Application note

General MCU RT_Thread Device Registration Application Note

Introduction

This document mainly describes the RT_Thread device registration of the N32G45x series, N32G4FR series, N32WB452 series, N32G43x series, N32L40x series, and N32L43x series MCUs, so that users can quickly familiarize themselves with the RT_Thread device driver.

Contents

1 Overview

1.1 Brief introduction

This document mainly describes the RT_Thread device registration of the N32G45x series, N32G4FR series, N32WB452 series, N32G43x series, N32L40x series, and N32L43x series MCUs, so that users can quickly familiarize themselves with the RT_Thread device driver.

2 Device registration

2.1 I/O device

2.1.1 Introduction of I/O device

RT-Thread provides a simple I/O device model framework, as shown in Figure 2-1. It is located between hardware and applications and is divided into three layers. From top to bottom are the I/O device management layer, device driver framework layer, device driver layer.

Figure 2-1 I/O device model framework
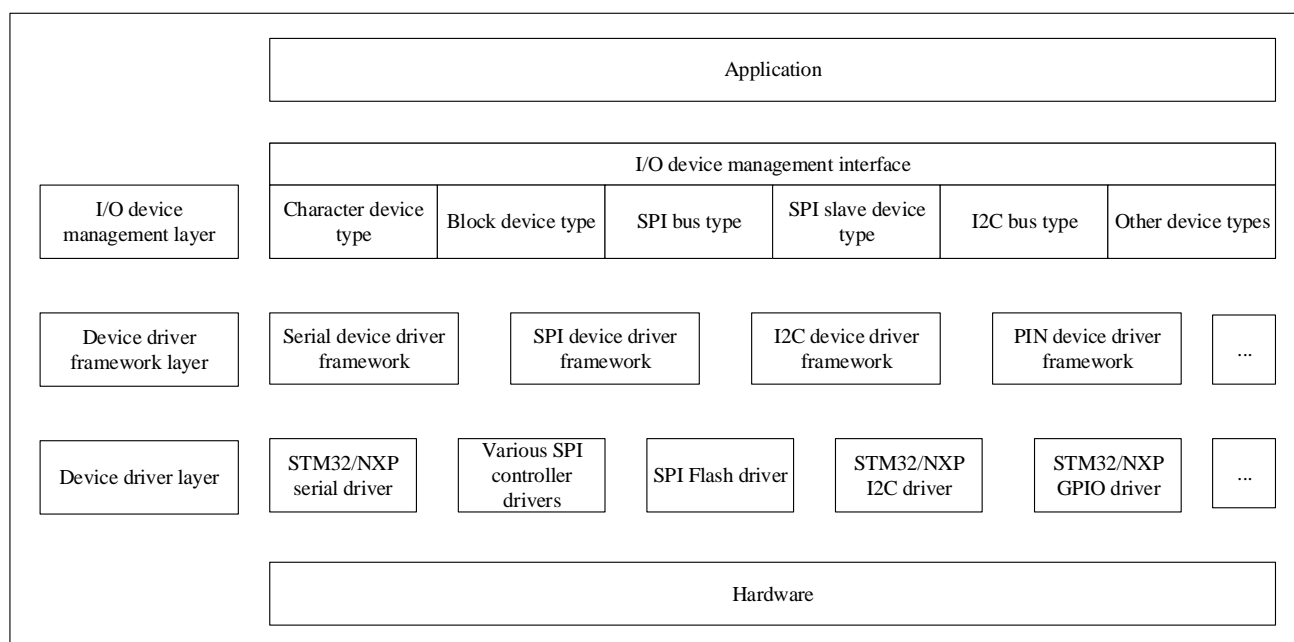
The application obtains the correct device driver through the I/O device management interface, and then interacts data (or control) with the underlying I/O hardware device through this device driver.

The I/O device management layer encapsulates device drivers. Application programs access lower-layer devices through standard interfaces provided by the I/O device layer. Upgrade or replacement of device drivers does not affect upper-layer applications. In this way, the code related to the hardware operation of the device can exist independently of the application program, and both parties only need to pay attention to the implementation of their own functions, which reduces the coupling and complexity of the code and improves the reliability of the system.

The device driver framework layer is an abstraction of the same type of hardware device drivers. It extracts the same parts from the same type of hardware device drivers from different manufacturers, and sets aside different parts for the interface, which is implemented by the driver.

The device driver layer is a group of programs that drive hardware devices to work, and realize the function of accessing hardware devices. It is responsible for creating and registering I/O devices. For devices with simple operation logic, the device can be directly registered in the I/O device manager without going through the device driver framework layer. Use the sequence diagram as shown in the figure below, mainly in the following two points:

■ The device driver creates a device instance with hardware access capability according to the device model

definition, and registers the device in the I/O device manager through the rt_device_register() interface.

■ The application finds the device through the rt_device_find () interface and then uses the I/O device management interface to access the hardware.

Figure 2-2 I/O device model framework
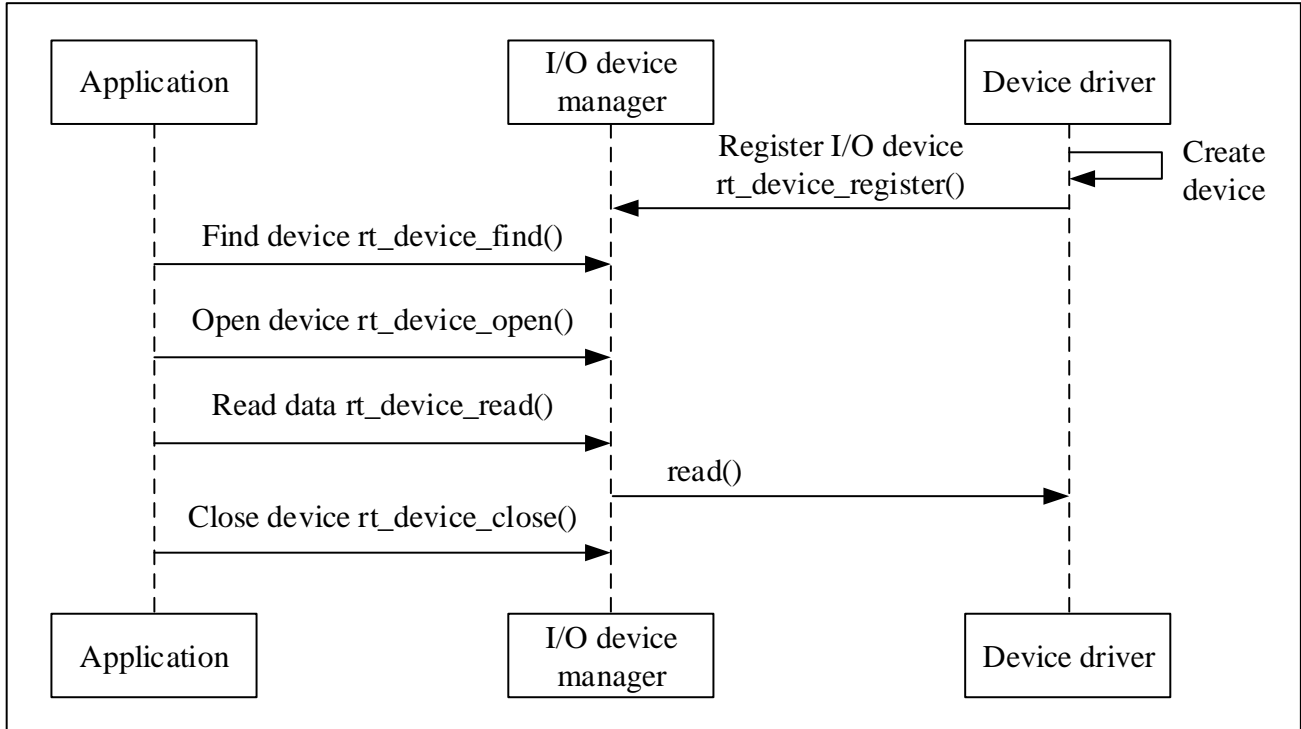
2.1.2 **Create and register I/O device**

The driver layer is responsible for creating device instances and registering them with the I/O device manager. Device instances can be created statically or dynamically using the following interface:

rt_device_t rt_device_create(int type, int attach_size)	
Parameter	**Describe**
type	Device type
attach_size	User data size
return	-
The device handle	Success
RT_NULL	Create failed, dynamic memory allocation failed

2.1.3 **Access I/O device**

The application program accesses the hardware device through the I/O device management interface. After the device driver is implemented, the application program can access the hardware. Figure 2-3 shows the mapping between the I/O device management interface and the operation methods of the I/O device.

Figure 2-3 I/O device interface
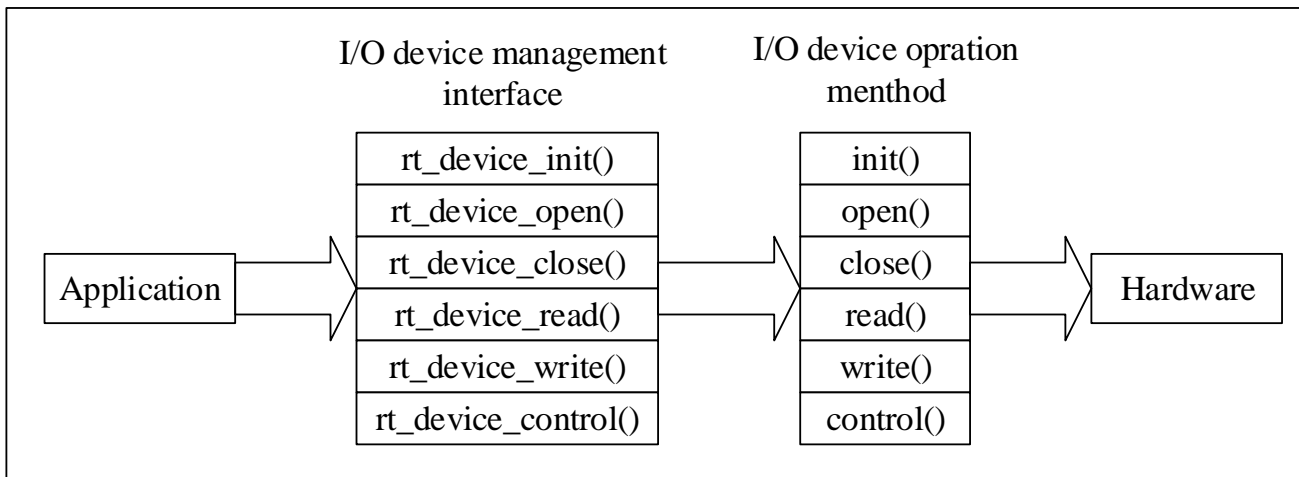
2.1.4 **Find device**

The application obtains a device handle based on the device name so that it can operate the device. The find device function looks like this:

rt_device_t rt_device_find(const char* name)	
Parameter	**Describe**
name	Device name
return	-
The device handle	If a device is found, the device handle is returned
RT_NULL	The corresponding device object was not found

2.1.5 **Initialize device**

After obtaining the device handle, the application can initialize the device using the following function:

rt_err_t rt_device_init(rt_device_t dev)	
Parameter	**Describe**
dev	The device handle
return	-
RT_EOK	The device is successfully initialized
Error code	Device initialization failed

2.1.6 **Open/close device**

Through the device handle, the application can open and close the device. When the device is opened, it will detect whether the device has been initialized. If it is not initialized, the initialization interface will be called by default to initialize the device. Open the device with the following function:

rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags)	
Parameter	**Describe**
dev	The device handle

oflags	The device opens the mode flag
return	-
RT_EOK	Device opened successfully
-RT_EBUSY	If the parameter specified when the device is registered includes the RT_DEVICE_FLAG_STANDALONE parameter, the device will not be allowed to open repeatedly
Other error codes	Device failed to open

Close device by using the following function:

rt_err_t rt_device_close(rt_device_t dev)	
Parameter	**Describe**
dev	The device handle
return	-
RT_EOK	Device close successfully
-RT_ERROR	The device has been completely closed and cannot be closed repeatedly
Other error codes	Failed to close device

2.1.7 **Control device**

Through the command control word, the application program can also control the device through the following function:

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
Parameter	**Describe**
dev	The device handle
cmd	Command control word, which is usually associated with the device driver
arg	Control parameter
return	-
RT_EOK	Function executed successfully
-RT_ENOSYS	Execution failed, dev is empty
Other error codes	Failed to execute

2.1.8 **Read/write device**

An application reading data from the device can be done with the following function:

rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)	
Parameter	**Describe**
dev	The device handle
pos	Read data offset
buffer	Memory buffer pointer, the read data will be saved in the buffer
size	The size of the read data
return	-
The actual size of the data read	If it is a character device, the returned size is in byte, if it is a block device, the returned size is in block

0	You need to read the current thread's errno to determine the error status

To write data to the device, you can use the following function:

rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)	
Parameter	**Describe**
dev	The device handle
pos	Write data offset
buffer	Memory buffer pointer, where the data to be written is placed
size	The size of the written data
return	-
The actual size of the data to be written	If it is a character device, the returned size is in byte, if it is a block device, the returned size is in block
0	You need to read the current thread's errno to determine the error status

2.1.9 **Data sending and receiving callback**

When the hardware device receives data, the following function can call back another function to set the data reception indication, and notify the upper-layer application thread that data arrives:

rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)(rt_device_t dev, rt_size_t size)	
Parameter	**Describe**
dev	The device handle
rx_ind	Callback function pointer
return	-
RT_EOK	Set successfully

The callback function for this function is provided by the caller. When the hardware device receives data, it will call back this function and pass the received data length in the size parameter to the upper-layer application. The upper application thread should read data from the device immediately after receiving the instruction.

When the application calls rt_device_write() to write data, if the underlying hardware can support automatic sending, the upper-layer application can set a callback function. This callback function will be called after the underlying hardware data transmission is completed (such as when the DMA transfer is completed or when the FIFO has been written and a completion interrupt is generated). You can set the device to send the completion indication through the following function:

rt_err_t rt_device_set_tx_complete(rt_device_t dev, rt_err_t (*tx_done)(rt_device_t dev,void *buffer))	
Parameter	**Describe**
dev	The device handle
tx_done	Callback function pointer
return	-
RT_EOK	Set successfully

When this function is called, the callback function is provided by the caller. When the hardware device finishes sending data, the driver calls back this function and passes the address buffer of the data block that has been sent as a parameter to the upper-layer application. When the upper-layer application (thread) receives the instruction, it will release the buffer memory block or use it as the buffer for the next write data according to the situation of sending

the buffer.

2.2 **PIN device**

2.2.1 **Introduction of PIN**

The pins on the chip are generally divided into 4 categories: power supply, clock, control and I/O. The I/O port is divided into General Purpose Input Output (general purpose input/output) in the usage mode, referred to as GPIO, which is multiplexed with functions I/O (eg SPI/I2C/UART, etc.).

Most MCU pins have more than one function. Different pins have different internal structures and different functions. The actual function of the pin can be switched with different configurations. The general I/O port has the following features:

Programmable control interrupt: the interrupt trigger mode can be configured, as shown in Figure 2-4:

Figure 2-4 PIN interrupt trigger mode
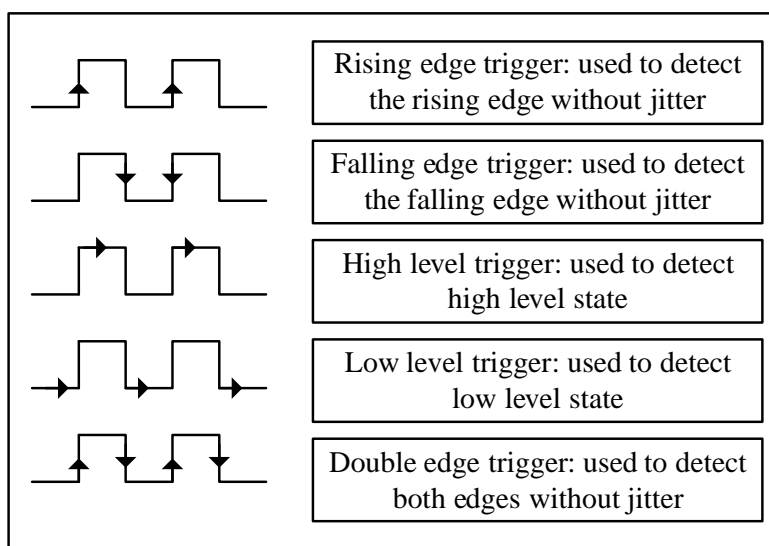
The input and output mode can be controlled.

Output mode generally includes: push-pull, open drain, pull-up, pull-down. When the pin is in output mode, the connected peripheral device can be controlled by configuring the level state of the pin output to be high or low.

Input modes generally includes: floating, pull-up, pull-down and analog. When the pin is in input mode, the level state of the pin can be read, i.e. high level or low level.

2.2.2 **Access PIN device**

The application accesses GPIO through the PIN device management interface provided by RT-Thread. The relevant interfaces are as follows:

Function	Describe
rt_pin_mode()	Set pin mode
rt_pin_write()	Set pin level
rt_pin_read()	Read pin level

rt_pin_attach_irq()	Bind pin interrupt callback function
rt_pin_irq_enable()	Enable pin interrupt
rt_pin_detach_irq()	Breakout pin interrupt callback function

2.2.3 **Set pin mode**

The input or output mode of the pin should be set before it is used. This can be done by using the following function:

void rt_pin_mode(rt_base_t pin, rt_base_t mode)	
Parameter	**Describe**
pin	Pin number
mode	Pin worke mode

2.2.4 **Set pin level**

The function to set the pin output level is as follows:

void rt_pin_write(rt_base_t pin, rt_base_t value)	
Parameter	**Describe**
pin	Pin number
value	Logical level value, which can be one of two macro definition values: PIN_LOW low level or PIN_HIGH high level

2.2.5 **Read pin level**

The function for reading pin levels is as follows:

int rt_pin_read(rt_base_t pin)	
Parameter	**Describe**
pin	Pin number
return	-
PIN_LOW	Low level
PIN_HIGH	High level

2.2.6 **Bind pin interrupt callback function**

To use the interrupt function of a pin, you can use the following function to configure a pin as an interrupt trigger mode and bind an interrupt callback function to the corresponding pin. When the pin interrupt occurs, the callback function will be executed:

rt_err_t rt_pin_attach_irq(rt_int32_t pin, rt_uint32_t mode, void (*hdr)(void *args), void *args)	
Parameter	**Describe**
pin	Pin number
mode	Interrupt trigger mode
hdr	Interrupt the callback function, which needs to be defined by the user
args	The parameter of the interrupt callback function, set to RT_NULL if not needed
return	-

RT_EOK	Binding success
Error code	Binding failed

2.2.7 **Enable pin interrupt**

After binding the pin interrupt callback, use the following function to enable pin interrupt:

rt_err_t rt_pin_irq_enable(rt_base_t pin, rt_uint32_t enabled)	
Parameter	**Describe**
pin	Pin number
enabled	PIN_IRQ_ENABLE (enabled) or PIN_IRQ_DISABLE (disabled)
return	-
RT_EOK	Enable success
Error code	Enable failed

2.2.8 **Breakout pin interrupt callback function**

You can use the following function to breakout pin interrupt callback function:

rt_err_t rt_pin_detach_irq(rt_int32_t pin)	
Parameter	**Describe**
pin	Pin number
return	-
RT_EOK	Breakout success
Error code	Breakout failed

2.3 **SPI device**

2.3.1 **Introduction of SPI**

SPI (Serial Peripheral Interface) is a high-speed, full-duplex, synchronous communication bus, often used for short distance communication. SPI generally uses four wires for communication, as shown in Figure 2-5:

Figure 2-5 SPI communication
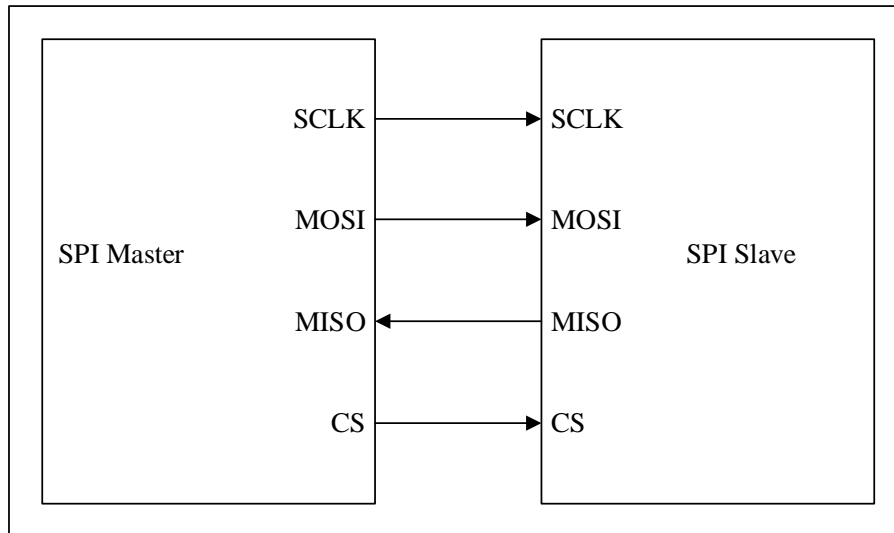
■ MOSI: SPI bus master output/slave input data wire.

■ MISO: SPI bus master input/slave output data wire.

■ SCLK: serial clock wire. The master device outputs clock signal to the slave device.

■ CS: slave device select wire (Chip select). Also called SS, CSB, CSN, EN, etc., the master device outputs the chip select signal to the slave device

SPI works in a master-slave mode, usually with one master and one or more slaves. The communication is initiated by the master device, the master device selects the slave device to be communicated through CS, and then provides a clock signal to the slave device through SCLK, the data is output to the slave device through MOSI, and the data sent by the slave device is received through MISO at the same time.

As shown in Figure 2-6, the chip has two SPI controllers. The SPI controller corresponds to the SPI master device. Each SPI controller can connect to multiple SPI slave devices. Slave devices mounted on the same SPI controller share 3 signal pins: SCK, MISO, MOSI, but the CS pin of each slave device is independent.

Figure 2-6 SPI controller-
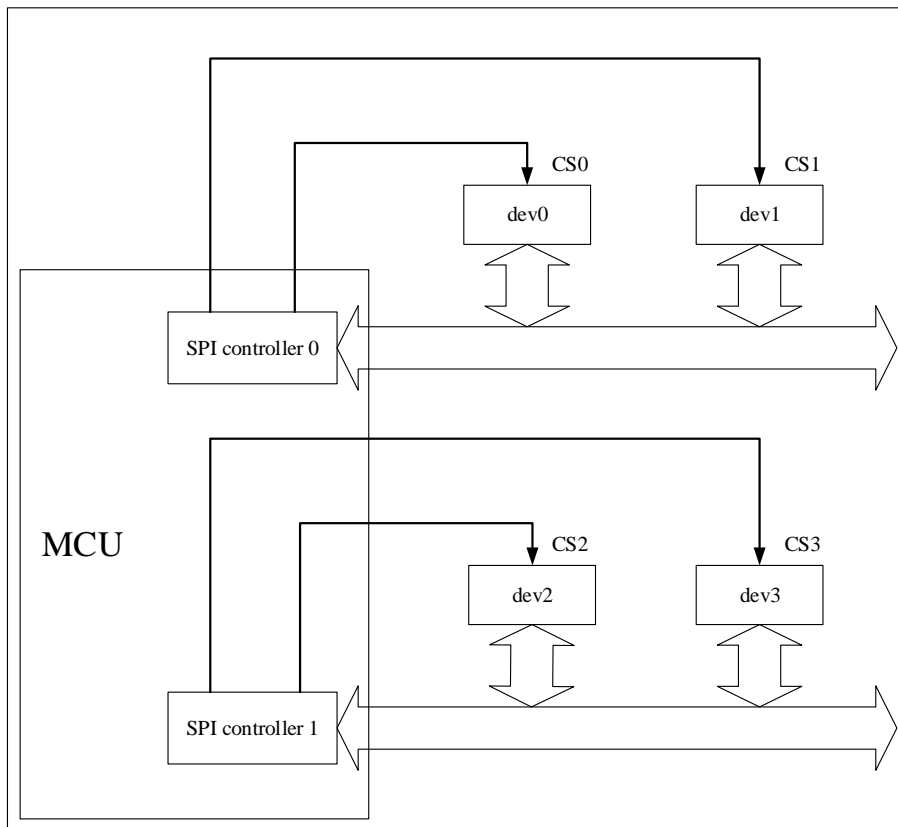
The master device selects the slave device by controlling the CS pin, which is generally active at low level. Only one CS pin on an SPI master device is in a valid state at any one time, and the slave device connected to this valid CS pin can communicate with the master device at this time.

2.3.2 **Mount SPI device**

The SPI device needs to be mounted to the registered SPI bus.

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
return	-
RT_EOK	Success
Other error codes	Failed

This function is used to mount an SPI device to the specified SPI bus, register the SPI device with the kernel, and save user_data to the SPI device control block.

Generally, the SPI bus is named spix, and the SPI device is named spixy. For example, SPI10 indicates the device 0 mounted on the SPI1 bus. User_data is generally the CS pin pointer of the SPI device. The SPI controller will operate this pin for chip selection during data transmission.

Mount the SPI device to the bus using the following function:

```
rt_err_t rt_hw_spi_device_attach( const char      *bus_name,
                                   const char      *device_name,
                                   GPIO_TypeDef *cs_gpiox,
                                   uint16_t         cs_gpio_pin)
```

2.3.3 **Configuring SPI device**

After mounting an SPI device to the SPI bus, you need to set transmission parameters for the SPI device.

rt_err_t rt_spi_configure(struct rt_spi_device *device, struct rt_spi_configuration *cfg)	
Parameter	**Describe**
device	SPI device handle
cfg	SPI configuration parameter pointer
return	-
RT_EOK	Success

This function will save the configuration parameters pointed to by cfg in the control block of the SPI device, which will be used when transferring data. The prototype of struct rt_spi_configuration is as follows:

```
struct rt_spi_configuration
{
    rt_uint8_t    mode;          // mode
    rt_uint8_t    data_width;     // data width, 8 bits, 16 bits, 32 bits
    rt_uint16_t   reserved;       // reserved
    rt_uint32_t   max_hz;        // maximum frequency
};
```

2.3.4 **Access SPI device**

In general, the SPI device of the MCU is used as a master and a slave to communicate. In RT-Thread, the SPI master is virtualized as an SPI bus device. The application uses the SPI device management interface to access the SPI slave device. The main interface is as follows:

Function	**Describe**
rt_device_find()	Obtain a device handle based on the SPI device name
rt_spi_transfer_message()	Customize transmission data
rt_spi_transfer()	Transfer data once
rt_spi_send()	Send data once
rt_spi_recv()	Receive data once
rt_spi_send_then_send()	Send twice in a row
rt_spi_send_then_recv()	Send first, then receive

2.3.5 **Find SPI device**

Before using the SPI device, the device handle should be obtained according to the name of the SPI device, and then the SPI device can be operated. The device find function is shown as follows:

rt_device_t rt_device_find(const char* name)	
Parameter	**Describe**
name	Device name
return	-
Device handle	If a device is found, the device handle is returned
RT_NULL	The corresponding device object was not found

2.3.6 **Customize transmission data**

After obtaining the SPI device handle, you can use the SPI device management interface to access the SPI device and send and receive data. Messages can be transferred using the following function:

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
return	-
RT_NULL	Successfully sent
non-null pointer	Send failed, return pointer to remaining unsent message

This function can transmit a string of messages. The user can customize the values of each parameter of the message structure to be transmitted, which makes it easy to control the data transmission mode. struct rt_spi_message prototype:

```
struct rt_spi_message
{
    const void          *send_buf;      // Send buffer pointer
    void                *recv_buf;      // Receive buffer pointer
    rt_size_t            length;        // Number of bytes of data sent/received
    struct rt_spi_message *next;        // A pointer to the next message
    unsigned cs_take     : 1;          // Select
    unsigned cs_release  : 1;           // Release
};
```

sendbuf is the send buffer pointer, and when its value is RT_NULL, it means that the current transmission is in a receive-only state, and no data needs to be sent.

recvbuf is the pointer to the receive buffer. When its value is RT_NULL, it means that this transmission is in a send-only state, and the received data does not need to be saved, so the received data is discarded directly.

The unit of length is word, that is, when the data length is 8 bits, each length occupies 1 byte; when the data length is 16 bits, each length occupies 2 bytes.

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

is RT_NULL. Multiple messages to be transmitted are connected together in the form of a singly linked list through the next pointer.

When the value of cs_take is 1, it means that the corresponding CS is set to a valid state before transmitting data. When the value of cs_release is 1, it means that the corresponding CS will be released after the data transmission ends.

2.3.7 **Transfer data once**

If the data is transmitted only once, the following function can be used:

rt_size_t rt_spi_transfer(struct rt_spi_device *device, const void *send_buf, void *recv_buf, rt_size_t length)	
Parameter	**Describe**
device	SPI device handle
send_buf	Send data buffer pointer
recv_buf	Receive data buffer pointer
length	Number of bytes of data sent/received
return	-
0	Transfer failed
Non-zero value	Number of bytes successfully transferred

This function is equivalent to calling rt_spi_transfer_message() to transfer a message. The chip selection is selected when the data is sent, and the chip selection is released when the function returns. The message parameter configuration is as follows:

```
struct rt_spi_message    msg;

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

2.3.8 **Send data once**

If the data is sent only once and the received data is ignored, the following function can be used:

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 data buffer pointer
length	Number of bytes of sent data
return	-
0	Send failed
Non-zero value	Number of bytes successfully sent

Call this function to send the data of the buffer pointed to by send_buf, ignoring the received data, this function is the

encapsulation of the rt_spi_transfer() function.

This function is equivalent to calling rt_spi_transfer_message() to transfer a message. The chip select is selected when data is sent, and the chip select is released when the function returns. The message parameter is 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;
```

2.3.9 **Receive data once**

If the data is received only once, the following function can be used:

rt_size_t rt_spi_recv(struct rt_spi_device *device, void *recv_buf, rt_size_t length)	
Parameter	**Describe**
device	SPI device handle
recv_buf	Receive data buffer pointer
length	Number of bytes of received data
return	-
0	Receive failed
Non-zero value	Number of bytes successfully received

Call this function to receive data and save it to the buffer pointed to by recv_buf. This function is a wrapper around the rt_spi_transfer() function. The SPI bus protocol stipulates that the clock can only be generated by the master device, so when receiving data, the master device will send data 0XFF.

This function is equivalent to calling rt_spi_transfer_message() to transfer a message, the chip select is selected when it starts to receive data, and the chip select is released when the function returns. The message parameter is configured as follows:

```
struct rt_spi_message msg;

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

2.3.10 **Send data twice in a row**

If you need to send the data of 2 buffers in succession, and the middle chip selection is not released, you can call the

following function:

```
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 device handle
send_buf1	Send data buffer 1 pointer
send_length1	Send data buffer 1 data bytes
send_buf2	Send data buffer 2 pointer
send_length2	Send data buffer 2 data bytes
return	-
RT_EOK	Send success
-RT_EIO	Send failed

This function can send two buffers in a row, ignore the received data, the chip select is selected when send_buf1 is sent, and the chip select is released after send_buf2 is sent.

This function is used to write a piece of data to the SPI device. The first time it sends the command and address data, and the second time it sends the specified length of data. The reason why it is sent twice instead of combined into one data block, or called rt_spi_send() twice, is because in most data write operations, the command and address need to be sent first, and the length is generally only a few bytes. If it is sent together with the following data, memory space application and a large amount of data handling will be required. If rt_spi_send() is called twice, the chip select will be released after the command and address are sent. Most SPI devices rely on setting the chip select to be valid once as the start of the command, therefore, the chip select is released after sending the command or address data, and the operation is discarded.

This function is equivalent to calling rt_spi_transfer_message() to transfer two messages. The message parameter configuration is 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;
```

2.3.11 **Send first, then receive**

If you need to send data to the slave device first, and then receive the data sent from the slave device, and the intermediate chip selection is not released, you can call the following function:

rt_err_t rt_spi_send_then_recv(struct rt_spi_device *device,	
const void *send_buf, rt_size_t send_length, void *recv_buf, rt_size_t recv_length)	
Parameter	**Describe**
device	SPI slave device handle
send_buf	Send data buffer pointer
send_length	Send data buffer data bytes
recv_buf	Receive data buffer pointer
recv_length	Received data bytes
return	-
RT_EOK	Success
-RT_EIO	Failed

This function starts chip selection when sending the first piece of data send_buf, at this time ignores the received data, and then sends the second piece of data, at this time the master device will send the data 0XFF, the received data is stored in recv_buf, and release the chip select when the function returns.

This function is suitable for reading a piece of data from the SPI slave device. For the first time, some commands and address data will be sent first, and then the data of the specified length will be received.

This function is equivalent to calling rt_spi_transfer_message() to transfer two messages. The message parameter configuration is as follows:

```
struct rt_spi_message msg1, msg2;


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


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

The SPI device management module also provides rt_spi_sendrecv8() and rt_spi_sendrecv16() functions, both of

which are encapsulations of this function, rt_spi_sendrecv8() sends one byte of data and receives one byte of data, rt_spi_sendrecv16() send 2 bytes of data and receive 2 bytes of data.

2.3.12 **Special application scenario**

In some special usage scenarios, a device wants to monopolize the bus for a period of time, and during this period, the chip selection must be kept valid, and the data transmission may be intermittent during this period, you can use the relevant interface according to the steps shown. The data transfer function must use rt_spi_transfer_message(), and the chip select control fields cs_take and cs_release of each message to be transferred in this function must be set to 0, because the chip select has already used other interface control, and does not need to be controlled during data transmission. .

2.3.13 **Get the bus**

In the case of multi-threading, the same SPI bus may be used in different threads. In order to prevent the loss of data being transmitted by the SPI bus, the slave device needs to obtain the right to use the SPI bus before starting to transmit data, only after the acquisition is successful, the bus can be used to transmit data. The following function can be used to acquire the SPI bus:

rt_err_t rt_spi_take_bus(struct rt_spi_device *device)	
Parameter	**Describe**
device	SPI device handle
return	-
RT_EOK	Success
Error code	Failed

2.3.14 **Select chip select**

After obtaining the right to use the bus from the device, you need to set the corresponding chip select signal to be valid. You can use the following function to select the chip select:

rt_err_t rt_spi_take(struct rt_spi_device *device)	
Parameter	**Describe**
device	SPI device handle
return	-
0	Success
Error code	Failed

2.3.15 **Add a message**

When rt_spi_transfer_message() is used to transfer messages, all the messages to be transferred are linked in a one-way list. You can add a new message to the list by using the following function:

void rt_spi_message_append(struct rt_spi_message *list, struct rt_spi_message *message)	
Parameter	**Describe**
list	The node of the linked list of messages to be transmitted

message	New message pointer

2.3.16 **Release chip select**

After the slave data transmission is completed, the chip selection needs to be released. The following function can be used to release the chip selection:

rt_err_t rt_spi_release(struct rt_spi_device *device)	
Parameter	**Describe**
device	SPI device handle
return	-
0	Success
Error code	Failed

2.3.17 **Release the bus**

The slave device is not using the SPI bus to transfer data. The bus must be released as soon as possible so that other slave devices can use the SPI bus to transfer data. The bus can be released using the following function:

rt_err_t rt_spi_release_bus(struct rt_spi_device *device)	
Parameter	**Describe**
device	SPI device handle
return	-
RT_EOK	Success

2.4 **UART device**

2.4.1 **Introduction of UART**

Universal Asynchronous Receiver/Transmitter (UART), as a kind of asynchronous serial communication protocol, works by transmitting each character of the transmitted data bit by bit. It is the most frequently used data bus during application development.

2.4.2 **Access serial port device**

Applications access serial port hardware through the I/O device management interfaces provided by RT-Thread. The interfaces are as follows:

Function	Describe
rt_device_find()	Find device
rt_device_open()	Open device
rt_device_read()	Read data
rt_device_write()	Write data
rt_device_control()	Control device
rt_device_set_rx_indicate()	Set the receive callback function
rt_device_set_tx_complete()	Set the send completion callback function
rt_device_close()	Close device

2.4.3 **Find serial port device**

The application program obtains the device handle according to the serial device name, and then can operate the serial device. The find device function is shown below.

rt_device_t rt_device_find(const char* name)	
Parameter	**Describe**
name	Device name
return	-
The device handle	If a device is found, the device handle is returned
RT_NULL	The corresponding device object was not found

2.4.4 **Open serial port device**

Through the device handle, the application can open and close the device. When the device is opened, it will detect whether the device has been initialized. If it is not initialized, the initialization interface will be called by default to initialize the device. Open the device with the following function:

rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags)	
Parameter	**Describe**
dev	The device handle
oflags	Device mode flag

return	-
RT_EOK	Device opened successfully
-RT_EBUSY	If the parameter specified when the device is registered includes the RT_DEVICE_FLAG_STANDALONE parameter, the device will not be allowed to open repeatedly
Other error codes	Device open failed

The oflags parameter supports the following values (multiple values can be supported in the form of OR):

```
#define RT_DEVICE_FLAG_STREAM       0x040    /* Stream mode */


/* Receive mode parameter */
#define RT_DEVICE_FLAG_INT_RX       0x100    /* Interrupt receive mode */
#define RT_DEVICE_FLAG_DMA_RX       0x200    /* DMA receive mode */


/* Send mode parameter */
#define RT_DEVICE_FLAG_INT_TX       0x400    /* Interrupt send mode */
#define RT_DEVICE_FLAG_DMA_TX       0x800    /* DMA send mode */
```

There are three modes of serial port data receiving and sending data: interrupt mode, polling mode, and DMA mode. When in use, only one of these three modes can be selected. If the open parameter oflags of the serial port does not specify the use of interrupt mode or DMA mode, the polling mode is used by default.

DMA (Direct Memory Access) means direct memory access. The DMA transmission method does not require the CPU to directly control the transmission, nor does it have the process of retaining the scene and restoring the scene like the interrupt processing method. A direct data transfer path is opened up for RAM and I/O devices through the DMA controller, which saves CPU resources for other operations. Using DMA transfers can continuously acquire or send a piece of information without interruption or delay, which is very useful when communication is frequent or when there are large pieces of information to transfer.

2.4.5 **Control serial port device**

Through the control interface, the application program can configure the serial port device, such as baud rate, data bit, check bit, receive buffer size, stop bit and other parameters modification. The control function is as follows:

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
Parameter	**Describe**
dev	The device handle
cmd	Command control word, available value:RT_DEVICE_CTRL_CONFIG
arg	Control parameter, available type: struct serial_configure
return	-
RT_EOK	Function executed successfully
-RT_ENOSYS	Execution failed, dev is empty
Other error codes	Execution failed

The prototype of the control parameter structure struct serial_configure is as follows:

struct serial_configure

```
{
    rt_uint32_t baud_rate;              /* Baud rate */
    rt_uint32_t data_bits    :4;        /* Data bits */
    rt_uint32_t stop_bits    :2;        /* Stop bit */
    rt_uint32_t parity       :2;         /* Parity bit */
    rt_uint32_t bit_order    :1;        /* The high value is in front or the low value is in front */
    rt_uint32_t invert       :1;         /* Mode */
    rt_uint32_t bufsz        :16;       /* Receive data buffer size */
    rt_uint32_t reserved     :4;         /* Reserved bit */
};
```

The default serial port configuration provided by RT-Thread is as follows, that is, each serial port device in the RT-Thread system uses the following configuration by default:

```
#define RT_SERIAL_CONFIG_DEFAULT              \
{                                             \
    BAUD_RATE_115200,         /* 115200 bits/s */     \
    DATA_BITS_8,              /* 8 data bits */        \
    STOP_BITS_1,             /* 1 stop bit */         \
    PARITY_NONE,             /* No parity    */       \
    BIT_ORDER_LSB,           /* LSB first sent */     \
    NRZ_NORMAL,              /* Normal mode */        \
    RT_SERIAL_RB_BUFSZ,   /* Buffer size */        \
    0                                         \
}
```

If the actual configuration parameters of the serial port are inconsistent with the default configuration parameters, the user can modify them through the application code. Modify serial port configuration parameters, such as baud rate, data bits, parity bits, buffer receiving buffsize, stop bits, etc.

2.4.6 Send data

To write data to the serial port, you can use the following function:

rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)	
Parameter	**Describe**
dev	The device handle
pos	Write data offset, this parameter is not used by serial device
buffer	Memory buffer pointer, where the data to be written is placed
size	The size of the written data
return	-
The actual size of the data to be written	If it is a character device, the return size is in byte
0	You need to read the current thread's errno to determine the error status

2.4.7 Set the send completion callback function

When the application calls rt_device_write() to write data, if the underlying hardware can support automatic sending,

the upper-layer application can set a callback function. This callback function will be called after the underlying hardware data transmission is completed (such as when the DMA transfer is completed or when the FIFO has been written and a completion interrupt is generated). You can set the device to send the completion indication through the following function:

rt_err_t rt_device_set_tx_complete(rt_device_t dev, rt_err_t (*tx_done)(rt_device_t dev, void *buffer))	
Parameter	**Describe**
dev	The device handle
tx_done	Callback function pointer
return	-
RT_EOK	Set successfully

When this function is called, the callback function is provided by the caller. When the hardware device finishes sending data, the device driver will call back this function and pass the sent data block address buffer as a parameter to the upper-layer application. When the upper-layer application (thread) receives the instruction, it will release the buffer memory block or use it as the buffer for the next write data according to the situation of sending the buffer.

2.4.8 **Set the receive callback function**

The data receiving indication can be set by the following function. When the serial port receives data, it notifies the upper application thread that data arrives:

rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)(rt_device_t dev, rt_size_t size))	
Parameter	**Describe**
dev	The device handle
rx_ind	Callback function pointer
dev	Device handle (callback function parameter)
size	Buffer data size (callback function parameter)
return	-
RT_EOK	Set successfully

The callback function for this function is provided by the caller. If the serial port is opened in the interrupt receiving mode, when the serial port receives a data and generates an interrupt, the callback function will be called, and the data size of the buffer at this time will be placed in the size parameter, and the serial port device handle will be placed in the dev parameter for the caller to obtain.

If the serial port is opened in DMA receive mode, this callback function will be called when DMA finishes receiving a batch of data.

In general, the receive callback function can send a semaphore or event to notify the serial port data processing thread that data arrives.

2.4.9 **Receive data**

The following function can be called to read the data received by the serial port:

rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)

2.4.10 **Close serial port device**

When the application completes the serial port operation, the serial port device can be closed by the following function:

rt_err_t rt_device_close(rt_device_t dev)	
Parameter	**Describe**
dev	The device handle
return	-
RT_EOK	Close the device successfully
-RT_ERROR	The device has been closed and cannot be closed repeatedly
Other error codes	Failed to close device

2.5 I2C device

2.5.1 Introduction of I2C

The I2C (Inter Integrated Circuit) bus is a half-duplex, bidirectional two-wire synchronous serial bus developed by PHILIPS. When the I2C bus transmits data, only two signal lines are needed, one is a bidirectional data line SDA (serial data), and the other is a bidirectional clock line SCL (serial clock). The SPI bus has two lines for receiving data and sending data between the master and slave devices, while the I2C bus only uses one line for data transmission and reception.

I2C works in the same master-slave mode as SPI, which is different from the one-master-multiple-slave structure of SPI. It allows multiple master devices to exist at the same time. Each device connected to the bus has a unique address, and the master device starts data transmission. And generate a clock signal, the slave device is addressed by the master device, and only one master device is allowed at the same time.

2.5.2 Access I2C bus device

In general, the I2C device of the MCU is used as a master and a slave to communicate. In RT-Thread, the I2C master is virtualized as an I2C bus device, and the I2C slave communicates with the I2C bus through the I2C device interface. The relevant interfaces are as follows:

Function	Describe
rt_device_find()	Obtain the device handle based on the I2C bus device name
rt_i2c_transfer()	Transfer data

2.5.3 Find I2C bus device

Before using the I2C bus device, you need to obtain the device handle according to the I2C bus device name, and then you can operate the I2C bus device. The function to find the device is as follows:

rt_device_t rt_device_find(const char* name)	
Parameter	Describe
name	I2C bus device name
return	-
The device handle	Find the corresponding device will return the corresponding device handle
RT_NULL	The corresponding device object was not found

2.5.4 Data transfer

After getting the I2C bus device handle, you can use rt_i2c_transfer() for data transfer. The function prototype looks like this:

rt_size_t rt_i2c_transfer(struct rt_i2c_bus_device *bus, struct rt_i2c_msg msgs[], rt_uint32_t num)	
Parameter	Describe
bus	I2C bus device handle
msgs[]	Pointer to an array of messages to transmit

num	The number of elements in the message array
return	-
The number of elements in the message array	Success
Error code	Failed

Like the custom transmission interface of the SPI bus, the data transmitted by the custom transmission interface of the I2C bus is also based on a message. The parameter msgs[] points to the message array to be transmitted, and the user can customize the content of each message to implement 2 different data transmission modes supported by the I2C bus. If the master needs to send a repeat start condition, it needs to send 2 messages.

The prototype of the I2C message data structure is as follows:

```
struct rt_i2c_msg
{
    rt_uint16_t   addr;       /* Slave address */
    rt_uint16_t   flags;       /* Read, write flag, etc. */
    rt_uint16_t   len;        /* Read and write data bytes */
    rt_uint8_t   *buf;        /* Read/write data buffer pointer */
}
```

Slave address addr: supports 7-bit and 10-bit binary addresses, please check the data sheet of different devices.

2.6 ADC device

2.6.1 Introduction of ADC

ADC(analog-to-digital converter) refers to an analog-to-digital converter. A device that converts continuously changing analog signals into discrete digital signals. Real-world analog signals, such as temperature, pressure, sound or images, need to be converted into digital forms that are easier to store, process and transmit. Analog-to-digital converters can achieve this function and can be found in a variety of different products. The corresponding DAC(digital-to-analog converter) is the reverse process of ADC analog-to-digital conversion. ADC was originally used to convert wireless signals to digital signals. Such as television signals, long and short broadcast radio send and receive.

2.6.2 Access ADC device

The application accesses the ADC hardware through the ADC device management interface provided by RT-Thread. The relevant interfaces are as follows:

Function	Describe
rt_device_find()	Obtain device handle based on ADC device name
rt_adc_enable()	Enable the ADC device
rt_adc_read()	Read ADC device data
rt_adc_disable()	Close ADC device

2.6.3 Find ADC device

The application obtains the device handle according to the ADC device name, and then can operate the ADC device. The function to find the device is as follows:

rt_device_t rt_device_find(const char* name)	
Parameter	Describe
name	ADC device name
return	-
The device handle	If a device is found, the device handle is returned
RT_NULL	No device found

2.6.4 Enable ADC channel

Before reading ADC device data, you need to enable the device first, and enable the device through the following function:

rt_err_t rt_adc_enable(rt_adc_device_t dev, rt_uint32_t channel)	
Parameter	Describe
dev	ADC device handle
channel	The ADC channel
return	-

RT_EOK	Success
-RT_ENOSYS	Failed. Device operation method is empty
Other error codes	Failed

2.6.5 **Read ADC channel sampling value**

Reading the ADC channel sample value can be done by the following function:

rt_uint32_t rt_adc_read(rt_adc_device_t dev, rt_uint32_t channel)	
Parameter	**Describe**
dev	ADC device handle
channel	The ADC channel
return	-
Value read	

2.6.6 **Close ADC channel**

Closing the ADC channel can be done with the following function:

rt_err_t rt_adc_disable(rt_adc_device_t dev, rt_uint32_t channel)	
Parameter	**Describe**
dev	ADC device handle
channel	The ADC channel
return	-
RT_EOK	Success
-RT_ENOSYS	Failed. Device operation method is empty
Other error codes	Failed

2.7 DAC device

2.7.1 Introduction of DAC

Digital-to-analog converter (DAC). It refers to a device that converts discrete digital signals in the form of binary digital quantities into continuously changing analog signals. In the digital world, it is not easy to deal with unstable and dynamic analog signals. Based on the characteristics of DAC, it can be found in various products. The corresponding ADC (Analog-to-Digital Converter)), which is the reverse process of DAC digital-to-analog conversion. DACs are mainly used in audio amplification, video encoding, motor control, digital potentiometers, etc.

2.7.2 Access DAC device

The application accesses the DAC hardware through the DAC device management interface provided by RT-Thread. The relevant interfaces are as follows:

Function	Describe
rt_device_find()	Obtain the device handle based on the DAC device name
rt_dac_enable()	Enable the DAC device
rt_dac_write()	Set DAC device output value
rt_dac_disable()	Close DAC device

2.7.3 Find DAC device

The application obtains the device handle according to the DAC device name, and then can operate the DAC device. The function of finding the device is as follows:

rt_device_t rt_device_find(const char* name)	
Parameter	**Describe**
name	DAC device name
return	-
The device handle	If a device is found, the device handle is returned
RT_NULL	No device found

2.7.4 Enable DAC channel

Before setting the DAC device data, you need to enable the device first, and enable the device through the following function:

rt_err_t rt_dac_enable(rt_dac_device_t dev, rt_uint32_t channel)	
Parameter	**Describe**
dev	DAC device handle
channel	DAC channel
return	-
RT_EOK	Success
-RT_ENOSYS	Failed. Device operation method is empty

Other error codes	Failed

2.7.5 **Set DAC channel output value**

Setting the DAC channel output value can be done with the following function:

rt_uint32_t rt_dac_write(rt_dac_device_t dev, rt_uint32_t channel, rt_uint32_t value)	
Parameter	**Describe**
dev	DAC device handle
channel	DAC channel
value	DAC output value
return	-
RT_EOK	Success
-RT_ENOSYS	Failed

2.7.6 **Close DAC channel**

Closing the DAC channel can be done with the following function:

rt_err_t rt_dac_disable(rt_dac_device_t dev, rt_uint32_t channel)	
Parameter	**Describe**
dev	DAC device handle
channel	DAC channel
return	-
RT_EOK	Success
-RT_ENOSYS	Failed. Device operation method is empty
Other error codes	Failed

2.8 CAN device

2.8.1 Introduction of CAN

CAN is the abbreviation of Controller Area Network (CAN). It was developed by BOSCH, A German company famous for its research and development and production of automotive electronic products, and eventually became an international standard (ISO 11898). It is one of the most widely used field buses in the world.

2.8.2 Access CAN device

The application program accesses the CAN hardware controller through the I/O device management interface provided by RT-Thread. The relevant interfaces are as follows:

Function	Describe
rt_device_find	Find device
rt_device_open	Open device
rt_device_read	Read data
rt_device_write	Write data
rt_device_control	Control device
rt_device_set_rx_indicate	Set the receive callback function
rt_device_close	Close device

2.8.3 Find CAN device

The application finds the device according to the name of the CAN device to obtain the device handle, and then can operate the CAN device. The function of finding the device is as follows:

rt_device_t rt_device_find(const char* name)	
Parameter	**Describe**
name	Device name
return	-
The device handle	Find the corresponding device will return the corresponding device handle
RT_NULL	The corresponding device object was not found

2.8.4 Open CAN device

Through the device handle, the application can open and close the device. When the device is opened, it will detect whether the device has been initialized. If it is not initialized, the initialization interface will be called by default to initialize the device. Open the device with the following function:

rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags)	
Parameter	**Describe**
dev	The device handle
oflags	Open device mode flag
return	-

RT_EOK	Device opened successfully
-RT_EBUSY	If the parameter specified when the device is registered includes the RT_DEVICE_FLAG_STANDALONE parameter, the device will not be allowed to open repeatedly
Other error codes	Device failed to open

2.8.5 **Control CAN device**

Through the command control word, the application program can configure the CAN device through the following function:

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
Parameter	**Describe**
dev	The device handle
cmd	Control command
arg	Control parameter
return	-
RT_EOK	Function executed successfully
Other error codes	Failed to execute

2.8.6 **Send data**

Sending data using a CAN device can be done through the following function:

rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)	
Parameter	**Describe**
dev	The device handle
pos	Write data offset. This parameter is not used by the CAN device
buffer	CAN message pointer
size	CAN message size
return	-
Is not zero	The actual size of the CAN message sent
0	Send failed

2.8.7 **Set receive callback function**

The data reception indication can be set by the following function. When CAN receives data, it notifies the upper application thread that data arrives:

rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)(rt_device_t dev, rt_size_t size))	
Parameter	**Describe**
dev	The device handle
rx_ind	Callback function pointer
dev	Device handle (callback function parameter)
size	Buffer data size (callback function parameter)
return	-

RT_EOK	Set successfully

2.8.8 **Receive data**

The following function can be called to read the data received by the CAN device:

rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)	
Parameter	**Describe**
dev	The device handle
pos	Read data offset. This parameter is not used by the CAN device
buffer	CAN message pointer, the read data will be stored in the buffer
size	CAN message size
return	-
Is not zero	CAN message size
0	Failed

2.8.9 **Close CAN device**

When the application completes the CAN operation, the CAN device can be closed by the following function:

rt_err_t rt_device_close(rt_device_t dev)	
Parameter	**Describe**
dev	The device handle
return	-
RT_EOK	Close device successfully
-RT_ERROR	The device has been completely closed and cannot be closed repeatedly
Other error codes	Close device failed

2.9 **HWTIMER device**

2.9.1 **Introduction of Timer**

Hardware timers generally have two working modes, timer mode and counter mode. No matter which mode it is working in, the essence is to count the pulse signal through the internal counter module. Below are some important concepts of timers.

Counter mode: Count the external pulse signal from the external input pin.

Timer mode: count the internal pulse signal. Timers are often used as timing clocks to achieve timing detection, timing response, and timing control.

2.9.2 **Access hardware timer device**

The application program accesses the hardware timer device through the I/O device management interface provided by RT-Thread. The relevant interface is as follows:

Function	Describe
rt_device_find()	Find timer device
rt_device_open()	Enable the timer device in read/write mode
rt_device_set_rx_indicate()	Set the timeout callback function
rt_device_control()	Control the timer device, you can set the timing mode (single/period)/counting frequency, or stop the timer
rt_device_write()	Set the timer timeout value and the timer will start immediately
rt_device_read()	Get the current timer value
rt_device_close()	Close timer device

2.9.3 **Find timer device**

The application obtains the device handle according to the name of the hardware timer device, and then can operate the hardware timer device. The function to find the device is as follows:

rt_device_t rt_device_find(const char* name)	
Parameter	**Describe**
name	Hardware timer device name
return	-
Timer device handle	If a device is found, the device handle is returned
RT_NULL	No device found

2.9.4 **Open timer device**

Through the device handle, the application can open the device. When the device is opened, it will detect whether the device has been initialized. If it is not initialized, the initialization interface will be called by default to initialize the device. Open the device with the following function:

rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags)

Parameter	Describe
dev	Hardware timer device handle
oflags	Device open mode, generally open in read and write mode, that is, the value: RT_DEVICE_OFLAG_RDWR
return	-
RT_EOK	Device opened successfully
Other error codes	Device open failed

2.9.5 **Set timeout callback function**

Set the timer timeout callback function through the following function, and this callback function will be called when the timer times out:

rt_err_t rt_device_set_rx_indicate(rt_device_t dev, rt_err_t (*rx_ind)(rt_device_t dev,rt_size_t size))	
Parameter	**Describe**
dev	The device handle
rx_ind	Timeout callback function, provided by the caller
return	-
RT_EOK	Success

2.9.6 **Control timer device**

Through the command control word, the application program can configure the hardware timer device through the following function:

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
Parameter	**Describe**
dev	The device handle
cmd	Command control word
arg	Control parameter
return	-
RT_EOK	Function executed successfully
-RT_ENOSYS	Execution failed. dev is empty
Other error codes	Execution failed

2.9.7 **Set timer timeout value**

The timeout value of the timer can be set by the following function:

rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)	
Parameter	**Describe**
dev	The device handle
pos	Write data offset, unused, can take a value of 0
buffer	Pointer to timer timeout structure
size	The size of the timeout structure
return	-

The actual size of the data to be written	
0	Failed

2.9.8 **Get current timer value**

The current value of the timer can be obtained by the following function:

rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)	
Parameter	**Describe**
dev	Timer device handle
pos	Write data offset, unused, can take a value of 0
buffer	Output parameter, pointer to timer timeout structure
size	The size of the timeout structure
return	-
The size of the timeout structure	Success
0	Failed

2.9.1 **Close timer device**

The timer device can be closed by the following function:

rt_err_t rt_device_close(rt_device_t dev)	
Parameter	**Describe**
dev	Timer device handle
return	-
RT_EOK	Close device successfully
-RT_ERROR	The device has been completely closed and cannot be closed repeatedly
Other error codes	Failed to close device

2.10 WATCHDOG device

2.10.1 Introduction of WATCHDOG

The hardware watchdog (watchdog timer) is a timer whose timing output is connected to the reset terminal of the circuit. In the productized embedded system, in order to make the system reset automatically under abnormal conditions, it is generally necessary to introduce a watchdog.

When the watchdog starts, the counter starts counting automatically. If it is not reset before the counter overflows, it will send a reset signal to the CPU to restart the system (commonly known as "bitten by the dog").When the system is running normally, it is necessary to clear the watchdog counter (commonly known as "feeding the dog") within the time interval allowed by the watchdog to prevent the reset signal from being generated. If the system doesn't break down, the program can "feed the dog" on time. Once the program runs away, there is no "feed the dog" and the system "gets bitten" reset.

2.10.2 Access watchdog device

The application program accesses the watchdog hardware through the I/O device management interface provided by RT-Thread. The relevant interfaces are as follows:

Function	Describe
rt_device_find()	Find the device based on the watchdog device name to obtain the device handle
rt_device_init()	Initialize watchdog device
rt_device_control()	Control watchdog device
rt_device_close()	Close watchdog device

2.10.3 Find watchdog

The application obtains the device handle according to the name of the watchdog device, and then can operate the watchdog device. The device search function is as follows:

rt_device_t rt_device_find(const char* name)	
Parameter	Describe
name	Watchdog device name
return	-
The device handle	If a device is found, the device handle is returned
RT_NULL	The corresponding device object was not found

2.10.4 Initialize watchdog

Before using the watchdog device, it needs to be initialized first. Initialize the watchdog device through the following function:

rt_err_t rt_device_init(rt_device_t dev)	
Parameter	Describe
dev	Watchdog device handle

return	-
RT_EOK	Device initialization succeeded
-RT_ENOSYS	Initialization failed. The watchdog device driver initialization function is empty
Other error codes	Device failed to open

2.10.5 **Control watchdog**

Through the command control word, the application can configure the watchdog device through the following function:

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
Parameter	**Describe**
dev	Watchdog device handle
cmd	Command control word
arg	Control parameter
return	-
RT_EOK	Function executed successfully
-RT_ENOSYS	Execution failed. dev is empty
Other error codes	Execution failed

2.10.6 **Feed dog in the idle thread hook function**

```
static void idle_hook(void)
{
    /* Feed the dog in the idle thread callback */
    rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_KEEPALIVE, NULL);
}
```

2.10.7 **Close watchdog**

When the application completes the watchdog operation, the watchdog device can be closed by the following function:

rt_err_t rt_device_close(rt_device_t dev)	
Parameter	**Describe**
dev	Watchdog device handle
return	-
RT_EOK	Close device successfully
-RT_ERROR	The device has been completely closed and cannot be closed repeatedly
Other error codes	Failed to close device

3 Version history

Date	Version	Remark
2021.05.07	V1.0	The initial version

4 **NOTICE**

This document is the exclusive property of Nations Technologies Inc. (Hereinafter referred to as NATIONS). This document, and the product of NATIONS described herein (Hereinafter referred to as the Product) are owned by NATIONS under the laws and treaties of the People's Republic of China and other applicable jurisdictions worldwide.

NATIONS does not grant any license under its patents, copyrights, trademarks, or other intellectual property rights. Names and brands of third party may be mentioned or referred thereto (if any) for identification purposes only.

NATIONS reserves the right to make changes, corrections, enhancements, modifications, and improvements to this document at any time without notice. Please contact NATIONS and obtain the latest version of this document before placing orders.

Although NATIONS has attempted to provide accurate and reliable information, NATIONS assumes no responsibility for the accuracy and reliability of this document.

It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. In no event shall NATIONS be liable for any direct, indirect, incidental, special, exemplary, or consequential damages arising in any way out of the use of this document or the Product.

NATIONS Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at user's risk. User shall indemnify NATIONS and hold NATIONS harmless from and against all claims, costs, damages, and other liabilities, arising from or related to any customer's Insecure Usage.

Any express or implied warranty with regard to this document or the Product, including, but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement are disclaimed to the fullest extent permitted by law.

Unless otherwise explicitly permitted by NATIONS, anyone may not use, duplicate, modify, transcribe or otherwise distribute this document for any purposes, in whole or in part.

Nations Technologies Inc.
Tel：+86-755-86309900
Email：info@nationstech.com
Address: Nations Tower, #109 Baoshen Road, Hi-tech Park North.
Nanshan District, Shenzhen, 518057, P.R.China