
SPI Device Application Notes

RT-THREAD Document Center

Copyright ©2019 Shanghai Ruisaide Electronic Technology Co., Ltd.



WWW.RT-THREAD.ORG

Friday 28th September, 2018

[Table of contents](#)

Table of contents	i
1 Purpose and structure of this paper	1
1.1 Purpose and Background of this Paper	1
1.2 Structure of this paper	1
2 Introduction to SPI device driver framework.	1
3 Run the example code.	2
3.1 Sample Code Software and Hardware Resources.	2
3.2 Configure the project.	4
3.3 Add sample code.	6
4 Detailed explanation of the use of SPI device driver interface.	7
4.1 Mounting SPI devices to the bus.	8
4.2 Configure SPI mode.	9
4.3 Data Transmission.	11
4.3.1. rt_spi_transfer_message()	11
4.3.2. rt_spi_send()	12
4.3.3. rt_spi_send_then_send()	14
4.3.4. rt_spi_send_then_recv()	15
4.4 SPI device driver application	16
5 References.	17
5.1 All APIs related to this article.	17
5.2 Detailed explanation of other core APIs	18
5.2.1. rt_spi_take_bus()	18
5.2.2. rt_spi_release_bus()	18

5.2.3. rt_spi_take() 19

5.2.4. rt_spi_release() 19

5.2.5. rt_spi_message_append(). 20

This application note uses driving an OLED display with an SPI interface as an example. It explains how to add the SPI device driver framework and underlying hardware drivers, and develop applications using the SPI device driver interface. It also provides code examples verified on the Atom STM32F4 Explorer development board.

1 Purpose and structure of this paper

1.1 Purpose and Background of this Paper

The Serial Peripheral Interface Bus (SPI) is a synchronous serial communication interface specification for short-range communication, primarily used in microcontroller systems. SPI is primarily used in EEPROMs, Flash memories, real-time clocks, analog-to-digital converters, digital signal processors, and digital signal decoders. Occupying four or three pins on the chip, SPI is simple and easy to use, leading to its integration in an increasing number of chips.

To facilitate application layer program development, RT-Thread introduces the SPI device driver framework. This article explains how to Use RT-Thread SPI device driver.

1.2 Structure of this paper

This article first briefly introduces the RT-Thread SPI device driver framework, then runs the SPI device driver sample code on the Zhengdian Atom STM32F4 Explorer development board. Finally, it details how to use the SPI device driver framework interface and parameter values.

2 Introduction to SPI Device Driver Framework

The RT-Thread SPI device driver framework virtualizes the MCU's SPI hardware controller into an SPI bus (SPI BUS#n). Multiple SPI devices (SPI BUS#0 CSm) can be connected to the bus, but each SPI device can only be connected to one SPI bus. Currently, RT-Thread has implemented drivers for many common SPI devices, such as SD cards, various Flash memory series, and the ENC28J60 Ethernet module. The following figure shows the hierarchical structure of the SPI device driver framework.

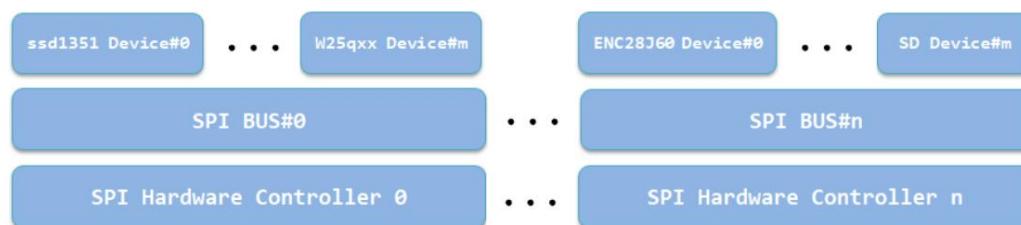


Figure 1: SPI Device driver framework hierarchy

Based on the previous introduction, users have a general understanding of the RT-Thread SPI device driver framework. So how do users use What about using the SPI device driver framework?

3. Run the sample code

This chapter is based on the Atom Explorer STM32F4 development board and SPI sample code, and gives the RT-Thread SPI How to use the device driver framework.

3.1 Sample Code Software and Hardware Resources

1. [RT-Thread Source Code](#)
2. [ENV Tool](#)
3. [SPI device driver sample code](#)
4. [Zhengdian Atom STM32F4 Explorer Development Board](#)
5. 1.5-inch color OLED display (SSD1351 controller)
6. MDK5

The MCU of the Atom Explorer STM32F4 development board is STM32F407ZGT6. This example uses USB to Serial port (USART1) sends data and supplies power, and uses SEGGER J-LINK to connect to JTAG debugging. STM32F4 has Multiple hardware SPI controllers, this example uses SPI1. Color OLED display onboard SSD1351 controller, resolution 128*128.

The pin connections between STM32F4 and OLED display are shown in the following table:

STM32 pins	OLED display pins	illustrate
PA5	D0	SPI1 SCK, clock
PA6		SPI1 MISO, not used
PA7	D1	SPI1 MOSI, master output, slave Machine Input
PC6	D/C	GPIO, output, command 0/data 1 Select
PC7	RES	GPIO, output, reset, low power Flat and effective
PC8	CS	GPIO, output, chip select, low power Flat and effective
3.3V	VCC	powered by
GND	GND	Grounding

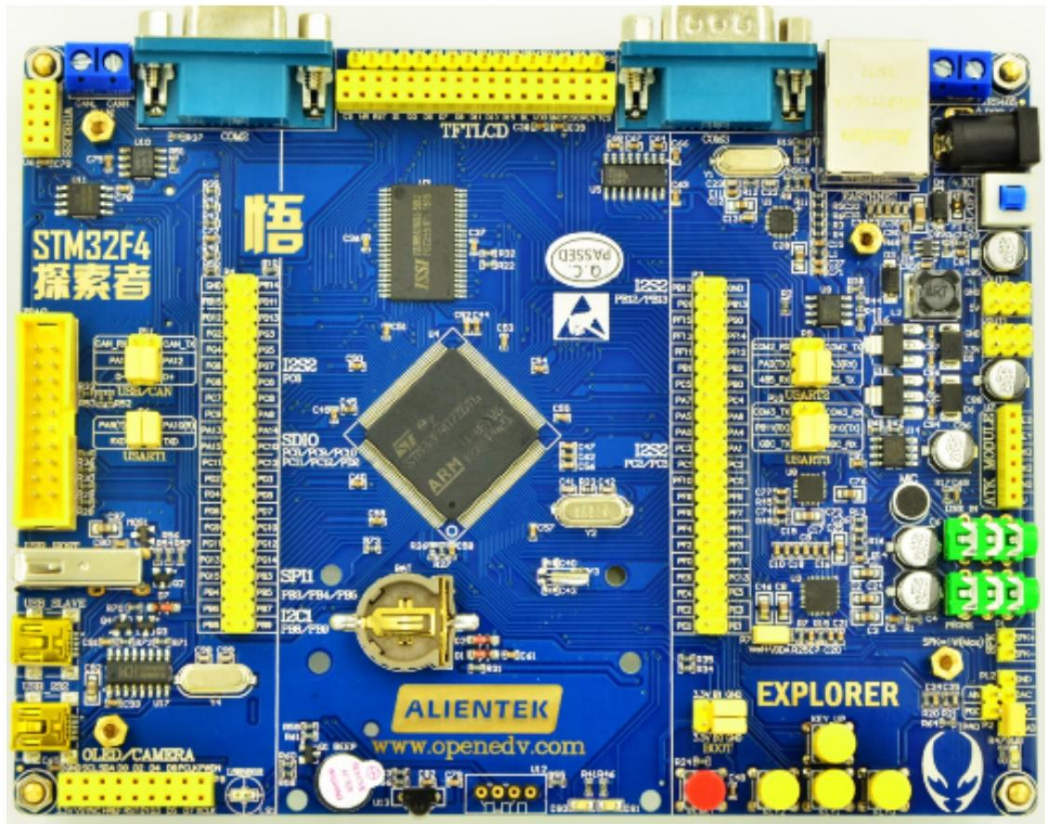


Figure 2: Zhengdian Atom Development Board



Figure 3: color OLED Display

The SPI device driver sample code includes three files: `app.c`, `drv_ssd1351.c`, and `drv_ssd1351.h`.

`drv_ssd1351.c` is the OLED display driver file, which contains the initialization of the SPI device `ssd1351`,

Mount to the system and control the OLED display through commands.

Therefore, the codes are not limited to a specific hardware platform and users can easily port them to other platforms.

3.2 Configuration Project

Use menuconfig to configure the project: Use the `cd` command in the `env` tool command line to enter `rt-thread/bsp/stm32f4xx-HAL` directory, and then enter the `menuconfig` command to enter the configuration interface.

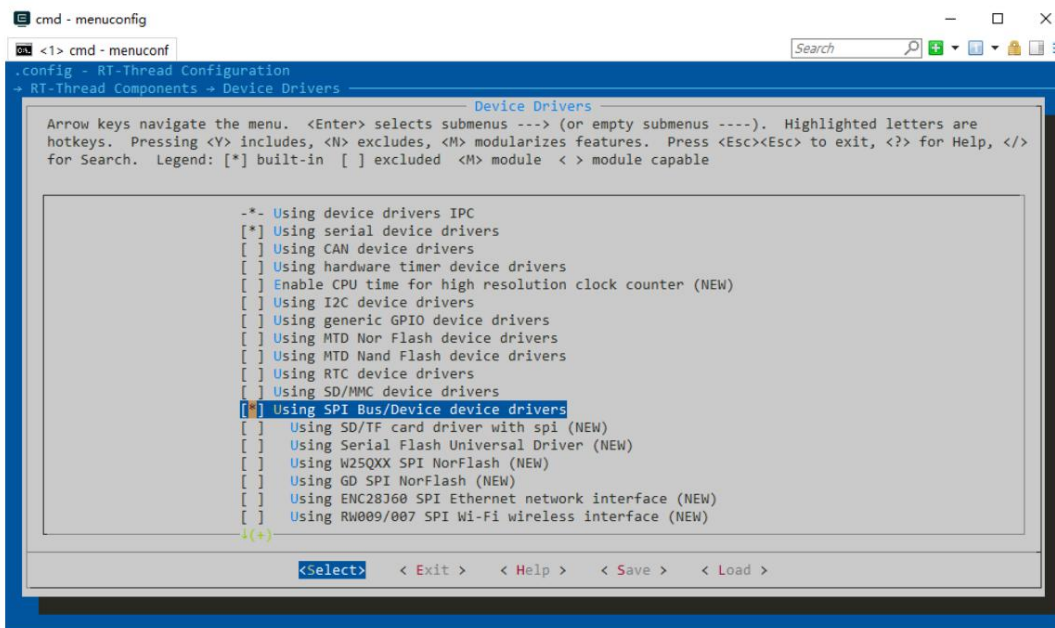


Figure 4: use `menuconfig` Open SPI

- Modify the project chip model: Change Device type to STM32F407ZG.
- Configure the shell to use serial port 1: Select Using UART1 and go to RT-Thread Kernel —> Kernel Device
In the Object menu, change the device name for console to `uart1`.
- Enable the SPI bus and device driver and register the SPI bus to the system: Enter RT-Thread Components
—> Device Drivers menu, select Using SPI Bus/Device drivers, RT-Thread
The Configuration interface will select Using SPI1 by default, and the SPI1 bus device will be registered with the operating system.
- Enable GPIO driver: Go to RT-Thread Components —> Device Drivers menu and select Using
Generic GPIO device drivers. OLED screens require 2 additional GPIOs for DC and RES signals.
The SPI bus driver also needs to operate the chip select pins and needs to call the system's GPIO driver interface.

Generate a new project and modify debugging options: Exit the menuconfig configuration interface and save the configuration, in the ENV command line Enter the `scons --target=mdk5 -s` command to generate an mdk5 project. The new project is named project. Use MDK5 to open Project, change the debugging option to J-LINK.

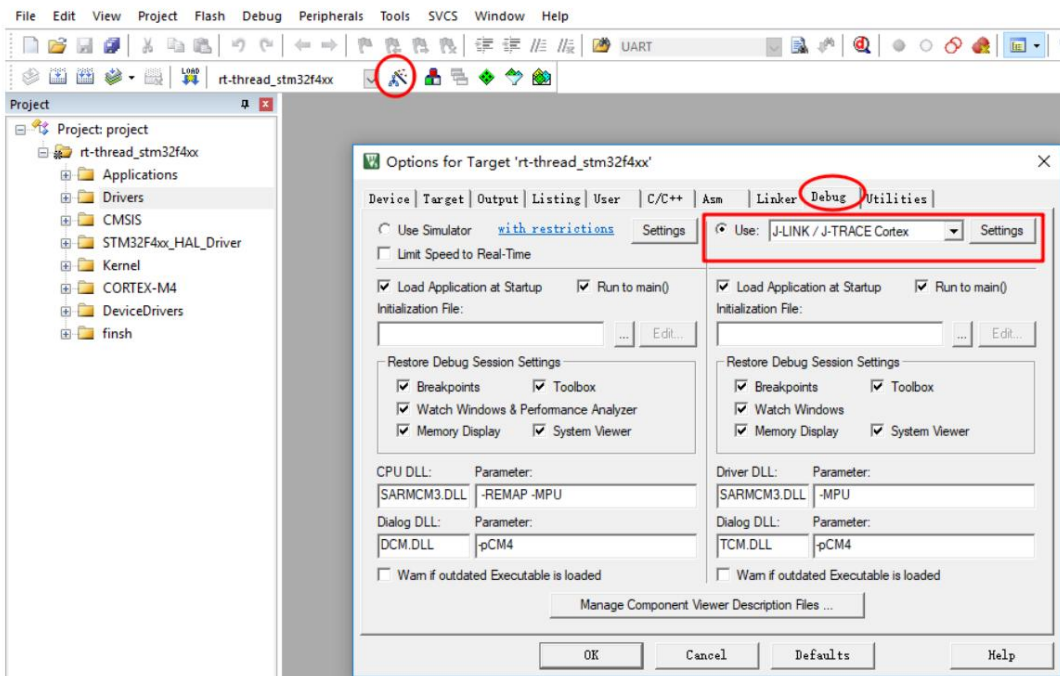


Figure 5: Modify debugging options

Use the `list_device` command to view the SPI bus: After adding the SPI underlying hardware driver correctly, open the PuTTY terminal (open the corresponding port and configure the baud rate to 115200) and use the `list_device` command to see the SPI bus.

To the UART device and PIN device we use.

COM12 - PuTTY

```

\ | /
- RT -      Thread Operating System
/ | \      3.0.3 build Mar 28 2018
2006 - 2018 Copyright by rt-thread team
msh >list_device
device      type          ref count
-----
spi1        SPI Bus        0
uart1       Character Device 2
pin         Miscellaneous Device 0
msh >

```

Figure 6: use `list_device` Command to view system devices

3.3 Add sample code

Copy `app.c` in the SPI device driver sample code to `/rt-thread/bsp/stm32f4xx-HAL/`

`Applications` directory . Copy `drv_ssd1351.c` and `drv_ssd1351.h` to the `/rt-thread/bsp/stm32f4xx- HAL/drivers` directory and add them to the corresponding groups in the project, as shown in the figure:

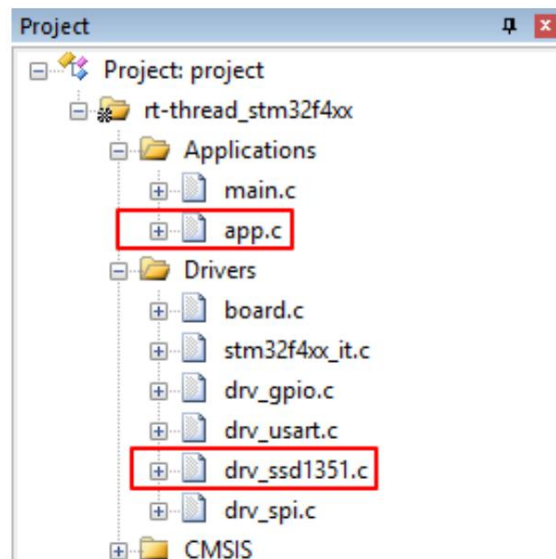


Figure 7: Add sample code to the project

Call `app_init()` in `main.c`, `app_init()` will create an oled thread, and the thread will display the rainbow colors in a loop.

Color pattern and square color pattern.

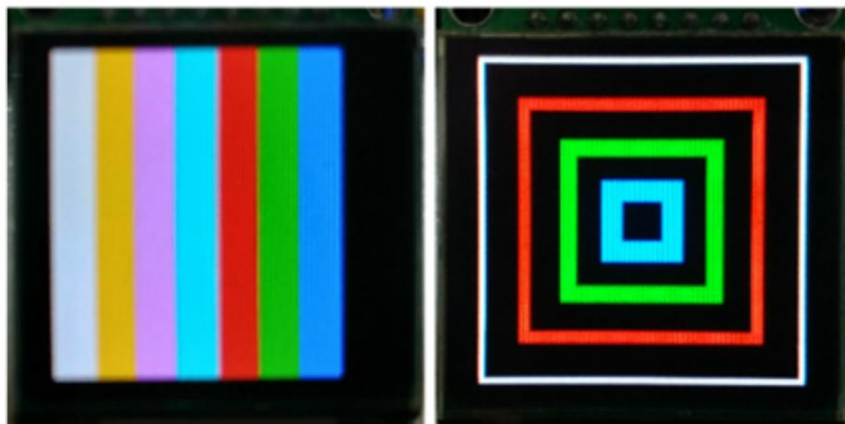


Figure 8: Experimental phenomenon

The source code of `main.c` calling the test code is as follows:

```
#include <rtthread.h>
#include <board.h>
```

```

extern int app_init(void);

int main(void)
{
    /* user app entry */

    app_init();

    return 0;
}

```

```

\ | /
- RT -   Thread Operating System
/ | \   3.0.3 build Apr 3 2018
2006 - 2018 Copyright by rt-thread team
msh >list_device
device          type          ref count
-----
spi10  SPI Device          0
spi1   SPI Bus             0
uart1  Character Device     2
pin    Miscellaneous Device 0
msh >

```

Figure 9: Use `list_device` Command View `SPI` Device Driver

4 Detailed explanation of SPI device driver interface usage

Following the steps above, I believe readers can quickly run the RT-Thread SPI device driver. So how to
How to develop applications using SPI device driver interface?

The RT-Thread SPI device driver usage process is as follows:

1. Define the SPI device object and call `rt_spi_bus_attach_device()` to attach the SPI device to the SPI bus.
2. Call `rt_spi_configure()` to configure the SPI bus mode.
3. Use `rt_spi_send()` and other related data transmission interfaces to transmit data.

Next, this chapter will explain in detail the main SPI device driver interfaces used in the sample code.

4.1 Mount SPI device to bus

After the user defines the SPI device object, he can call this function to mount the SPI device to the SPI bus.

Function prototype:

```
rt_err_t rt_spi_bus_attach_device(struct rt_spi_device *device,
                                const char             *name,
                                const char             *bus_name,
                                void                   *user_data)
```

parameter	describe
device	SPI device handle
name	SPI device name
bus_name	SPI bus name
user_data	User data pointer

Function return: Returns RT_EOK if successful, otherwise returns an error code.

This function is used to mount an SPI device to the specified SPI bus, register the SPI device with the kernel, and pass user_data Save to SPI device device.

Notice

1. The user first needs to define the SPI device object device
2. The recommended naming principle for SPI bus is spix and the naming principle for SPI device is spixy. For example, spi10 in this example means hanging Device number 0 on the spi1 bus.
3. The SPI bus name can be checked by entering the list_device command in the msh shell to determine the SPI device to be mounted. SPI bus.
4. user_data is usually the CS pin pointer of the SPI device. The SPI controller will operate this pin when transmitting data. Pin for chip select.

The sample code in this article drives the `rt_hw_ssd1351_config()` in `drv_ssd1351.c` to mount the `ssd1351` device to The SPI bus source code is as follows:

```
#define SPI_BUS_NAME           "spi1" /* SPI bus name*/
#define SPI_SSD1351_DEVICE_NAME "spi10" /* SPI device name*/

... ..
```

```
static struct rt_spi_device spi_dev_ssd1351; /* SPI device ssd1351 object*/ static struct stm32_hw_spi_cs
spi_cs; /* SPI device CS chip select pin*/

... ..

static int rt_hw_ssd1351_config(void) {

    rt_err_t res;

    /* oled use PC8 as CS */
    spi_cs.pin = CS_PIN;
    rt_pin_mode(spi_cs.pin, PIN_MODE_OUTPUT); /* Set the chip select pin mode to input
    out*/

    res = rt_spi_bus_attach_device(&spi_dev_ssd1351,
        SPI_SSD1351_DEVICE_NAME, SPI_BUS_NAME, (void*)&spi_cs); if (res !=
    RT_EOK) {

        OLED_TRACE("rt_spi_bus_attach_device\n\n"); return res;

    }

    ... ..

}
```

4.2 Configuring SPI Mode

After mounting the SPI device to the SPI bus, it is usually necessary to configure the SPI bus to meet the clock and data width requirements of different devices.

Set SPI mode and frequency parameters.

The SPI slave mode determines the master mode, so the SPI master mode must be the same as the slave mode.

Can communicate normally.

Function prototype:

```
rt_err_t rt_spi_configure(struct rt_spi_device *device,
                        struct rt_spi_configuration *cfg)
```

parameter	describe
device	SPI device handle
cfg	SPI transmission configuration parameter pointer

Function returns: returns RT_EOK.

This function will save the mode parameters pointed to by cfg to the device. When the device calls the data transfer function, it will use

Use this configuration information.

The prototype of struct rt_spi_configuration is as follows:

```
struct rt_spi_configuration
{
    rt_uint8_t mode;           //spi mode
    rt_uint8_t data_width; //Data width, can be 8 bits, 16 bits, 32 bits
    rt_uint16_t reserved; //Reserved
    rt_uint32_t max_hz;       //Maximum frequency
};
```

Mode: Use the macro definitions in spi.h, including MSB/LSB, master-slave mode, timing mode, etc.

The combination is as follows.

```
/* Set the data transmission order to be MSB first or LSB first*/
#define RT_SPI_LSB          (0<<2)           /* bit[2]: 0-LSB */
#define RT_SPI_MSB          (1<<2)           /* bit[2]: 1-MSB */

/* Set SPI master/slave mode*/
#define RT_SPI_MASTER (0<<3)                 /* SPI master
device */
#define RT_SPI_SLAVE */ (1<<3)              /* SPI slave device

/* Set clock polarity and clock phase */
#define RT_SPI_MODE_0 (0 | 0) 0 */          /* CPOL = 0, CPHA =
1 */
#define RT_SPI_MODE_1 (0 | RT_SPI_CPHA)     /* CPOL = 0, CPHA =
0 */
#define RT_SPI_MODE_2 (RT_SPI_CPOL | 0)     /* CPOL = 1, CPHA =
1 */
#define RT_SPI_MODE_3 (RT_SPI_CPOL | RT_SPI_CPHA) /* CPOL = 1, CPHA =
1 */

#define RT_SPI_CS_HIGH (1<<4)               /* Chipselect
active high */
#define RT_SPI_NO_CS      (1<<5)            /* No chipselect */
#define RT_SPI_3WIRE */ (1<<6)             /* SI/SO pin shared

#define RT_SPI_READY to (1<<7)             /* Slave pulls low
pause */
```

Data width/data_width: According to the data width format that can be sent and received by the SPI master device and SPI slave device Set to 8, 16, or 32 bits.

Maximum frequency/max_hz: Set the baud rate of data transmission, which is also set according to the baud rate range of the SPI master device and SPI slave device.

Notice

After mounting the SPI device to the SPI bus, this function must be used to configure the transmission parameters of the SPI device.

The source code for configuring SPI transmission parameters in `rt_hw_ssd1351_config()` in the underlying driver `drv_ssd1351.c` of this article's sample code is as follows:

```
static int rt_hw_ssd1351_config(void) {

    ... ..

    /* config spi */ {

        struct rt_spi_configuration cfg; cfg.data_width
        = 8; cfg.mode =
        RT_SPI_MASTER | RT_SPI_MODE_0 | RT_SPI_MSB;
        cfg.max_hz = 20 * 1000 * 1000; /* 20M, SPI max 42MHz, ssd1351 4-wire
        spi */

        rt_spi_configure(&spi_dev_ssd1351, &cfg);
    }

    ... ..
}
```

4.3 Data Transmission

After the SPI device is mounted on the SPI bus and the relevant SPI transmission parameters are configured, a series of SPI device driver data transmission functions provided by RT-Thread can be called.

4.3.1. `rt_spi_transfer_message()`

Function prototype:

```
struct rt_spi_message *rt_spi_transfer_message(struct rt_spi_device *
    device,

                                                struct rt_spi_message *message)
```

parameter	describe
device	SPI device handle
message	Message pointer

Function returns: Returns RT_NULL if the message is sent successfully, otherwise returns a pointer to the remaining unsent message

This function can transmit a series of messages. Users can flexibly set the values of the parameters of the message structure.

And the data transmission method can be easily controlled.

The prototype of struct rt_spi_message is as follows:

```

struct rt_spi_message
{
    const void *send_buf; void /* Send buffer pointer*/
    *recv_buf; /* Receive buffer pointer*/
    rt_size_t length; struct /* Number of bytes of data sent/received*/
    rt_spi_message *next; /* Pointer to the next message to be sent*/

    unsigned cs_take : 1; /* If the value is 1, the CS pin is pulled low, and if the value is 0, the pin is not changed.
        status*/
    unsigned cs_release : 1; status*/ /* If the value is 1, the CS pin is pulled high, and if the value is 0, the pin is not changed.

};

```

SPI is a full-duplex communication bus that sends one byte of data while receiving one byte of data. The parameter length

The number of bytes of data sent or received during a data transmission. The data sent is the buffer data pointed to by send_buf.

The received data is saved in the buffer pointed to by recv_buf. If the received data is ignored, the value of recv_buf is NULL.

If you ignore the sent data and only receive data, the send_buf value is NULL.

The parameter next is a pointer to the next message to be sent. If only one message is sent, the pointer value is set to NULL.

4.3.2. rt_spi_send()

Function prototype:

```

rt_size_t rt_spi_send(struct rt_spi_device *device,
                    const void *send_buf,
                    rt_size_t length)

```

parameter	describe
device	SPI device handle
send_buf	Send buffer pointer
length	The number of bytes of data sent

Function returns: number of data bytes sent successfully

Call this function to send the data in the buffer pointed to by send_buf and ignore the received data.

This function is equivalent to calling `rt_spi_transfer_message()` to transfer a message. The message parameters are configured as follows:

```
struct rt_spi_message msg;

msg.send_buf = send_buf;
msg.recv_buf = RT_NULL;
msg.length    = length;
msg.cs_take    = 1;
msg.cs_release = 1;
msg.next      = RT_NULL;
```

Notice

Calling this function will send data once. Chip select starts when data starts to be sent and ends when the function returns.

The underlying driver `drv_ssd1351.c` in this sample code calls `rt_spi_send()` to send instructions and data to the SSD1351.

The function source code is as follows:

```
rt_err_t ssd1351_write_cmd(const rt_uint8_t cmd)
{
    rt_size_t len;

    rt_pin_write(DC_PIN, PIN_LOW);          /* Command low level */

    len = rt_spi_send(&spi_dev_ssd1351, &cmd, 1);

    if (len != 1)
    {
        OLED_TRACE("ssd1351_write_cmd error. %d\r\n", len);
        return -RT_ERROR;
    }
    else
    {

```



```
        return RT_EOK;
    }

}

rt_err_t ssd1351_write_data(const rt_uint8_t data)
{
    rt_size_t len;

    rt_pin_write(DC_PIN, PIN_HIGH);          /*Data high level*/

    len = rt_spi_send(&spi_dev_ssd1351, &data, 1);

    if (len != 1)
    {
        OLED_TRACE("ssd1351_write_data error. %d\r\n",len);
        return -RT_ERROR;
    }
    else
    {
        return RT_EOK;
    }
}
```

4.3.3. rt_spi_send_then_send()

Function prototype:

```
rt_err_t rt_spi_send_then_send(struct rt_spi_device *device,
                                const void          *send_buf1,
                                rt_size_t           send_length1,
                                const void          *send_buf2,
                                rt_size_t           send_length2);
```

parameter	describe
device	SPI bus device handle
send_buf1	Send buffer 1 data pointer
send_length1	Number of bytes of data in the send buffer
send_buf2	Send buffer 2 data pointer
send_length2	Number of data bytes in send buffer 2

Function return: Returns RT_EOK if successful, otherwise returns an error code

This function can send data from 2 buffers continuously and ignore the received data.

Start, and the chip select ends after sending send_buf2.

This function is equivalent to calling rt_spi_transfer_message() to transfer two messages. The message parameters are configured as follows:

```
struct rt_spi_message msg1,msg2;

msg1.send_buf = send_buf1;
msg1.recv_buf = RT_NULL;
msg1.length      = send_length1;
msg1.cs_take      = 1;
msg1.cs_release = 0;
msg1.next = &msg2;

msg2.send_buf = send_buf2;
msg2.recv_buf = RT_NULL;
msg2.length      = send_length2;
msg2.cs_take      = 0;
msg2.cs_release = 1;
msg2.next        = RT_NULL;
```

4.3.4. rt_spi_send_then_rcv()

Function prototype:

```
rt_err_t rt_spi_send_then_rcv(struct rt_spi_device *device,
                             const void *send_buf,
                             rt_size_t send_length,
                             void *recv_buf,
                             rt_size_t rcv_length);
```

parameter	describe
device	SPI bus device handle
send_buf	Send buffer data pointer
send_length	Number of bytes of data in the send buffer
recv_buf	Receive buffer data pointer, SPI is full-duplex, supports Support simultaneous sending and receiving
length	Number of data bytes in the receive buffer

Function return: Returns RT_EOK if successful, otherwise returns an error code

This function starts chip selection when sending the first message send_buf, ignores the received data, and then sends the second

The data sent at this time is empty, and the received data is saved in rcv_buf. The chip selection ends when the function returns.

This function is equivalent to calling rt_spi_transfer_message() to transfer two messages. The message parameters are configured as follows:

```
struct rt_spi_message msg1,msg2;

msg1.send_buf = send_buf;
msg1.rcv_buf = RT_NULL;
msg1.length = send_length;
msg1.cs_take      = 1;
msg1.cs_release = 0;
msg1.next = &msg2;

msg2.send_buf = RT_NULL;
msg2.rcv_buf = rcv_buf;
msg2.length      = rcv_length;
msg2.cs_take      = 0;
msg2.cs_release = 1;
msg2.next        = RT_NULL;
```

rt_spi_sendrcv8() and rt_spi_sendrcv16() functions are encapsulations of this function.

() sends one byte of data and receives one byte of data at the same time, rt_spi_sendrcv16() sends 2 bytes of data and receives one byte of data at the same time.

Received 2 bytes of data.

4.4 SPI device driver application

This article uses SSD1351 to display image information. First, we need to determine the row and column starting addresses of the information on the display.

Call ssd1351_write_cmd() to send commands to SSD1351 , and call ssd1351_write_data() to send data to SSD1351.

Send data, the source code is as follows:

```
void set_column_address(rt_uint8_t start_address, rt_uint8_t end_address)
{
    ssd1351_write_cmd(0x15);                // Set Column Address
    ssd1351_write_data(start_address);      // Default => 0x00 (Start
    Address
    ssd1351_write_data(end_address);        // Default => 0x7F (End
    Address
}
void set_row_address(rt_uint8_t start_address, rt_uint8_t end_address)
{
    ssd1351_write_cmd(0x75);                // Set Row Address
```

```

    ssd1351_write_data(start_address);           // Default => 0x00 (Start
        Address
    ssd1351_write_data(end_address);             // Default => 0x7F (End
        Address
}

```

5References

5.1 All related APIs in this article

SPI device driver framework all APIs	header files
rt_spi_bus_register()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_bus_attach_device()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_configure()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_send_then_send()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_send_then_recv()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_transfer()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_transfer_message()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_take_bus()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_release_bus()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_take()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_release()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_recv()	rt-thread/components/drivers/include/drivers/spi.h
rt_spi_send()	rt-thread/components/drivers/include/drivers/spi.h

SPI device driver framework all APIs	header files
rt_spi_sendrecv8()	rt-thread/components/drivers/include/drivers/ spi.h
rt_spi_sendrecv16()	rt-thread/components/drivers/include/drivers/ spi.h
rt_spi_message_append()	rt-thread/components/drivers/include/drivers/ spi.h
Sample code related API	Location
ssd1351_write_cmd()	drv_ssd1351.c
ssd1351_write_data()	drv_ssd1351.c
rt_hw_ssd1351_config()	drv_ssd1351.c

5.2 Detailed explanation of other core APIs

5.2.1. rt_spi_take_bus()

Function prototype:

```
rt_err_t rt_spi_take_bus(struct rt_spi_device *device);
```

parameter	describe
device	SPI device handle

Function return: Returns RT_EOK if successful, otherwise returns an error code

The device calling this function can occupy the SPI bus resources, and other devices cannot use the SPI bus.

5.2.2. rt_spi_release_bus()

Function prototype:

```
rt_err_t rt_spi_release_bus(struct rt_spi_device *device);
```

parameter	describe
device	SPI device handle

Function return: Returns RT_EOK if successful, otherwise returns an error code

After the device calls `rt_spi_take_bus()` to obtain bus resources, it needs to call this function to release the SPI bus resources.

Other devices can access the SPI bus.

5.2.3. `rt_spi_take()`

Function prototype:

```
rt_err_t rt_spi_take(struct rt_spi_device *device);
```

parameter	describe
device	SPI device handle

Function returns: Return 0

Calling this function starts chip selection.

5.2.4. `rt_spi_release()`

Function prototype:

```
rt_err_t rt_spi_release(struct rt_spi_device *device);
```

parameter	describe
device	SPI device handle

Function returns: Return 0

Calling this function ends the chip selection.

5.2.5. `rt_spi_message_append()`

Function prototype:

```
rt_inline void rt_spi_message_append(struct rt_spi_message *list,
                                     struct rt_spi_message *message)
```

parameter	describe
list	Message list pointer
message	Message pointer

Function returns: No return value

Call this function to insert a message message into the message list list.