_t address_length, uint32_t wait_cycles, spi_instruction_address_trans_mode_t instruction_address_trans_mode)**

Parameter:

Parameter name	Description	Input/Output
spi_num	SPI number	Input
instruction_length	Number of bits sent	Input
address_length	Number of bits in the sending address	Input
wait_cycles	Number of waiting clocks	Input
instruction_address_trans_mode	Method of instruction address transmission	Input

Return value: No

spi_send_data_standard

Description: SPI standard mode transmits data.

Function prototype: **void spi_send_data_standard(spi_device_num_t spi_num, spi_chip_select_t chip_select, const uint8_t *cmd_buff, size_t cmd_len, const uint8_t *tx_buff, size_t tx_len)**

Parameter:

Parameter name	Description	Input/Output
spi_num	SPI number	Input
chip_select	Chip select signal	Input
cmd_buff	Peripheral instruction address data, if not, set to NULL	Input
cmd_len	Peripheral instruction address data length, if not set to 0	Input
tx_buff	Receive data	Input
tx_len	Receive data length	Input

Return value: No

spi_send_data_standard_dma

Description: Using DMA to transfer data in SPI standard mode.

Function prototype: **void spi_send_data_standard_dma(dmac_channel_number_t channel_num, spi_device_num_t spi_num, spi_chip_select_t chip_select, const uint8_t *cmd_buff, size_t cmd_len, const uint8_t *tx_buff, size_t tx_len)**

Parameter:

Parameter name	Description	Input/Output
channel_num	DMA channel number	Input
spi_num	SPI number	Input
chip_select	Chip select signal	Input
cmd_buff	Peripheral instruction address data, if not, set to NULL	Input
cmd_len	Peripheral instruction address data length, if not set to 0	Input
tx_buff	Send data	Input
tx_len	Send data length	Input

Return value: No

spi_receive_data_standard

Description: Receive data in standard mode.

Function prototype: **void spi_receive_data_standard(spi_device_num_t spi_num, spi_chip_select_t chip_select, const uint8_t *cmd_buff, size_t cmd_len, uint8_t *rx_buff, size_t rx_len)**

Parameter:

Parameter name	Description	Input/Output
spi_num	SPI number	Input
chip_select	Chip select signal	Input
cmd_buff	Peripheral instruction address data, if not, set to NULL	Input
cmd_len	Peripheral instruction address data length, if not set to 0	Input
rx_buff	Receive data	Output
rx_len	Receive data length	Input

Return value: No

spi_receive_data_standard_dma

Description: Receive data through DMA in standard mode.

Function prototype: **void spi_receive_data_standard_dma(dmac_channel_number_t dma_send_channel_num, dmac_channel_number_t dma_receive_channel_num, spi_device_num_t spi_num, spi_chip_select_t chip_select, const uint8_t *cmd_buff, size_t cmd_len, uint8_t *rx_buff, size_t rx_len)**

Parameter:

Parameter name	Description	Input/Output
dma_send_channel_num	DMA channel number used to send instruction address	Input
dma_receive_channel_num	DMA channel number used to receive data	Input
spi_num	SPI number	Input
chip_select	Chip select signal	Input
cmd_buff	Peripheral instruction address data, if not, set to NULL	Input
cmd_len	Peripheral instruction address data length, if not set to 0	Input
rx_buff	Receive data	Output
rx_len	Receive data length	Input

Return value: No

spi_send_data_multiple

Description: Send data in multi-wire mode.

Function prototype: **void spi_send_data_multiple(spi_device_num_t spi_num, spi_chip_select_t chip_select, const uint32_t *cmd_buff, size_t cmd_len, uint8_t *tx_buff, size_t tx_len)**

Parameter:

Parameter name	Description	Input/Output

Parameter name	Description	Input/Output
spi_num	SPI number	Input
chip_select	Chip select signal	Input
cmd_buff	Peripheral instruction address data, if not, set to NULL	Input
cmd_len	Peripheral instruction address data length, if not set to 0	Input
tx_buff	Send data	Input
tx_len	Send data length	Input

Return value: No

spi_send_data_multiple_dma

Description: Send data by DMA in multi-wire mode.

Function prototype: **void spi_send_data_multiple_dma(dmac_channel_number_t channel_num,spi_device_num_t spi_num, spi_chip_select_t chip_select, const uint32_t *cmd_buff, size_t cmd_len, const uint8_t *tx_buff, size_t tx_len)**

Parameter:

Parameter name	Description	Input/Output
channel_num	DMA channel number	Input
spi_num	SPI number	Input
chip_select	Chip select signal	Input
cmd_buff	Peripheral instruction address data, if not, set to NULL	Input
cmd_len	Peripheral instruction address data length, if not set to 0	Input
tx_buff	Send data	Input
tx_len	Send data length	Input

Return value: No

spi_receive_data_multiple

Description: Receive data in multi-line mode.

Function prototype: **void spi_receive_data_multiple(spi_device_num_t spi_num, spi_chip_select_t chip_select, const uint32_t *cmd_buff, size_t cmd_len, uint8_t *rx_buff, size_t rx_len)**

Parameter:

Parameter name	Description	Input/Output
spi_num	SPI number	Input
chip_select	Chip select signal	Input
cmd_buff	Peripheral instruction address data, if not, set to NULL	Input
cmd_len	Peripheral instruction address data length, if not set to 0	Input
rx_buff	Receive data	Output
rx_len	Receive data length	Input

Return value: No

spi_receive_data_multiple_dma

Description: Receive data by DMA in multi-line mode.

Function prototype: **void spi_receive_data_multiple_dma(dmac_channel_number_t dma_send_channel_num, dmac_channel_number_t dma_receive_channel_num, spi_device_num_t spi_num, spi_chip_select_t chip_select, uint32_t const *cmd_buff, size_t cmd_len, uint8_t *rx_buff, size_t rx_len);**

Parameter:

Parameter name	Description	Input/Output
dma_send_channel_num	DMA channel number used to send instruction address	Input
dma_receive_channel_num	DMA channel number used to receive data	Input
spi_num	SPI number	Input
chip_select	Chip select signal	Input
cmd_buff	Peripheral instruction address data, if not, set to NULL	Input
cmd_len	Peripheral instruction address data length, if not set to 0	Input
rx_buff	Receive data	Input
rx_len	Receive data length	Input

Return value: No

spi_fill_data_dma

Description: The same data is always sent through DMA, which can be used to refresh data.

Function prototype: **void spi_fill_data_dma(dmac_channel_number_t channel_num,spi_device_num_t spi_num, spi_chip_select_t chip_select, const uint32_t *tx_buff, size_t tx_len);**

Parameter:

Parameter name	Description	Input/Output
channel_num	DMA channel number	Input
spi_num	SPI number	Input
chip_select	Chip select signal	Input
tx_buff	Send data, just send tx_buff	Input
tx_len	Receive data length	Input

Return value: No

spi_send_data_normal_dma

Description: Send data by DMA.

Function prototype: **void spi_send_data_normal_dma(dmac_channel_number_t channel_num, spi_device_num_t spi_num, spi_chip_select_t chip_select, const void *tx_buff, size_t tx_len, spi_transfer_width_t spi_transfer_width)**

Parameter:

Parameter name	Description	Input/Output
channel_num	DMA channel number	Input
spi_num	SPI number	Input
chip_select	Chip select signal	Input
tx_buff	Send data, just send tx_buff	Input
tx_len	Receive data length	Input
spi_transfer_width	Receive data bit wight	Input

Return value: No

spi_handle_data_dma

Description: SPI transfer data by DMA

Function prototype: **void spi_handle_data_dma(spi_device_num_t spi_num, spi_chip_select_t chip_select, spi_data_t data, plic_interrupt_t *cb)**

Parameter:

Parameter name	Description	Input/Output
spi_num	SPI number	Input
data	SPI data related parameters	Input
cb	DMA interrupt callback function, if it is set to NULL, it is in blocking mode, and the function exits until the transmission is completed	Input

Return value: No

spi_slave_config

Description: SPI slave configuration. The SPI slave of K210 is a data line multiplexed with MOSI and MISO functions.

Function prototype: **void spi_slave_config(uint8_t int_pin, uint8_t ready_pin, dmac_channel_number_t dmac_channel, size_t data_bit_length, uint8_t *data, uint32_t len, spi_slave_receive_callback_t callback);**

Parameter:

Parameter name	Description	Input/Output
spi_num	SPI number	Input
ready_pin	When the spi slave is ready, it will pull up this	Input

Parameter name	Description	Input/Output
	pin level, and the master will trigger an interrupt.	
dmac_channel	DMA channel used when transmitting data	Input
data_bit_length	Set the data width when the spi transmits data	Input
data	Spi slave receives data buffer	Input
len	Size of spi slave receives data buffer	Input
callback	Callback function after spi slave receives data	Input

Return value: No

spi_slave_dual_config

Description: SPI slave configuration.

void spi_slave_dual_config(uint8_t int_pin,
uint8_t ready_pin,
uint8_t mosi_pin,
uint8_t miso_pin,
dmac_channel_number_t dmac_channel,
size_t data_bit_length,
uint8_t *data,
uint32_t len,
spi_slave_receive_callback_t callback);

Parameter:

Parameter name	Description	Input/Output
spi_num	SPI number	Input
ready_pin	When the spi slave is ready, it will pull up this pin level, and the master will trigger an interrupt.	Input
mosi_pin	IO number of MOSI	Input
miso_pin	IO number of MISO	Input
dmac_channel	DMA channel used when transmitting data	Input
data_bit_length	Set the data width when the spi transmits data	Input
data	spi slave receives data buffer	Input
len	Size of spi slave receives data buffer	Input
callback	Callback function after spi slave receives data	Input

Return value: No

Eg:

```
/*SPI0 works in MODE0 mode    Standard SPI mode    Sends 8 bits of data at a time*/
spi_init(SPI_DEVICE_0, SPI_WORK_MODE_0, SPI_FF_STANDARD, 8, 0);
uint8_t cmd[4];
cmd[0] = 0x06;
```

```
cmd[1] = 0x01;
cmd[2] = 0x02;
cmd[3] = 0x04;
uint8_t data_buf[4] = {0,1,2,3};

/*SPI0 uses chip select 0 to send instruction 0x06 to address 0x010204 to send four bytes of data 0, 1, 2, 3*/
spi_send_data_standard(SPI_DEVICE_0, SPI_CHIP_SELECT_0, cmd, 4, data_buf, 4);

/* SPI0 uses chip select 0 to send instructions 0x06 address 0x010204 to receive 4 bytes of data */
spi_receive_data_standard(SPI_DEVICE_0, SPI_CHIP_SELECT_0, cmd, 4, data_buf, 4);

/* SPI0 works in MODE0 mode, four-wire SPI mode, sends 8-bit data in a single time */
spi_init(SPI_DEVICE_0, SPI_WORK_MODE_0, SPI_FF_QUAD, 8, 0);

/* 8-bit instruction length 32-bit address length After sending the instruction address, wait for 4 clk, the instruction is sent through the standard SPI method, and the address is sent through the four-wire method */
spi_init_non_standard(SPI_DEVICE_0, 8, 32, 4, SPI_AITM_ADDR_STANDARD);
uint32 cmd[2];
cmd[0] = 0x06;
cmd[1] = 0x010204;
uint8_t data_buf[4] = {0,1,2,3};

/* SPI0 uses chip select 0 to send instruction 0x06 to address 0x010204 to send four bytes of data 0, 1, 2, 3 */
spi_send_data_multiple(SPI_DEVICE_0, SPI_CHIP_SELECT_0, cmd, 2, data_buf, 4);

/* SPI0 uses chip select 0 to send instructions 0x06 address 0x010204 to receive 4 bytes of data */
spi_receive_data_multiple(SPI_DEVICE_0, SPI_CHIP_SELECT_0, cmd, 2, data_buf, 4);

/* SPI0 works in MODE2 mode, 8-wire SPI mode, sends 32-bit data at a time */
spi_init(SPI_DEVICE_0, SPI_WORK_MODE_2, SPI_FF_OCTAL, 32, 0);

/* No instruction 32-bit address length Wait for 0 clk after sending the instruction address, the instruction address is sent through 8-wire */
spi_init_non_standard(SPI_DEVICE_0, 0, 32, 0, SPI_AITM_AS_FRAME_FORMAT);
uint32_t data_buf[256] = {0};
/* Use DMA channel 0 chip select 0 to send 256 int data*/
spi_send_data_normal_dma(DMAC_CHANNEL0, SPI_DEVICE_0, SPI_CHIP_SELECT_0, data_buf, 256, SPI_TRANS_INT);
```

```
uint32_t data = 0x55AA55AA;

/* Use DMA channel 0 chip select 0 to continuously send 256 0x55AA55AA*/
spi_fill_data_dma(DMAC_CHANNEL0, SPI_DEVICE_0, SPI_CHIP_SELECT_0,&data, 256);
```

spi_set_clk_rate

Description: Set SPI clock frequency

Function prototype: **uint32_t spi_set_clk_rate(spi_device_num_t spi_num, uint32_t spi_clk)**

Parameter:

Parameter name	Description	Input/Output
spi_num	SPI number	Input
spi_clk	The clock frequency of the target SPI device	Input

Return value: The clock frequency of the SPI device after setting

Data type

The related data types and data structure are defined as follows:

- spi_device_num_t: SPI number.
- spi_mode_t: SPI mode.
- spi_frame_format_t: SPI frame format.
- spi_instruction_address_trans_mode_t: SPI instruction and address transmission mode.
- spi_data_t: Parameters associated with data when using DMA transfers.
- spi_transfer_mode_t: SPI transmission mode.

spi_device_num_t

Description: SPI number

Define

```
typedef enum _spi_device_num
{
    SPI_DEVICE_0,
    SPI_DEVICE_1,
    SPI_DEVICE_2,
    SPI_DEVICE_3,
    SPI_DEVICE_MAX,
} spi_device_num_t;
```

member

member name	Description
SPI_DEVICE_0	SPI 0 master device

member name	Description
SPI_DEVICE_1	SPI 1 master device
SPI_DEVICE_2	SPI 2 slave device
SPI_DEVICE_3	SPI 3 master device

spi_mode_t

Description: SPI mode.

Define

```
typedef enum _spi_mode
{
    SPI_WORK_MODE_0,
    SPI_WORK_MODE_1,
    SPI_WORK_MODE_2,
    SPI_WORK_MODE_3,
} spi_mode_t;
```

member

member name	Description
SPI_WORK_MODE_0	SPI mode 0
SPI_WORK_MODE_1	SPI mode 1
SPI_WORK_MODE_2	SPI mode 2
SPI_WORK_MODE_3	SPI mode 3

spi_frame_format_t

Description: SPI frame format.

Define

```
typedef enum _spi_frame_format
{
    SPI_FF_STANDARD,
    SPI_FF_DUAL,
    SPI_FF_QUAD,
    SPI_FF_OCTAL
} spi_frame_format_t;
```

member

member name	Description
SPI_FF_STANDARD	standard
SPI_FF_DUAL	dual-line
SPI_FF_QUAD	4-line
SPI_FF_OCTAL	8-line (SPI3 not supported)

spi_instruction_address_trans_mode_t

Description: SPI instruction and address transmission mode.

Define

```
typedef enum _spi_instruction_address_trans_mode
{
    SPI_AITM_STANDARD,
    SPI_AITM_ADDR_STANDARD,
    SPI_AITM_AS_FRAME_FORMAT
} spi_instruction_address_trans_mode_t;
```

member

member name	Description
SPI_AITM_STANDARD	Standard frame format
SPI_AITM_ADDR_STANDARD	The command uses the configured value, and the address uses the standard frame format
SPI_AITM_AS_FRAME_FORMAT	Both use configured values

spi_data_t

Description: Parameters associated with data when using DMA transfers.

Define

```
typedef struct _spi_data_t
{
    dmac_channel_number_t tx_channel;
    dmac_channel_number_t rx_channel;
    uint32_t *tx_buf;
    size_t tx_len;
    uint32_t *rx_buf;
    size_t rx_len;
    spi_transfer_mode_t transfer_mode;
    bool fill_mode;
} spi_data_t;
```

member

member name	Description
tx_channel	DMA channel number used when sending
rx_channel	DMA channel number used when receiving
tx_buf	Data sent
tx_len	Length of data sent
rx_buf	Data received
rx_len	Length of data received
transfer_mode	Transmission mode, send or receive
fill_mode	Choose whether to transfer data in padding mode. In this case the source address of DMA transfer will not increase

spi_transfer_mode_t

Description: SPI transmission mode.

Define

```
typedef enum _spi_transfer_mode
{
    SPI_TMOD_TRANS_RECV,
    SPI_TMOD_TRANS,
    SPI_TMOD_RECV,
    SPI_TMOD_EEROM
} spi_transfer_mode_t;
```

member

member name	Description
SPI_TMOD_TRANS_RECV	Full duplex
SPI_TMOD_TRANS	Just send
SPI_TMOD_RECV	Just receive
SPI_TMOD_EEROM	First send, then receive