

---

## General Purpose **GPIO** Device Application Notes

---

**RT-THREAD** Documentation Center

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



[WWW.RT-THREAD.ORG](http://WWW.RT-THREAD.ORG)

Friday 28th September, 2018

Table of contents

Table of contents	i
1 Purpose and structure of this paper . . . . .	1
1.1 Purpose and background of this paper . . . . .	1
1.2 Structure of this paper . . . . .	1
2 Problem description. . . . .	1
3 Problem solving. . . . .	2
3.1 Prepare and configure the project. . . . .	3
3.2 GPIO output configuration. . . . .	4
3.3 GPIO input configuration. . . . .	6
3.4 GPIO interrupt configuration. . . . .	7
3.5 The relationship between the I/O device management framework and general GPIO devices. . . . .	9
4 References. . . . .	9
4.1 All APIs related to this article . . . . .	9
4.1.1. API List . . . . .	10
4.1.2. Detailed explanation of core API. . . . .	10
4.1.2.1. rt_pin_mode() . . . . .	10
4.1.2.2. rt_pin_write() . . . . .	11
4.1.2.3. rt_pin_read() . . . . .	12
4.1.2.4. rt_pin_attach_irq() . . . . .	12
4.1.2.5. rt_pin_detach_irq() . . . . .	14
4.1.2.6. rt_pin_irq_enable() . . . . .	14

This application note describes how to use the general GPIO device driver of RT-Thread, including the configuration of the driver and the application of related APIs. It also provides code examples verified on the Atom STM32F4 Explorer development board.

## 1 Purpose and structure of this paper

### 1.1 Purpose and Background of this Paper

In order to provide users with a general API for operating GPIO and facilitate application development, RT-Thread introduces a general GPIO device driver. It also provides an Arduino-style API for operating GPIO, such as setting GPIO mode and output level, reading GPIO input level, and configuring GPIO external interrupts. This article explains how to use RT-Thread's general GPIO device driver.

### 1.2 Structure of this paper

This article first describes the basic situation of RT-Thread general GPIO device driver, then gives the code example verified on the Zhengdian Atom STM32F4 Explorer development board, and finally describes in detail the parameter values and precautions of the general GPIO device driver API.

## 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 a low-level driver framework to lower-level devices. For general GPIO devices, applications can access them through the device operation interface or directly through the general GPIO device driver. Generally speaking, we use the second method. So how do we use the general GPIO device driver in RT-Thread to operate GPIO?



Figure 1: RT-Thread

Device Management Framework

### 3. Problem Solving

This article is based on the Zhengdian Atom STM32F4 Explorer development board and gives a specific application example of general GPIO devices. Code, including the use of pin input, output and external interrupt. Due to the universality of RT-Thread upper application API. Therefore, the codes are not limited to a specific hardware platform and users can easily port them to other platforms. The MCU used in the STM32F4 Explorer Development Board is STM32F407ZGT6, with 2 LEDs and 4 independent. The LEDs are connected to GPIOF9 and GPIOF10 of the MCU, and the KEY0 button is connected to GPIOE4, KEY1. The KEY2 button is connected to GPIOE2, and the WK\_UP button is connected to GPIOA0. The LEDs are all lit at low level. The independent buttons KEY0, KEY1, and KEY2 are at low level when pressed; WK\_UP is at high level when pressed. Level.

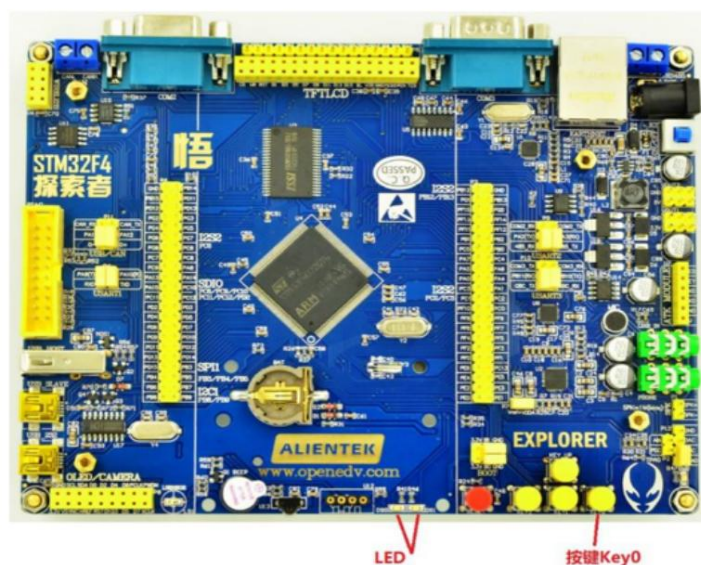


figure 2: Zhengdian Atom Development Board

### 3.1 Prepare and configure the project

1. Download [RT-Thread source code](#)

2. Download [the GPIO device driver sample code](#)

3. 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) In the `menuconfig` configuration interface, select `RT-Thread Components` → `Device Drivers` →

Using generic GPIO device drivers, as shown in the figure:

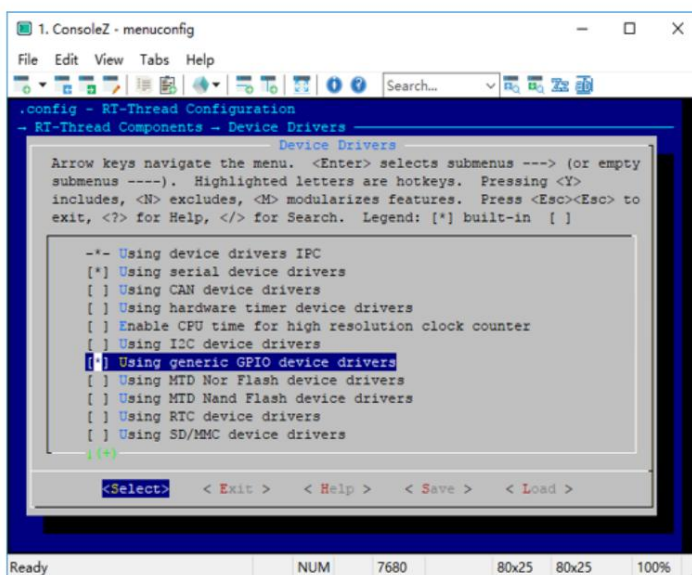


Figure 3: `menuconfig` Open in `GPIO` drive

(2) Enter the command `scons -target=mdk5 -s` to generate the `mdk5` project. Replace the `main.c` included with the sample code with

Delete `main.c` in `bsp`, as shown in the figure:

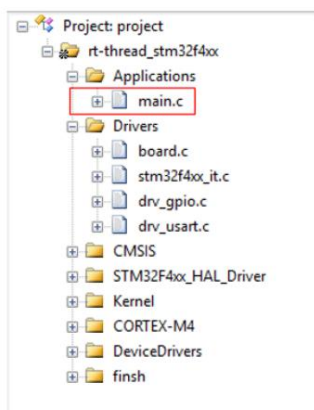
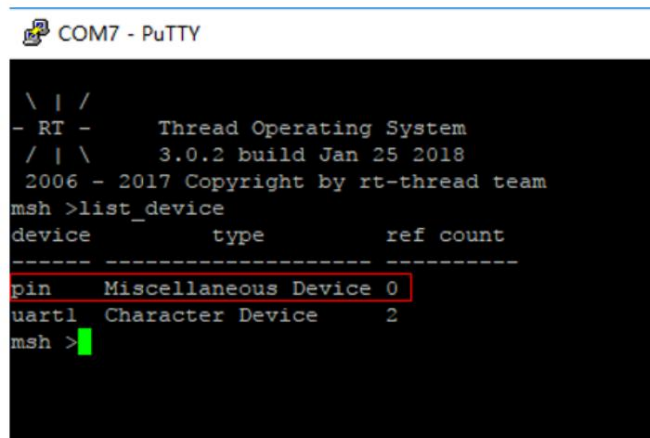


Figure 4: Add test code

- (3) Compile and download the program. Enter the `list_device` command in the terminal to see that the device is `pin` and the type is `Miscellaneous Device` means that the general GPIO device driver has been added successfully.



```

\ | /
- RT -   Thread Operating System
/ | \   3.0.2 build Jan 25 2018
2006 - 2017 Copyright by rt-thread team

msh >list_device
device      type      ref count
-----
pin         Miscellaneous Device 0
uart1      Character Device 2
msh >

```

Figure 5: Check `pin` equipment

The following are 3 general GPIO device driver API application examples: GPIO output, GPIO input, GPIO external interrupt, these codes have been verified on the Zhengdian Atom STM32F4 Explorer development board.

### 3.2 GPIO output configuration

Example 1: Configure GPIO as output to light up the LED. According to the schematic diagram, GPIOF9 is connected to the onboard red LED.

The silkscreen is DS0; GPIOF10 is connected to the onboard green LED, the silkscreen is DS1. GPIOF9 lights up when the output is low DS0, GPIOF9 outputs high level, DS0 is off; GPIOF10 outputs low level, DS1 is on, GPIOF10 outputs

If the output is high level, DS1 will not light up.

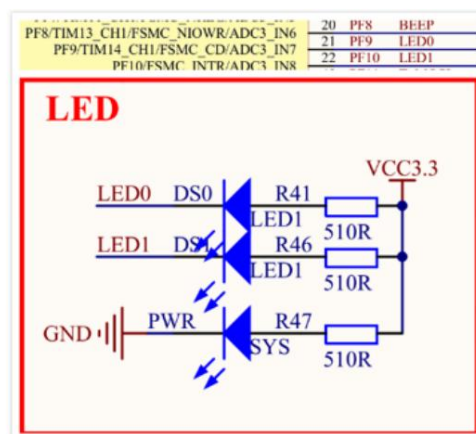


Figure 6: LED Schematic

```

#define LED0 21 //PF9--21, find PF9 in the pin_index pins[] in the drv_gpio.c file
                Number 21
#define LED1 22 //PF10--22, find PF10 number 22 in the pin_index pins[] in the drv_gpio.c file

void led_thread_entry(void* parameter) {

    // Set the pin to output mode
    rt_pin_mode(LED0, PIN_MODE_OUTPUT); //
    Set the pin to output mode
    rt_pin_mode(LED1, PIN_MODE_OUTPUT);
    while (1) {

        // Output low level, LED0 is on
        rt_pin_write(LED0, PIN_LOW); //
        Output low level, LED1 is on
        rt_pin_write(LED1, PIN_LOW); //
        suspend for 500ms
        rt_thread_delay(rt_tick_from_millisecond(500));

        // Output high level, LED0 off
        rt_pin_write(LED0, PIN_HIGH); // Output
        high level, LED1 off
        rt_pin_write(LED1, PIN_HIGH); //
        Suspend for 500ms
        rt_thread_delay(rt_tick_from_millisecond(500));
    }
}

```

In the thread entry function `led_thread_entry`, `rt_pin_mode` is first called to set the pin mode to output mode, and then it enters the `while(1)` loop, calling `rt_pin_write` every 500ms to change the GPIO output level. The following is the code for creating a thread:

```

rt_thread_t tid; // thread handle /*
create led thread */ tid =
rt_thread_create("led", led_thread_entry,

                RT_NULL,
                1024,
                3,
                10); /

/* Start the thread if creation is successful*/
if (tid != RT_NULL)
    rt_thread_startup(tid);

```

Compile and download the program, and we will see the LED flashing at 500ms intervals.

### 3.3 GPIO input configuration

Example 2: Configure GPIOE3 and GPIOE2 as pull-up inputs, GPIOA0 as pull-down input, and detect key signals. According to the schematic diagram, GPIOE3 is connected to key KEY1. When the key is pressed, GPIOE3 should read a low level, and when the key is not pressed, GPIOE3 should read a high level; GPIOE2 is connected to key KEY2. When the key is pressed, GPIOE2 should read a low level, and when the key is not pressed, GPIOE2 should read a high level; GPIOA0 is connected to key WK\_UP. When the key is pressed, GPIOA0 should read a high level, and when the key is not pressed, GPIOA0 should read a low level.

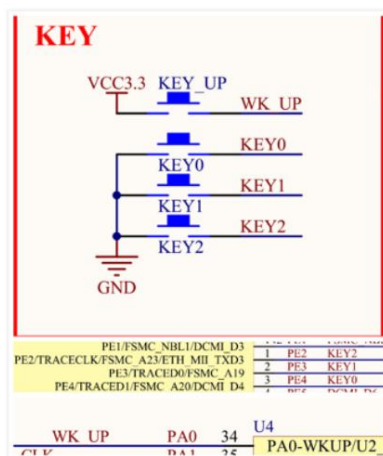


Figure 7: Button schematic diagram

```
#define KEY1      2 //PE3--2, found in the drv_gpio.c file pin_index pins[]
                PE3 number 2
#define KEY2      1 //PE2--1, find PE2 number 1 in pin_index pins[] in drv_gpio.c file

#define WK_UP 34 //PA0--34, find PA0 number 34 in the pin_index pins[] file of drv_gpio.c

void key_thread_entry(void* parameter) {

    //PE2, PE3 set pull-up input
    rt_pin_mode(KEY1, PIN_MODE_INPUT_PULLUP);
    rt_pin_mode(KEY2, PIN_MODE_INPUT_PULLUP);
    //PA0 is set as pull-down
    input rt_pin_mode(WK_UP, PIN_MODE_INPUT_PULLDOWN);

    while (1) {

        // Detect low level, that is, button 1 is pressed
```



```

        if (rt_pin_read(KEY1) == PIN_LOW) {

            rt_kprintf("key1 pressed\n");

        } // Low level detected, that is, button 2 is pressed
        if (rt_pin_read(KEY2) == PIN_LOW) {

            rt_kprintf("key2 pressed\n");

        } // High level detected, that is, key wp is pressed
        if (rt_pin_read(WK_UP) == PIN_HIGH) {

            rt_kprintf("WK_UP pressed\n");

        } // Suspend 10ms
        rt_thread_delay(rt_tick_from_millisecond(10));
    }
}

```

In the thread entry function `key_thread_entry`, first call `rt_pin_mode` to set the pin GPIOE3 to pull-up input mode. In this way, when the user presses the key KEY1, the level read by GPIOE3 is low; when the key is not pressed, the level read by GPIOE3 is high. Then enter the `while(1)` loop and call `rt_pin_read` to read the level of the pin GPIOE3. If the low level is read, it means that the key KEY1 is pressed, and the string "key1 pressed!" is printed in the terminal. The key input status is detected every 10ms. The following is the code to create a thread:

```

rt_thread_t tid; /* Create
key thread*/ tid =
    rt_thread_create("key", key_thread_entry,

                    RT_NULL,
                    1024,
                    2,
                    10); /

    * Start the thread if creation is successful*/
    if (tid != RT_NULL)
        rt_thread_startup(tid);

```

Compile and download the program. We press the user button on the development board and the terminal will print prompt characters.

### 3.4 GPIO interrupt configuration

Example 3: Configure GPIO to external interrupt mode, falling edge trigger, and detect key signals. According to the schematic diagram, GPIOE4 Connected to key KEY0, the MCU should detect a falling edge when the key is pressed.

```

#define KEY0          3 //PE4--3, find PE4 in the gpio.c file pin_index pins[]
                        Number 3
void hdr_callback(void *args)// callback function {

    char *a = args; // Get parameters
    rt_kprintf("key0 down! %s\n",a);
}

void irq_thread_entry(void* parameter) {

    // Pull up input
    rt_pin_mode(KEY0, PIN_MODE_INPUT_PULLUP); // Bind
    interrupt, falling edge mode, callback function name is hdr_callback
    rt_pin_attach_irq(KEY0, PIN_IRQ_MODE_FALLING, hdr_callback, (void*)"
        callback
args"); //
    Enable interrupt
    rt_pin_irq_enable(KEY0, PIN_IRQ_ENABLE);

}

```

In the thread entry function `irq_thread_entry`, first call `rt_pin_attach_irq` to set the pin GPIOE4 to the falling edge interrupt mode, bind the interrupt callback function, and pass in the string "callback args". Then call `rt_pin_irq_enable` to enable the interrupt, so that when the key KEY0 is pressed, the MCU will detect the falling edge of the level and trigger the external interrupt. The callback function `hdr_callback` will be called in the interrupt service program, and the passed parameters and prompt information will be printed in the callback function. The following is the code to create a thread:

```

rt_thread_t tid; // thread handle /* create
irq thread */ tid =
rt_thread_create("exirq",
                irq_thread_entry,
                RT_NULL,
                1024,
                4,
                10); /

/* Start thread if creation is successful*/
if (tid != RT_NULL)
    rt_thread_startup(tid);

```

Compile and download the program. We press KEY0 and the terminal will print a prompt character.

### 3.5 Relationship between I/O device management framework and general GPIO devices

RT-Thread automatic initialization function calls `rt_hw_pin_init` ==> `rt_device_pin_register` in turn ==> `rt_device_register` completes GPIO hardware initialization. `rt_device_register` registers device type `RT_Device_Class_Miscellaneous`, which means miscellaneous devices, so we can use a unified API operation GPIO.

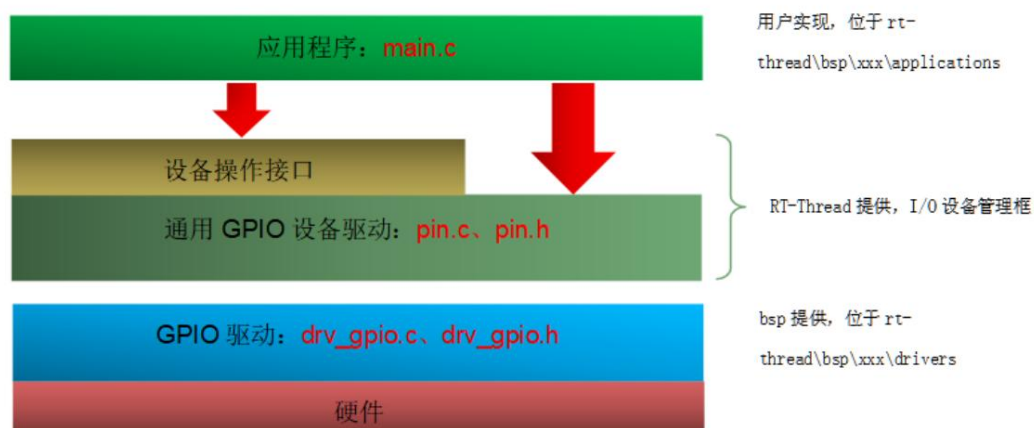


Figure 8: General GPIO Driver and device management framework connection

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](#)

## 4 References

### 4.1 All relevant APIs in this article

If the user application code wants to use the RT-Thread GPIO driver interface, it needs to enable the GPIO driver in menuconfig.

Reference the header file `rtdevice.h`.

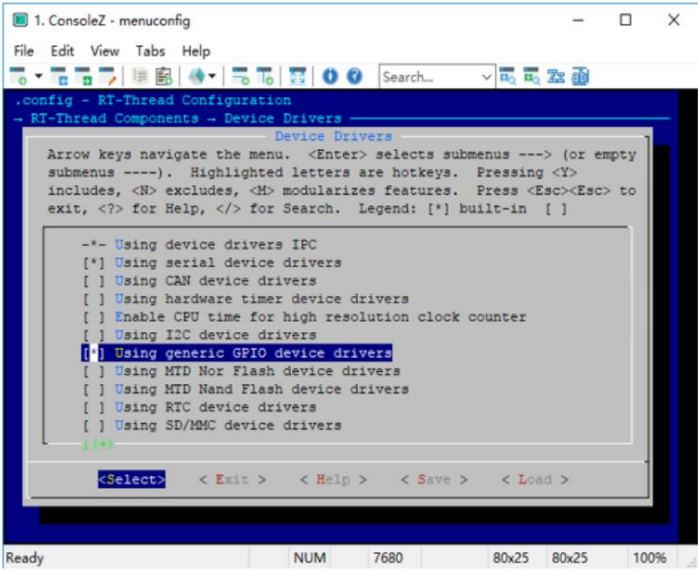


Figure 9: menuconfig Open in GPIO drive

API List

API	head File
rt_pin_mode	rt-thread\components\drivers\include\drivers\pin.h
rt_pin_write	rt-thread\components\drivers\include\drivers\pin.h
rt_pin_read	rt-thread\components\drivers\include\drivers\pin.h
rt_pin_attach_irq	rt-thread\components\drivers\include\drivers\pin.h
rt_pin_detach_irq	rt-thread\components\drivers\include\drivers\pin.h
rt_pin_irq_enable	rt-thread\components\drivers\include\drivers\pin.h

4.1.2. Detailed explanation of core API

4.1.2.1. rt\_pin\_mode() function prototype

void rt\_pin\_mode(rt\_base\_t pin, rt\_base\_t mode)

Function parameters

parameter	describe
pin	Pin Number
mode	model

Function returns None

This function sets the pin mode.

The pin number is defined by the driver and can be found in pin\_index pins[] of drv\_gpio.c .

Take STM32F407ZGT6 as an example. The number of pins of this chip is 100. The following code can be found in pin\_index pins[]:

```
drv_gpio.c
352 #endif
353 #if (STM32F4xx_PIN_NUMBERS == 100 && !defined(STM32F469xx) && !defined(STM32F479xx))
354     __STM32_PIN_DEFAULT,
355     __STM32_PIN(1, E, 2),
356     __STM32_PIN(2, E, 3),
357     __STM32_PIN(3, E, 4),
358     __STM32_PIN(4, E, 5),
359     __STM32_PIN(5, E, 6),
360     __STM32_PIN_DEFAULT,
361     __STM32_PIN(7, C, 13),
362     __STM32_PIN(8, C, 14),
363     __STM32_PIN(9, C, 15),
364     __STM32_PIN_DEFAULT,
365     __STM32_PIN_DEFAULT,
366     __STM32_PIN_DEFAULT,
367     __STM32_PIN_DEFAULT,
368     __STM32_PIN_DEFAULT,
369     __STM32_PIN(15, C, 0),
370     __STM32_PIN(16, C, 1),
371     __STM32_PIN(17, C, 2),
372     __STM32_PIN(18, C, 3),
373     __STM32_PIN_DEFAULT,
374     __STM32_PIN_DEFAULT,
375     __STM32_PIN_DEFAULT,
376     __STM32_PIN_DEFAULT,
```

Figure 10: pin serial number

STM32\_PIN(1, E, 2) means GPIOE2 is numbered 1, and STM32\_PIN(9, C, 15) means GPIOC15 is numbered 9, and so on. The first parameter of STM32\_PIN() is the pin number, and the second parameter is Port, the third parameter is the pin number.

The mode can be one of the following five:

- PIN\_MODE\_OUTPUT output, see drv\_gpio.c source code for specific mode implementation, this article uses Push-pull output
- PIN\_MODE\_INPUT Input
- PIN\_MODE\_INPUT\_PULLUP Pull-up input
- PIN\_MODE\_INPUT\_PULLDOWN Pull-down input
- PIN\_MODE\_OUTPUT\_OD Open-drain output

4.1.2.2. rt\_pin\_write() function prototype

```
void rt_pin_write(rt_base_t pin, rt_base_t value)
```

Function parameters

parameter	describe
pin	Pin Number
value	Level logic value, can take one of two values, PIN_LOW low level, PIN_HIGH High level

Function returns None

This function can set the pin output level.

#### 4.1.2.3. rt\_pin\_read() function prototype

```
int rt_pin_read(rt_base_t pin)
```

Function parameters

parameter	describe
pin	Pin Number

Function Returns

return value	describe
PIN_LOW	Low level
PIN_HIGH	High level

This function reads the input pin level.

#### 4.1.2.4. rt\_pin\_attach\_irq() function prototype

```
rt_err_t rt_pin_attach_irq( rt_int32_t pin, rt_uint32_t mode,  
                             void (*hdr)(void *args), void *args)
```

Function parameters

parameter	describe
pin	Pin Number
mode	Interrupt trigger mode
hdr	Interrupt callback function, the user needs to define this function by himself, and its return value is void
args	Parameters of the interrupt callback function, set to RT_NULL when not needed

Function Returns	
return value	describe
RT_EOK	success
RT_ENOSYS	No system
RT_EBUSY	busy

The interrupt trigger mode can take one of the following three values:

PIN\_IRQ\_MODE\_RISING rising edge trigger PIN\_IRQ\_MODE\_FALLING falling edge trigger

PIN\_IRQ\_MODE\_RISING\_FALLING Edge trigger (both rising and falling edges are triggered)

This function can bind the interrupt callback function.

Bind interrupt callback function passing string example:

```
rt_pin_attach_irq(3, PIN_IRQ_MODE_FALLING, hdr_callback, (void*)"callback
args");
void hdr_callback(void *args)
{
    char *a = args;

    rt_kprintf("%s", a);
}
```

The output is "callback args".

Example of passing a value:

```
rt_pin_attach_irq(3, PIN_IRQ_MODE_FALLING, hdr_callback, (void*)6);
void hdr_callback(void *args)
```

```
{
    Int a = (int)args;

    rt_kprintf("%d",a);
}
```

The output is 6.

4.1.2.5. **rt\_pin\_detach\_irq()** function prototype

```
rt_err_t rt_pin_detach_irq(rt_int32_t pin)
```

Function parameters

parameter	describe
pin	Pin Number

Function Returns

return value	describe
RT_EOK	success
RT_ENOSYS	Error

This function can disengage the pin interrupt callback function.

4.1.2.6. **rt\_pin\_irq\_enable()** function prototype

```
rt_err_t rt_pin_irq_enable(rt_base_t pin, rt_uint32_t enabled)
```

Function parameters

parameter	describe
pin	Pin Number
enabled	Status, can be one of 2 values: PIN_IRQ_ENABLE is enabled, PIN_IRQ_DISABLE Disable



Function Returns

return value	describe
RT_EOK	success
RT_ENOSYS	Error