Serial Device Application Notes

RT-THREAD Documentation Center

WWW.RT-THREAD.ORG

Friday 28th September, 2018

Machine Translated by Google

Table of contents

This application note describes how to use RT-Thread's serial port devices, including serial port configuration,

The application of the device operation interface and the code examples verified on the Zhengdian Atom STM32F4 Explorer development board are given.

1 Purpose and structure of this paper

1.1 Purpose and Background of this Paper

The serial port (Universal Asynchronous Receiver/Transmitter, often written as UART, uart) is one of the most widely used communication interfaces. On a bare metal platform or an RTOS platform without a device management framework, we usually only need to write the serial port hardware initialization code according to the official manual. After the introduction of the real-time operating system RT-Thread with a device management framework, the use of serial ports is very different from bare metal or other RTOS. RT-Thread comes with an I/O device management layer, which encapsulates various hardware devices into logical devices with a unified interface for easy management and use. This article explains how to use serial ports in RT-Thread.

1.2 Structure of this paper

This article first gives the sample code for developing a serial port data receiving and sending program using the RT-Thread device operation interface, and verifies it on the Zhengdian Atom STM32F4 Explorer development board. Then it analyzes the implementation of the sample code, and finally describes in depth the connection between the RT-Thread device management framework and the serial port.

2 Problem Statement

RT-Thread provides a simple I/O device management framework, which divides I/O devices into three layers for processing: application layer, I/O device management layer, and hardware driver layer. The application obtains the correct device driver through the device operation interface of RT-Thread, and then interacts with the underlying I/O hardware device for data (or control) through this device driver.

RT-Thread provides an abstract device operation interface to upper-level applications and an underlying driver framework to lower-level devices.

Figure 1: *RT-Thread* Device Management Framework

So how can users use the device operation interface to develop cross-platform serial port application code?

3. Problem Solving

This article is based on the Zhengdian Atom STM32F4 Explorer development board and gives the serial port configuration process and application code examples. Due to the universality of the RT-Thread device operation interface, these codes are independent of the hardware platform and readers can directly use them. Use it on your own hardware platform. The Zhengdian Atom STM32F4 Explorer development board uses STM32F407ZGT6. It has multiple serial ports. We use serial port 1 as the shell terminal and serial port 2 as the experimental serial port to test data transmission and reception. The terminal software uses putty. The onboard serial port 1 has a USB to serial port chip, so use a USB cable to connect serial port 1 and PC is enough; serial port 2 needs to use USB to serial port module to connect to PC.

Machine Translated by Google

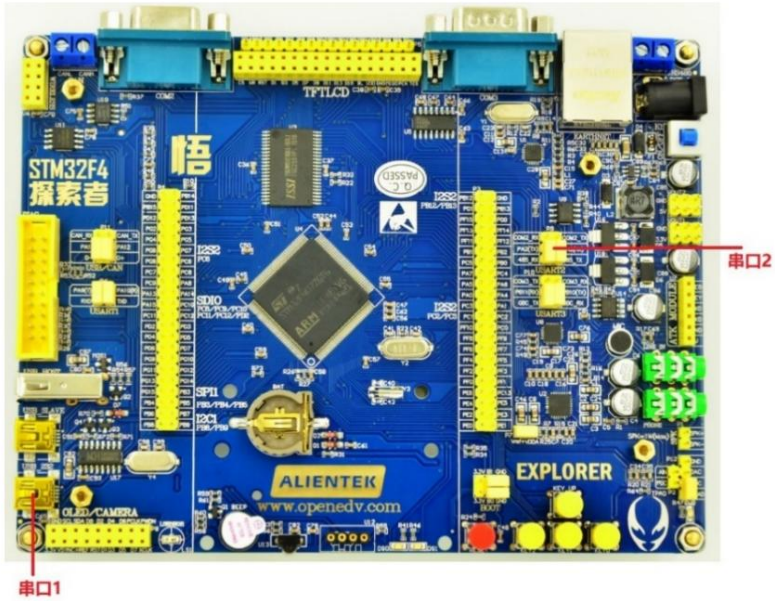

figure 2: Punctual Atom *STM32F4* Explorer

3.1 Prepare and configure the project

1. Download RT-Thread source code

2. Enter the rt-thread\bsp\stm32f4xx-HAL directory and enter menuconfig in the env command line to enter the configuration Use the menuconfig tool (learn how to use it) to configure the project.

(1) Configure the shell to use serial port 1: RT-Thread Kernel —> Kernel Device Object —> Modify the device name for console ÿ uart1ÿ

(2) Check Using UART1 and Using UART2, select the chip model as STM32F407ZG, and the clock source as external 8MHz, as shown in the figure:

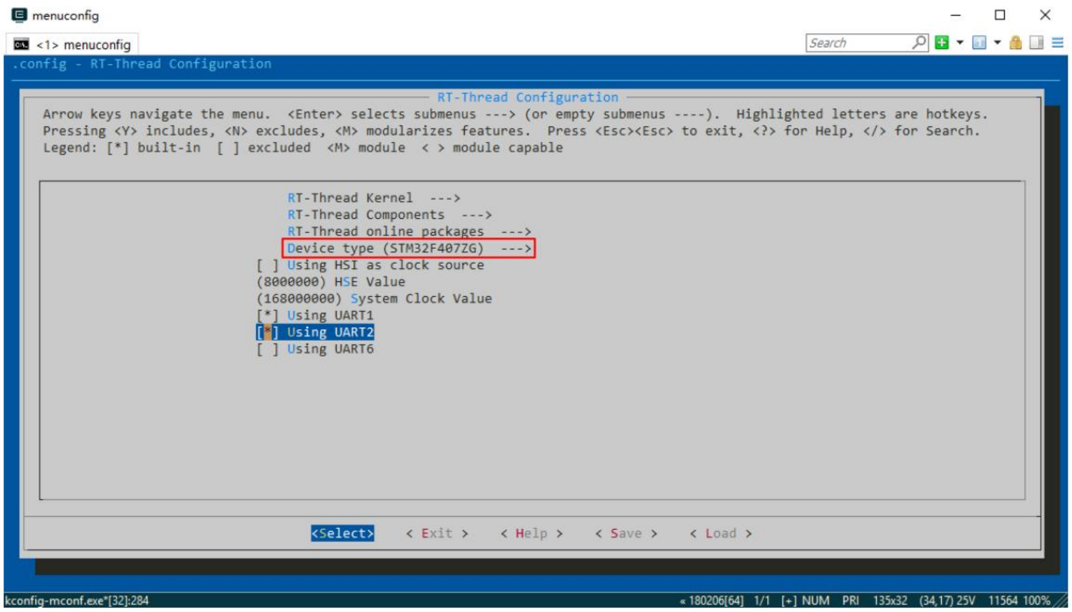

image 3: use *menuconfig* Configuring the Serial Port

3. Enter the command scons –target=mdk5 -s to generate a keil project. After opening the project, change the MCU model to STM32F407ZETx, as shown in the figure:

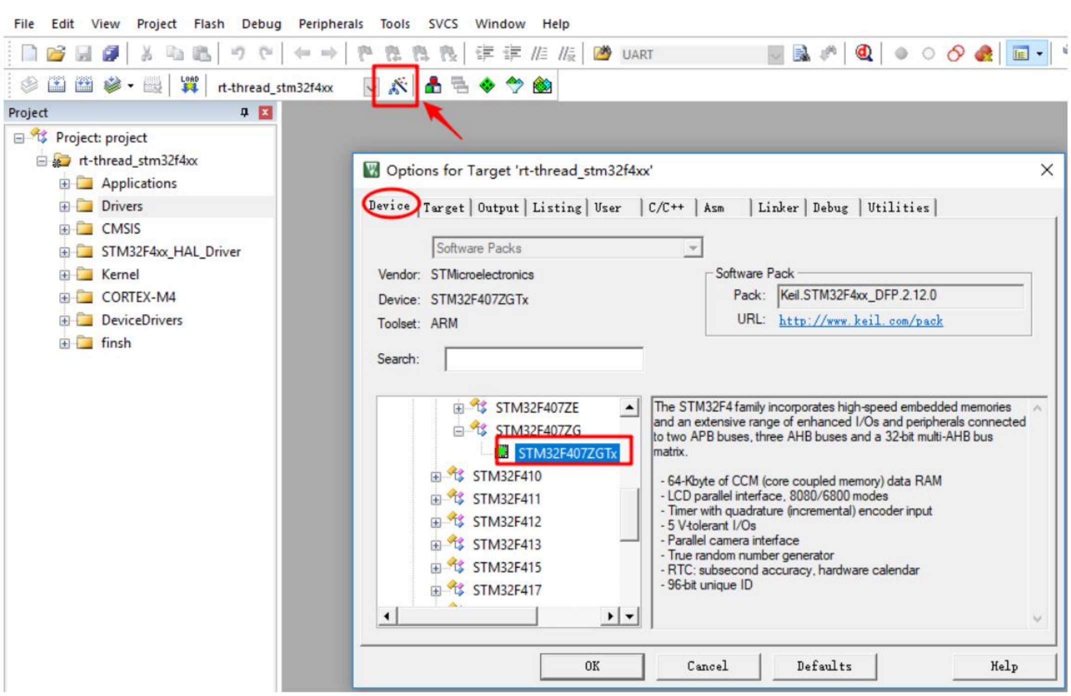

Figure 4: Check chip model

4. Open putty, select the correct serial port, and configure the software parameters to 115200-8-1-N and no flow control. As shown in the figure:

Machine Translated by Google

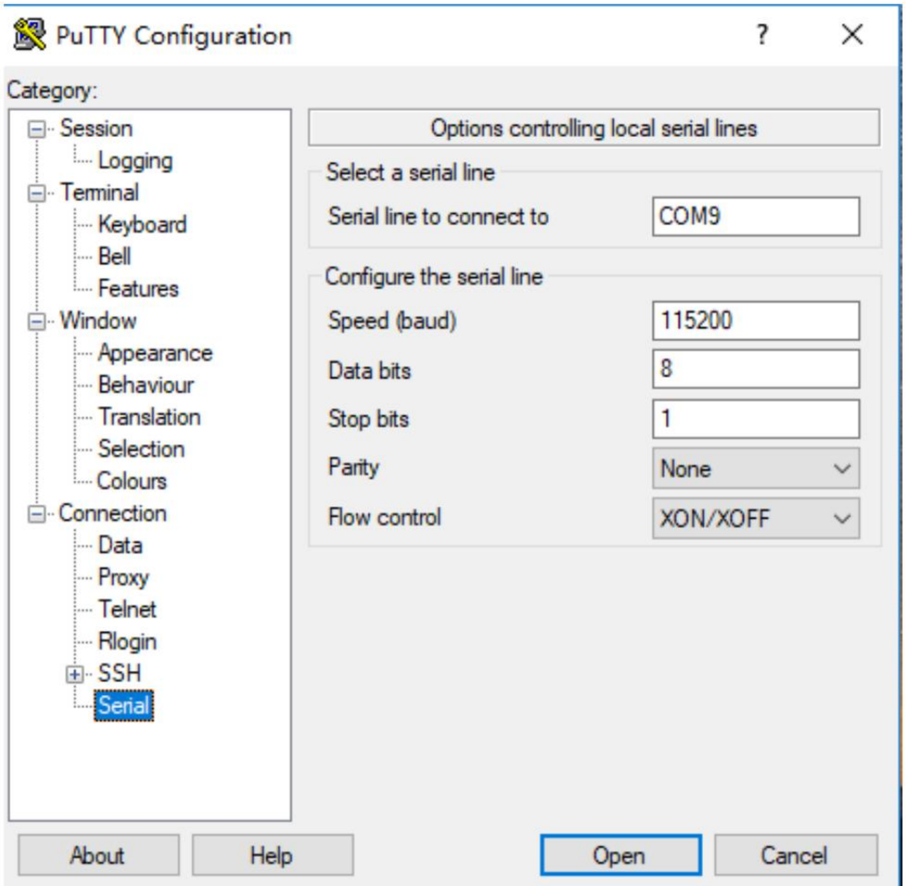

Figure 5: *putty* Configuration

5. Compile and download the program. After pressing reset, you can see the RT-Thread logo log on the terminal connected to serial port 1.

Enter the list_device command to view uart1 and uart2 Character Device, which means the serial port is configured well.

■

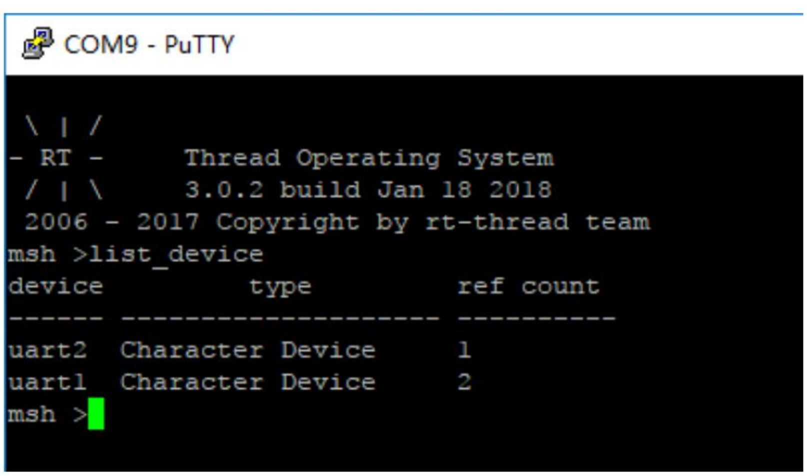

Figure 6: use *list_device* Command View *uart* equipment

3.2 Add serial port related code

Download serial port sample code

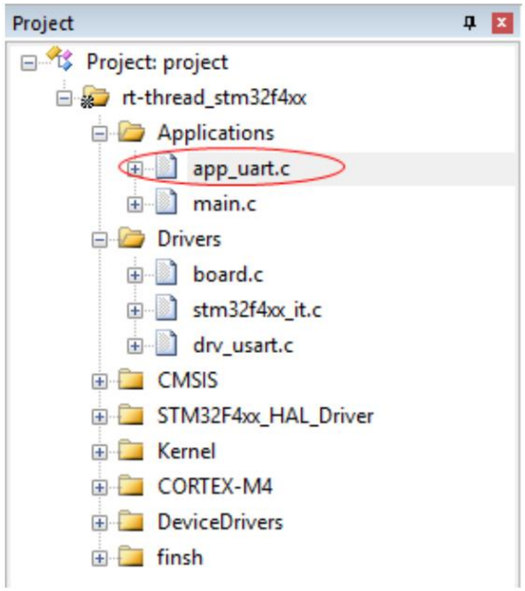

Figure 7: Add sample code to the project

The sample code of this application note is app_uart.c and app_uart.h. The code of app_uart.c is related to the serial port, which is easy to read. app_uart.c provides four functions uart_open, uart_putchar, uart_putstring, and uart_getchar to facilitate the use of the serial port. The code in app_uart.c is independent of the hardware platform, and readers can add it directly to their own projects. Use these functions to write test code in main.c. The source code of main.c is as follows:

```
#include "app_uart.h"
#include "board.h"
void test_thread_entry(void* parameter) {

    rt_uint8_t uart_rx_data; /* Open the
    serial port */
    if (uart_open("uart2") != RT_EOK) {

        rt_kprintf("uart open error.\n");
         while (1) {

            rt_thread_delay(10);
         }

    } /* single character write*/
    uart_putchar('2');
    uart_putchar('0');
    uart_putchar('1');
```

Machine Translated by Google


```
        uart_putchar('8');
        uart_putchar('\n'); /* write
        string*/
        uart_putstring("Hello RT-Thread!\r\n"); while (1) {


            /* Read data */
            uart_rx_data = uart_getchar(); /* misalignment*/

            uart_rx_data = uart_rx_data + 1; /* output */

            uart_putchar(uart_rx_data);
        }

} int main(void) {

        rt_thread_t tid; /* Create
        test thread */ tid =
        rt_thread_create("test",
                                test_thread_entry,
                                RT_NULL,
                                1024,
                                2,
                                10); /
        * Start the thread if creation is successful*/
        if (tid != RT_NULL)
            rt_thread_startup(tid); return 0;

}
```

This program implements the following functions:

1. The test thread test_thread_entry is created and started in the main function.

2. After the test thread calls the uart_open function to open the specified serial port, it first uses the uart_putchar function to send characters and the uart_putstring function to send strings.

3. Then call the uart_getchar function in the while loop to read the received data and save it to the local variable In uart_rx_data, the data is finally output after being shifted.

3.3 Operation Results

Compile, download the code to the board, reset, and connect the terminal software putty to serial port 2 (the software parameters are configured as 115200-8-1-N, no flow control) outputs characters 2, 0, 1, 8 and the string Hello RT-Thread!. Input character 'A', serial port 2 receives it and outputs it after shifting. The experimental phenomenon is shown in the figure:

Figure 8: Experimental phenomena

In the figure, putty is connected to the serial port 2 of the development board as the test serial port.

4 Advanced Reading

The serial port is usually configured to receive interrupts and polling transmission mode. In interrupt mode, the CPU does not need to keep querying and waiting for the serial port related flag register. The serial port triggers an interrupt after receiving data. We process the data in the interrupt service program, which is more efficient. The official bsp of RT-Thread uses this mode by default.

4.1 Which serial port to use

The uart_open function is used to open the specified serial port. It completes the serial port device callback function settings, serial port device opening Initialization of startup and events. The source code is as follows:

```
rt_err_t uart_open(const char *name) {

    rt_err_t res; /* Find
    the serial port device in the system*/
    uart_device = rt_device_find(name); /* Open the
    device after finding it */
    if (uart_device != RT_NULL) {

        res = rt_device_set_rx_indicate(uart_device, uart_intput); /* Check return value */

        if (res != RT_EOK) {

            rt_kprintf("set %s rx indicate error.%d\n",name,res);
            return -RT_ERROR;

        } /* Open the device in readable and writable,
        interrupt mode*/ res = rt_device_open(uart_device, RT_DEVICE_OFLAG_RDWR |
                                              RT_DEVICE_FLAG_INT_RX );
```

```
        /* Check return value */
        if (res != RT_EOK) {

            rt_kprintf("open %s device error.%d\n",name,res);
            return -RT_ERROR;
        }
    }
    else
    {
        rt_kprintf("can't find %s device.\n",name); return -RT_ERROR;

    } /* Initialize event object */
    rt_event_init(&event, "event", RT_IPC_FLAG_FIFO); return RT_EOK;

}
```

The brief process is as follows:

Machine Translated by Google

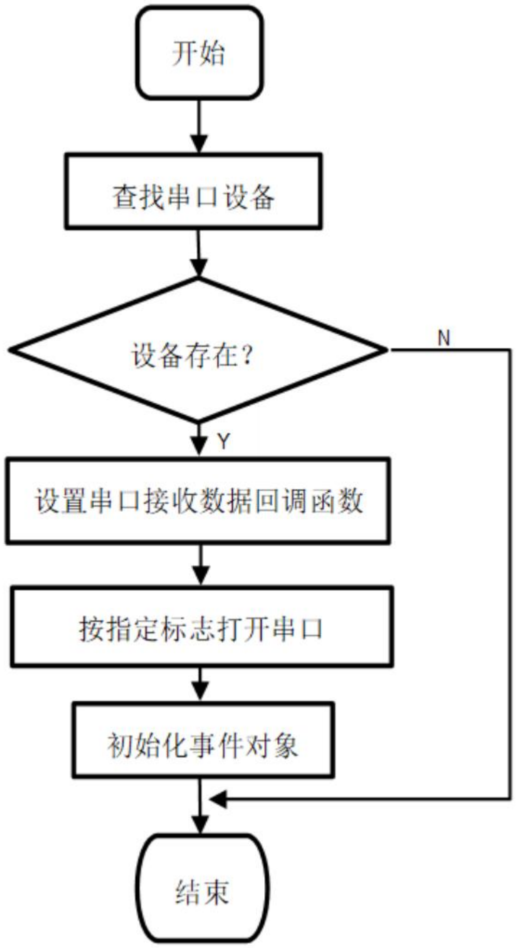

Figure **9:** *uart_open* Function Flowchart

The device operation interfaces used by the uart_open function are: rt_device_find, rt_device_set_rx_indicate, and rt_device_open. The uart_open function first calls rt_device_find to obtain the serial port handle according to the serial port name, and saves it in the static global variable uart_device. All subsequent serial port operations are based on this serial port handle. The name here is determined by calling the registration function rt_hw_serial_register in drv_usart.c, which links the serial port hardware driver with the RT-Thread device management framework.

```
/* register UART2 device */
rt_hw_serial_register(&serial2,
                      "uart2",
                      RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX, uart);
```

Then call rt_device_set_rx_indicate to set the callback function of the serial port receiving interrupt. Finally, call rt_device_open to open the serial port in readable and writable, interrupt receiving mode. Its second parameter is the flag, which should be consistent with the registration function rt_hw_serial_register mentioned above.

```
rt_device_open(uart_device, RT_DEVICE_OFLAG_RDWR | RT_DEVICE_FLAG_INT_RX)
        ;
```

Finally, call rt_event_init to initialize the event. The automatic initialization mechanism is enabled by default in RT-Thread, so users do not need to manually call the serial port initialization function in the application (INIT_BOARD_EXPORT in drv_usart.c implements automatic initialization). The user-implemented serial port hardware driver selected by the macro RT_USING_UARTx will be automatically associated with RT-Thread (rt_hw_serial_register in drv_usart.c implements serial port hardware registration).

4.2 Serial port sending

The uart_putchar function is used to send 1 byte of data. The uart_putchar function actually calls rt_device_write to send a byte, and takes error prevention measures, that is, checks the return value, resends if it fails, and limits the timeout. The source code is as follows:

```
void uart_putchar(const rt_uint8_t c) {

        rt_size_t len = 0; rt_uint32_t
        timeout = 0;
        do
        {
                len = rt_device_write(uart_device, 0, &c, 1); timeout++;

        }
        while (len != 1 && timeout < 500);
}
```

The data flow diagram of calling uart_putchar is as follows:

Figure **10:** *uart_putchar* data flow

When the application calls uart_putchar, the actual calling relationship is: rt_device_write ==> rt_serial_write ==> drv_putc, the final data is sent out through the serial port data register.

4.3 Serial port receiving

The uart_getchar function is used to receive data. The implementation of the uart_getchar function uses the serial port receive interrupt callback Mechanisms and events are used for asynchronous communication, which has blocking characteristics. The relevant source code is as follows:

```
/*Serial port receive event flag*/
#define UART_RX_EVENT (1 << 0) /* Event control
block*/
static struct rt_event event; /* device handle*/

static rt_device_t uart_device = RT_NULL;

/* ÿ ÿ ÿ ÿ */ static
rt_err_t uart_intput(rt_device_t dev, rt_size_t size) {

    /* Send event */
    rt_event_send(&event, UART_RX_EVENT);
    return RT_EOK;
}
```

```
rt_uint8_t uart_getchar(void) {

    rt_uint32_t e; rt_uint8_t
    ch; /* read 1 byte of
    data*/
    while (rt_device_read(uart_device, 0, &ch, 1) != 1)
{
        /* Receive events */
        rt_event_recv(&event, UART_RX_EVENT,RT_EVENT_FLAG_AND |
                              RT_EVENT_FLAG_CLEAR,RT_WAITING_FOREVER, &e);
}
    return ch;
}
```

The uart_getchar function has a while() loop inside. It first calls rt_device_read to read a byte of data. If it does not read, it calls rt_event_recv to wait for the event flag and suspends the calling thread. After the serial port receives a byte of data, an interrupt is generated and the callback function uart_intput is called. The callback function calls rt_event_send to send the event flag to wake up the thread waiting for the event. The data flow diagram of calling the uart_getchar function is as follows:

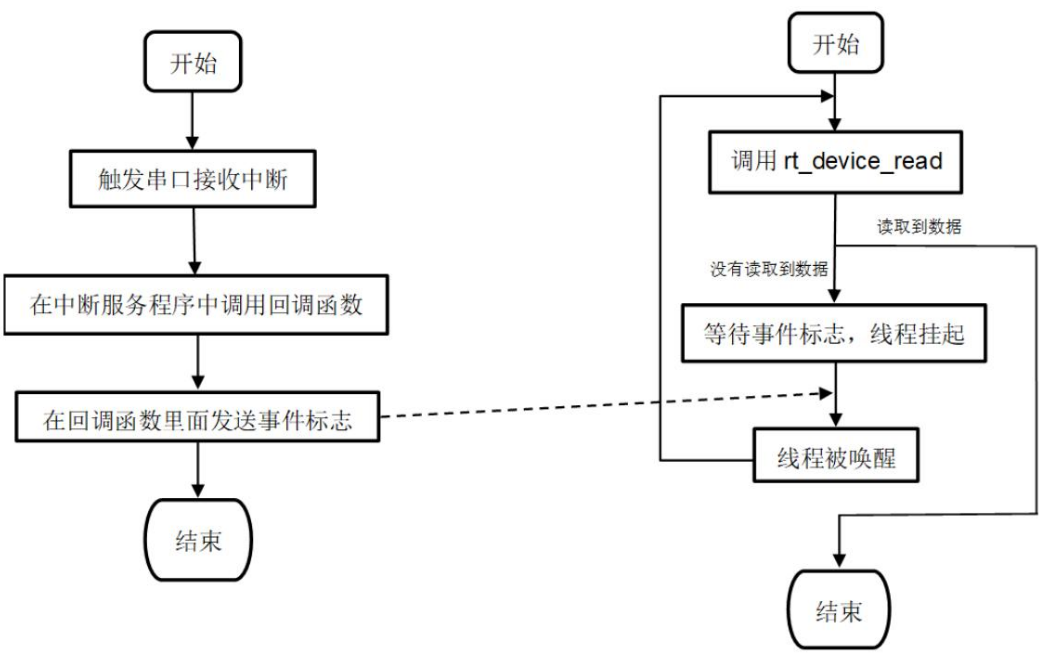

Figure **11:** *uart_getchar* data flow

When the application calls uart_getchar, the actual calling relationship is: rt_device_read ==> rt_serial_read ==> drv_getc, and finally the data is read from the serial port data register.

4.4 Relationship between I/O device management framework and serial port

RT-Thread automatic initialization function calls hw_usart_init ==> rt_hw_serial_register ==> rt_device_register completes the serial port hardware initialization, thereby connecting the device operation interface and the serial port driver. We can use the device operation interface to operate the serial port.

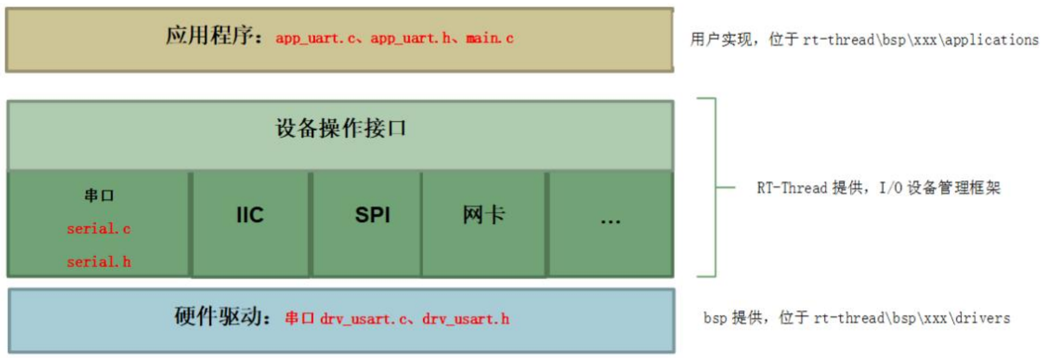

Figure **12:** Serial port driver and device management framework connection

For more information about the I/O device management framework and serial port driver implementation details, please refer to the RT-Thread Programming Manual. Chapter **6** I /**O** Device Management

Online view address: link

5 API Reference

Note: app_uart.h file does not belong to RT-Thread.

5.1 API List

API	head File
uart_open	app_uart.h
uart_getchar	app_uart.h
uart_putchar	app_uart.h
rt_event_send	rt-thread\include\rtthread.h
rt_event_recv	rt-thread\include\rtthread.h
rt_device_find	rt-thread\include\rtthread.h
rt_device_set_rx_indicate	rt-thread\include\rtthread.h
rt_device_open	rt-thread\include\rtthread.h
rt_device_write	rt-thread\include\rtthread.h

API	head File
rt_device_read	rt-thread\include\rtthread.h

5.2 Core API Detailed Explanation

5.2.1. rt_device_open()

Function prototype

```
rt_err_t rt_device_open (rt_device_t dev, rt_uint16_t oflag)
```

Function parameters

parameter	describe
dev	Device handle, used to operate the device
order	Access Mode

Function Returns

return value	describe
RT_EOK	normal
- RT_EBUSY	If the parameters specified when registering the device include RT_DEVICE_FLAG_STANDALONE, this device Duplicate opening will not be allowed

This function opens the device based on the device handle.

oflag supports the following parameters:

```
RT_DEVICE_OFLAG_CLOSE /* The device has been closed (for internal use) */
RT_DEVICE_OFLAG_RDONLY /* Open the device in read-only mode */
RT_DEVICE_OFLAG_WRONLY /* Open the device in write-only mode */
RT_DEVICE_OFLAG_RDWR /*             /* Open the device in read-write mode */
RT_DEVICE_OFLAG_OPEN     Device is already opened (for internal use) */
RT_DEVICE_FLAG_STREAM /* Device is opened in stream mode*/
RT_DEVICE_FLAG_INT_RX /* Device is turned on in interrupt receive mode*/
RT_DEVICE_FLAG_DMA_RX /* Device is turned on in DMA receive mode*/
```

```
RT_DEVICE_FLAG_INT_TX /* Device is turned on in interrupt transmit mode*/
RT_DEVICE_FLAG_DMA_TX /* Device is turned on in DMA transmit mode*/
```

Precautions

If the upper layer application needs to set the device's receive callback function, it must use INT_RX or DMA_RX

The device is opened in this way, otherwise the function will not be called back.

5.2.2. rt_device_find()

Function prototype

```
rt_device_t rt_device_find(const char *name)
```

Function parameters

parameter	describe
name	Device Name

Function Returns

If the corresponding device is found, the corresponding device handle will be returned; otherwise, RT_NULL will be returned.

This function finds a device by the specified device name.

5.2.3. rt_device_set_rx_indicate()

Function prototype

```
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))
```

Function parameters

parameter	describe
dev	Device handle
rx_ind	Receive interrupt callback function

Function Returns

return value	describe
RT_EOK	success

This function can set a callback function, which is called back when the hardware device receives data to notify the application that data has arrived.

When the hardware device receives data, it will call back this function and pass the length of the received data in the size parameter.

To the upper application. The upper application thread should read data from the device immediately after receiving the instruction.

5.2.4. rt_device_read()

Function prototype

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

Function parameters

parameter	describe
dev	Device handle
pos	Read data offset
buffer	Memory buffer pointer. The read data will be saved in the buffer.
size	The size of the data to be read

Function Returns

Returns the actual size of the data read (if it is a character device, the returned size is in bytes; if it is a block device,

The returned size is in blocks); if it returns 0, you need to read the errno of the current thread to determine the error status.

This function reads data from the device.

Calling this function will get data from the device dev and store it in the buffer.

The maximum length is size. pos has different meanings depending on the device class.

5.2.5. rt_device_write()

Function prototype

```
rt_size_t rt_device_write(rt_device_t dev,
                          rt_off_t       pos,
                          const void *buffer,
                          rt_size_t size)
```

Function parameters

parameter	describe
dev	Device handle
pos	Write data offset
buffer	Memory buffer pointer where the data to be written is placed
size	The size of the data to be written

Function Returns

Returns the actual size of the data written (if it is a character device, the returned size is in bytes; if it is a block device, the returned size is in bytes).

If the return value is 0, you need to read the errno of the current thread to determine the error status.

This function will write the data in buffer buffer to device dev. The maximum length of the written data is size.

pos has different meanings depending on the device class.

This function writes data to the device.