
通用 MCU RT_Thread 设备注册应用笔记

简介

本文档主要描述 N32G45x 系列、N32G4FR 系列、N32WB452 系列、N32G43x 系列、N32L40x 系列、N32L43x 系列 MCU 的 RT_Thread 设备注册，便于使用者快速熟悉 RT_Thread 设备驱动。

目录

1 概述	1
1.1 简述	1
2 设备注册	2
2.1 I/O 设备	2
2.1.1 I/O 设备介绍	2
2.1.2 创建和注册 I/O 设备	3
2.1.3 访问 I/O 设备	3
2.1.4 查找设备	4
2.1.5 初始化设备	4
2.1.6 打开和关闭设备	4
2.1.7 控制设备	5
2.1.8 读写设备	5
2.1.9 数据收发回调	6
2.2 PIN 设备	7
2.2.1 PIN 简介	7
2.2.2 访问 PIN 设备	7
2.2.3 设置引脚模式	8
2.2.4 设置引脚电平	8
2.2.5 读取引脚电平	8
2.2.6 绑定引脚中断回调函数	8
2.2.7 使能引脚中断	9
2.2.8 脱离引脚中断回调函数	9
2.3 SPI 设备	10
2.3.1 SPI 简介	10
2.3.2 挂载 SPI 设备	11
2.3.3 配置 SPI 设备	12
2.3.4 访问 SPI 设备	12
2.3.5 查找 SPI 设备	13
2.3.6 自定义传输数据	13
2.3.7 传输一次数据	14
2.3.8 发送一次数据	14
2.3.9 接收一次数据	15
2.3.10 连续两次发送数据	15
2.3.11 先发送后接收数据	16
2.3.12 特殊使用场景	17
2.3.13 获取总线	18
2.3.14 选中片选	18
2.3.15 增加一条消息	18
2.3.16 释放片选	18
2.3.17 释放总线	19

2.4 UART 设备	20
2.4.1 UART 简介	20
2.4.2 访问串口设备	20
2.4.3 查找串口设备	20
2.4.4 打开串口设备	20
2.4.5 控制串口设备	21
2.4.6 发送数据	22
2.4.7 设置发送完成回调函数	22
2.4.8 设置接收回调函数	23
2.4.9 接收数据	23
2.4.10 关闭串口设备	23
2.5 I2C 设备	24
2.5.1 I2C 简介	24
2.5.2 访问 I2C 总线设备	24
2.5.3 查找 I2C 总线设备	24
2.5.4 数据传输	24
2.6 ADC 设备	26
2.6.1 ADC 简介	26
2.6.2 访问 ADC 设备	26
2.6.3 查找 ADC 设备	26
2.6.4 使能 ADC 通道	26
2.6.5 读取 ADC 通道采样值	27
2.6.6 关闭 ADC 通道	27
2.7 DAC 设备	28
2.7.1 DAC 简介	28
2.7.2 访问 DAC 设备	28
2.7.3 查找 DAC 设备	28
2.7.4 使能 DAC 通道	28
2.7.5 设置 DAC 通道输出值	29
2.7.6 关闭 DAC 通道	29
2.8 CAN 设备	30
2.8.1 CAN 简介	30
2.8.2 访问 CAN 设备	30
2.8.3 查找 CAN 设备	30
2.8.4 打开 CAN 设备	30
2.8.5 控制 CAN 设备	31
2.8.6 发送数据	31
2.8.7 设置接收回调函数	31
2.8.8 接收数据	31
2.8.9 关闭 CAN 设备	32
2.9 HWTIMER 设备	33
2.9.1 定时器简介	33
2.9.2 访问硬件定时器设备	33
2.9.3 查找定时器设备	33

2.9.4 打开定时器设备	33
2.9.5 设置超时回调函数	34
2.9.6 控制定时器设备	34
2.9.7 设置定时器超时值	34
2.9.8 获取定时器当前值	34
2.9.9 关闭定时器设备	35
2.10 WATCHDOG 设备	36
2.10.1 WATCHDOG 简介	36
2.10.2 访问看门狗设备	36
2.10.3 查找看门狗	36
2.10.4 初始化看门狗	36
2.10.5 控制看门狗	37
2.10.6 在空闲线程钩子函数里喂狗	37
2.10.7 关闭看门狗	37
3 版本历史	38
4 声明	39

1 概述

1.1 简述

本文档主要描述 N32G45x 系列、N32G4FR 系列、N32WB452 系列、N32G43x 系列、N32L40x 系列、N32L43x 系列 MCU 的 RT_Thread 设备注册，便于使用者快速熟悉 RT_Thread 设备驱动。

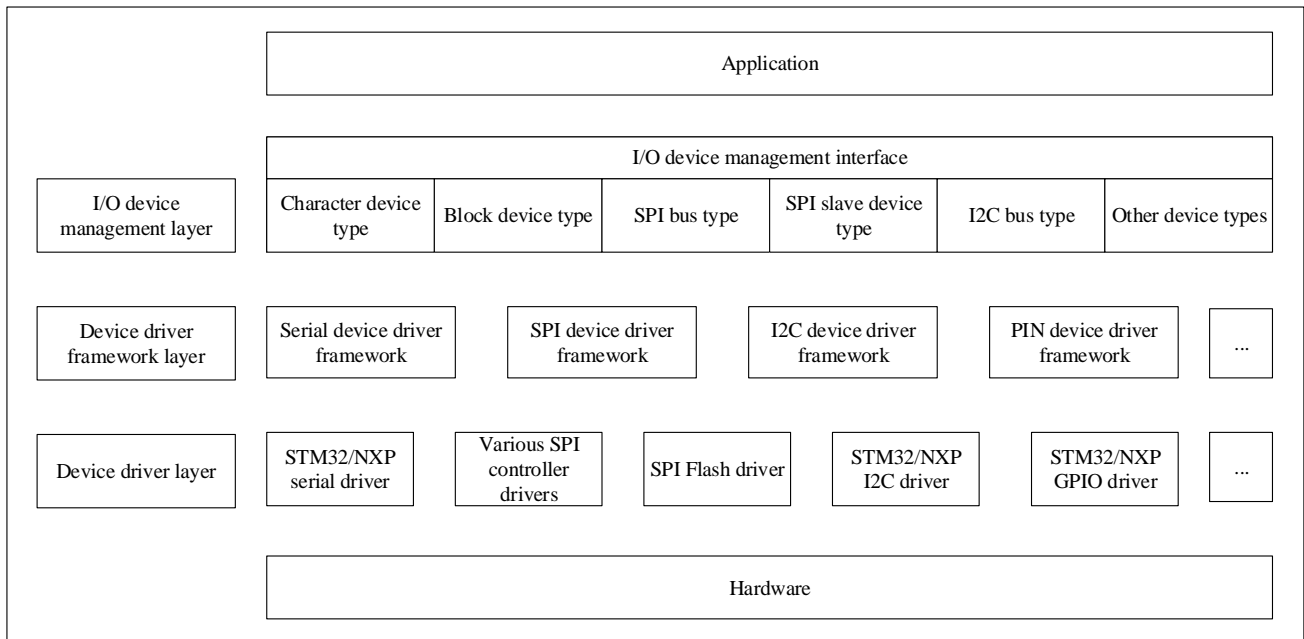
2 设备注册

2.1 I/O 设备

2.1.1 I/O 设备介绍

RT-Thread 提供了一套简单的 I/O 设备模型框架，如图 2-1 所示，它位于硬件和应用程序之间，共分成三层，从上到下分别是 I/O 设备管理层、设备驱动框架层、设备驱动层。

图 2-1 I/O 设备模型框架



应用程序通过 I/O 设备管理接口获得正确的设备驱动，然后通过这个设备驱动与底层 I/O 硬件设备进行数据（或控制）交互。

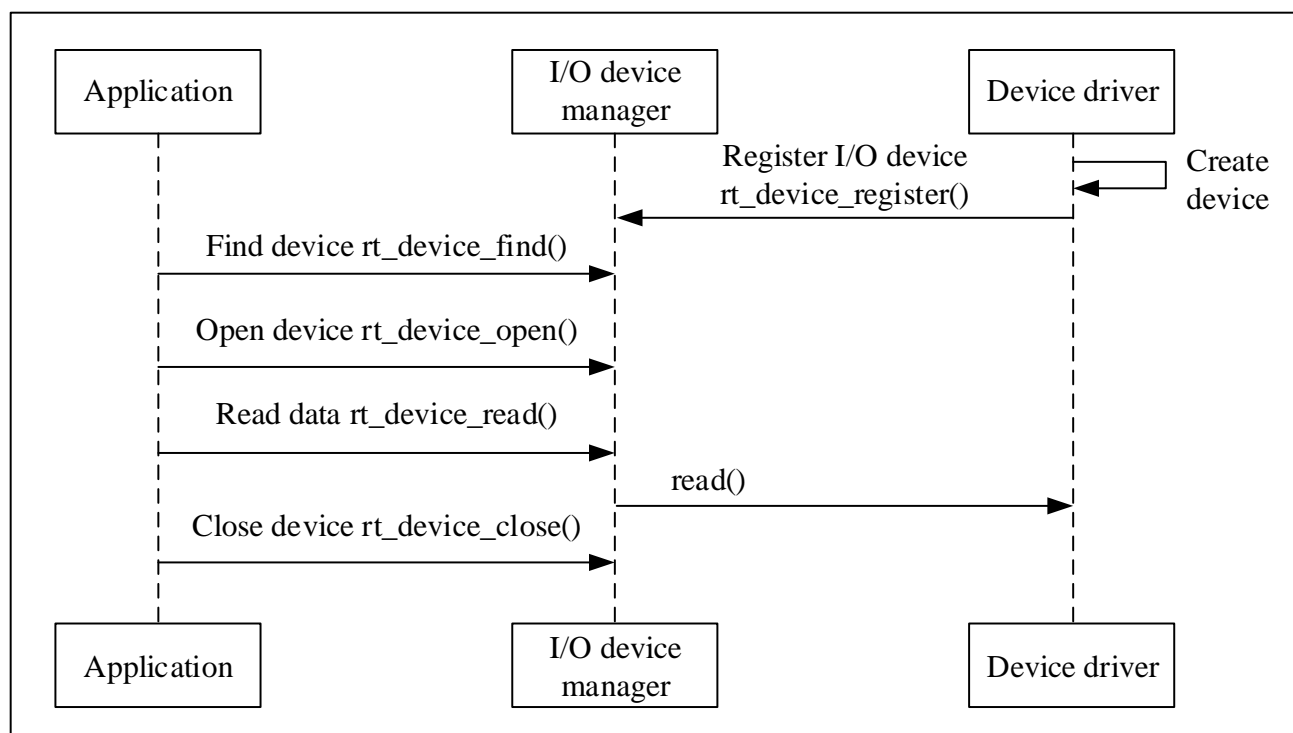
I/O 设备管理层实现了对设备驱动程序的封装。应用程序通过 I/O 设备层提供的标准接口访问底层设备，设备驱动程序的升级、更替不会对上层应用产生影响。这种方式使得设备的硬件操作相关的代码能够独立于应用程序而存在，双方只需关注各自的功能实现，从而降低了代码的耦合性、复杂性，提高了系统的可靠性。

设备驱动框架层是对同类硬件设备驱动的抽象，将不同厂家的同类硬件设备驱动中相同的部分抽取出来，将不同部分留出接口，由驱动程序实现。

设备驱动层是一组驱使硬件设备工作的程序，实现访问硬件设备的功能。它负责创建和注册 I/O 设备，对于操作逻辑简单的设备，可以不经设备驱动框架层，直接将设备注册到 I/O 设备管理器中，使用序列图如图 2-2 所示，主要有以下 2 点：

- 设备驱动根据设备模型定义，创建出具备硬件访问能力的设备实例，将该设备通过 `rt_device_register()` 接口注册到 I/O 设备管理器中。
- 应用程序通过 `rt_device_find()` 接口查找到设备，然后使用 I/O 设备管理接口来访问硬件。

图 2-2 I/O 设备模型框架



2.1.2 创建和注册 I/O 设备

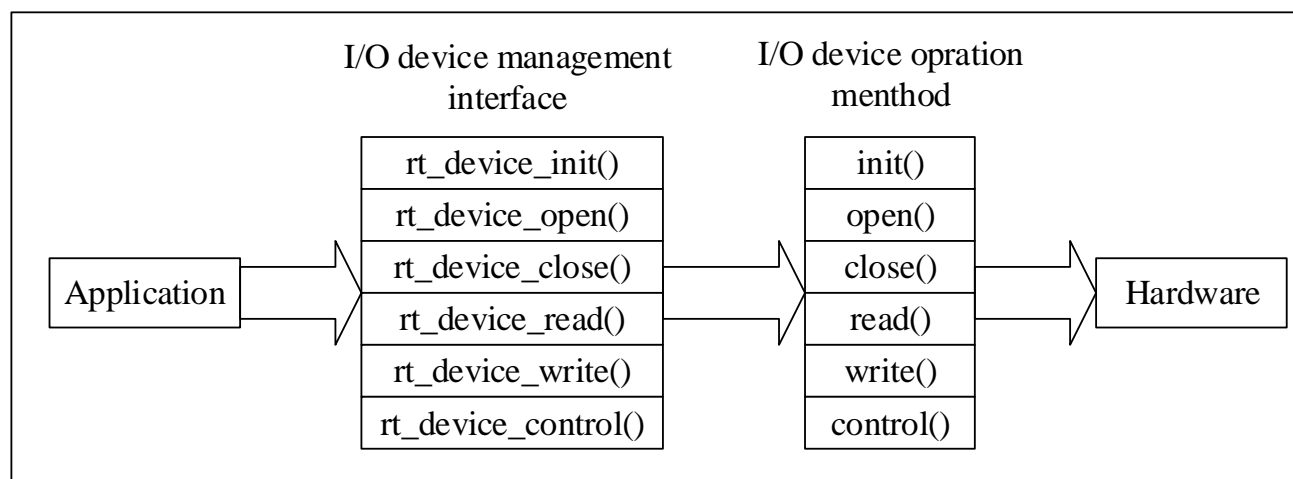
驱动层负责创建设备实例，并注册到 I/O 设备管理器中，可以通过静态声明的方式创建设备实例，也可以用下面的接口进行动态创建：

rt_device_t rt_device_create(int type, int attach_size)	
参数	描述
type	设备类型
attach_size	用户数据大小
返回	——
设备句柄	创建成功
RT_NULL	创建失败，动态内存分配失败

2.1.3 访问 I/O 设备

应用程序通过 I/O 设备管理接口来访问硬件设备，当设备驱动实现后，应用程序就可以访问该硬件。I/O 设备管理接口与 I/O 设备的操作方法的映射关系如图 2-3 所示：

图 2-3 I/O 设备接口



2.1.4 查找设备

应用程序根据设备名称获取设备句柄，进而可以操作设备。查找设备函数如下所示：

rt_device_t rt_device_find(const char* name)	
参数	描述
name	设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

2.1.5 初始化设备

获得设备句柄后，应用程序可使用如下函数对设备进行初始化操作：

rt_err_t rt_device_init(rt_device_t dev)	
参数	描述
dev	设备句柄
返回	——
RT_EOK	设备初始化成功
错误码	设备初始化失败

2.1.6 打开和关闭设备

通过设备句柄，应用程序可以打开和关闭设备，打开设备时，会检测设备是否已经初始化，没有初始化则会默认调用初始化接口初始化设备。通过如下函数打开设备：

rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags)	
参数	描述
dev	设备句柄
oflags	设备打开模式标志

返回	——
RT_EOK	设备打开成功
-RT_EBUSY	如果设备注册时指定的参数中包括 RT_DEVICE_FLAG_STANDALONE 参数，此设备将不允许重复打开
其他错误码	设备打开失败

通过如下函数关闭设备：

rt_err_t rt_device_close(rt_device_t dev)	
参数	描述
dev	设备句柄
返回	——
RT_EOK	关闭设备成功
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

2.1.7 控制设备

通过命令控制字，应用程序也可以对设备进行控制，通过如下函数完成：

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
参数	描述
dev	设备句柄
cmd	命令控制字，这个参数通常与设备驱动程序相关
arg	控制的参数
返回	——
RT_EOK	函数执行成功
-RT_ENOSYS	执行失败，dev 为空
其他错误码	执行失败

2.1.8 读写设备

应用程序从设备中读取数据可以通过如下函数完成：

rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)	
参数	描述
dev	设备句柄
pos	读取数据偏移量
buffer	内存缓冲区指针，读取的数据将会被保存在缓冲区中
size	读取数据的大小
返回	——
读到数据的实际大小	如果是字符设备，返回大小以字节为单位，如果是块设备，返回的大小以块为单位
0	需要读取当前线程的 errno 来判断错误状态

向设备中写入数据，可以通过如下函数完成：

rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)	
参数	描述
dev	设备句柄
pos	写入数据偏移量
buffer	内存缓冲区指针，放置要写入的数据
size	写入数据的大小
返回	——
写入数据的实际大小	如果是字符设备，返回大小以字节为单位；如果是块设备，返回的大小以块为单位
0	需要读取当前线程的 <code>errno</code> 来判断错误状态

2.1.9 数据收发回调

当硬件设备收到数据时，可以通过如下函数回调另一个函数来设置数据接收指示，通知上层应用线程有数据到达：

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))	
参数	描述
dev	设备句柄
rx_ind	回调函数指针
返回	——
RT_EOK	设置成功

该函数的回调函数由调用者提供。当硬件设备接收到数据时，会回调这个函数并把收到的数据长度放在 `size` 参数中传递给上层应用。上层应用线程应在收到指示后，立刻从设备中读取数据。

在应用程序调用 `rt_device_write()` 写入数据时，如果底层硬件能够支持自动发送，那么上层应用可以设置一个回调函数。这个回调函数会在底层硬件数据发送完成后(例如 DMA 传送完成或 FIFO 已经写入完毕产生完成中断时)调用。可以通过如下函数设置设备发送完成指示：

rt_err_t rt_device_set_tx_complete(rt_device_t dev, rt_err_t (*tx_done)(rt_device_t dev, void *buffer))	
参数	描述
dev	设备句柄
tx_done	回调函数指针
返回	——
RT_EOK	设置成功

调用这个函数时，回调函数由调用者提供，当硬件设备发送完数据时，由驱动程序回调这个函数并把发送完成的数据块地址 `buffer` 作为参数传递给上层应用。上层应用（线程）在收到指示时会根据发送 `buffer` 的情况，释放 `buffer` 内存块或将其作为下一个写数据的缓存。

2.2 PIN 设备

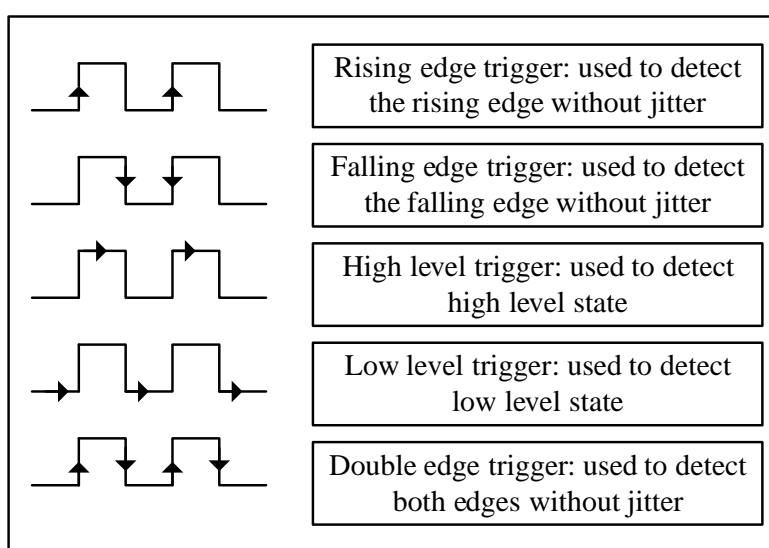
2.2.1 PIN 简介

芯片上的引脚一般分为 4 类：电源、时钟、控制与 I/O，I/O 口在使用模式上又分为 General Purpose Input Output（通用输入/输出），简称 GPIO，与功能复用 I/O（如 SPI/I2C/UART 等）。

大多数 MCU 的引脚都不止一个功能。不同引脚内部结构不一样，拥有的功能也不一样。可以通过不同的配置，切换引脚的实际功能。通用 I/O 口主要特性如下：

可编程控制中断：中断触发模式可配置，如图 2-4 所示：

图 2-4 PIN 中断触发方式



输入输出模式可控制。

输出模式一般包括：推挽、开漏、上拉、下拉。引脚为输出模式时，可以通过配置引脚输出的电平状态为高电平或低电平来控制连接的外围设备。

输入模式一般包括：浮空、上拉、下拉、模拟。引脚为输入模式时，可以读取引脚的电平状态，即高电平或低电平。

2.2.2 访问 PIN 设备

应用程序通过 RT-Thread 提供的 PIN 设备管理接口来访问 GPIO，相关接口如下所示：

函数	描述
rt_pin_mode()	设置引脚模式
rt_pin_write()	设置引脚电平
rt_pin_read()	读取引脚电平
rt_pin_attach_irq()	绑定引脚中断回调函数
rt_pin_irq_enable()	使能引脚中断
rt_pin_detach_irq()	脱离引脚中断回调函数

2.2.3 设置引脚模式

引脚在使用前需要先设置好输入或者输出模式，通过如下函数完成：

void rt_pin_mode(rt_base_t pin, rt_base_t mode)	
参数	描述
pin	引脚编号
mode	引脚工作模式

2.2.4 设置引脚电平

设置引脚输出电平的函数如下所示：

void rt_pin_write(rt_base_t pin, rt_base_t value)	
参数	描述
pin	引脚编号
value	电平逻辑值，可取 2 种宏定义值之一：PIN_LOW 低电平，PIN_HIGH 高电平

2.2.5 读取引脚电平

读取引脚电平的函数如下所示：

int rt_pin_read(rt_base_t pin)	
参数	描述
pin	引脚编号
返回	——
PIN_LOW	低电平
PIN_HIGH	高电平

2.2.6 绑定引脚中断回调函数

若要使用到引脚的中断功能，可以使用如下函数将某个引脚配置为某种中断触发模式并绑定一个中断回调函数到对应引脚，当引脚中断发生时，就会执行回调函数：

rt_err_t rt_pin_attach_irq(rt_int32_t pin, rt_uint32_t mode, void (*hdr)(void *args), void *args)	
参数	描述
pin	引脚编号
mode	中断触发模式
hdr	中断回调函数，用户需要自行定义这个函数
args	中断回调函数的参数，不需要时设置为 RT_NULL
返回	——
RT_EOK	绑定成功
错误码	绑定失败

2.2.7 使能引脚中断

绑定好引脚中断回调函数后使用下面的函数使能引脚中断：

rt_err_t rt_pin_irq_enable(rt_base_t pin, rt_uint32_t enabled)	
参数	描述
pin	引脚编号
enabled	状态，可取 2 种值之一：PIN_IRQ_ENABLE（开启），PIN_IRQ_DISABLE（关闭）
返回	——
RT_EOK	使能成功
错误码	使能失败

2.2.8 脱离引脚中断回调函数

可以使用如下函数脱离引脚中断回调函数：

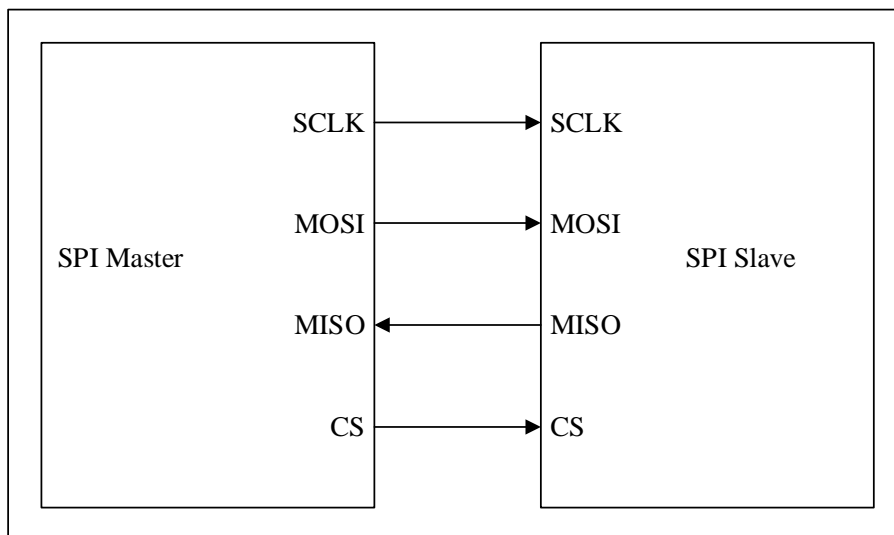
rt_err_t rt_pin_detach_irq(rt_int32_t pin)	
参数	描述
pin	引脚编号
返回	——
RT_EOK	脱离成功
错误码	脱离失败

2.3 SPI 设备

2.3.1 SPI 简介

SPI（Serial Peripheral Interface，串行外设接口）是一种高速、全双工、同步通信总线，常用于短距离通讯。SPI 一般使用 4 根线通信，如图 2-5 所示：

图 2-5 SPI 通信

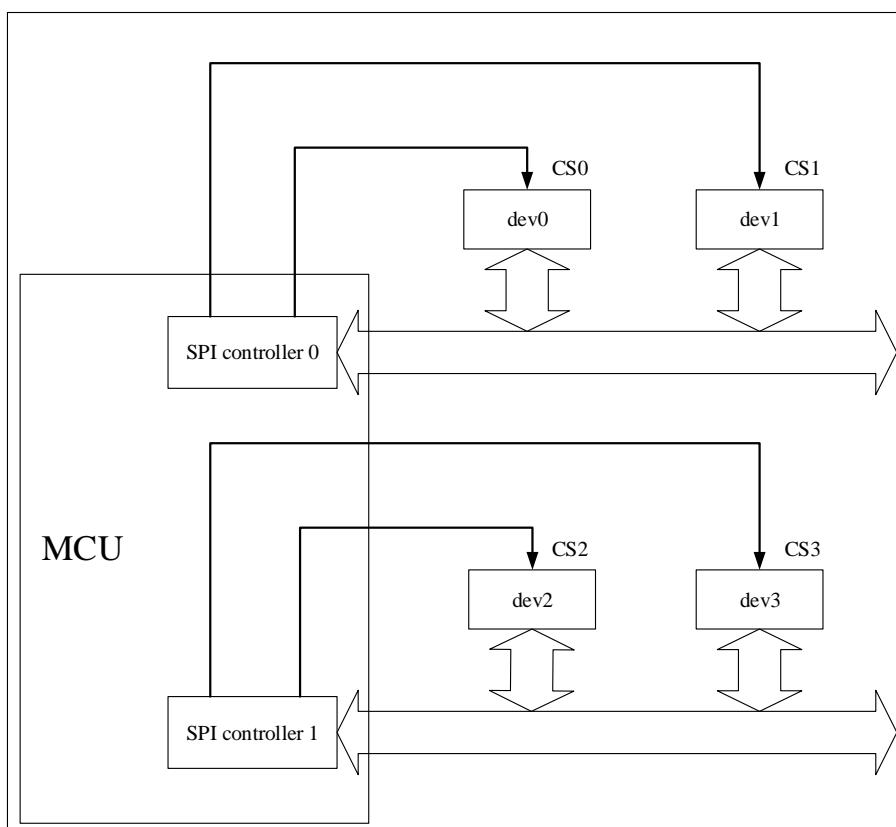


- MOSI：主机输出/从机输入数据线（SPI Bus Master Output/Slave Input）。
- MISO：主机输入/从机输出数据线（SPI Bus Master Input/Slave Output）。
- SCLK：串行时钟线（Serial Clock），主设备输出时钟信号至从设备。
- CS：从设备选择线(Chip select)。也叫 SS、CSB、CSN、EN 等，主设备输出片选信号至从设备。

SPI 以主从方式工作，通常有一个主设备和一个或多个从设备。通信由主设备发起，主设备通过 CS 选择要通信的从设备，然后通过 SCLK 给从设备提供时钟信号，数据通过 MOSI 输出给从设备，同时通过 MISO 接收从设备发送的数据。

如图 2-6 所示芯片有 2 个 SPI 控制器，SPI 控制器对应 SPI 主设备，每个 SPI 控制器可以连接多个 SPI 从设备。挂载在同一个 SPI 控制器上的从设备共享 3 个信号引脚：SCK、MISO、MOSI，但每个从设备的 CS 引脚是独立的。

图 2-6 SPI 控制器



主设备通过控制 CS 引脚对从设备进行片选，一般为低电平有效。任何时刻，一个 SPI 主设备上只有一个 CS 引脚处于有效状态，与该有效 CS 引脚连接的从设备此时可以与主设备通信。

2.3.2 挂载 SPI 设备

SPI 设备需要挂载到已经注册好的 SPI 总线上。

<pre>rt_err_t rt_spi_bus_attach_device(struct rt_spi_device *device, const char *name, const char *bus_name, void *user_data)</pre>	
参数	描述
device	SPI 设备句柄
name	SPI 设备名称
bus_name	SPI 总线名称
user_data	用户数据指针
返回	——
RT_EOK	成功
其他错误码	失败

此函数用于挂载一个 SPI 设备到指定的 SPI 总线，并向内核注册 SPI 设备，并将 user_data 保存到 SPI 设备的控制块里。

一般 SPI 总线命名原则为 `spix`，SPI 设备命名原则为 `spixy`，如 `spi10` 表示挂载在 `spi1` 总线上的 0 号设备。`user_data` 一般为 SPI 设备的 CS 引脚指针，进行数据传输时 SPI 控制器会操作此引脚进行片选。

使用下面的函数挂载 SPI 设备到总线：

```
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 配置 SPI 设备

挂载 SPI 设备到 SPI 总线后需要配置 SPI 设备的传输参数。

rt_err_t rt_spi_configure(struct rt_spi_device *device, struct rt_spi_configuration *cfg)	
参数	描述
device	SPI 设备句柄
cfg	SPI 配置参数指针
返回	——
RT_EOK	成功

此函数会保存 `cfg` 指向的配置参数到 SPI 设备 `device` 的控制块里，当传输数据时会使用此配置参数。`struct rt_spi_configuration` 原型如下：

```
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 访问 SPI 设备

一般情况下 MCU 的 SPI 器件都是作为主机和从机通讯，在 RT-Thread 中将 SPI 主机虚拟为 SPI 总线设备，应用程序使用 SPI 设备管理接口来访问 SPI 从机器件，主要接口如下所示：

函数	描述
<code>rt_device_find()</code>	根据 SPI 设备名称查找设备获取设备句柄
<code>rt_spi_transfer_message()</code>	自定义传输数据
<code>rt_spi_transfer()</code>	传输一次数据
<code>rt_spi_send()</code>	发送一次数据
<code>rt_spi_recv()</code>	接受一次数据
<code>rt_spi_send_then_send()</code>	连续两次发送
<code>rt_spi_send_then_recv()</code>	先发送后接收

2.3.5 查找 SPI 设备

在使用 SPI 设备前需要根据 SPI 设备名称获取设备句柄，进而才可以操作 SPI 设备，查找设备函数如下所示：

rt_device_t rt_device_find(const char* name)	
参数	描述
name	设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

2.3.6 自定义传输数据

获取到 SPI 设备句柄就可以使用 SPI 设备管理接口访问 SPI 设备器件，进行数据收发。可以通过如下函数传输消息：

struct rt_spi_message *rt_spi_transfer_message(struct rt_spi_device *device, struct rt_spi_message *message)	
参数	描述
device	SPI 设备句柄
message	消息指针
返回	——
RT_NULL	成功发送
非空指针	发送失败，返回指向剩余未发送的 message 的指针

此函数可以传输一连串消息，用户可以自定义每个待传输的 message 结构体各参数的数值，从而可以很方便的 control 数据传输方式。struct rt_spi_message 原型如下：

struct rt_spi_message	
{	
const void	*send_buf; //发送缓冲区指针
void	*recv_buf; //接收缓冲区指针
rt_size_t	length; //发送/接收 数据字节数
struct rt_spi_message	*next; //指向继续发送的下一条消息的指针
unsigned cs_take	: 1; //片选选中
unsigned cs_release	: 1; //释放片选
};	

sendbuf 为发送缓冲区指针，其值为 RT_NULL 时，表示本次传输为只接收状态，不需要发送数据。

recvbuf 为接收缓冲区指针，其值为 RT_NULL 时，表示本次传输为只发送状态，不需要保存接收到的数据，所以收到的数据直接丢弃。

length 的单位为 word，即数据长度为 8 位时，每个 length 占用 1 个字节；当数据长度为 16 位时，每个 length 占用 2 个字节。

参数 next 是指向继续发送的下一条消息的指针，若只发送一条消息，则此指针值为 RT_NULL。多个待传输的消息通过 next 指针以单向链表的形式连接在一起。

cs_take 值为 1 时，表示在传输数据前，设置对应的 CS 为有效状态。cs_release 值为 1 时，表示在数据传输结束后，释放对应的 CS。

2.3.7 传输一次数据

如果只传输一次数据可以通过如下函数：

rt_size_t rt_spi_transfer(struct rt_spi_device *device, const void *send_buf, void *recv_buf, rt_size_t length)	
参数	描述
device	SPI 设备句柄
send_buf	发送数据缓冲区指针
recv_buf	接收数据缓冲区指针
length	发送/接收数据字节数
返回	——
0	传输失败
非 0 值	成功传输的字节数

此函数等同于调用 rt_spi_transfer_message() 传输一条消息，开始发送数据时片选选中，函数返回时释放片选，message 参数配置如下：

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 发送一次数据

如果只发送一次数据，而忽略接收到的数据可以通过如下函数：

rt_size_t rt_spi_send(struct rt_spi_device *device, const void *send_buf, rt_size_t length)	
参数	描述
device	SPI 设备句柄
send_buf	发送数据缓冲区指针
length	发送数据字节数
返回	——
0	发送失败
非 0 值	成功发送的字节数

调用此函数发送 send_buf 指向的缓冲区的数据，忽略接收到的数据，此函数是对 rt_spi_transfer() 函数的封装。

此函数等同于调用 rt_spi_transfer_message() 传输一条消息，开始发送数据时片选选中，函数返回时释放片选，message 参数配置如下：

```
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 接收一次数据

如果只接收一次数据可以通过如下函数：

rt_size_t rt_spi_recv(struct rt_spi_device *device, void *recv_buf, rt_size_t length)	
参数	描述
device	SPI 设备句柄
recv_buf	接收数据缓冲区指针
length	接收数据字节数
返回	——
0	接收失败
非 0 值	成功接收的字节数

调用此函数接收数据并保存到 `recv_buf` 指向的缓冲区。此函数是对 `rt_spi_transfer()` 函数的封装。SPI 总线协议规定只能由主设备产生时钟，因此在接收数据时，主设备会发送数据 `0XFF`。

此函数等同于调用 `rt_spi_transfer_message()` 传输一条消息，开始接收数据时片选选中，函数返回时释放片选，`message` 参数配置如下：

```
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 连续两次发送数据

如果需要先后连续发送 2 个缓冲区的数据，并且中间片选不释放，可以调用如下函数：

```
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)
```

参数	描述
device	SPI 设备句柄
send_buf1	发送数据缓冲区 1 指针
send_length1	发送数据缓冲区 1 数据字节数
send_buf2	发送数据缓冲区 2 指针
send_length2	发送数据缓冲区 2 数据字节数
返回	——
RT_EOK	发送成功
-RT_EIO	发送失败

此函数可以连续发送 2 个缓冲区的数据，忽略接收到的数据，发送 send_buf1 时片选选中，发送完 send_buf2 后释放片选。

本函数适合向 SPI 设备中写入一块数据，第一次先发送命令和地址等数据，第二次再发送指定长度的数据。之所以分两次发送而不是合并成一个数据块发送，或调用两次 rt_spi_send()，是因为在大部分的数据写操作中，都需要先发命令和地址，长度一般只有几个字节。如果与后面的数据合并在一起发送，将需要进行内存空间申请和大量的数据搬运。而如果调用两次 rt_spi_send()，那么在发送完命令和地址后，片选会被释放，大部分 SPI 设备都依靠设置片选一次有效为命令的起始，所以片选在发送完命令或地址数据后被释放，则此次操作被丢弃。

此函数等同于调用 rt_spi_transfer_message() 传输 2 条消息，message 参数配置如下：

```
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 先发送后接收数据

如果需要向从设备先发送数据，然后接收从设备发送的数据，并且中间片选不释放，可以调用如下函数：

```
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)
参数	描述
device	SPI 从设备句柄
send_buf	发送数据缓冲区指针
send_length	发送数据缓冲区数据字节数
recv_buf	接收数据缓冲区指针
recv_length	接收数据字节数
返回	——
RT_EOK	成功
-RT_EIO	失败

此函数发送第一条数据 send_buf 时开始片选，此时忽略接收到的数据，然后发送第二条数据，此时主设备会发送数据 0XFF，接收到的数据保存在 recv_buf 里，函数返回时释放片选。

本函数适合从 SPI 从设备中读取一块数据，第一次会先发送一些命令和地址数据，然后再接收指定长度的数据。

此函数等同于调用 rt_spi_transfer_message() 传输 2 条消息，message 参数配置如下：

```

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;

```

SPI 设备管理模块还提供 rt_spi_sendrecv8() 和 rt_spi_sendrecv16() 函数，这两个函数都是对此函数的封装，rt_spi_sendrecv8() 发送一个字节数据同时收到一个字节数据，rt_spi_sendrecv16() 发送 2 个字节数据同时收到 2 个字节数据。

2.3.12 特殊使用场景

在一些特殊的使用场景，某个设备希望独占总线一段时间，且期间要保持片选一直有效，期间数据传输可能是间断的，则可以按照如所示步骤使用相关接口。传输数据函数必须使用 rt_spi_transfer_message()，并且此函数每个待传输消息的片选控制域 cs_take 和 cs_release 都要设置为 0 值，因为片选已经使用了其他接口控制，不需要在数据传输的时候控制。

2.3.13 获取总线

在多线程的情况下，同一个 SPI 总线可能会在不同的线程中使用，为了防止 SPI 总线正在传输的数据丢失，从设备在开始传输数据前需要先获取 SPI 总线的使用权，获取成功才能够使用总线传输数据，可使用如下函数获取 SPI 总线：

rt_err_t rt_spi_take_bus(struct rt_spi_device *device)	
参数	描述
device	SPI 设备句柄
返回	——
RT_EOK	成功
错误码	失败

2.3.14 选中片选

从设备获取总线的使用权后，需要设置自己对应的片选信号为有效，可使用如下函数选中片选：

rt_err_t rt_spi_take(struct rt_spi_device *device)	
参数	描述
device	SPI 设备句柄
返回	——
0	成功
错误码	失败

2.3.15 增加一条消息

使用 rt_spi_transfer_message() 传输消息时，所有待传输的消息都是以单向链表的形式连接起来的，可使用如下函数往消息链表里增加一条新的待传输消息：

void rt_spi_message_append(struct rt_spi_message *list, struct rt_spi_message *message)	
参数	描述
list	待传输的消息链表节点
message	新增消息指针

2.3.16 释放片选

从设备数据传输完成后，需要释放片选，可使用如下函数释放片选：

rt_err_t rt_spi_release(struct rt_spi_device *device)	
参数	描述
device	SPI 设备句柄
返回	——
0	成功
错误码	失败

2.3.17 释放总线

从设备不在使用 SPI 总线传输数据，必须尽快释放总线，这样其他从设备才能使用 SPI 总线传输数据，可使用如下函数释放总线：

rt_err_t rt_spi_release_bus(struct rt_spi_device *device)	
参数	描述
device	SPI 设备句柄
返回	——
RT_EOK	成功

2.4 UART 设备

2.4.1 UART 简介

UART（Universal Asynchronous Receiver/Transmitter）通用异步收发传输器，UART 作为异步串口通信协议的一种，工作原理是将传输数据的每个字符一位接一位地传输。是在应用程序开发过程中使用频率最高的数据总线。

2.4.2 访问串口设备

应用程序通过 RT-Thread 提供的 I/O 设备管理接口来访问串口硬件，相关接口如下所示：

函数	描述
rt_device_find()	查找设备
rt_device_open()	打开设备
rt_device_read()	读取数据
rt_device_write()	写入数据
rt_device_control()	控制设备
rt_device_set_rx_indicate()	设置接收回调函数
rt_device_set_tx_complete()	设置发送完成回调函数
rt_device_close()	关闭设备

2.4.3 查找串口设备

应用程序根据串口设备名称获取设备句柄，进而可以操作串口设备，查找设备函数如下所示：

rt_device_t rt_device_find(const char* name)	
参数	描述
name	设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

2.4.4 打开串口设备

通过设备句柄，应用程序可以打开和关闭设备，打开设备时，会检测设备是否已经初始化，没有初始化则会默认调用初始化接口初始化设备。通过如下函数打开设备：

rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags)	
参数	描述
dev	设备句柄
oflags	设备模式标志
返回	——
RT_EOK	设备打开成功

-RT_EBUSY	如果设备注册时指定的参数中包括 RT_DEVICE_FLAG_STANDALONE 参数，此设备将不允许重复打开
其他错误码	设备打开失败

oflags 参数支持下列取值(可以采用或的方式支持多种取值):

#define RT_DEVICE_FLAG_STREAM	0x040	/* 流模式 */
/* Receive mode parameter */		
#define RT_DEVICE_FLAG_INT_RX	0x100	/* 中断接收模式 */
#define RT_DEVICE_FLAG_DMA_RX	0x200	/* DMA 接收模式 */
/* Send mode parameter */		
#define RT_DEVICE_FLAG_INT_TX	0x400	/* 中断发送模式 */
#define RT_DEVICE_FLAG_DMA_TX	0x800	/* DMA 发送模式 */

串口数据接收和发送数据的模式分为 3 种：中断模式、轮询模式、DMA 模式。在使用的时候，这 3 种模式只能选其一，若串口的打开参数 oflags 没有指定使用中断模式或者 DMA 模式，则默认使用轮询模式。

DMA（Direct Memory Access）即直接存储器访问。DMA 传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过 DMA 控制器为 RAM 与 I/O 设备开辟一条直接传送数据的通路，这就节省了 CPU 的资源来做其他操作。使用 DMA 传输可以连续获取或发送一段信息而不占用中断或延时，在通信频繁或有大量信息要传输时非常有用。

2.4.5 控制串口设备

通过控制接口，应用程序可以对串口设备进行配置，如波特率、数据位、校验位、接收缓冲区大小、停止位等参数的修改。控制函数如下所示：

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
参数	描述
dev	设备句柄
cmd	命令控制字，可取值：RT_DEVICE_CTRL_CONFIG
arg	控制的参数，可取类型：struct serial_configure
返回	_____
RT_EOK	函数执行成功
-RT_ENOSYS	执行失败，dev 为空
其他错误码	执行失败

控制参数结构体 struct serial_configure 原型如下：

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 */
};

```

RT-Thread 提供的默认串口配置如下，即 RT-Thread 系统中默认每个串口设备都使用如下配置：

```

#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 \
}

```

若实际使用串口的配置参数与默认配置参数不符，则用户可以通过应用代码进行修改。修改串口配置参数，如波特率、数据位、校验位、缓冲区接收 `bufsize`、停止位等。

2.4.6 发送数据

向串口中写入数据，可以通过如下函数完成：

rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)	
参数	描述
dev	设备句柄
pos	写入数据偏移量，此参数串口设备未使用
buffer	内存缓冲区指针，放置要写入的数据
size	写入数据的大小
返回	——
写入数据的实际大小	如果是字符设备，返回大小以字节为单位；
0	需要读取当前线程的 <code>errno</code> 来判断错误状态

2.4.7 设置发送完成回调函数

在应用程序调用 `rt_device_write()` 写入数据时，如果底层硬件能够支持自动发送，那么上层应用可以设置一个回调函数。这个回调函数会在底层硬件数据发送完成后(例如 DMA 传送完成或 FIFO 已经写入完毕产生完成中断时)调用。可以通过如下函数设置设备发送完成指示：

rt_err_t rt_device_set_tx_complete(rt_device_t dev, rt_err_t (*tx_done)(rt_device_t dev, void *buffer))	
参数	描述
dev	设备句柄

tx_done	回调函数指针
返回	——
RT_EOK	设置成功

调用这个函数时，回调函数由调用者提供，当硬件设备发送完数据时，由设备驱动程序回调这个函数并把发送完成的数据块地址 **buffer** 作为参数传递给上层应用。上层应用（线程）在收到指示时会根据发送 **buffer** 的情况，释放 **buffer** 内存块或将其作为下一个写数据的缓存。

2.4.8 设置接收回调函数

可以通过如下函数来设置数据接收指示，当串口收到数据时，通知上层应用线程有数据到达：

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))	
参数	描述
dev	设备句柄
rx_ind	回调函数指针
dev	设备句柄（回调函数参数）
size	缓冲区数据大小（回调函数参数）
返回	——
RT_EOK	设置成功

该函数的回调函数由调用者提供。若串口以中断接收模式打开，当串口接收到一个数据产生中断时，就会调用回调函数，并且会把此时缓冲区的数据大小放在 **size** 参数里，把串口设备句柄放在 **dev** 参数里供调用者获取。

若串口以 DMA 接收模式打开，当 DMA 完成一批数据的接收后会调用此回调函数。

一般情况下接收回调函数可以发送一个信号量或者事件通知串口数据处理线程有数据到达。

2.4.9 接收数据

可调用如下函数读取串口接收到的数据：

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

2.4.10 关闭串口设备

当应用程序完成串口操作后，可以关闭串口设备，通过如下函数完成：

rt_err_t rt_device_close(rt_device_t dev)	
参数	描述
dev	设备句柄
返回	——
RT_EOK	关闭设备成功
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

2.5 I2C 设备

2.5.1 I2C 简介

I2C（Inter Integrated Circuit）总线是 PHILIPS 公司开发的一种半双工、双向二线制同步串行总线。I2C 总线传输数据时只需两根信号线，一根是双向数据线 SDA（serial data），另一根是双向时钟线 SCL（serial clock）。SPI 总线有两根线分别用于主从设备之间接收数据和发送数据，而 I2C 总线只使用一根线进行数据收发。

I2C 和 SPI 一样以主从的方式工作，不同于 SPI 一主多从的结构，它允许同时有多个主设备存在，每个连接到总线上的器件都有唯一的地址，主设备启动数据传输并产生时钟信号，从设备被主设备寻址，同一时刻只允许有一个主设备。

2.5.2 访问 I2C 总线设备

一般情况下 MCU 的 I2C 器件都是作为主机和从机通讯，在 RT-Thread 中将 I2C 主机虚拟为 I2C 总线设备，I2C 从机通过 I2C 设备接口和 I2C 总线通讯，相关接口如下所示：

函数	描述
rt_device_find()	根据 I2C 总线设备名称查找设备获取设备句柄
rt_i2c_transfer()	传输数据

2.5.3 查找 I2C 总线设备

在使用 I2C 总线设备前需要根据 I2C 总线设备名称获取设备句柄，进而才可以操作 I2C 总线设备，查找设备函数如下所示：

rt_device_t rt_device_find(const char* name)	
参数	描述
name	I2C 总线设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

2.5.4 数据传输

获取到 I2C 总线设备句柄就可以使用 rt_i2c_transfer() 进行数据传输。函数原型如下所示：

rt_size_t rt_i2c_transfer(struct rt_i2c_bus_device *bus, struct rt_i2c_msg msgs[], rt_uint32_t num)	
参数	描述
bus	I2C 总线设备句柄
msgs[]	待传输的消息数组指针
num	消息数组的元素个数
返回	——
消息数组的元素个数	成功
错误码	失败

和 SPI 总线的自定义传输接口一样，I2C 总线的自定义传输接口传输的数据也是以一个消息为单位。参数 `msgs[]` 指向待传输的消息数组，用户可以自定义每条消息的内容，实现 I2C 总线所支持的 2 种不同的数据传输模式。如果主设备需要发送重复开始条件，则需要发送 2 个消息。

I2C 消息数据结构原型如下：

```
struct rt_i2c_msg
{
    rt_uint16_t  addr;    /* 从机地址 */
    rt_uint16_t  flags;   /* 读、写标志等 */
    rt_uint16_t  len;     /* 读写数据字节数 */
    rt_uint8_t   *buf;    /* 读写数据缓冲区指针 */
}
```

从机地址 `addr`：支持 7 位和 10 位二进制地址，需查看不同设备的数据手册。

2.6 ADC 设备

2.6.1 ADC 简介

ADC(Analog-to-Digital Converter)指模数转换器。是指将连续变化的模拟信号转换为离散的数字信号的器件。真实世界的模拟信号，例如温度、压力、声音或者图像等，需要转换成更容易储存、处理和发射的数字形式。模数转换器可以实现这个功能，在各种不同的产品中都可以找到它的身影。与之相对应的 DAC(Digital-to-Analog Converter)，它是 ADC 模数转换的逆向过程。ADC 最早用于对无线信号向数字信号转换。如电视信号，长短播电台发接收等。

2.6.2 访问 ADC 设备

应用程序通过 RT-Thread 提供的 ADC 设备管理接口来访问 ADC 硬件，相关接口如下所示：

函数	描述
rt_device_find()	根据 ADC 设备名称查找设备获取设备句柄
rt_adc_enable()	使能 ADC 设备
rt_adc_read()	读取 ADC 设备数据
rt_adc_disable()	关闭 ADC 设备

2.6.3 查找 ADC 设备

应用程序根据 ADC 设备名称获取设备句柄，进而可以操作 ADC 设备，查找设备函数如下所示：

rt_device_t rt_device_find(const char* name)	
参数	描述
name	ADC 设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到设备

2.6.4 使能 ADC 通道

在读取 ADC 设备数据前需要先使能设备，通过如下函数使能设备：

rt_err_t rt_adc_enable(rt_adc_device_t dev, rt_uint32_t channel)	
参数	描述
dev	ADC 设备句柄
channel	ADC 通道
返回	——
RT_EOK	成功
-RT_ENOSYS	失败，设备操作方法为空
其他错误码	失败

2.6.5 读取 ADC 通道采样值

读取 ADC 通道采样值可通过如下函数完成：

rt_uint32_t rt_adc_read(rt_adc_device_t dev, rt_uint32_t channel)	
参数	描述
dev	ADC 设备句柄
channel	ADC 通道
返回	——
读取的数值	

2.6.6 关闭 ADC 通道

关闭 ADC 通道可通过如下函数完成：

rt_err_t rt_adc_disable(rt_adc_device_t dev, rt_uint32_t channel)	
参数	描述
dev	ADC 设备句柄
channel	ADC 通道
返回	——
RT_EOK	成功
-RT_ENOSYS	失败，设备操作方法为空
其他错误码	失败

2.7 DAC 设备

2.7.1 DAC 简介

DAC(Digital-to-Analog Converter)指数模转换器。是指把二进制数字量形式的离散数字信号转换为连续变化的模拟信号的器件。在数字世界中,要处理不稳定和动态的模拟信号并不容易,基于 DAC 的特性,在各种不同的产品中都可以找到它的身影。与之相对应的 ADC(Analog-to-Digital Converter),它是 DAC 数模转换的逆向过程。DAC 主要应用于音频放大,视频编码,电机控制,数字电位计等。

2.7.2 访问 DAC 设备

应用程序通过 RT-Thread 提供的 DAC 设备管理接口来访问 DAC 硬件,相关接口如下所示:

函数	描述
rt_device_find()	根据 DAC 设备名称查找设备获取设备句柄
rt_dac_enable()	使能 DAC 设备
rt_dac_write()	设置 DAC 设备输出值
rt_dac_disable()	关闭 DAC 设备

2.7.3 查找 DAC 设备

应用程序根据 DAC 设备名称获取设备句柄,进而可以操作 DAC 设备,查找设备函数如下所示:

rt_device_t rt_device_find(const char* name)	
参数	描述
name	DAC 设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到设备

2.7.4 使能 DAC 通道

在设置 DAC 设备数据前需要先使能设备,通过如下函数使能设备:

rt_err_t rt_dac_enable(rt_dac_device_t dev, rt_uint32_t channel)	
参数	描述
dev	DAC 设备句柄
channel	DAC 通道
返回	——
RT_EOK	成功
-RT_ENOSYS	失败,设备操作方法为空
其他错误码	失败

2.7.5 设置 DAC 通道输出值

设置 DAC 通道输出值可通过如下函数完成：

rt_uint32_t rt_dac_write(rt_dac_device_t dev, rt_uint32_t channel, rt_uint32_t value)	
参数	描述
dev	DAC 设备句柄
channel	DAC 通道
value	DAC 输出值
返回	——
RT_EOK	成功
-RT_ENOSYS	失败

2.7.6 关闭 DAC 通道

关闭 DAC 通道可通过如下函数完成：

rt_err_t rt_dac_disable(rt_dac_device_t dev, rt_uint32_t channel)	
参数	描述
dev	DAC 设备句柄
channel	DAC 通道
返回	——
RT_EOK	成功
-RT_ENOSYS	失败，设备操作方法为空
其他错误码	失败

2.8 CAN 设备

2.8.1 CAN 简介

CAN 是控制器局域网(Controller Area Network, CAN)的简称, 是由以研发和生产汽车电子产品著称的德国 BOSCH 公司开发的, 并最终成为国际标准 (ISO 11898), 是国际上应用最广泛的现场总线之一。

2.8.2 访问 CAN 设备

应用程序通过 RT-Thread 提供的 I/O 设备管理接口来访问 CAN 硬件控制器, 相关接口如下所示:

函数	描述
rt_device_find	查找设备
rt_device_open	打开设备
rt_device_read	读取数据
rt_device_write	写入数据
rt_device_control	控制设备
rt_device_set_rx_indicate	设置接收回调函数
rt_device_close	关闭设备

2.8.3 查找 CAN 设备

应用程序根据 CAN 设备名称查找设备获取设备句柄, 进而可以操作 CAN 设备, 查找设备函数如下所示:

rt_device_t rt_device_find(const char* name)	
参数	描述
name	设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

2.8.4 打开 CAN 设备

通过设备句柄, 应用程序可以打开和关闭设备, 打开设备时, 会检测设备是否已经初始化, 没有初始化则会默认调用初始化接口初始化设备。通过如下函数打开设备:

rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags)	
参数	描述
dev	设备句柄
oflags	打开设备模式标志
返回	——
RT_EOK	设备打开成功
-RT_EBUSY	如果设备注册时指定的参数中包括 RT_DEVICE_FLAG_STANDALONE 参数, 此设备将不允许重复打开

其他错误码	设备打开失败
-------	--------

2.8.5 控制 CAN 设备

通过命令控制字，应用程序可以对 CAN 设备进行配置，通过如下函数完成：

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
参数	描述
dev	设备句柄
cmd	控制命令
arg	控制参数
返回	——
RT_EOK	函数执行成功
其他错误码	执行失败

2.8.6 发送数据

使用 CAN 设备发送数据，可以通过如下函数完成：

rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)	
参数	描述
dev	设备句柄
pos	写入数据偏移量，此参数 CAN 设备未使用
buffer	CAN 消息指针
size	CAN 消息大小
返回	——
不为 0	实际发送的 CAN 消息大小
0	发送失败

2.8.7 设置接收回调函数

可以通过如下函数来设置数据接收指示，当 CAN 收到数据时，通知上层应用线程有数据到达：

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))	
参数	描述
dev	设备句柄
rx_ind	回调函数指针
dev	设备句柄（回调函数参数）
size	缓冲区数据大小（回调函数参数）
返回	——
RT_EOK	设置成功

2.8.8 接收数据

可调用如下函数读取 CAN 设备接收到的数据：

rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos, void* buffer, rt_size_t size)	
参数	描述
dev	设备句柄
pos	读取数据偏移量，此参数 CAN 设备未使用
buffer	CAN 消息指针，读取的数据将会被保存在缓冲区中
size	CAN 消息大小
返回	——
不为 0	CAN 消息大小
0	失败

2.8.9 关闭 CAN 设备

当应用程序完成 CAN 操作后，可以关闭 CAN 设备，通过如下函数完成：

rt_err_t rt_device_close(rt_device_t dev)	
参数	描述
dev	设备句柄
返回	——
RT_EOK	关闭设备成功
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

2.9 HWTIMER 设备

2.9.1 定时器简介

硬件定时器一般有 2 种工作模式，定时器模式和计数器模式。不管是工作在哪一种模式，实质都是通过内部计数器模块对脉冲信号进行计数。下面是定时器的一些重要概念。

计数器模式：对外部输入引脚的外部脉冲信号计数。

定时器模式：对内部脉冲信号计数。定时器常用作定时时钟，以实现定时检测，定时响应、定时控制。

2.9.2 访问硬件定时器设备

应用程序通过 RT-Thread 提供的 I/O 设备管理接口来访问硬件定时器设备，相关接口如下所示：

函数	描述
rt_device_find()	查找定时器设备
rt_device_open()	以读写方式打开定时器设备
rt_device_set_rx_indicate()	设置超时回调函数
rt_device_control()	控制定时器设备，可以设置定时模式（单次/周期）/计数频率，或者停止定时器
rt_device_write()	设置定时器超时值，定时器随即启动
rt_device_read()	获取定时器当前值
rt_device_close()	关闭定时器设备

2.9.3 查找定时器设备

应用程序根据硬件定时器设备名称获取设备句柄，进而可以操作硬件定时器设备，查找设备函数如下所示：

rt_device_t rt_device_find(const char* name)	
参数	描述
name	硬件定时器设备名称
返回	——
定时器设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到设备

2.9.4 打开定时器设备

通过设备句柄，应用程序可以打开设备。打开设备时，会检测设备是否已经初始化，没有初始化则会默认调用初始化接口初始化设备。通过如下函数打开设备：

rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags)	
参数	描述
dev	硬件定时器设备句柄
oflags	设备打开模式，一般以读写方式打开，即取值 RT_DEVICE_OFLAG_RDWR
返回	——
RT_EOK	设备打开成功

其他错误码	设备打开失败
-------	--------

2.9.5 设置超时回调函数

通过如下函数设置定时器超时回调函数，当定时器超时将会调用此回调函数：

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))	
参数	描述
dev	设备句柄
rx_ind	超时回调函数，由调用者提供
返回	——
RT_EOK	成功

2.9.6 控制定时器设备

通过命令控制字，应用程序可以对硬件定时器设备进行配置，通过如下函数完成：

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
参数	描述
dev	设备句柄
cmd	命令控制字
arg	控制的参数
返回	——
RT_EOK	函数执行成功
-RT_ENOSYS	执行失败，dev 为空
其他错误码	执行失败

2.9.7 设置定时器超时值

通过如下函数可以设置定时器的超时值：

rt_size_t rt_device_write(rt_device_t dev, rt_off_t pos, const void* buffer, rt_size_t size)	
参数	描述
dev	设备句柄
pos	写入数据偏移量，未使用，可取 0 值
buffer	指向定时器超时时间结构体的指针
size	超时时间结构体的大小
返回	——
写入数据的实际大小	
0	失败

2.9.8 获取定时器当前值

通过如下函数可以获取定时器当前值：

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

参数	描述
dev	定时器设备句柄
pos	写入数据偏移量，未使用，可取 0 值
buffer	输出参数，指向定时器超时时间结构体的指针
size	超时时间结构体的大小
返回	——
超时时间结构体的大小	成功
0	失败

2.9.9 关闭定时器设备

通过如下函数可以关闭定时器设备：

rt_err_t rt_device_close(rt_device_t dev)	
参数	描述
dev	定时器设备句柄
返回	——
RT_EOK	关闭设备成功
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

2.10 WATCHDOG 设备

2.10.1 WATCHDOG 简介

硬件看门狗（watchdog timer）是一个定时器，其定时输出连接到电路的复位端。在产品化的嵌入式系统中，为了使系统在异常情况下能自动复位，一般都需要引入看门狗。

当看门狗启动后，计数器开始自动计数，在计数器溢出前如果没有被复位，计数器溢出就会对 CPU 产生一个复位信号使系统重启（俗称“被狗咬”）。系统正常运行时，需要在看门狗允许的时间间隔内对看门狗计数器清零（俗称“喂狗”），不让复位信号产生。如果系统不出问题，程序能够按时“喂狗”。一旦程序跑飞，没有“喂狗”，系统“被咬”复位。

2.10.2 访问看门狗设备

应用程序通过 RT-Thread 提供的 I/O 设备管理接口来访问看门狗硬件，相关接口如下所示：

函数	描述
rt_device_find()	根据看门狗设备名称查找设备获取设备句柄
rt_device_init()	初始化看门狗设备
rt_device_control()	控制看门狗设备
rt_device_close()	关闭看门狗设备

2.10.3 查找看门狗

应用程序根据看门狗设备名称获取设备句柄，进而可以操作看门狗设备，查找设备函数如下所示：

rt_device_t rt_device_find(const char* name)	
参数	描述
name	看门狗设备名称
返回	——
设备句柄	查找到对应设备将返回相应的设备句柄
RT_NULL	没有找到相应的设备对象

2.10.4 初始化看门狗

使用看门狗设备前需要先初始化，通过如下函数初始化看门狗设备：

rt_err_t rt_device_init(rt_device_t dev)	
参数	描述
dev	看门狗设备句柄
返回	——
RT_EOK	设备初始化成功
-RT_ENOSYS	初始化失败，看门狗设备驱动初始化函数为空
其他错误码	设备打开失败

2.10.5 控制看门狗

通过命令控制字，应用程序可以对看门狗设备进行配置，通过如下函数完成：

rt_err_t rt_device_control(rt_device_t dev, rt_uint8_t cmd, void* arg)	
参数	描述
dev	看门狗设备句柄
cmd	命令控制字
arg	控制的参数
返回	——
RT_EOK	函数执行成功
-RT_ENOSYS	执行失败，dev 为空
其他错误码	执行失败

2.10.6 在空闲线程钩子函数里喂狗

<pre>static void idle_hook(void) { /* 在空闲线程的回调函数里喂狗 */ rt_device_control(wdg_dev, RT_DEVICE_CTRL_WDT_KEEPAIVE, NULL); }</pre>	
---	--

2.10.7 关闭看门狗

当应用程序完成看门狗操作后，可以关闭看门狗设备，通过如下函数完成：

rt_err_t rt_device_close(rt_device_t dev)	
参数	描述
dev	看门狗设备句柄
返回	——
RT_EOK	关闭设备成功
-RT_ERROR	设备已经完全关闭，不能重复关闭设备
其他错误码	关闭设备失败

3 版本历史

日期	版本	修改
2021.05.07	V1.0	初始版本

4 声明

国民技术股份有限公司（下称“国民技术”）对此文档拥有专属产权。依据中华人民共和国的法律、条约以及世界其他法域相适用的管辖，此文档及其中描述的国民技术产品（下称“产品”）为公司所有。

国民技术在此并未授予专利权、著作权、商标权或其他任何知识产权许可。所提到或引用的第三方名称或品牌（如有）仅用作区别之目的。

国民技术保留随时变更、订正、增强、修改和改良此文档的权利，恕不另行通知。请使用人在下单购买前联系国民技术获取此文档的最新版本。

国民技术竭力提供准确可信的资讯，但即便如此，并不推定国民技术对此文档准确性和可靠性承担责任。

使用此文档信息以及生成产品时，使用者应当进行合理的设计、编程并测试其功能性和安全性，国民技术不对任何因使用此文档或本产品而产生的任何直接、间接、意外、特殊、惩罚性或衍生性损害结果承担责任。

国民技术对于产品在系统或设备中的应用效果没有任何故意或保证，如有任何应用在其发生操作不当或故障情况下，有可能致使人员伤亡、人身伤害或严重财产损失，则此类应用被视为“不安全使用”。

不安全使用包括但不限于：外科手术设备、原子能控制仪器、飞机或宇宙飞船仪器、所有类型的安全装置以及其他旨在支持或维持生命的应用。

所有不安全使用的风险应由使用人承担，同时使用人应使国民技术免于因为这类不安全使用而导致被诉、支付费用、发生损害或承担责任时的赔偿。

对于此文档和产品的任何明示、默示之保证，包括但不限于适销性、特定用途适用性和不侵权的保证，国民技术可在法律允许范围内进行免责。

未经明确许可，任何人不得以任何理由对此文档的全部或部分进行使用、复制、修改、抄录和传播。