



夏令营培训 RT-Thread 设备驱动

培训内容目录

- RT-Thread I/O设备框架概念
- RT-Thread I/O API
- GPIO 应用与驱动开发
- I2C 从机应用与驱动开发
- SPI 从机应用与驱动开发

The logo for RT-Thread, featuring a blue stylized wave icon above the text "RT-Thread" in a blue sans-serif font. The logo is centered within a large, light gray circle that has a subtle drop shadow.

RT-Thread

培训1： RT-Thread I/O设备框架概念

1. 驱动开发问题思考

不同厂家的 **MCU** 关于 **SPI** 接口 **API** 的设计:

- GD: spi_i2s_data_transmit
- ST: HAL_SPI_Transmit
- NXP: LPSPI_MasterTransferBlocking
- LPC: SPI_MasterTransferBlocking

开发项目:

1. STM32 + SPI + W25Q128 + FATFS(A同事)
2. 项目1上更换芯片为LPC(B同事)(由于开发一般都是直接使用spi编写w25q128, 所以基本是重写)
3. LPC + SPI + RW007(WIFI) + Lwip(B同事)

疑问:

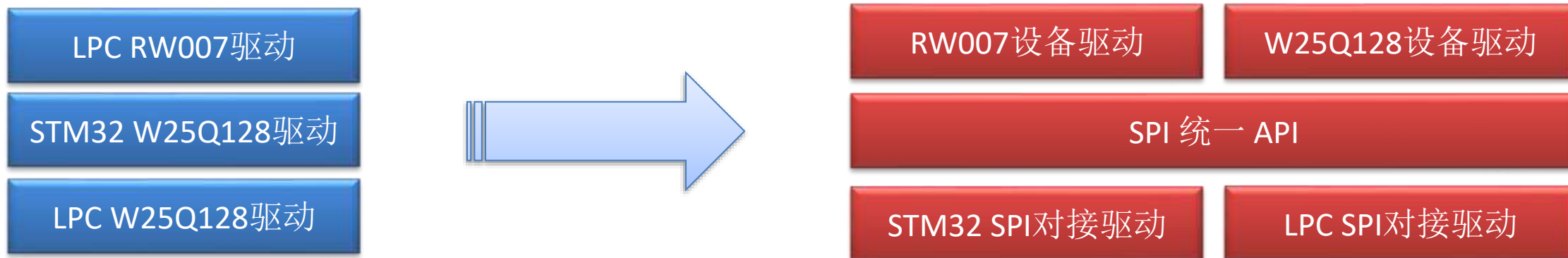
1. 他们能够复用工作吗? 对于团队的效率高吗?

2. 驱动开发碎片化问题

- 学习成本高：同一个工程师需要时间去学习不同厂家的API设计
- 代码复用率低：都是SPI的代码驱动设备，没有办法做到代码复用

那有没有什么办法统一解决这种问题呢？

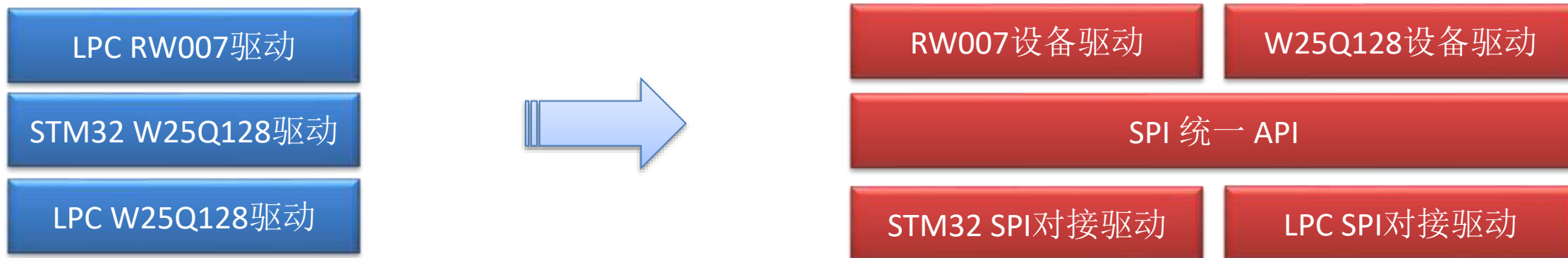
- 不同的厂家 同一外设 开发逻辑 和 API 相同？
- 驱动代码和驱动设备代码分离？



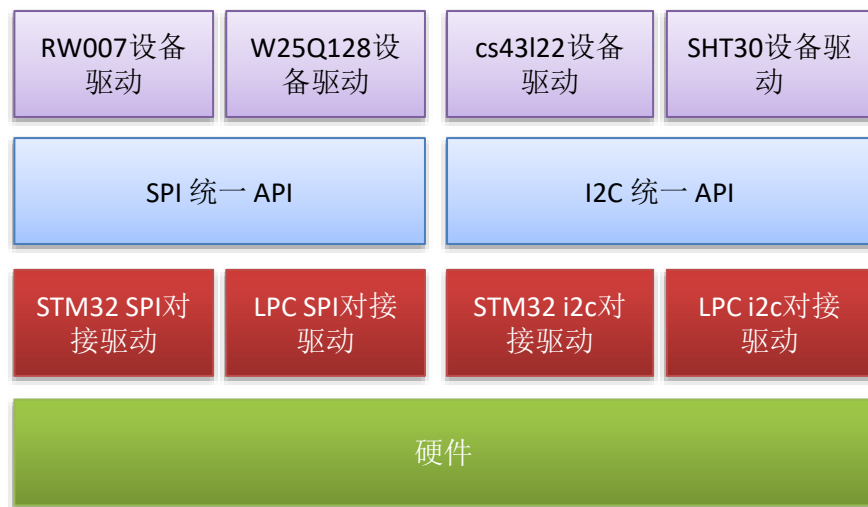
3. 优劣势

SPI驱动和设备驱动分离了，提供统一的API:

- 更换 MCU 只需要改变对应的对接驱动
- 重新驱动设备，只需要重新编写设备驱动相关的代码
- 同一 API 接口，学习成本低
- 分离后设备驱动可以入库，供公司其他项目使用，减少碎片化开发，防止反复造轮子
- 代码框架会变复杂，但是从上面的优点来看是值得的



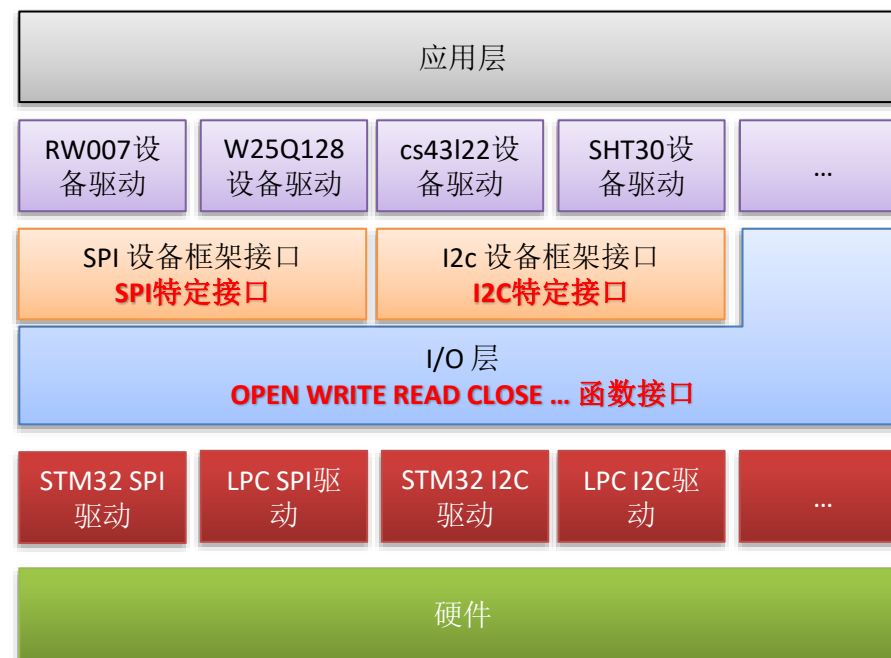
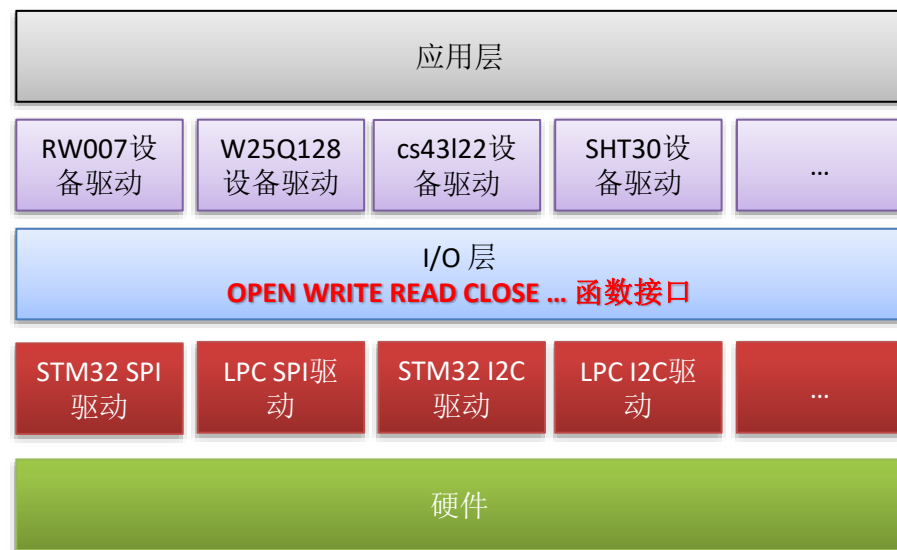
4. 框架演进



5. 框架再演进

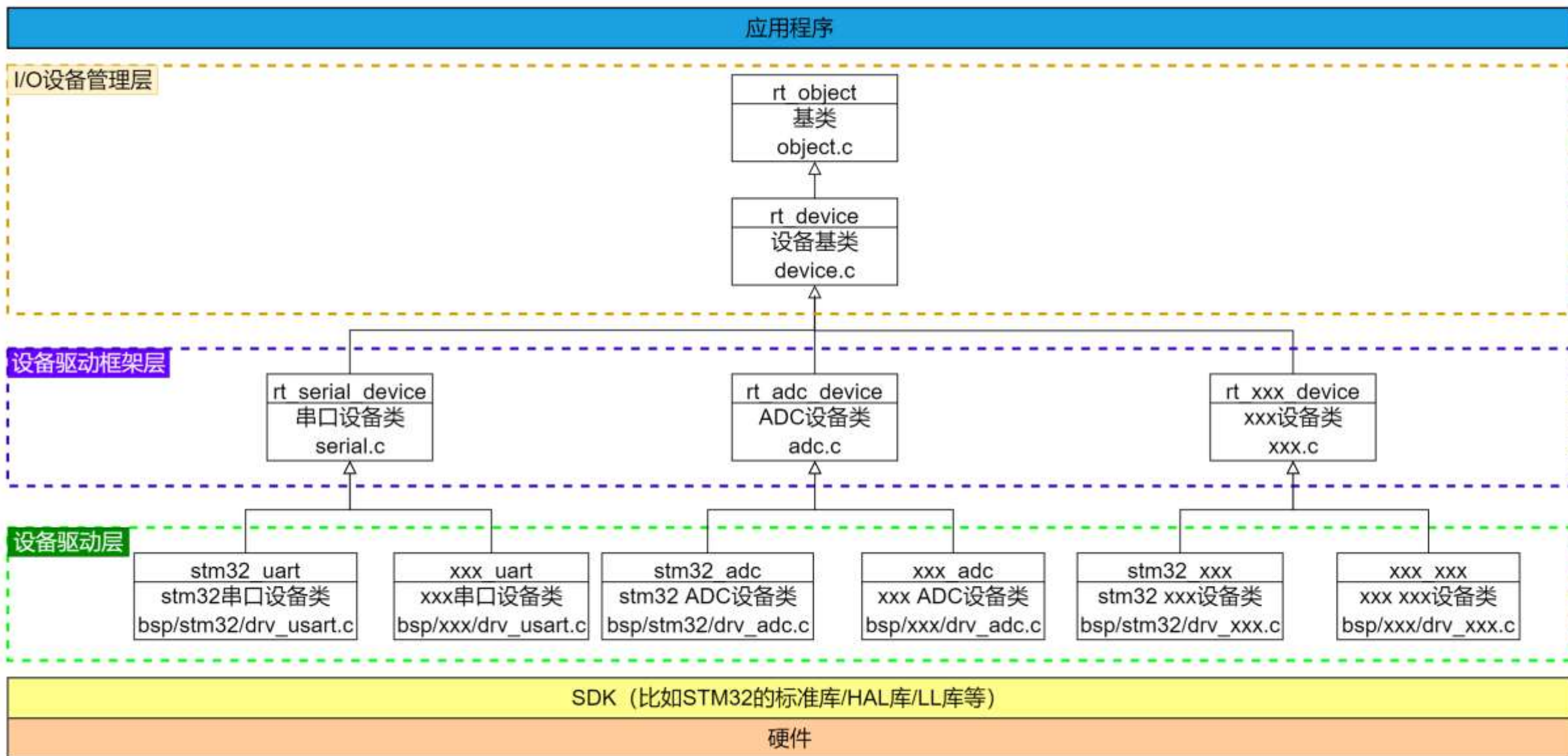
编写应用： **RTT应用开发工程师**

可以做成Package： **RTT驱动开发工程师**



做成特定平台的BSP驱动： **RTT板级BSP移植工程师**

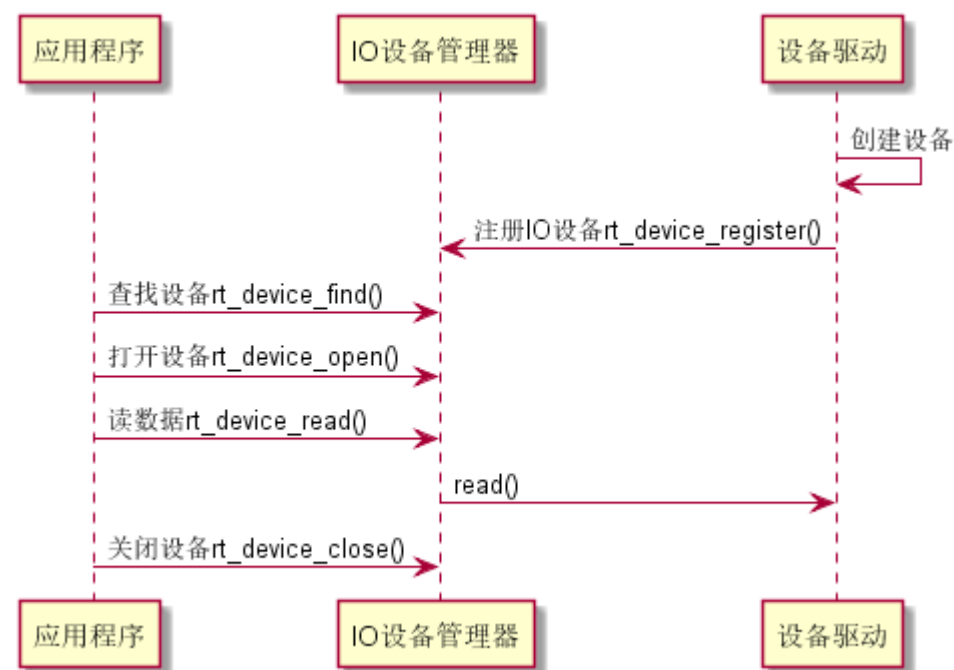
6.RT-Thread 设备驱动框架分析



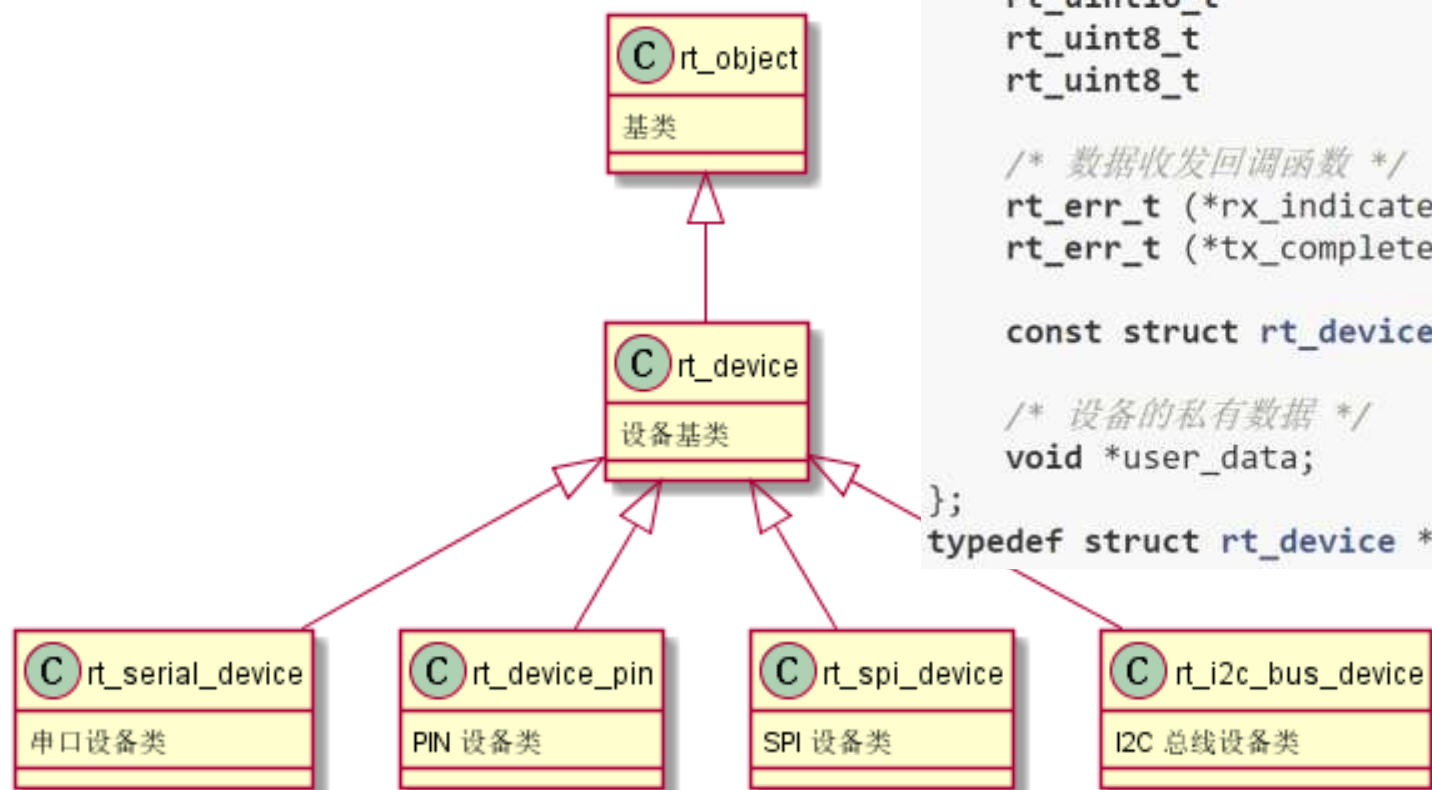
7. RT-Thread I/O框架

- 绝大部分的嵌入式系统都包括一些 I/O（Input/Output，输入 / 输出）设备，例如仪器上的数据显示屏、工业设备上的串口通信、数据采集设备上用于保存数据的 Flash 或 SD 卡，以及网络设备的以太网接口等，都是嵌入式系统中容易找到的 I/O 设备例子。

- I/O 提供一套接口 open write read control close
- SPI I2C GPIO RTC WDG 特定 API



8. I/O派生设备种类



```
struct rt_device
{
    struct rt_object      parent;          /* 内核对象基类 */
    enum rt_device_class_type type;        /* 设备类型 */
    rt_uint16_t          flag;            /* 设备参数 */
    rt_uint16_t          open_flag;       /* 设备打开标志 */
    rt_uint8_t           ref_count;        /* 设备被引用次数 */
    rt_uint8_t           device_id;       /* 设备 ID, 0 - 255 */

    /* 数据收发回调函数 */
    rt_err_t (*rx_indicate)(rt_device_t dev, rt_size_t size);
    rt_err_t (*tx_complete)(rt_device_t dev, void *buffer);

    const struct rt_device_ops *ops;      /* 设备操作方法 */

    /* 设备的私有数据 */
    void *user_data;
};
typedef struct rt_device *rt_device_t;
```

9. RT-Thread 支持的 I/O 设备类型

| | | |
|-------------------------------|------------------|----|
| RT_Device_Class_Char | /* 字符设备 | */ |
| RT_Device_Class_Block | /* 块设备 | */ |
| RT_Device_Class_NetIf | /* 网络接口设备 | */ |
| RT_Device_Class_MTD | /* 内存设备 | */ |
| RT_Device_Class_RTC | /* RTC 设备 | */ |
| RT_Device_Class_Sound | /* 声音设备 | */ |
| RT_Device_Class_Graphic | /* 图形设备 | */ |
| RT_Device_Class_I2CBUS | /* I2C 总线设备 | */ |
| RT_Device_Class_USBDevice | /* USB device 设备 | */ |
| RT_Device_Class_USBHost | /* USB host 设备 | */ |
| RT_Device_Class_SPIBUS | /* SPI 总线设备 | */ |
| RT_Device_Class_SPIDevice | /* SPI 设备 | */ |
| RT_Device_Class_SDIO | /* SDIO 设备 | */ |
| RT_Device_Class_Miscellaneous | /* 杂类设备 | */ |

10. 字符/块设备特点

字符设备和块设备的特点与区别:

- 字符设备: 提供连续的数据流, 应用程序可以顺序读取, 通常不支持随机存取。相反, 此类设备支持按字节/字符来读写数据。举例来说, 键盘、串口、调制解调器都是典型的字符设备
- 块设备: 应用程序可以随机访问设备数据, 程序可自行确定读取数据的位置。硬盘、软盘、**CD-ROM**驱动器和闪存都是典型的块设备, 应用程序可以寻址磁盘上的任何位置, 并由此读取数据。此外, 数据的读写只能以块(通常是**512B**)的倍数进行。与字符设备不同, 块设备并不支持基于字符的寻址。

总结一下, 这两种类型的设备的根本区别在于它们是否可以被随机访问。字符设备只能**顺序读取**, 块设备可以**随机读取**。

11. 为什么要对设备分类

- **MSH** 可以 重定向到任意的 字符设备 上，例如将**lcd**模拟成字符设备，就可以将打印输出到**LCD**上，或者是实现一套空字符设备，将**msh**重定向到这里。
- **Fatfs** 文件系统依赖 块设备驱动，我们将**SD**卡读写实现成块设备，但是也可以用**ram**来模拟块设备驱动，
- 不同的组件和应用会依赖不同的设备，对设备进行分类，可以做到对一类设备同样的控制

12. 来分别下面属于什么设备

- 串口设备 (RT_Device_Class_Char)
- SDIO 网卡 (RT_Device_Class_SDIO)
- CS43L22(音频codec) (RT_Device_Class_Sound)
- GPIO (RT_Device_Class_Pin)
- LCD屏幕 (RT_Device_Class_Graphic)
- 录音驱动 (RT_Device_Class_Sound)

Refer→rt-thread\include\rtdef.h: rt_device_class_type

培训2: RT-Thread I/O API



1. 创建/销毁设备

- `rt_device_t rt_device_create(int type, int attach_size);`
- `void rt_device_destroy(rt_device_t device);`

| 参数 | 描述 |
|-------------|----------------------|
| type | 设备类型, 可取前面小节列出的设备类型值 |
| 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 |

| 参数 | 描述 |
|--------|------|
| device | 设备句柄 |
| 返回 | 无 |

2. 注册/注销设备

- `rt_err_t rt_device_register(rt_device_t dev, const char* name, rt_uint8_t flags);`
- `rt_err_t rt_device_unregister(rt_device_t dev);`

| 参数 | 描述 |
|-----------|--|
| dev | 设备句柄 |
| name | 设备名称, 设备名称的最大长度由 rtconfig.h 中定义的宏 RT_NAME_MAX 指定, 多余部分会被自动截掉 |
| flags | 设备模式标志 |
| 返回 | -- |
| RT_EOK | 注册成功 |
| -RT_ERROR | 注册失败, dev 为空或者 name 已经存在 |

| 参数 | 描述 |
|--------|------|
| dev | 设备句柄 |
| 返回 | -- |
| RT_EOK | 成功 |

3. 注册设备flags

```
#define RT_DEVICE_FLAG_RDONLY      0x001 /* 只读 */
#define RT_DEVICE_FLAG_WRONLY      0x002 /* 只写 */
#define RT_DEVICE_FLAG_RDWR        0x003 /* 读写 */
#define RT_DEVICE_FLAG_REMOVABLE   0x004 /* 可移除 */
#define RT_DEVICE_FLAG_STANDALONE  0x008 /* 独立 */
#define RT_DEVICE_FLAG_SUSPENDED   0x020 /* 挂起 */
#define RT_DEVICE_FLAG_STREAM      0x040 /* 流模式 */
#define RT_DEVICE_FLAG_INT_RX      0x100 /* 中断接收 */
#define RT_DEVICE_FLAG_DMA_RX      0x200 /* DMA 接收 */
#define RT_DEVICE_FLAG_INT_TX      0x400 /* 中断发送 */
#define RT_DEVICE_FLAG_DMA_TX      0x800 /* DMA 发送 */
```

4. 实验1 | 注册字符设备 test

在自己的开发板上注册一个设备：

- 字符设备
- 驱动名称为 **test**

参考：

```
rt_device_t rt_device_create(int type, int attach_size);  
void rt_device_destroy(rt_device_t device);  
rt_err_t rt_device_register(rt_device_t dev, const char* name,  
                             rt_uint8_t flags);  
rt_err_t rt_device_unregister(rt_device_t dev);
```

```
RT_Device_Class_Char      /* 字符设备 */  
RT_Device_Class_Block    /* 块设备 */  
RT_Device_Class_NetIf    /* 网络接口设备 */  
RT_Device_Class_MTD      /* 内存设备 */  
RT_Device_Class_RTC      /* RTC 设备 */  
RT_Device_Class_Sound    /* 声音设备 */  
RT_Device_Class_Graphic  /* 图形设备 */  
RT_Device_Class_I2CBUS   /* I2C 总线设备 */  
RT_Device_Class_USBDevice /* USB device 设备 */  
RT_Device_Class_USBHost  /* USB host 设备 */  
RT_Device_Class_SPIBUS   /* SPI 总线设备 */  
RT_Device_Class_SPIDevice /* SPI 设备 */  
RT_Device_Class_SDIO     /* SDIO 设备 */  
RT_Device_Class_Miscellaneous /* 杂类设备 */
```

```
#define RT_DEVICE_FLAG_RDONLY 0x001 /* 只读 */  
#define RT_DEVICE_FLAG_WRONLY 0x002 /* 只写 */  
#define RT_DEVICE_FLAG_RDWR 0x003 /* 读写 */  
#define RT_DEVICE_FLAG_REMOVABLE 0x004 /* 可移除 */  
#define RT_DEVICE_FLAG_STANDALONE 0x008 /* 独立 */  
#define RT_DEVICE_FLAG_SUSPENDED 0x020 /* 挂起 */  
#define RT_DEVICE_FLAG_STREAM 0x040 /* 流模式 */  
#define RT_DEVICE_FLAG_INT_RX 0x100 /* 中断接收 */  
#define RT_DEVICE_FLAG_DMA_RX 0x200 /* DMA 接收 */  
#define RT_DEVICE_FLAG_INT_TX 0x400 /* 中断发送 */  
#define RT_DEVICE_FLAG_DMA_TX 0x800 /* DMA 发送 */
```

5. 实验1 | 注册字符设备 test

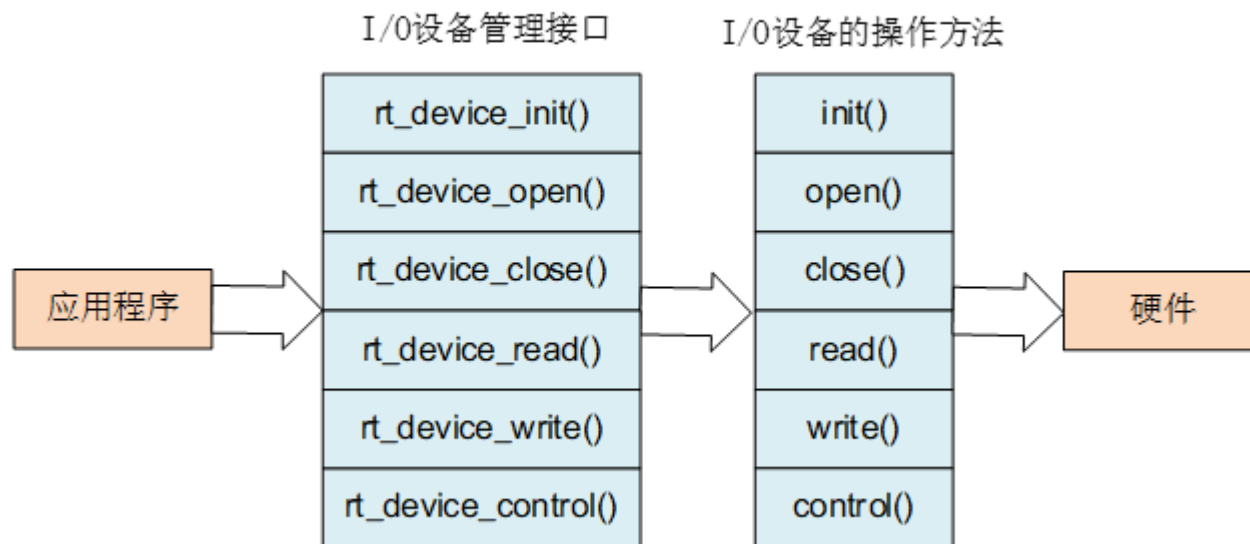
```
applications > C dev_test.c > rt_dev_test_init(void)
1  ✓ #include <rtthread.h>
2  ✓ #include <rtdevice.h>
3
4  ✓ static int rt_dev_test_init(void)
5  {
6      rt_device_t test_dev = rt_device_create(RT_Device_Class_Char, 0);
7  ✓  if(!test_dev)
8      {
9          rt_kprintf("test_dev create failed");
10         return -RT_ERROR;
11     }
12
13  ✓  if(rt_device_register(test_dev, "test_dev", RT_DEVICE_FLAG_RDWR) != RT_EOK)
14      {
15          rt_kprintf("test_dev register failed!\n");
16          return -RT_ERROR;
17      }
18
19      return RT_EOK;
20  }
21  INIT_DEVICE_EXPORT(rt_dev_test_init);
```

```
\ | /
- RT -   Thread Operating System
/ | \   5.2.0 build Jul 23 2024 11:02:52
2006 - 2024 Copyright by RT-Thread team
msh >list device
device          type          ref count
-----
test_dev Character Device      0
uart1  Character Device      2
pin     Pin Device            0
msh >
```

6. 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下图所示：

```
struct rt_device_ops
{
    /* common device interface */
    rt_err_t (*init) (rt_device_t dev);
    rt_err_t (*open) (rt_device_t dev, rt_uint16_t oflag);
    rt_err_t (*close) (rt_device_t dev);
    rt_size_t (*read) (rt_device_t dev, rt_off_t pos, void *buffer, rt_size_t size);
    rt_size_t (*write) (rt_device_t dev, rt_off_t pos, const void *buffer, rt_size_t size);
    rt_err_t (*control)(rt_device_t dev, int cmd, void *args);
};
```



7. 查找设备/初始化设备

- `rt_device_t rt_device_find(const char* name);`
- `rt_err_t rt_device_init(rt_device_t dev);`

| 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 |

| 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 |

8. 打开和关闭设备

- `rt_err_t rt_device_open(rt_device_t dev, rt_uint16_t oflags);`
- `rt_err_t rt_device_close(rt_device_t dev);`

| 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 |

| 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 |

9. 打开标志位

```
#define RT_DEVICE_OFLAG_CLOSE 0x000 /* 设备已经关闭（内部使用） */
#define RT_DEVICE_OFLAG_RDONLY 0x001 /* 以只读方式打开设备 */
#define RT_DEVICE_OFLAG_WRONLY 0x002 /* 以只写方式打开设备 */
#define RT_DEVICE_OFLAG_RDWR 0x003 /* 以读写方式打开设备 */
#define RT_DEVICE_OFLAG_OPEN 0x008 /* 设备已经打开（内部使用） */
#define RT_DEVICE_FLAG_STREAM 0x040 /* 设备以流模式打开 */
#define RT_DEVICE_FLAG_INT_RX 0x100 /* 设备以中断接收模式打开 */
#define RT_DEVICE_FLAG_DMA_RX 0x200 /* 设备以 DMA 接收模式打开 */
#define RT_DEVICE_FLAG_INT_TX 0x400 /* 设备以中断发送模式打开 */
#define RT_DEVICE_FLAG_DMA_TX 0x800 /* 设备以 DMA 发送模式打开 */
```

❶ 注意事项

如果上层应用程序需要设置设备的接收回调函数，则必须以 `RT_DEVICE_FLAG_INT_RX` 或者 `RT_DEVICE_FLAG_DMA_RX` 的方式打开设备，否则不会回调函数。

10. 控制设备

- `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 |

参数 cmd 的通用设备命令可取如下宏定义:

```
#define RT_DEVICE_CTRL_RESUME      0x01  /* 恢复设备 */
#define RT_DEVICE_CTRL_SUSPEND    0x02  /* 挂起设备 */
#define RT_DEVICE_CTRL_CONFIG     0x03  /* 配置设备 */
#define RT_DEVICE_CTRL_SET_INT    0x10  /* 设置中断 */
#define RT_DEVICE_CTRL_CLR_INT    0x11  /* 清中断 */
#define RT_DEVICE_CTRL_GET_INT    0x12  /* 获取中断状态 */
```

11. 读写设备

- `rt_size_t rt_device_read(rt_device_t dev, rt_off_t pos,void* buffer, rt_size_t size);`
- `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 | 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 <code>errno</code> 来判断错误状态 |

| 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 |

12. 数据接收回调

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));`

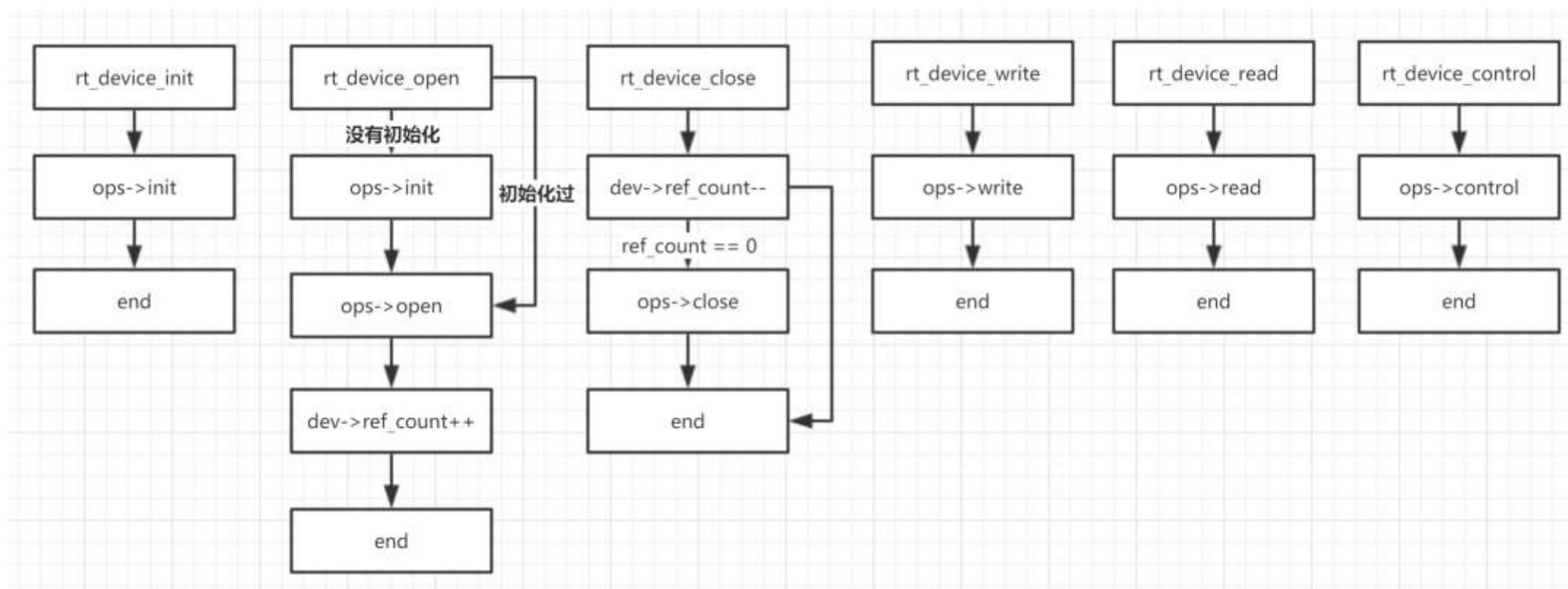
| 参数 | 描述 |
|--------|--------|
| dev | 设备句柄 |
| rx_ind | 回调函数指针 |
| 返回 | —— |
| RT_EOK | 设置成功 |

13. 数据接收回调

- 在应用程序调用 `rt_device_write()` 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，函数参数及返回值见：
- `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 |

14. IO框架调用关系图



15. 实验二： 添加 TEST 驱动的OPS接口

```
static rt_err_t drv_test_init(rt_device_t dev)
{
    LOG_I("test drv init\n");
    return RT_EOK;
}

static rt_err_t drv_test_open(rt_device_t dev, rt_uint16_t oflag)
{
    LOG_I("test drv open flag = %d\n", oflag);
    return RT_EOK;
}

static rt_err_t drv_test_close(rt_device_t dev)
{
    LOG_I("test drv close");
    return RT_EOK;
}

static rt_ssize_t drv_test_read(rt_device_t dev, rt_off_t pos, void *buffer, rt_size_t size)
{
    LOG_I("test drv read pos = %d, size = %d\n", pos, size);
    return size;
}

static rt_ssize_t drv_test_write(rt_device_t dev, rt_off_t pos, const void *buffer, rt_size_t size)
{
    LOG_I("test drv write pos = %d, size = %d\n", pos, size);
    return size;
}

static rt_err_t drv_test_control(rt_device_t dev, int cmd, void *args)
{
    LOG_I("test drv control cmd = %d\n", cmd);
    return RT_EOK;
}
```

```
int rt_drv_test_init(void)
{
    rt_device_t test_dev = rt_device_create(RT_Device_Class_Char, 0);
    if(!test_dev)
    {
        LOG_E("test drv create failed!\n");
        return -RT_ERROR;
    }

    test_dev->init = drv_test_init;
    test_dev->open = drv_test_open;
    test_dev->close = drv_test_close;
    test_dev->read = drv_test_read;
    test_dev->write = drv_test_write;
    test_dev->control = drv_test_control;

    if(rt_device_register(test_dev, "test_drv", RT_DEVICE_FLAG_RDWR) != RT_EOK)
    {
        LOG_E("test drv register failed!\n");
        return -RT_ERROR;
    }

    return RT_EOK;
}

INIT_BOARD_EXPORT(rt_drv_test_init);
```


16. 代码分析

- 分析这段代码的打印结果

```
static int drv_test_app(void)
{
    rt_device_t test_dev = rt_device_find("test_drv");
    if(test_dev == RT_NULL)
    {
        LOG_E("can not find test drv!");
        return -RT_ERROR;
    }

    rt_device_open(test_dev, RT_DEVICE_OFLAG_RDWR);
    rt_device_control(test_dev, RT_DEVICE_CTRL_CONFIG, RT_NULL);
    rt_device_write(test_dev, 100, RT_NULL, 1024);
    rt_device_read(test_dev, 20, RT_NULL, 128);

    rt_device_close(test_dev);

    return RT_EOK;
}
_MSH_CMD_EXPORT(drv_test_app, enable test drv app);
```

```
\ | /
- RT -   Thread Operating System
/ | \   5.2.0 build Jul 23 2024 11:46:45
2006 - 2024 Copyright by RT-Thread team
msh >list device
device          type          ref count
-----
test_drv        Character Device  0
uart1           Character Device  2
pin             Pin Device      0
msh >drv_test_app
[12609] I/drv.test: test drv init

[12614] I/drv.test: test drv open flag = 3

[12619] I/drv.test: test drv control cmd = 3

[12625] I/drv.test: test drv write pos = 100, size = 1024

[12631] I/drv.test: test drv read pos = 20, size = 128

[12638] I/drv.test: test drv close
msh >
```




培训3 - RT-Thread GPIO 外设开发

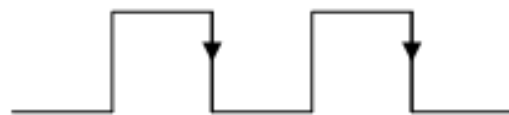
2.1 GPIO概念与原理

- 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，一般有以下所示 5 种中断触发模式：

2.2 可编程控制中断



上升沿触发：用于检测无抖动的上升沿



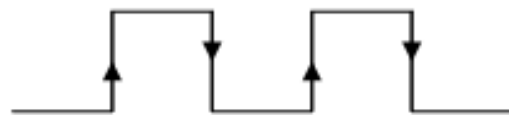
下降沿触发：用于检测无抖动的下降沿



高电平触发：用于检测高电平状态



低电平触发：用于检测低电平状态



双边沿触发：用于检测无抖动的双边沿

2.3 应用开发 | 常用接口

- 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:
- `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

2.4 配置GPIO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);
```

- mode 可取以下宏定义值之一：

```
#define PIN_MODE_OUTPUT 0x00      /* 输出 */  
#define PIN_MODE_INPUT 0x01      /* 输入 */  
#define PIN_MODE_INPUT_PULLUP 0x02 /* 上拉输入 */  
#define PIN_MODE_INPUT_PULLDOWN 0x03 /* 下拉输入 */  
#define PIN_MODE_OUTPUT_OD 0x04 /* 开漏输出 */
```

2.5 应用开发 | 输出高低电平

- 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);
```

- value 可取 2 种宏定义值之一：PIN_LOW 低电平，PIN_HIGH 高电平

2.6 应用开发 | 读取引脚电平

- 读取引脚输入电平的函数如下所示：

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

- 返回值为 2 种宏定义值之一：PIN_LOW 低电平，PIN_HIGH 高电平

2.7 实验三：外部中断

```
#include <rtthread.h>
#include <rtdevice.h>
#include <drv_gpio.h>

#define LOG_TAG    "drv_test.app"
#define LOG_LVL    LOG_LVL_DBG
#include <ulog.h>

#define KEY_UP      GET_PIN(C, 5)
#define KEY_DOWN    GET_PIN(C, 1)
#define KEY_LEFT    GET_PIN(C, 0)
#define KEY_RIGHT   GET_PIN(C, 4)

void key_up_callback(void *args)
{
    int value = rt_pin_read(KEY_UP);
    LOG_I("key up! %d", value);
}

void key_down_callback(void *args)
{
    int value = rt_pin_read(KEY_DOWN);
    LOG_I("key down! %d", value);
}

void key_left_callback(void *args)
{
    int value = rt_pin_read(KEY_LEFT);
    LOG_I("key left! %d", value);
}

void key_right_callback(void *args)
{
    int value = rt_pin_read(KEY_RIGHT);
    LOG_I("key right! %d", value);
}
```

```
void irq_key_enable()
{
    rt_pin_mode(KEY_UP,    PIN_MODE_INPUT_PULLUP);
    rt_pin_mode(KEY_DOWN,  PIN_MODE_INPUT_PULLUP);
    rt_pin_mode(KEY_LEFT,  PIN_MODE_INPUT_PULLUP);
    rt_pin_mode(KEY_RIGHT, PIN_MODE_INPUT_PULLUP);

    rt_pin_attach_irq(KEY_UP,    PIN_IRQ_MODE_FALLING, key_up_callback,    RT_NULL);
    rt_pin_attach_irq(KEY_DOWN,  PIN_IRQ_MODE_FALLING, key_down_callback,  RT_NULL);
    rt_pin_attach_irq(KEY_LEFT,  PIN_IRQ_MODE_FALLING, key_left_callback,  RT_NULL);
    rt_pin_attach_irq(KEY_RIGHT, PIN_IRQ_MODE_FALLING, key_right_callback, RT_NULL);

    rt_pin_irq_enable(KEY_UP,    PIN_IRQ_ENABLE);
    rt_pin_irq_enable(KEY_DOWN,  PIN_IRQ_ENABLE);
    rt_pin_irq_enable(KEY_LEFT,  PIN_IRQ_ENABLE);
    rt_pin_irq_enable(KEY_RIGHT, PIN_IRQ_ENABLE);
}

MSH_CMD_EXPORT(irq_key_enable, enable key irq);
```

```
\ | /
- RT -   Thread Operating System
/ | \   5.2.0 build Jul 23 2024 14:13:25
2006 - 2024 Copyright by RT-Thread team
msh >irq_key_enable
msh >[13153] I/drv_test.app: key up! 0
[14960] I/drv_test.app: key left! 0
[15596] I/drv_test.app: key down! 0
[16334] I/drv_test.app: key right! 0
```


2.8 按键库的使用演示

- 勾选 FlexibleButton 按键库

```
(Top) → RT-Thread online packages → miscellaneouspackages r
RT-Thread Configuration
project laboratory --->
samples: kernel and components samples --->
entertainment: terminal games and other interesting software packages --->
[ ] libcsv: a small, simple and fast CSV library written in pure ANSI C89 that can read and write CSV data. --->
[ ] optparse: a public domain, portable, reentrant, embeddable, getopt-like option parser. --->
[ ] Fastlz: A portable real-time compression library --->
[ ] minilzo: A mini subset of the LZ0 real-time data compression library --->
[ ] QuickLZ : Fast data compression library --->
[ ] LZMA: A compression library with high compression ratio --->
[ ] ralarm: Infinitely scalable alarm components --->
[ ] MultiButton: A compact and easy to use event driven button driver --->
[*] FlexibleButton: Small and flexible button driver --->
[ ] CanFestival: A free software CANopen framework --->
[ ] zlib: general purpose data compression library --->
[ ] minizip: zip manipulation library --->
[ ] heatshrink: A data compression/decompression library for embedded/real-time systems --->
[ ] dstr: a dynamic string package for rt-thread --->
[ ] TinyFrame: Serial communication protocol. --->
[ ] kendryte k210 demo package --->
[ ] upacker: building and parsing data frames to be sent over a serial interface --->
[ ] uparam: Manage system parameters with FLASH --->
[ ] Hello: A example package --->
[ ] vi: A screen-oriented text editor --->
[ ] ki: A small text editor in less than 1K lines of code --->
[ ] armv7m_dwt: High precision timing and delay --->
↓↓↓↓↓↓↓↓↓↓↓↓ show-all mode enabled
[Space/Enter] Toggle/enter [ESC] Leave menu [S] Save
[O] Load [?] Symbol info [/] Jump to symbol
[F] Toggle show-help mode [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```



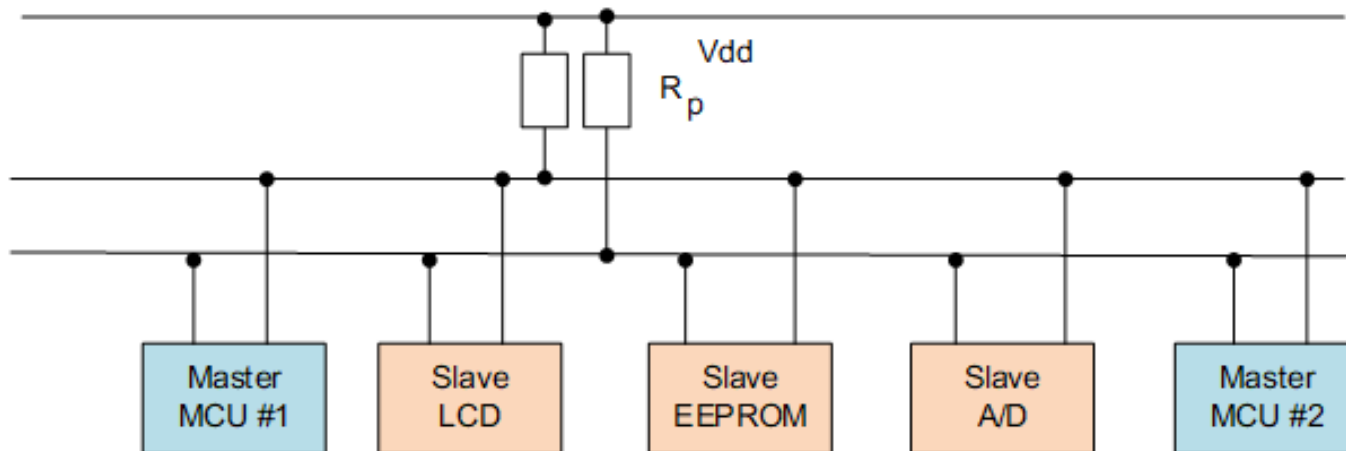
培训3 - RT-Thread I2C 外设开发

3.1 I2C总线简介

I2C 是 Inter-Integrated Circuit 的简称，读作：I-squared-C。由飞利浦公司于1980年代提出，为了让主板、嵌入式系统或手机用以连接低速周边外部设备而发展。

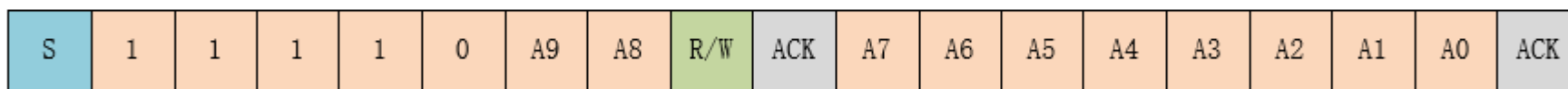
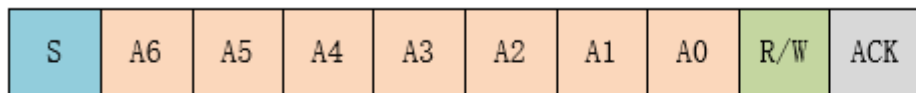
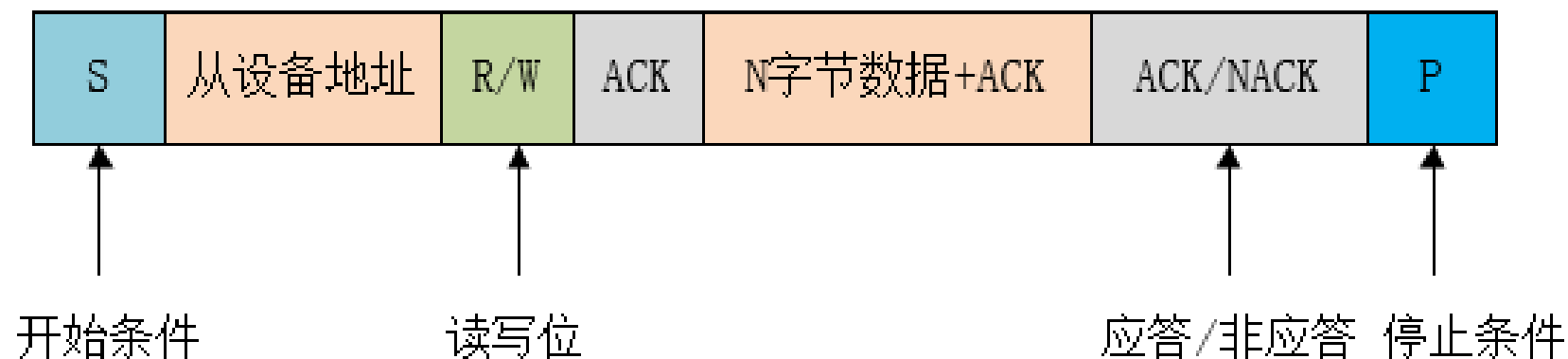
常见的I2C总线以传输速率的不同分为不同的模式：

- 低速模式：10Kbit/s
- 标准模式：100Kbit/s
- 快速模式：400Kbit/s
- 高速模式：3.4Mbit/s



3.3 I2C 总线协议 | 传输格式

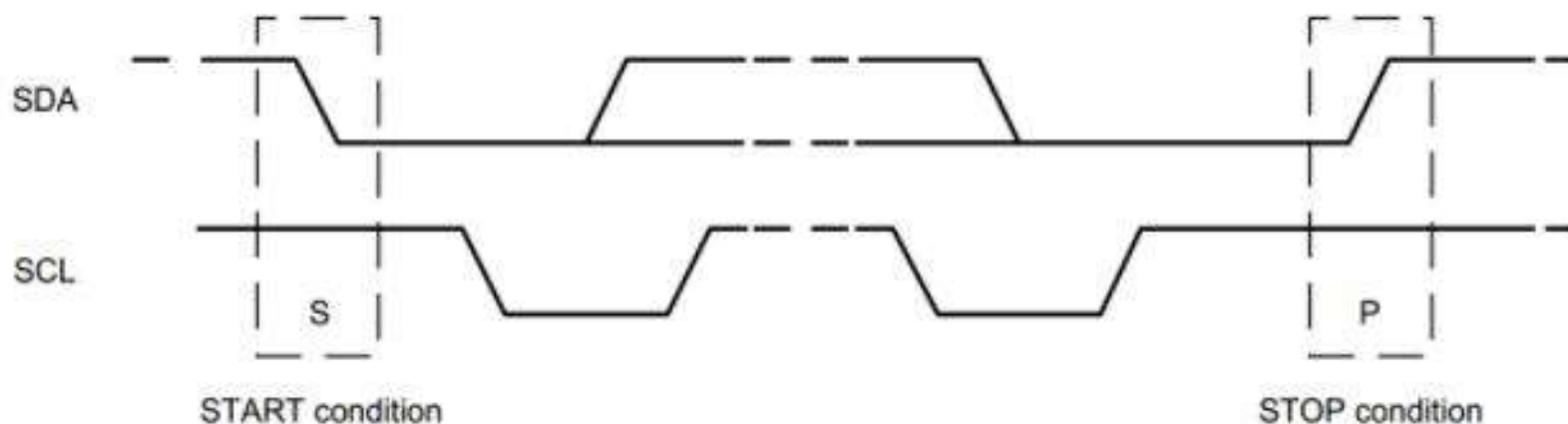
I2C 总线数据传输格式：



3.2 I2C 总线协议 | 起始位和结束位

起始位和结束位：

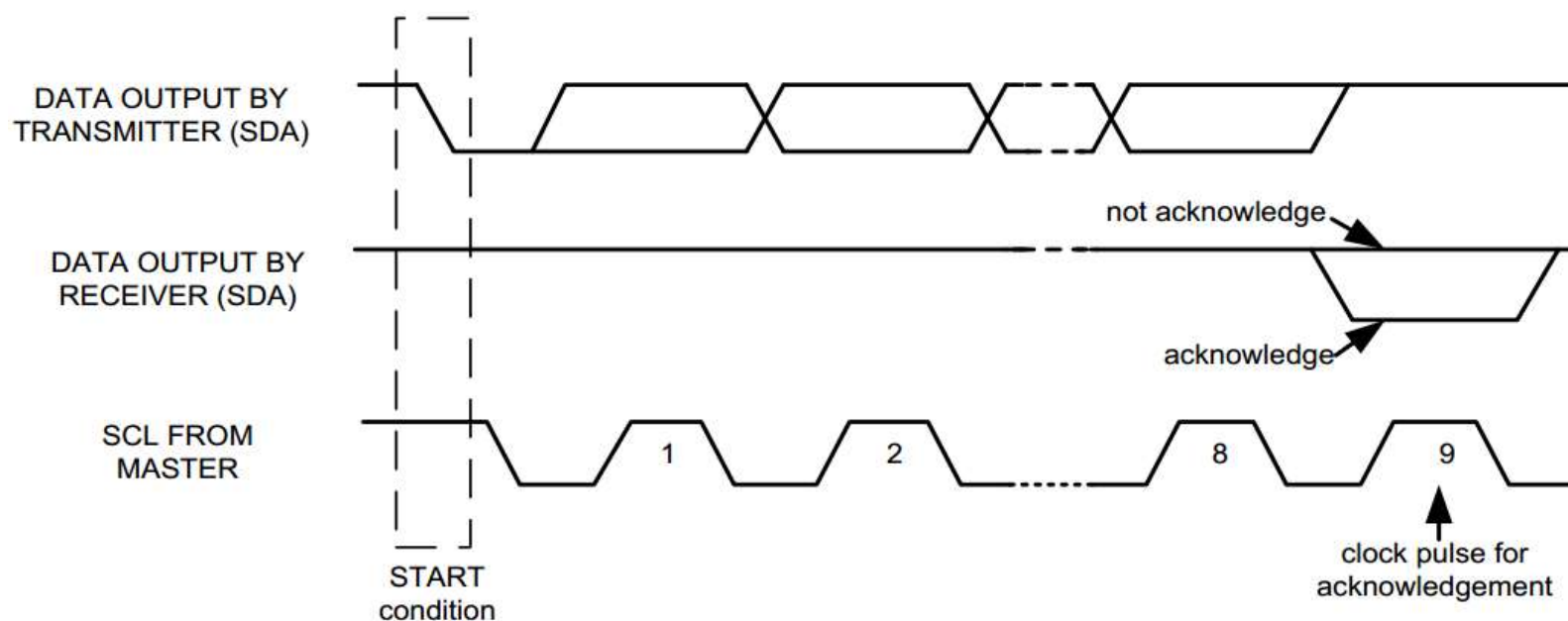
- 起始位（S）：在SCL为高电平时，SDA由高电平变为低电平
- 结束位（P）：在SCL为高电平时，SDA由低电平变为高电平



START and STOP Conditions

3.3 I2C 总线协议 | ACK NACK

- **应答 (ACK)**：拉低SDA线，并在SCL为高电平期间保持SDA线为低电平
- **非应答 (NOACK)**：不要拉低SDA线（此时SDA线为高电平），并在SCL为高电平期间保持SDA线为高电平

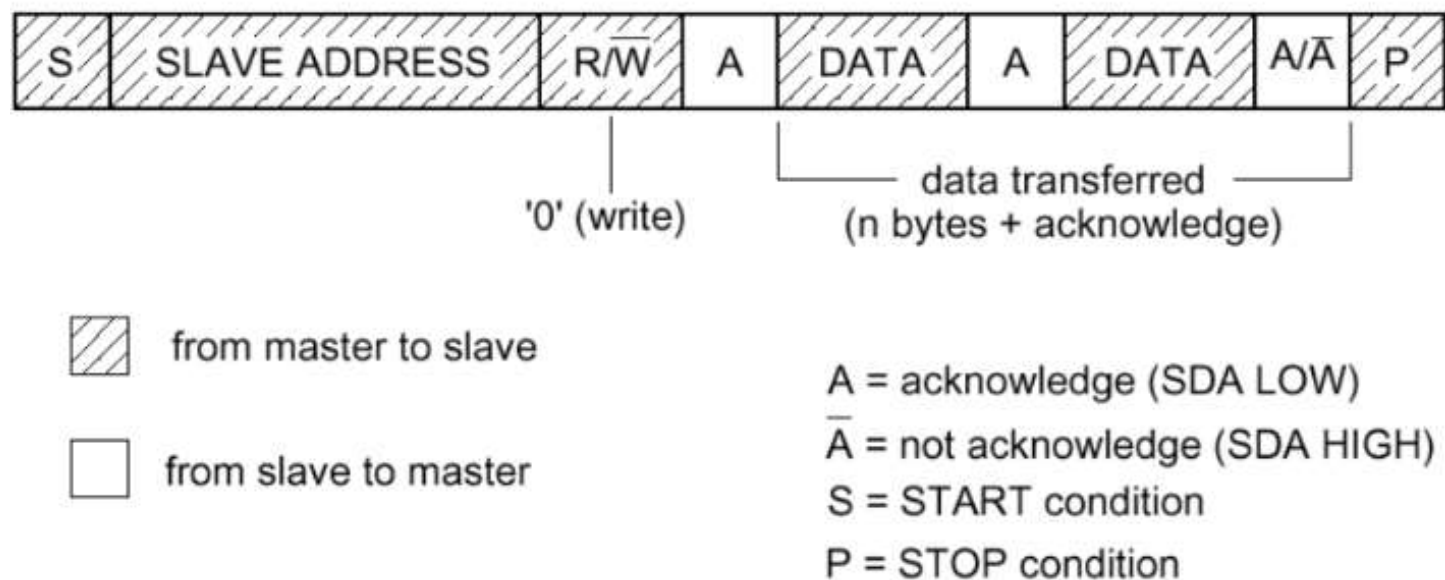


Acknowledge on the I²C Bus

3.4 I2C 总线协议 | 主机向从机写数据

- 写数据：

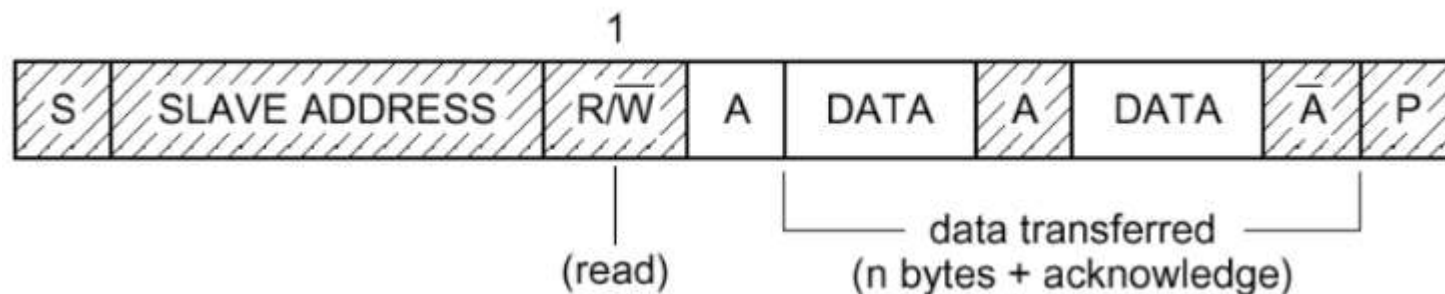
开始数据传输后，先发送一个**起始位（S）**，主设备**发送一个地址数据**（由7bit的从设备地址，和最低位的写标志位组成的8bit字节数据，该读写标志位决定数据的传输方向），然后，主设备释放SDA线，并**等待从设备的应答信号（ACK）**。每一个字节数据的传输都要跟一个应答信号位。数据传输以**停止位（P）结束**，并且释放I2C总线。



3.5 I2C 总线协议 | 主机向从机读数据

- 读数据:

开始通讯时，主设备先发送一个**起始信号 (S)**，主设备发送一个**地址数据**（由7bit的从设备地址，和最低位的写标志位组成的8bit字节数据），然后，主设备释放SDA线，并**等待从设备的应答信号 (ACK)**，从设备应答主设备后，主设备再**发送要读取的寄存器地址**，从设备应答主设备**(ACK)**，主设备再次发送**起始信号 (S)**，主设备发送**设备地址**（包含读标志），从设备**应答**主设备，并将该**寄存器的值**发送给主设备；



3.7 I2C从机常用模式

- 向I2C从机设备，某个寄存器写一个字节数据：**改变传感器寄存器的值。**
- 向I2C从机设备，某个寄存器写多个字节数据：**同上。**
- 向I2C从机设备，从某个寄存器读取一个字节数据：**读取传感器寄存器状态。**
- 向I2C从机设备，从某个寄存器读取多个字节数据：**读取传感器数据。**

3.8 查看I2C总线设备

- 开启 I2C 驱动后，使用 `list_device` 命令查看总线设备注册情况。

```
\ | /
- RT -   Thread Operating System
/ | \   5.2.0 build Jul 23 2024 15:38:54
2006 - 2024 Copyright by RT-Thread team

msh >
RT-Thread shell commands:
reboot      - Reboot System
pwm         - control pwm device
pin         - pin [option]
clear       - clear the terminal screen
version     - show RT-Thread version information
list        - list objects
help        - RT-Thread shell help
ps          - List threads in the system
free        - Show the memory usage in the system
backtrace   - print backtrace of a thread

msh >list device
device      type      ref count
-----
i2c3       I2C Bus      0
uart1      Character Device 2
pin        Pin Device  0
msh >
```

3.9 I2C设备探测Package

- 勾选使用 i2c-tools 软件包，方便对 i2c 设备进行调试。

```
(Top) → RT-Thread online packages → peripheral libraries and drivers
↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑ RT-Thread Configuration
[ ] Signalled: a signal led package for rt-thread ----
[ ] LedBlink: easy led blink support lib ----
[ ] littled: Little LED Daemon for LED driver ----
[ ] lkdGui: a monochrome graphic library. ----
[ ] infrared : infrared is base on rt-thread pin, hwtimer and pwm. ----
[ ] multi_infrared : multi_infrared is base on rt-thread pin ----
[ ] agile_button: A agile button package. ----
[ ] agile_led: A agile led package. ----
[ ] at24cxx: eeprom at24cxx driver library. ----
[ ] MotionDriver2RTT: A package porting MotionDriver to RTT ----
[ ] pca9685: I2C-bus controlled 16-channel PWM controller ----
[ ] TFT-LCD ILI9341 SPI screen driver software package ----
[*] i2c-tools: a collection of i2c tools including scan/read/write --->
[ ] nrf24l01: Single-chip 2.4GHz wireless transceiver. ----
[ ] RPLIDAR: a low cost LIDAR sensor suitable for indoor robotic SLAM application. ----
[ ] AS608 fingerprint module driver ----
[ ] rc522: rfid module driver ----
[ ] ws2812b: Ws2812b software driver package using SPI+DMA ----
[ ] extern rtc drivers ----
[ ] multi_rtimer : a real-time and low power software timer module. ----
[ ] MAX7219: for the digital tube ----
[ ] beep: Control the buzzer to make beeps at different intervals. ----
[ ] easyblink: Blink the LED easily and use a little RAM ----
[ ] pms_series: Digital universal particle concentration sensor driver library ----
[ ] CAN_YMODEM: a device connect can & ymodem ----
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
[Space/Enter] Toggle/enter [ESC] Leave menu [S] Save
[O] Load [?] Symbol info [/] Jump to symbol
[F] Toggle show-help mode [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

3.10 设备探测效果

- 探测总线上的设备：i2c scan i2c3

```
\ | /  
- RT -   Thread Operating System  
/ | \    5.2.0 build Jul 24 2024 10:07:05  
2006 - 2024 Copyright by RT-Thread team  
msh >i2c scan 65 64  
      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  
00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  
30: -- -- -- -- -- -- -- 38 -- -- -- -- -- -- --  
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  
70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- --  
msh >
```

3.11 API | 查找设备

- 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);
```

- 一般情况下，注册到系统的 I2C 设备名称为 i2c0 ， i2c1 等，使用示例如下所示：

```
struct rt_i2c_bus_device *i2c_bus;          /* I2C总线设备句柄 */  
  
/* 查找I2C总线设备，获取I2C总线设备句柄 */  
i2c_bus = (struct rt_i2c_bus_device *)rt_device_find(name);
```

3.12 API | 传输

```
rt_size_t rt_i2c_transfer(struct rt_i2c_bus_device *bus,
                          struct rt_i2c_msg         msgs[],
                          rt_uint32_t               num);

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 */
}
```

- I2C 设备接口使用的从机地址均不包含读写位，读写位控制需修改标志 flags。

```
#define RT_I2C_WR          0x0000    /* 写标志 */
#define RT_I2C_RD          (1u << 0) /* 读标志 */
#define RT_I2C_ADDR_10BIT  (1u << 2) /* 10 位地址模式 */
#define RT_I2C_NO_START    (1u << 4) /* 无开始条件 */
#define RT_I2C_IGNORE_NACK (1u << 5) /* 忽视 NACK */
#define RT_I2C_NO_READ_ACK (1u << 6) /* 读的时候不发送 ACK */
```

❗ 注意事项

此函数会调用 `rt_mutex_take()`，不能在中断服务程序里面调用，会导致 assertion 报错。

3.12 I2C消息数据类型

I2C 消息数据结构原型如下：包括从机地址、读写标志、读写数据字节数、读写缓冲区指针

这里需要注意从机地址 **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。

标志 **flags** 可取值为以下宏定义，根据需要可以与其他宏使用位运算 “|” 组合起来使用。

注：RT-Thread I2C 设备接口使用的从机地址均不包含读写位，读写位控制需修改标志 **flags**。

```
struct rt_i2c_msg
{
    rt_uint16_t addr;
    rt_uint16_t flags;
    rt_uint16_t len;
    rt_uint8_t *buf;
};
```

```
#define RT_I2C_WR          0x0000    /* 写标志，不可以和读标志进行“|”操作 */
#define RT_I2C_RD          (1u << 0) /* 读标志，不可以和写标志进行“|”操作 */
#define RT_I2C_ADDR_10BIT  (1u << 2) /* 10 位地址模式 */
#define RT_I2C_NO_START    (1u << 4) /* 无开始条件 */
#define RT_I2C_IGNORE_NACK (1u << 5) /* 忽视 NACK */
#define RT_I2C_NO_READ_ACK (1u << 6) /* 读的时候不发送 ACK */
#define RT_I2C_NO_STOP     (1u << 7) /* 不发送结束位 */
```

3.13 RT-Thread I2C 使用思路

1. 查找 I2C 总线设备
2. 构造 msgs 消息
3. 启动 transfer 传输
4. 处理结果

3.14 写一个字节数据

如何编写代码

```
#include <rtthread.h>
#include <rtdevice.h>

void i2c_sample_single_byte_write(void)
{
    struct rt_i2c_bus_device *i2c_bus;
    struct rt_i2c_msg msgs;
    rt_uint8_t buf[2];

    i2c_bus = (struct rt_i2c_bus_device *)rt_device_find("i2c2");
    if(i2c_bus == RT_NULL)
    {
        rt_kprintf("can't find %s device!\n", "i2c2");
    }

    buf[0] = 0x6B;

    msgs.addr = 0x68;
    msgs.flags = RT_I2C_WR;
    msgs.buf = buf;
    msgs.len = 1;

    if(rt_i2c_transfer(i2c_bus, &msgs, 1) == 1)
        rt_kprintf("single byte write success!\n");
    else
        rt_kprintf("single byte write failed...\n");
}

MSH_CMD_EXPORT(i2c_sample_single_byte_write, i2c_sample_single_byte_write);
```

| SINGLE-BYTE WRITE | | |
|-------------------|-------|---------------|
| MASTER | START | SLAVE ADDRESS |
| SLAVE | | |

3.15 写多字节数据

如何编写代码

```
void i2c_sample_multi_byte_write(void)
{
    struct rt_i2c_bus_device *i2c_bus;
    struct rt_i2c_msg msgs;
    rt_uint8_t buf[3];

    i2c_bus = (struct rt_i2c_bus_device *)rt_device_find("i2c2");
    if(i2c_bus == RT_NULL)
    {
        rt_kprintf("can't find %s device!\n", "i2c2");
    }

    buf[0] = 0x01;
    buf[1] = 0x02;
    buf[2] = 0x03;

    msgs.addr = 0x68;
    msgs.flags = RT_I2C_WR;
    msgs.buf = buf;
    msgs.len = 3;

    if(rt_i2c_transfer(i2c_bus, &msgs, 1) == 1)
        rt_kprintf("multi byte write success!\n");
    else
        rt_kprintf("multi byte write failed...\n");
}

MSH_CMD_EXPORT(i2c_sample_multi_byte_write, i2c_sample_multi_byte_write);
```

| MULTIPLE-BYTE WRITE | | |
|---------------------|-------|---------------|
| MASTER | START | SLAVE ADDRESS |
| SLAVE | | |

| | | |
|---|-----|------|
| A | | STOP |
| | ACK | |

3.16 读数据

如

```
void i2c_sample_single_byte_read(void)
{
    struct rt_i2c_bus_device *i2c_bus;
    struct rt_i2c_msg msgs[2];
    rt_uint8_t send_buf[1], recv_buf[1];

    i2c_bus = (struct rt_i2c_bus_device *)rt_device_find("i2c2");
    if(i2c_bus == RT_NULL)
    {
        rt_kprintf("can't find %s device!\n", "i2c2");
    }

    send_buf[0] = 0x6B;
    recv_buf[0] = 0x6A;

    msgs[0].addr = 0x68;
    msgs[0].flags = RT_I2C_WR;
    msgs[0].buf = send_buf;
    msgs[0].len = 1;

    msgs[1].addr = 0x68;
    msgs[1].flags = RT_I2C_RD;
    msgs[1].buf = recv_buf;
    msgs[1].len = 1;

    if(rt_i2c_transfer(i2c_bus, msgs, 2) == 2)
        rt_kprintf("single byte read: 0x%02x success!\n", recv_buf[0]);
    else
        rt_kprintf("single byte read failed...\n");
}

MSH_CMD_EXPORT(i2c_sample_single_byte_read, i2c_sample_single_byte_read);
```

| | |
|-------------|-----|
| SINGLE-BYTE | |
| MASTER | STA |
| SLAVE | |

| | |
|------|------|
| | |
| NACK | STOP |
| | |

3.17 I2C应用开发常见遇到的错误

- 单个设备挂掉导致总线死锁
- `rt_i2c_transfer`函数执行返回-5

3.18 i2c总线死锁原因

- 当 I2C 主机正在和从机通信时，如果主机正好发生打算发第9个时钟，此时**SCL**为高，而从机开始拉低**SDA**为低做准备（作为**ACK**信号），等待主机**SCL**变低后，从再释放**SDA**为高。
- 如果此时正好主机复位，主机**SCL**还没来得及变低，直接变成高电平，此时从机还在等待**SCL**变低，所以一直拉低**SDA**；而主机由于复位，发现**SDA**一直为低，也在等待从释放**SDA**为高。因此主机和从机都进入一个相互等待的死锁状态。

3.19 i2c总线死锁解锁思路

I2C主设备中增加I2C总线解锁程序，方法如下：

I2C 主设备启动传输前，先控制 I2C 中的 SCL 时钟线产生 9 个时钟脉冲（针对8位数据的情况）这样I2C从设备就可以完成被挂起的读操作，从死锁状态中恢复过来。

```
173 static rt_err_t stm32_i2c_bus_unlock(const struct stm32_soft_i2c_config *cfg)
174 {
175     rt_int32_t i = 0;
176
177     if (PIN_LOW == rt_pin_read(cfg->sda))
178     {
179         while (i++ < 9)
180         {
181             rt_pin_write(cfg->scl, PIN_HIGH);
182             stm32_udelay(100);
183             rt_pin_write(cfg->scl, PIN_LOW);
184             stm32_udelay(100);
185         }
186     }
187     if (PIN_LOW == rt_pin_read(cfg->sda))
188     {
189         return -RT_ERROR;
190     }
191
192     return RT_EOK;
193 }
```

3.20 rt_i2c_transfer函数执行返回 -8

- 设备地址错误:

```
\ | /  
- RT -   Thread Operating System  
/ | \  
2006 - 2024 Copyright by RT-Thread team  
msh > i2c_sample_error  
single byte read -5 failed...  
msh >
```

```
void i2c_sample_error(void)  
{  
    rt_err_t ret = RT_EOK;  
    struct rt_i2c_bus_device *i2c_bus;  
    struct rt_i2c_msg msgs[2];  
    rt_uint8_t send_buf[1], recv_buf[1];  
  
    i2c_bus = (struct rt_i2c_bus_device *)rt_device_find("i2c2");  
    if(i2c_bus == RT_NULL)  
    {  
        rt_kprintf("can't find i2c device!\n");  
    }  
  
    send_buf[0] = 0x6B;  
    recv_buf[0] = 0;  
  
    msgs[0].addr = 0x68;  
    msgs[0].flags = RT_I2C_WR;  
    msgs[0].buf = send_buf;  
    msgs[0].len = 1;  
  
    msgs[1].addr = 0x6D;  
    msgs[1].flags = RT_I2C_RD;  
    msgs[1].buf = recv_buf;  
    msgs[1].len = 1;  
  
    ret = rt_i2c_transfer(i2c_bus, msgs, 2);  
    if(ret == 2)  
        rt_kprintf("single byte read: %02x success!\n", recv_buf[0]);  
    else  
        rt_kprintf("single byte read %d failed...\n", ret);  
}  
MESH_CMD_EXPORT(i2c_sample_error, i2c_sample_error);
```

3.21 软件i2c驱动编写

- 开启 I2C 框架
- 选中 I2C 软件模拟设备功能
- 编写 I2C 软件模拟驱动

3.22 menuconfig配置内核I2C

```
(Top) → RT-Thread Components→ Device Drivers
RT-Thread Configuration
[ ] Enable device driver model with device tree
[ ] Using Device Bus device drivers
-*- Using device drivers IPC --->
-*- USING Serial device drivers --->
[ ] Using CAN device drivers
[ ] Enable CPU time for high resolution clock counter
-*- Using I2C device drivers
[ ]     Use I2C debug message
-*-     Use GPIO to simulate I2C
[ ]         Use simulate I2C debug message
[ ]         Use GPIO to soft simulate I2C
[ ] Using ethernet phy device drivers
[ ] Using ADC device drivers
[ ] Using DAC device drivers
[ ] Using NULL device drivers
[ ] Using ZERO device drivers
[ ] Using RANDOM device drivers
[*] Using PWM device drivers
[ ] Using PULSE ENCODER device drivers
[ ] Using INPUT CAPTURE device drivers
[ ] Using MTD Nor Flash device drivers
[ ] Using MTD Nand Flash device drivers
[ ] Using Power Management device drivers
[ ] Using RTC device drivers
[ ] Using SD/MMC device drivers
↓↓↓↓↓↓↓↓↓↓↓↓↓↓
[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                [?] Symbol info          [/] Jump to symbol
[F] Toggle show-help mode [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

3.23 编写软件I2C驱动模板

```
rw007 sn: [rw0072795b244008b49]
rw007 ver: [1.2.8-f7698586-18979]

list_de
list_device
msh />list_device
device          type          ref count
-----
w1              Network Interface 0
w0              Network Interface 0
wlan0           Network Interface 1
wlan1           Network Interface 1
wspi            SPI Device       0
sd0             Block Device     1
rtc             RTC              0
record          Pipe             0
mic0            Sound Device     0
sound0          Sound Device     0
i2c_sim         I2C Bus          0
i2c2            I2C Bus          1
i2c1            I2C Bus          0
spi2            SPI Bus          0
uart3           Character Device 2
pin             Miscellaneous Device 0
msh />
```

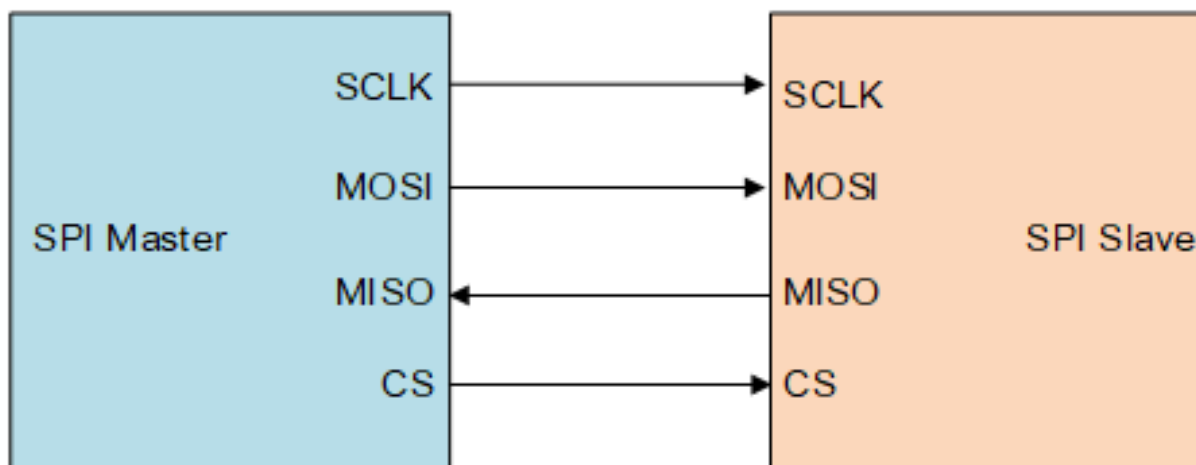
```
1  #include <rtthread.h>
2  #include <rtdevice.h>
3  #include <rthw.h>
4
5  static void udelay(rt_uint32_t us)
6  {
7      volatile rt_int32_t i = us;
8      while (i-- > 0) i = i;
9  }
10
11 static void set_sda(void *data, rt_int32_t state){}
12 static void set_scl(void *data, rt_int32_t state){}
13 static rt_int32_t get_sda(void *data){return 0;}
14 static rt_int32_t get_scl(void *data){return 0;}
15
16 static const struct rt_i2c_bit_ops bit_ops =
17 {
18     RT_NULL,
19     set_sda, set_scl,
20     get_sda, get_scl,
21     udelay, 20, 50
22 };
23 static struct rt_i2c_bus_device i2c_bus;
24
25 int rt_sw_i2c_init(void)
26 {
27     i2c_bus.priv = (void *)&bit_ops;
28     rt_i2c_bit_add_bus(&i2c_bus, "i2c_sim");
29     return RT_EOK;
30 }
31 INIT_DEVICE_EXPORT(rt_sw_i2c_init);
```



培训4 - RT-Thread SPI 外设开发

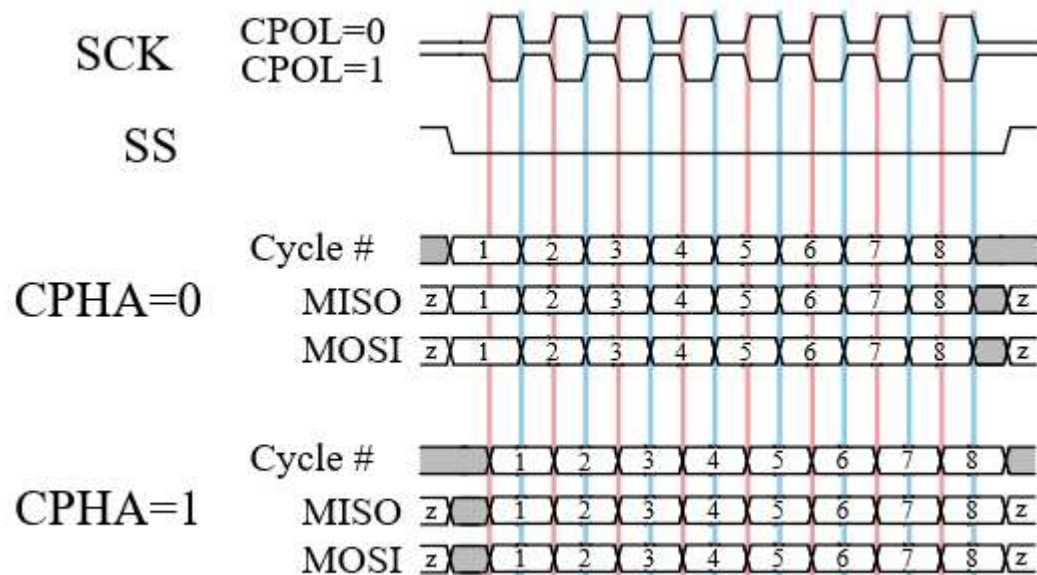
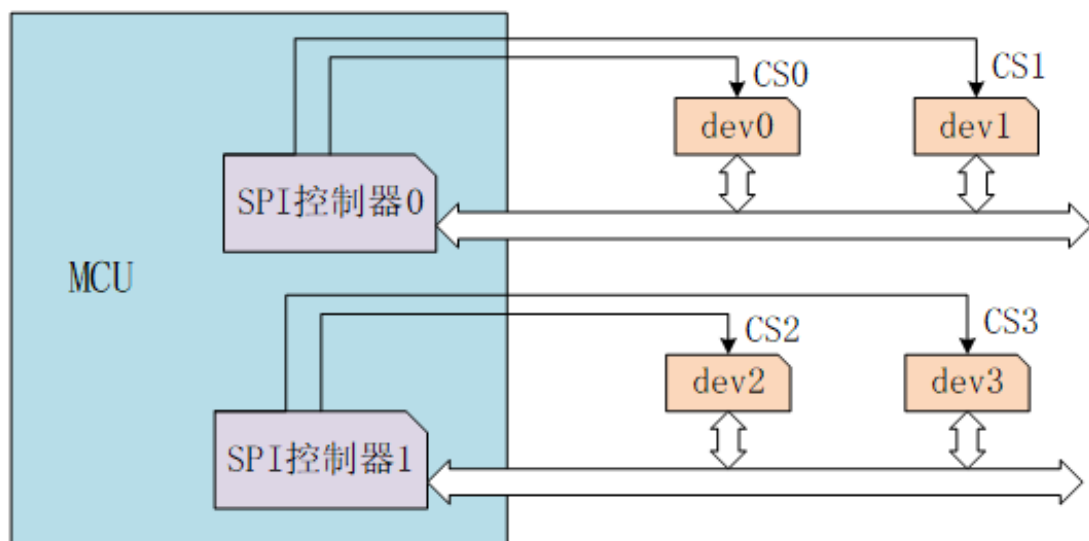
4.1 SPI总线概念与原理

- **SPI**（**S**erial **P**eripheral **I**nterface，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，主要应用于 **EEPROM**、**FLASH**、实时时钟、**AD** 转换器、还有数字信号处理器和数字信号解码器之间。**SPI** 一般使用 4 根线通信，如下图所示：



4.2 概念与原理

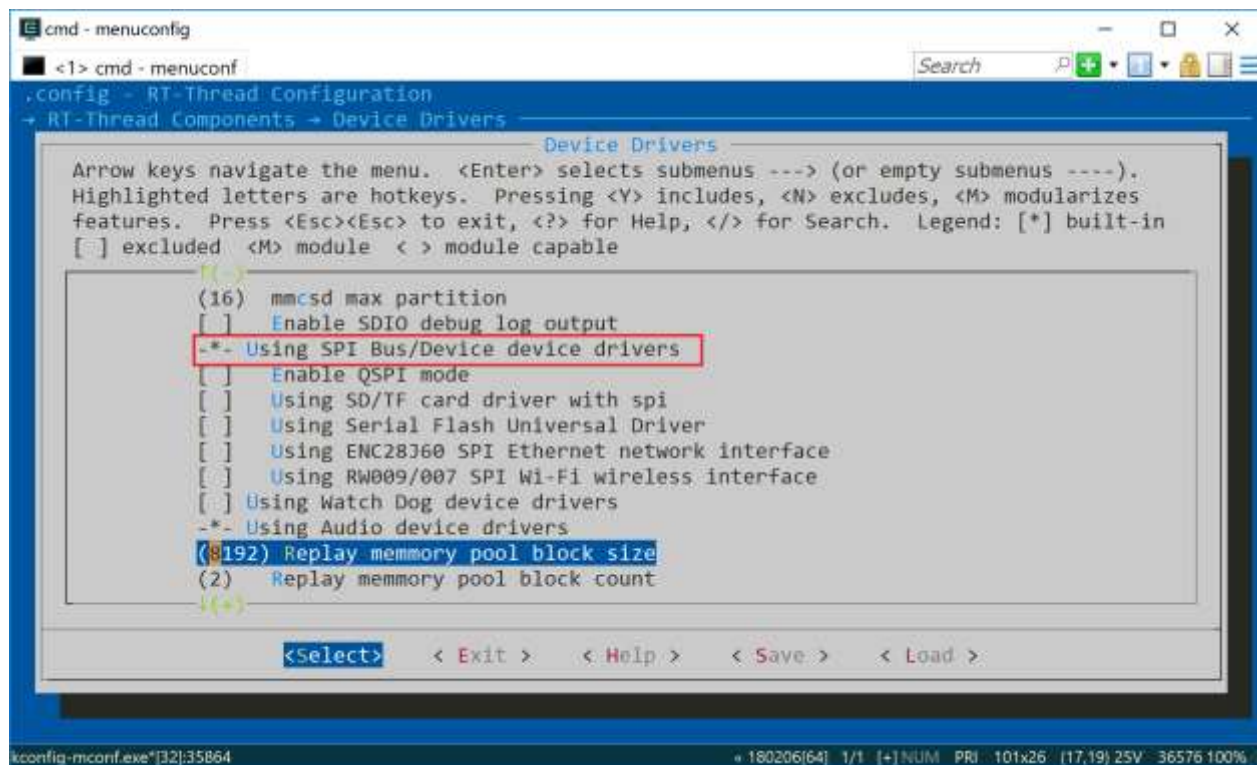
- 在 RT-Thread 中，SPI 设备分为“SPI 总线”和“SPI 设备”两大类，SPI 总线对应 SPI 控制器，SPI 设备对应不同 CS 连接的从设备，使用前需要先注册 SPI 总线，再把从设备挂载到总线上。



4.3 RT-Thread SPI 开发模式

- 编写 SPI BUS 驱动
- 注册 SPI Device 设备
- 打开 SPI Device 设备
- 使用 SPI 框架提供 API 编程发送接收数据
- 关闭 SPI Device 设备

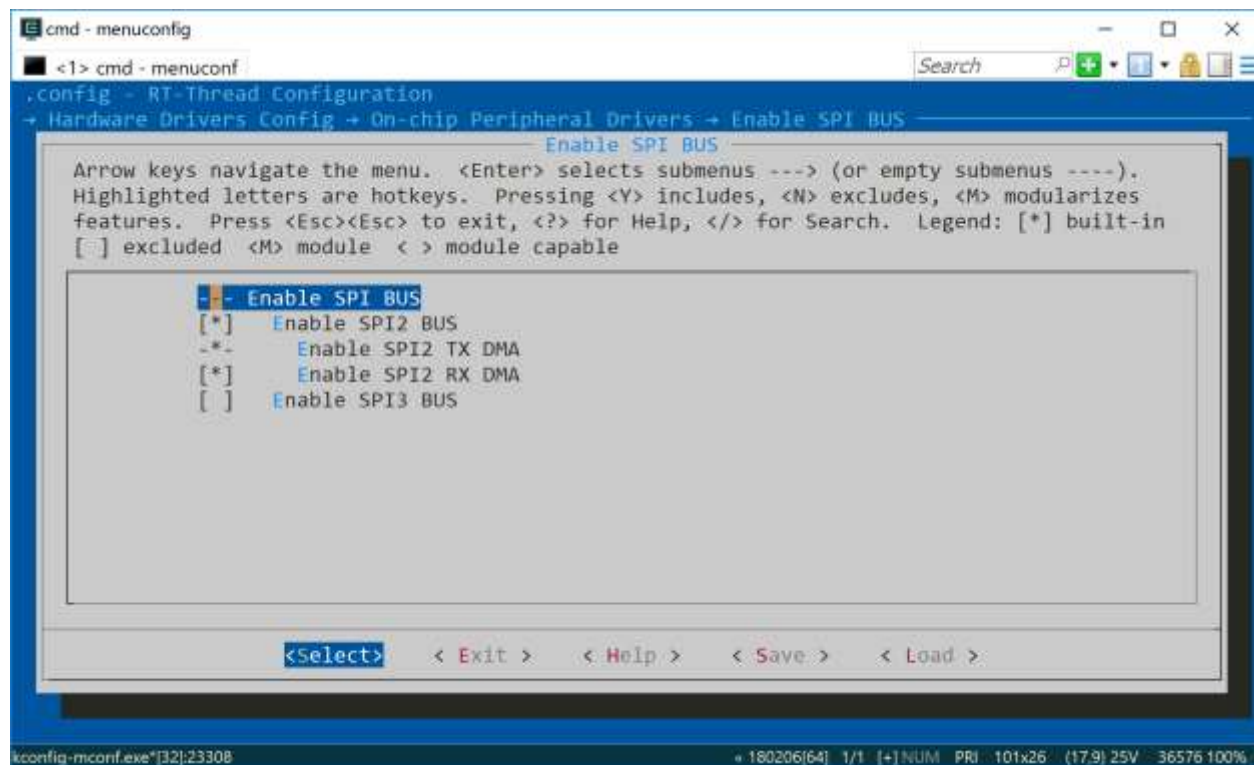
4.4 打开 SPI 框架 / 添加 SPI BUS 驱动



```
cmd - menuconfig
.config - RT-Thread Configuration
-> RT-Thread Components -> Device Drivers
    Device Drivers
    Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----).
    Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
    features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in
    [-] excluded <M> module < > module capable
    (16) mmc sd max partition
    [-] Enable SDIO debug log output
    [*] Using SPI Bus/Device drivers
    [-] Enable QSPI mode
    [-] Using SD/TF card driver with spi
    [-] Using Serial Flash Universal Driver
    [-] Using ENC28J60 SPI Ethernet network interface
    [-] Using RW009/007 SPI Wi-Fi wireless interface
    [-] Using Watch Dog device drivers
    [*] Using Audio device drivers
    (192) Replay memory pool block size
    (2)  Replay memory pool block count
    <Select> < Exit > < Help > < Save > < Load >
```

kconfig-mconf.exe*[32]:35864

• 180206[64] 1/1 [+] NUM PRI 101x26 (17,19) 25V 36576 100%



```
cmd - menuconfig
.config - RT-Thread Configuration
-> Hardware Drivers Config -> On-chip Peripheral Drivers -> Enable SPI BUS
    Enable SPI BUS
    Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----).
    Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
    features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in
    [-] excluded <M> module < > module capable
    [*] Enable SPI BUS
    [*] Enable SPI2 BUS
    [*] Enable SPI2 TX DMA
    [*] Enable SPI2 RX DMA
    [-] Enable SPI3 BUS
    <Select> < Exit > < Help > < Save > < Load >
```

kconfig-mconf.exe*[32]:23308

• 180206[64] 1/1 [+] NUM PRI 101x26 (17,9) 25V 36576 100%

4.5 注册SPI设备

- 挂载 SPI 设备：SPI 驱动会注册 SPI 总线，SPI 设备需要挂载到已经注册好的 SPI 总线上。

```
rt_err_t rt_spi_bus_attach_device(struct rt_spi_device *device,  
                                const char             *name,  
                                const char             *bus_name,  
                                void                   *user_data)
```

- 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。

4.6 SPI 注册设备(STM32 BSP)

- 若使用 `rt-thread/bsp/stm32` 目录下的 BSP 则可以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);
```

- 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。

4.7 演示 | 注册 SPI 设备

```
#include <rtthread.h>
#include <rtdevice.h>

#include "drv_spi.h"
#include "drv_gpio.h"

static int spi_attach(void)
{
    return rt_hw_spi_device_attach("spi2", "spi20", GET_PIN(B, 12));
}

INIT_DEVICE_EXPORT(spi_attach);
```

```
msh >
\ | /
- RT -   Thread Operating System
/ | \   5.2.0 build Jul 24 2024 19:31:34
2006 - 2024 Copyright by RT-Thread team
msh >list device
device          type          ref count
-----
spi20           SPI Device      0
spi2            SPI Bus         0
i2c2            I2C Bus         0
i2c1            I2C Bus         0
uart1           Character Device 2
pin             Pin Device      0
msh >
```

4.8 控制 SPI 设备相关API

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

❶ 注意事项

SPI 数据传输相关接口会调用 `rt_mutex_take()`, 此函数不能在中断服务程序里面调用, 会导致 assertion 报错。

4.9 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);
```

- 一般情况下，注册到系统的 SPI 设备名称为 spi10，使用示例如下所示：

```
#define W25Q_SPI_DEVICE_NAME    "qspi10"    /* SPI 设备名称 */
struct rt_spi_device *spi_dev_w25q;        /* SPI 设备句柄 */

/* 查找 spi 设备获取设备句柄 */
spi_dev_w25q = (struct rt_spi_device *)rt_device_find(W25Q_SPI_DEVICE_NAME);
```

4.10 SPI设备驱动配置

```
rt_err_t rt_spi_configure(struct rt_spi_device *device,  
                          struct rt_spi_configuration *cfg)
```

```
struct rt_spi_configuration  
{  
    rt_uint8_t mode;           /* 模式 */  
    rt_uint8_t data_width;     /* 数据宽度, 可取8位、16位、32位 */  
    rt_uint16_t reserved;      /* 保留 */  
    rt_uint32_t max_hz;        /* 最大频率 */  
};  
  
/* 设置数据传输顺序是MSB位在前还是LSB位在前 */  
#define RT_SPI_LSB      (0<<2)      /* bit[2]: 0-LSB */  
#define RT_SPI_MSB      (1<<2)      /* bit[2]: 1-MSB */  
  
/* 设置SPI的主从模式 */  
#define RT_SPI_MASTER    (0<<3)      /* SPI master device */  
#define RT_SPI_SLAVE     (1<<3)      /* SPI slave device */  
  
/* 设置时钟极性和时钟相位 */  
#define RT_SPI_MODE_0    (0 | 0)      /* CPOL = 0, CPHA = 0 */  
#define RT_SPI_MODE_1    (0 | RT_SPI_CPHA) /* CPOL = 0, CPHA = 1 */  
#define RT_SPI_MODE_2    (RT_SPI_CPOL | 0) /* CPOL = 1, CPHA = 0 */  
#define RT_SPI_MODE_3    (RT_SPI_CPOL | RT_SPI_CPHA) /* CPOL = 1, CPHA = 1 */  
  
#define RT_SPI_CS_HIGH   (1<<4)      /* Chipselect active high */  
#define RT_SPI_NO_CS     (1<<5)      /* No chipselect */  
#define RT_SPI_3WIRE     (1<<6)      /* SI/SO pin shared */  
#define RT_SPI_READY     (1<<7)      /* Slave pulls Low to pause */
```

4.11 SPI设备传输数据(一次通信)

- 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);
```

- 此函数不需要手动控制片选，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。

4.12 演示 | SPI设备传输数据(一次通信)

```
#include <rtthread.h>
#include <rtdevice.h>

#include "drv_spi.h"
#include "drv_gpio.h"

static int spi_attach(void)
{
    return rt_hw_spi_device_attach("spi2", "spi20", GET_PIN(B, 12));
}
MSH_CMD_EXPORT(spi_attach, spi_attach);

static int spi_example(void)
{
    rt_err_t ret = RT_EOK;
    struct rt_spi_device * spi20 = (struct rt_spi_device *)rt_device_find("spi20");

    struct rt_spi_configuration cfg;
    cfg.data_width = 8;
    cfg.mode = RT_SPI_MASTER | RT_SPI_MODE_0 | RT_SPI_MSB;
    cfg.max_hz = 1 * 1000 * 1000; /* 1M */
    rt_spi_configure(spi20, &cfg);

    rt_uint8_t sendBuff = 0xDA;
    rt_uint8_t recvBuff = 0xF1;
    ret = rt_spi_transfer(spi20, &sendBuff, &recvBuff, 1);
    rt_kprintf("ret = %d\n", ret);

    return ret;
}
MSH_CMD_EXPORT(spi_example, spi_example);
```

4.12 单独发送数据(一次通信)

- 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)
```

- 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);
```


4.12 演示 | 单独发送数据(一次通信)

```
static int spi_send_one_data(void)
{
    rt_err_t ret = RT_EOK;
    struct rt_spi_device * spi20 = (struct rt_spi_device *)rt_device_find("spi20");

    struct rt_spi_configuration cfg;
    cfg.data_width = 8;
    cfg.mode = RT_SPI_MASTER | RT_SPI_MODE_0 | RT_SPI_MSB;
    cfg.max_hz = 1 * 1000 * 1000; /* 1M */
    rt_spi_configure(spi20, &cfg);

    rt_uint8_t sendBuff = 0x1A;
    ret = rt_spi_send(spi20, &sendBuff, 1);
    rt_kprintf("ret = %d\n", ret); // return actual num

    return ret;
}
MSH_CMD_EXPORT(spi_send_one_data, spi_send_one_data);
```

4.12 演示 | 单独接收数据(一次通信)

```
static int spi_recv_one_data(void)
{
    rt_err_t ret = RT_EOK;
    struct rt_spi_device * spi20 = (struct rt_spi_device *)rt_device_find("spi20");

    struct rt_spi_configuration cfg;
    cfg.data_width = 8;
    cfg.mode = RT_SPI_MASTER | RT_SPI_MODE_0 | RT_SPI_MSB;
    cfg.max_hz = 1 * 1000 * 1000; /* 1M */
    rt_spi_configure(spi20, &cfg);

    rt_uint8_t recvBuff = 0x1A;
    ret = rt_spi_recv(spi20, &recvBuff, 1); // return actual num
    rt_kprintf("ret = %d\n", ret);

    return ret;
}
_MSH_CMD_EXPORT(spi_recv_one_data, spi_recv_one_data);
```

4.13 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);
```

- 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，适用于先发送地址，再发送指定长度的数据，中途不释放片选的情况。

4.13 演示 | 连续两次发送数据

```
static int spi_send_then_send_data(void)
{
    rt_err_t ret = RT_EOK;
    struct rt_spi_device * spi20 = (struct rt_spi_device *)rt_device_find("spi20");

    struct rt_spi_configuration cfg;
    cfg.data_width = 8;
    cfg.mode = RT_SPI_MASTER | RT_SPI_MODE_0 | RT_SPI_MSB;
    cfg.max_hz = 1 * 1000 * 1000; /* 1M */
    rt_spi_configure(spi20, &cfg);

    rt_uint8_t sendBuff1[2] = {0x1A, 0x99};
    rt_uint8_t sendBuff2[2] = {0x12, 0x39};
    ret = rt_spi_send_then_send(spi20, &sendBuff1, 2, &sendBuff2, 2);
    rt_kprintf("ret = %d\n", ret); // RT_EOK means success

    return ret;
}
MSH_CMD_EXPORT(spi_send_then_send_data, spi_send_then_send_data);
```

4.14 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_rcv(struct rt_spi_device *device,  
                               const void          *send_buf,  
                               rt_size_t           send_length,  
                               void               *recv_buf,  
                               rt_size_t           rcv_length);
```

- 函数发送第一条数据时开始片选，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。

4.15 演示 | 先发送后接收数据

```
static int spi_send_then_recv_data(void)
{
    rt_err_t ret = RT_EOK;
    struct rt_spi_device * spi20 = (struct rt_spi_device *)rt_device_find("spi20");

    struct rt_spi_configuration cfg;
    cfg.data_width = 8;
    cfg.mode = RT_SPI_MASTER | RT_SPI_MODE_0 | RT_SPI_MSB;
    cfg.max_hz = 1 * 1000 * 1000; /* 1M */
    rt_spi_configure(spi20, &cfg);

    rt_uint8_t sendBuff1[2] = {0x1A, 0x99};
    rt_uint8_t sendBuff2[2] = {0x12, 0x39};
    ret = rt_spi_send_then_recv(spi20, &sendBuff1, 2, &sendBuff2, 2);
    rt_kprintf("ret = %d\n", ret); // RT_EOK means success

    return ret;
}

MSH_CMD_EXPORT(spi_send_then_recv_data, spi_send_then_recv_data);
```