# Serial Device Application Notes

**RT-THREAD** Document Center



**WWW.RT-THREAD.ORG**

**Friday 28th September, 2018**

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 is given, and a code example verified on the Zhengdian Atom STM32F4 Explorer development board is 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 or UART) is one of the most widely used communication interfaces. On bare-metal platforms or RTOS platforms without a device management framework, we usually only need to write serial port hardware initialization code according to the official manual. With the introduction of RT-Thread, a real-time operating system with a device management framework, the use of serial ports is significantly different from that of bare-metal or other RTOS platforms. RT-Thread includes a built-in 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 presents sample code for developing a serial port data transmission and reception program using RT-Thread's device operation interface. This code is then verified on the Zhengdian Atom STM32F4 Explorer development board. The implementation of the sample code is then analyzed, and finally, the connection between the RT-Thread device management framework and serial ports is described in depth.

# 2 Problem Description

RT-Thread provides a simple I/O device management framework that divides I/O device management into three layers: the application layer, the I/O device management layer, and the hardware driver layer. Applications obtain the correct device driver through RT-Thread's device operation interface and then use this device driver to exchange data (or control) with the underlying I/O hardware.

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 provides the serial port configuration process and application code examples based on the Zhengdian Atom STM32F4 Explorer development board. 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 the serial port 1 and PC is enough; serial port 2 needs to use a USB to serial port module to connect to the PC.

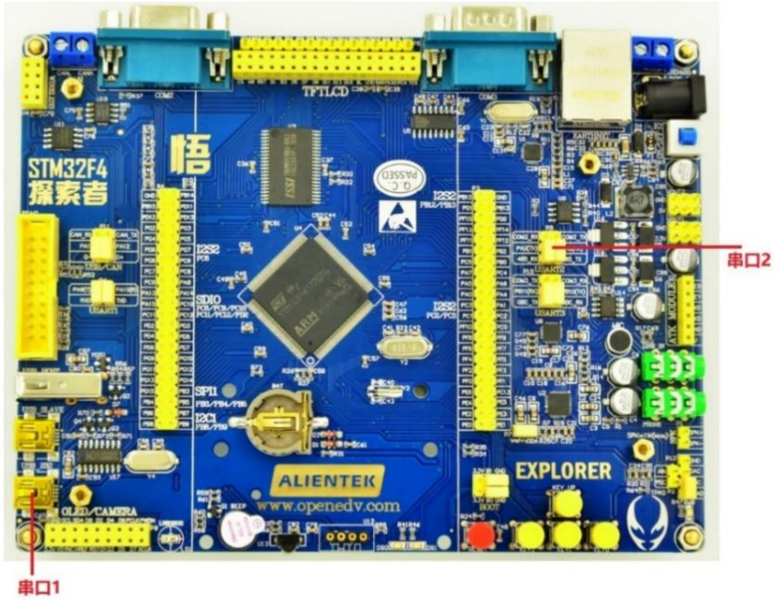


Figure 2: Punctual Atom *STM32F4* Explorer

## 3.1 Prepare and configure the project

1. Download the 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

The device name for console is uart1.

(2) Check Using UART1, Using UART2, select the chip model as STM32F407ZG, and the clock source as external

8MHz, as shown in the figure:

Machine Translated by Google



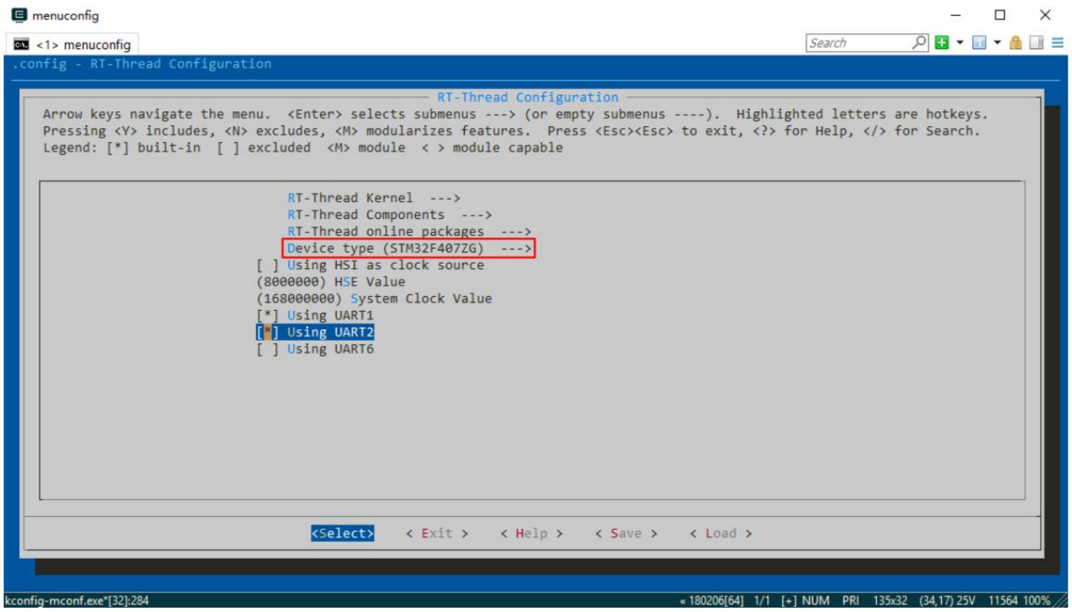


Figure 3: use *menuconfig* Configuring the serial port

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

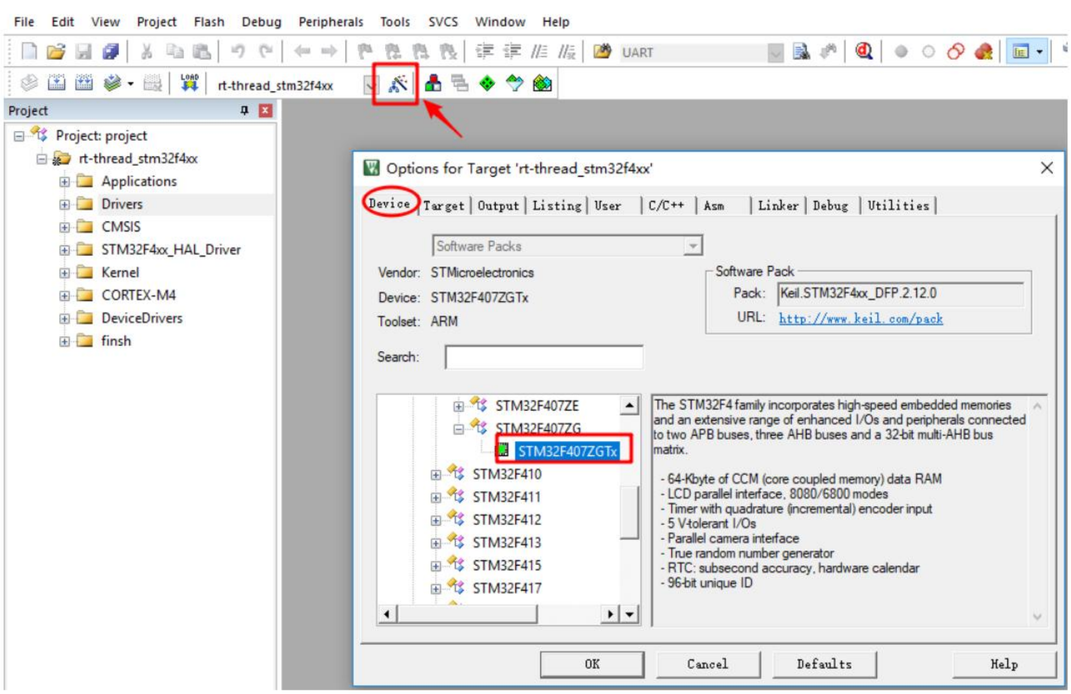


Figure 4: Check chip model

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:

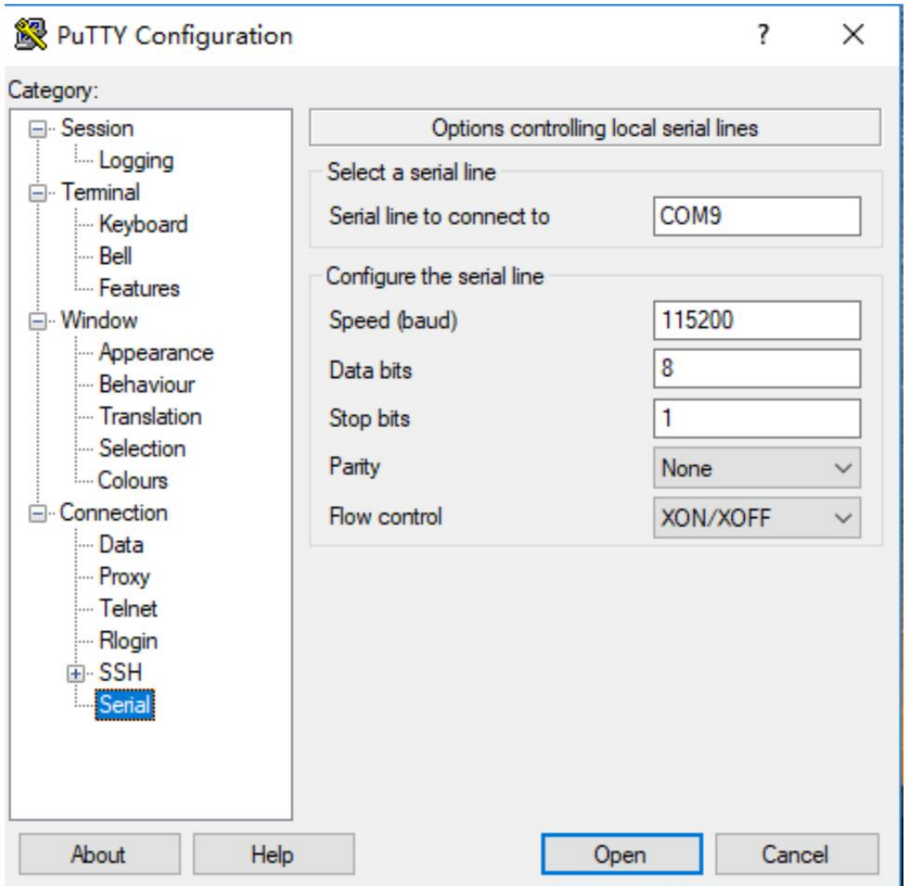


Figure 5: *Putty* Configuration

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.

■

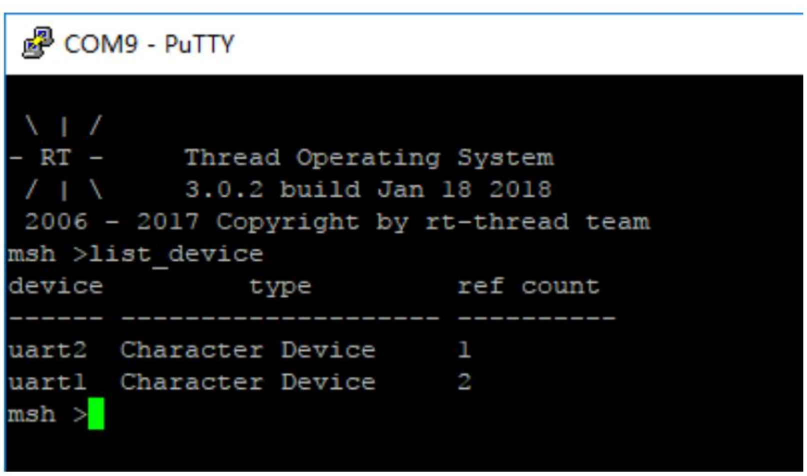


Figure 6: use *list_device* Command View *uart* equipment

## 3.2 Add serial port related code

Download the serial port sample code

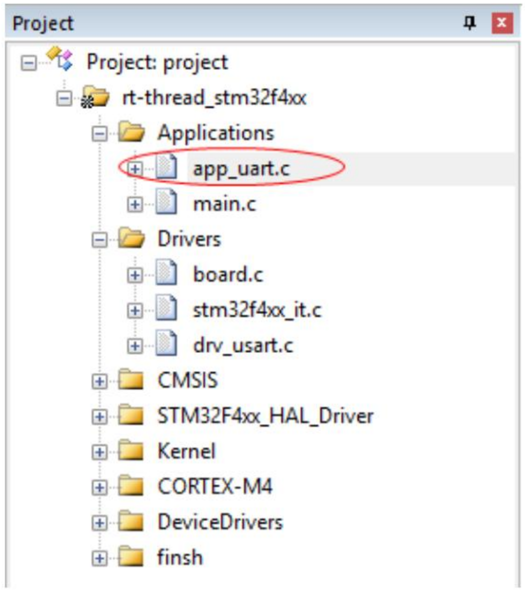


Figure 7: Add sample code to the project

This application note's sample code is contained in app_uart.c and app_uart.h. The code for serial port operations in app_uart.c is easy to read. app_uart.c provides four functions: uart_open, uart_putchar, uart_putstring, and uart_getchar, to facilitate serial port usage. The code in app_uart.c is hardware-platform-independent and can be directly added to your own project. These functions are used to write test code in main.c. The source code for 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');
```

```
    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 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 dislocated.

### 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!". The character 'A' is received by serial port 2, which then shifts the character and outputs it. The experimental results are shown in the figure:


Figure 8: Experimental phenomenon

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

## 4 Advanced Reading

Serial ports are typically configured in receive interrupt and polled transmit mode. In interrupt mode, the CPU doesn't need to constantly query the serial port's flag registers. Receiving data triggers an interrupt, and the interrupt service routine processes the data, resulting in higher efficiency. This mode is the default setting in the official RT-Thread BSP.

### 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 the startup event. 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, writable and
        interrupt mode*/ res = rt_device_open(uart_device, RT_DEVICE_OFLAG_RDWR |
                                          RT_DEVICE_FLAG_INT_RX );
```

Machine Translated by Google



```
        /* 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



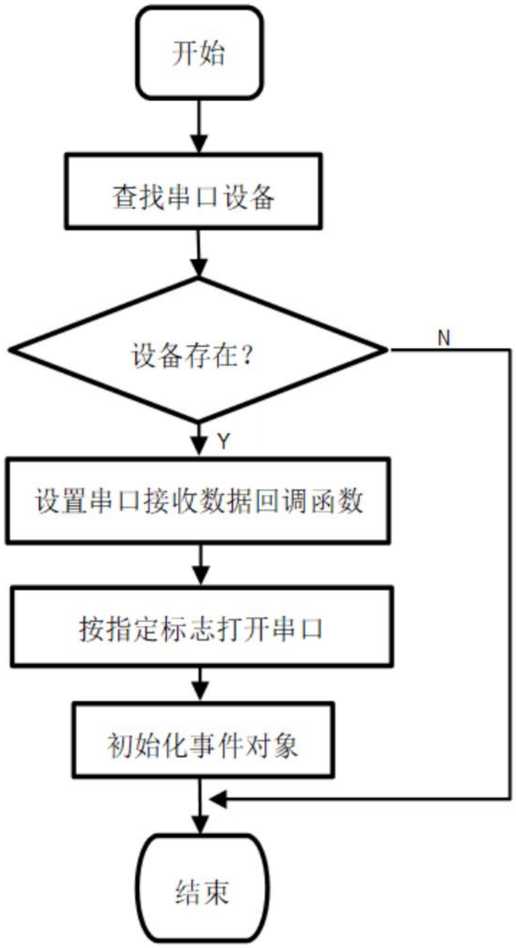


Figure 9: *uart_open* Function Flowchart

The uart_open function uses the following device operation interfaces: 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 based on the serial port name. This handle is stored in the static global variable uart_device. All subsequent serial port operations are based on this handle. The name is determined by calling the registration function rt_hw_serial_register in drv_usart.c, which connects the serial port hardware driver to 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);
```

Next, call rt_device_set_rx_indicate to set the callback function for the serial port receive interrupt. Finally, call rt_device_open to open the serial port in read-write, interrupt-receive mode. Its second parameter is a 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. RT-Thread automatically initializes the serial port by default, so users don't need to manually call the serial port initialization function in their applications (INIT_BOARD_EXPORT in drv_usart.c implements automatic initialization). The user-implemented serial port hardware driver selected by the RT_USING_UARTx macro is 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-proofing measures, namely checking the return value and resending if it fails, and setting a 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:

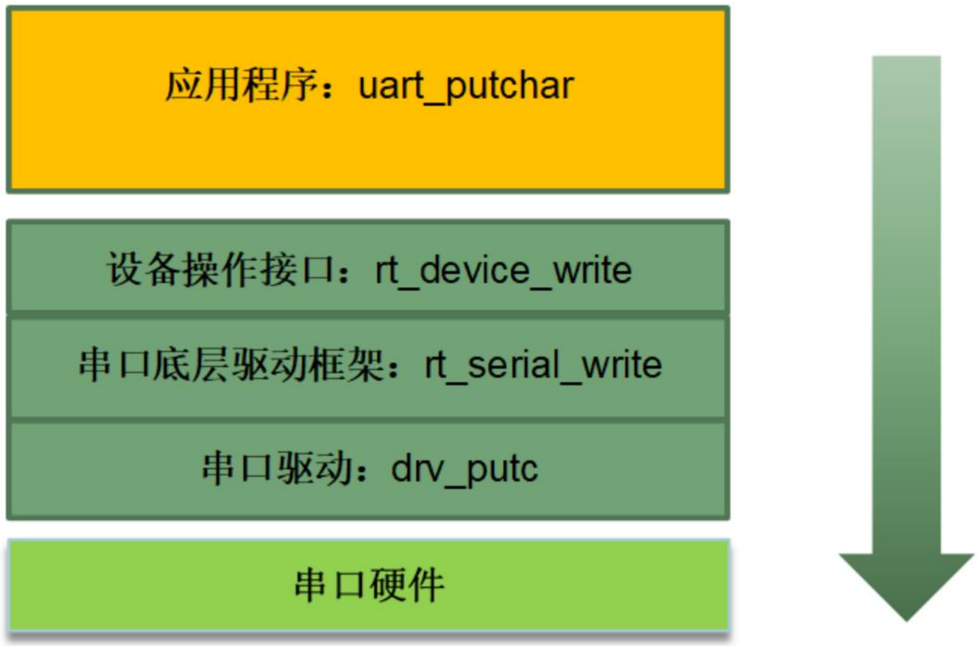


Figure **10:** *uart_putchar* Data Flow

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;

/* callback function
*/ 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;
}
```

Machine Translated by Google



```
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 contains a while() loop. It first calls rt_device_read to read a byte of data. If no data is read, it calls rt_event_recv to wait for the event flag, suspending 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 and wake up the thread waiting for the event. The data flow diagram of the uart_getchar function call is as follows:

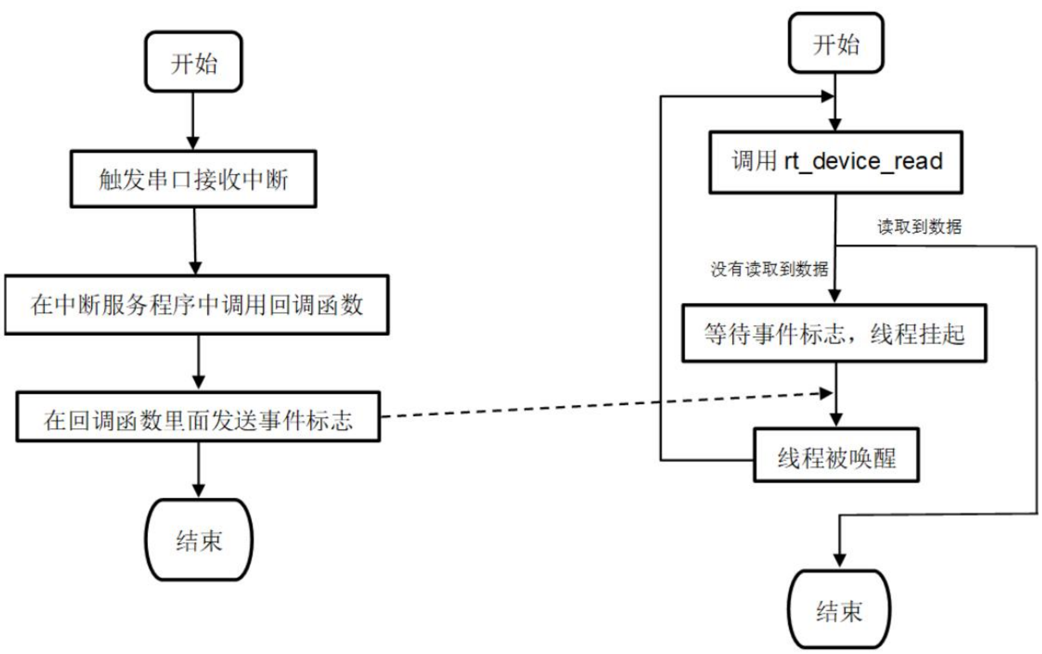


Figure **11:** *uart_getchar* Data Flow

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 ==> in sequence rt_device_register completes the serial port hardware initialization, thus connecting the device operation interface and the serial port driver. We can use the device operation interface to operate the serial port.

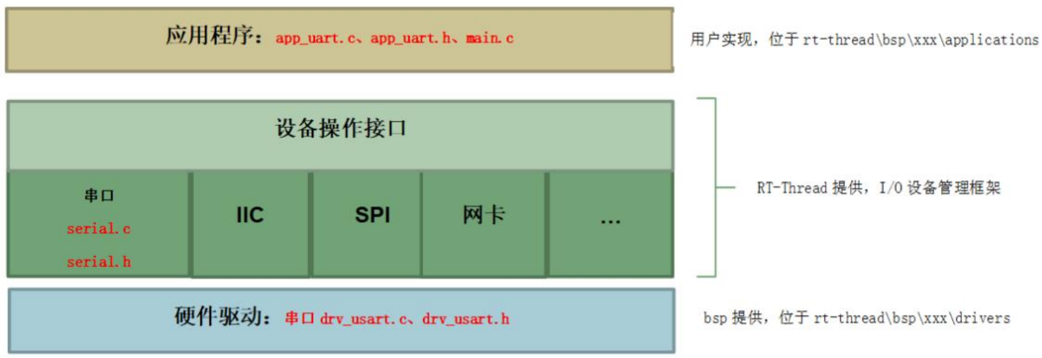


Figure 12: Serial port driver and device management framework connection

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 | header files |
|---|---|
| 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 | header files |
|---|---|
| rt_device_read | rt-thread\include\rtthread.h |

## 5.2 Detailed explanation of core API

### 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 |
| oflag | 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     The device has been opened (for internal use) */
RT_DEVICE_FLAG_STREAM /* Device 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 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 sets a callback function that is called 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 return 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 type.

#### 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 | Size of written data |

Function Returns

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

This function writes data to the device.