RT-THREAD USER GUIDE

RT-THREAD 文档中心

WWW.RT-THREAD.ORG

Monday 6th May, 2019

目录

第 1 章

Env 用户手册

Env 是 RT-Thread 推出的开发辅助工具，针对基于 RT-Thread 操作系统的项目工程，提供编译构建环境、图形化系统配置及软件包管理功能。

其内置的 menuconfig 提供了简单易用的配置剪裁工具，可对内核、组件和软件包进行自由裁剪，使系统以搭积木的方式进行构建。

1.1 主要特性

- menuconfig 图形化配置界面，交互性好，操作逻辑强；
- 丰富的文字帮助说明，配置无需查阅文档；
- 使用灵活，自动处理依赖，功能开关彻底；
- 自动生成 rtconfig.h，无需手动修改；
- 使用 scons 工具生成工程，提供编译环境，操作简单；
- 提供多种软件包，模块化软件包耦合关联少，可维护性好；
- 软件包可在线下载，软件包持续集成，包可靠性高；

1.2 准备工作

Env 工具包含了 RT-Thread 源代码开发编译环境和软件包管理系统。

- 从 RT-Thread 官网下载 Env 工具。
- 在电脑上装好 git，软件包管理功能需要 git 的支持。git 的下载地址为`https://git-scm.com/downloads`，根据向导正确安装 git，并将 git 添加到系统环境变量。
- 注意在工作环境中，所有的路径都不可以有中文字符或者空格。

1.3 Env 的使用方法

1.3.1 打开 Env 控制台

RT-Thread 软件包环境主要以命令行控制台为主，同时以字符型界面来进行辅助，使得尽量减少修改配置文件的方式即可搭建好 RT-Thread 开发环境的方式。打开 Env 控制台有两种方式：

1.3.1.1 方法一：点击 Env 目录下可执行文件

进入 Env 目录，可以运行本目录下的 `env.exe`，如果打开失败可以尝试使用 `env.bat`。

1.3.1.2 方法二：在文件夹中通过右键菜单打开 Env 控制台

Env 目录下有一张 `Add_Env_To_Right-click_Menu.png`(添加 Env 至右键菜单.png) 的图片，如下：

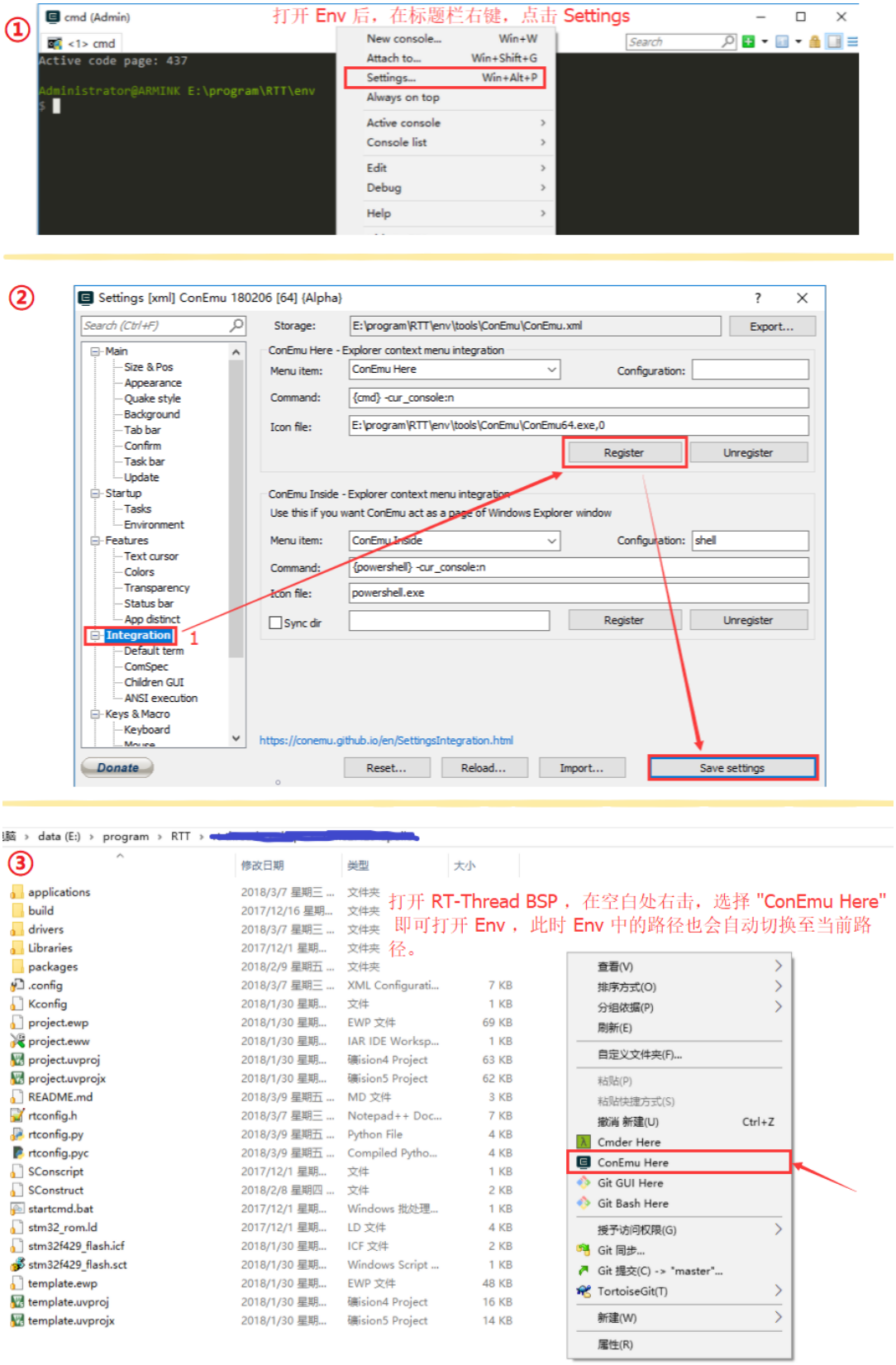


图 1.1: *添加 Env 控制台到右键菜单*

根据图片上的步骤操作，就可以在任意文件夹下通过右键菜单来启动 Env 控制台。效果如下：

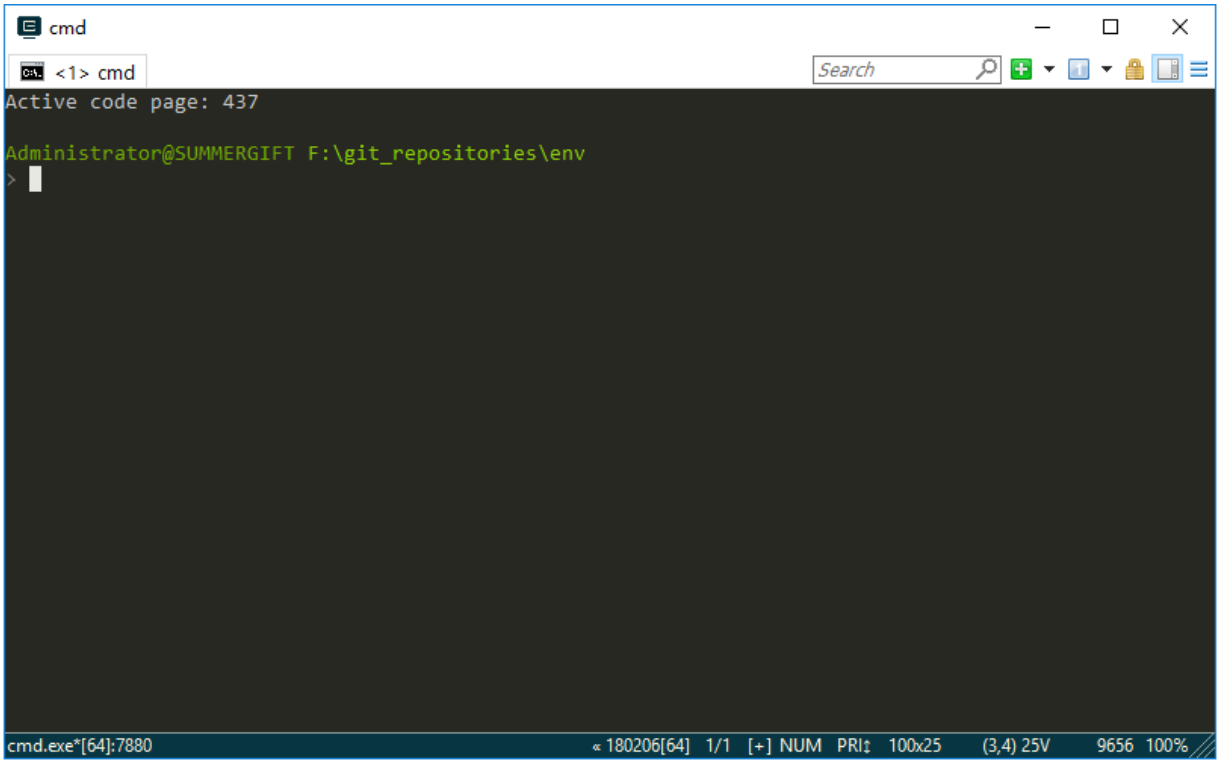


图 1.2: *通过右键菜单来启动 Env 控制台*

!!! note "注意事项" 因为需要设置 Env 进程的环境变量，第一次启动可能会出现杀毒软件误报的情况，如果遇到了 **杀毒软件误报**，允许 Env 相关程序运行，然后将相关程序添加至白名单即可。

1.3.2 编译工程

scons 是 RT-Thread 使用的编译构建工具，可以使用 scons 相关命令来编译 RT-Thread 。

- Env 中携带了 `Python` & `scons` 环境，只需在 `bk7221u` 目录中运行 `scons` 命令即可使用默认的 ARM_GCC 工具链编译工程。

```
rtthread@LAPTOP-A25O1HO7 D:\work\beken\beken_audio_release\bk7221u
> scons -j7
scons: Reading SConscript files ...
RTT_ROOT is: D:\work\beken\beken_audio_release\bk7221u\rt-thread
scons: done reading SConscript files.

scons: warning: you do not seem to have the pywin32 extensions installed;
        parallel (-j) builds may not work reliably with open Python files.
File "D:\work\env\tools\Python27\Scripts\scons.py", line 199, in <module>
scons: Building targets ...
scons: building associated VariantDir targets: build
CC build\applications\main.o
CC build\applications\msh_evm.o
CC build\applications\romfs.o
CC build\beken378\app\app.o
CC build\beken378\app\config\param_config.o
CC build\beken378\app\standalone-ap\sa_ap.o
CC build\beken378\app\standalone-station\sa_station.o
CC build\beken378\demo\ieee802_11_demo.o
CC build\beken378\driver\codec\driver_codec_es8374.o
CC build\beken378\driver\common\dd.o
CC build\beken378\driver\common\drv_model.o
CC build\beken378\driver\dma\dma.o
CC build\beken378\driver\driver.o
CC build\beken378\driver\entry\arch_main.o
CC build\beken378\driver\fft\fft.o
```

图 1.3: *scons 命令编译工程*

编译成功：

```
        ^
CC packages\zlib\zlib\zutil.o
LINK rtthread.elf
arm-none-eabi-objcopy -O binary rtthread.elf rtthread.bin
arm-none-eabi-size rtthread.elf
   text    data     bss     dec     hex filename
 942334   13884   65180 1021398   f95d6 rtthread.elf
scons: done building targets.

rtthread@LAPTOP-A25O1HO7 D:\work\beken\beken_audio_release\bk7221u
>
```

图 1.4: *编译工程成功*

第 2 章

romfs 用户手册

2.1 mkromfs 介绍

mkromfs 是 RT-Thread 开发的将内部指定区域的 flash 制作成文件系统的一款工具，可以通过操作文件系统的方式操作内部 flash 。通常 romfs 用来存放提示音的音频文件。

2.2 文件目录结构

```
romfs
|    dfs_romfs.c
|    dfs_romfs.h
|    mkromfs.py
|    romfs.c
|    SConscript
```

2.3 mkromfs 使用说明

进入 `rt-thread\components\dfs\filesystems\romfs\mkromfs.py` , `mkromfs.py` 就是制作 romfs 的指令

2.3.1 创建文件夹

在 `rt-thread\components\dfs\filesystems\romfs` 路径下新建文件夹 romfs , 在该文件夹下放入 `mp3/pcm` 文件, 并且在该路径下打开 env 工具。使用 `mkromfs.py romfs romfs_root.bin --binary --addr 0x220000` 生成 romfs_root.bin ,

参数介绍

参数	说明
`mkromfs.py`	制作 romfs 的脚本

参数	说明
romfs	自己新建的文件夹名称，可修改为其他名称
romfs_root.bin	生成后的 bin 文件
--binary	表示生成 bin 文件
--addr 0x220000	对应分区表 romfs 中的逻辑地址

2.3.2 修改 romfs.c

在 applications\romfs.c 路径下，添加如下代码

```
static const struct romfs_dirent _romfs_root_romfs[] = {

};

static const struct romfs_dirent _romfs_root[] = {
    {ROMFS_DIRENT_DIR, "sd", (rt_uint8_t *)_romfs_root_sd, sizeof(_romfs_root_sd)/sizeof(_romfs_root_sd[0])},
    {ROMFS_DIRENT_DIR, "romfs", (rt_uint8_t *)_romfs_root_romfs, sizeof(_romfs_root_romfs)/sizeof(_romfs_root_romfs[0])}
};
                    与之前创建的文件夹要保持一致
```

这时创建了一个节点，名称为 romfs ，但是该节点还未 mount 到根节点上。

2.3.3 挂载 romfs 节点到文件系统上

挂载文件系统的根路径，在 main.c 中添加如下代码

```
#define TONE_START_OFFSETADDR    (0x220000)    ///romfs分区表中逻辑地址，此处要和--
    addr对应

/* mount ROMFS as root directory */
if (dfs_mount(RT_NULL, "/", "rom", 0, (const void *)DFS_ROMFS_ROOT) == 0)
{
    rt_kprintf("ROMFS File System initialized!\n");
}
else
{
    rt_kprintf("ROMFS File System initialized Failed!\n");
}
```

```
if (dfs_mount(RT_NULL, "/romfs", "rom", 0, (const void *)TONE_START_OFFSETADDR) ==
    0)
        rt_kprintf("ROMFS tone File System initialized!\n");
    else
        rt_kprintf("ROMFS tone File System initialized Failed!\n");
```

保存，编译，下载到开发板。

2.4 使用 ota 打包工具将 romfs_root.bin 打包成 romfs_root.rbl

使用 ota 打包时，注意 app 区改成 romfs 。然后 ota 升级，下载 `romfs_root.rbl` 到开发板。

```
http_ota http://192.168.0.85/romfs_root.rbl
```

升级成功后，msh 下输入如下命令，就可以看见音频文件了。

```
msh />ls
Directory /:
sd                  <DIR>
romfs               <DIR>
msh />cd romfs
msh /romfs>ls
Directory /romfs:
gushi.mp3           14112
gushi1.mp3          14112
```

```
gushi2.mp3          14112
gushi3.mp3          14112
gushi4.mp3          14112
gushi5.mp3          14112
gushi6.mp3          14112
msh /romfs>
```

第 3 章

fatdisk 用户手册

3.1 fatdisk 介绍

fatdisk 是 RT-Thread 开发的将 flash 制作成 fat 文件系统的一款工具，可以通过操作文件系统的方式操作 flash 。

3.2 文件目录结构

```
fatdisk
|    fatdisk.exe
|    fatdisk.xml
|    README.md
|    rtthread-win32.exe
```

3.3 fatdisk 使用说明

3.3.1 fatdisk.xml 说明

配置文件，默认不需要修改。

3.3.2 fatdisk.xml 中字段含义

参数	介绍
disk_size	flash 空间的总大小，单位 kbytes(注：此大小为 download 分区大小减去 1k。eg：如果 download 分区大小为 512k，此处填 511)
sector_size	flash 扇区大小，(默认 512)

参数	介绍
root_dir	存放音频文件的文件名 (用户自己创建)
output	生成 bin 文件的路径
strip	默认为 1 即可

注：FAT 复用分区表的 download 分区。

3.3.3 fatdisk.exe

镜像打包工具

3.4 使用说明

3.4.1 生成 flash0.rbl

1. 在 `tools/fatdisk` 路径下新建一个目录，如 flash0。
2. 把需要打包的文件放到 flash0 目录下。
3. 然后运行 fatdisk.exe 即可生成 `flash0.bin`。

留意最后生成的 flash0.bin 体积不要超过`679KB`。因为 BK7251 默认 download 分区的大小是 680KB，再加上 RBL 文件头 96 字节，所以需要减少 1 个删区。

生成镜像后，可以使用 rt_ota_packaging_tool，把 `flash0.bin`制作成 `flash0.rbl`。注意分区名修改为 `filesystem`, 压缩算法，加密算法选择不加密，不压缩。

然后通过 OTA 下载到 download 分区即可。

重启成功后，就能看见打包的文件

```
msh />cd flash0
msh /flash0>ls
Directory /flash0:
1.txt               6
2.txt               6
3.txt               6
4.txt               6
5.txt               6
6.txt               6
test.txt            6
```

第 4 章

bk7251 挂载外部 flash 应用指南

RT-Thread 采用 SFUD 技术挂载外部 FLASH.

4.1 SFUD 介绍

SFUD 是一款开源的串行 SPI Flash 通用驱动库。由于现有市面的串行 Flash 种类居多，各个 Flash 的规格及命令存在差异，SFUD 就是为了解决这些 Flash 的差异现状而设计，让我们的产品能够支持不同品牌及规格的 Flash，提高了涉及到 Flash 功能的软件的可重用性及可扩展性，同时也可以规避 Flash 缺货或停产给产品所带来的风险。

4.2 如何挂载外部 FLASH

4.2.1 开启 SFUD 功能

`rtconfig.h` 添加下面的宏

```
#define RT_USING_SPI
#define RT_USING_SFUD
#define RT_SFUD_USING_SFDP
#define RT_SFUD_USING_FLASH_INFO_TABLE
#define BEKEN_USING_SPI
#define BEKEN_USING_SPI_FLASH
#define RT_DFS_ELM_MAX_SECTOR_SIZE 4096
#define RT_DFS_ELM_DRIVES 2
```

4.2.2 在代码中挂载外部 flash

- (a) spi_init 自动初始化，位置在 `drivers/drv_spi.c/rt_hw_spi_device_init`
- (b) sfud_probe 自动初始化，位置在 `drivers/drv_spi_flash.c/rt_hw_spi_flash_with_sfud_init`
- (c) 挂载 flash 手动挂载，在 main.c 的 main 函数添加如下代码

```
#if 1
    /* mount spi flash */
    if(dfs_mount("spi_flash", "/udisk", "elm", 0, 0) == 0)
        rt_kprintf("SPI FLASH system initialized\n");
    else
        rt_kprintf("SPI FLASH system initialzation failed\n");
#endif
```

编译，ota 升级。

重启设备，会看到如下日志

```
[SPI]:data_width = 8
[SPI]:max_hz = 50000000
[SPI]:config spi clk source DCO
[SPI]:input clk > 12MHz, set input clk = 12MHz
[SPI]:div = 8
[SPI]:spi_clk = 12000000
[SPI]:source_clk = 180000000
[SPI]:target frequency = 50000000, actual frequency = 10000000
[SPI]:mode = 0x00000004
[SPI]:[CTRL]:0x00c30800
[SFUD] Find a Winbond flash chip. Size is 4194304 bytes.
[SFUD] spi_flash flash device is initialize success.
SPI FLASH system initialized
```

4.2.3 格式化外部 flash

使用 `mkfs` 命令初始化外部 flash 为文件系统。

```
mkfs spi_flash
```

新建 test.txt 文件，并写入 hello world

```
msh /udisk>cd /udisk
msh /udisk>echo "hello world" test.txt
```

查看文件

```
msh /udisk>ls
Directory /udisk:
test.txt            11
```

此时，外部 flash 挂载成功

4.3 问题排查

4.3.1 添加目前不支持的 Flash

修改 rt-thread\components\drivers\spi\sfud\inc\sfud_flash_def.h ，所有已经支持的 Flash 见 SFUD_FLASH_CHIP_TABLE 宏定义，需要提前准备的 Flash 参数内容分别为：| 名称 | 制造商 ID | 类型 ID | 容量 ID | 容量 | 写模式 | 擦除粒度（擦除的最小单位）| 擦除粒度对应的命令 | 。这里以添加兆易创新（GigaDevice）的 GD25Q64B Flash 来举例。

此款 Flash 为兆易创新的早期生产的型号，所以不支持 SFDP 标准。首先需要下载其数据手册，找到 0x9F 命令返回的 3 种 ID，这里需要最后面两字节 ID，即 type id 及 capacity id 。GD25Q64B 对应这两个 ID 分别为 0x40 及 0x17 。上面要求的其他 Flash 参数都可以在数据手册中找到，这里要重点说明下 **写模式**这个参数，库本身提供的写模式共计有 4 种，详见文件顶部的 sfud_write_mode 枚举类型，同一款 Flash 可以同时支持多种写模式，视情况而定。对于 GD25Q64B 而言，其支持的写模式应该为 SFUD_WM_PAGE_256B ，即写 1-256 字节每页。结合上述 GD25Q64B 的 Flash 参数应如下：

```
    {"GD25Q64B", SFUD_MF_ID_GIGADEVICE, 0x40, 0x17, 8*1024*1024, SFUD_WM_PAGE_256B,
        4096, 0x20},
```

再将其增加到 SFUD_FLASH_CHIP_TABLE 宏定义末尾，即可完成该库对 GD25Q64B 的支持。

4.3.2 SFUD 测试命令

```
Usage:
sf probe [spi_device]           - probe and init SPI flash by given 'spi_device'
sf read addr size               - read 'size' bytes starting at 'addr'
sf write addr data1 ... dataN   - write some bytes 'data' to flash starting at 'addr
    '
sf erase addr size              - erase 'size' bytes starting at 'addr'
sf status [<volatile> <status>] - read or write '1:volatile|0:non-volatile' 'status'
sf bench                        - full chip benchmark. DANGER: It will erase full
    chip!
```

初始化 SFUD

```
msh /udisk>sf probe spi01
[SPI]:data_width = 8
[SPI]:max_hz = 50000000
[SPI]:config spi clk source DCO
[SPI]:input clk > 12MHz, set input clk = 12MHz
[SPI]:div = 8
[SPI]:spi_clk = 12000000
[SPI]:source_clk = 180000000
[SPI]:target frequency = 50000000, actual frequency = 10000000
[SPI]:mode = 0x00000004
[SPI]:[CTRL]:0x00c30800
[SFUD] Find a Winbond flash chip. Size is 4194304 bytes.
[SFUD] sf_cmd flash device is initialize success.
4 MB sf_cmd is current selected device.
```

擦除块设备

```
msh /udisk>sf erase 0 8
Erase the sf_cmd flash data success. Start from 0x00000000, size is 8.
```

读取块设备

```
msh /udisk>sf read 0 8
Read the sf_cmd flash data success. Start from 0x00000000, size is 8. The data is:
Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
[00000000] FF FF FF FF FF FF FF FF
```

写入块设备

```
msh /udisk>sf write 0 1 2 3 4 5 6 7 8
Write the sf_cmd flash data success. Start from 0x00000000, size is 8.
Write data: 1 2 3 4 5 6 7 8 .
```

读取块设备

```
msh /udisk>sf read 0 8
Read the sf_cmd flash data success. Start from 0x00000000, size is 8. The data is:
Offset (h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
[00000000] 01 02 03 04 05 06 07 08
```

4.3.3 flash 无法写入数据

问题解决: 查看 `sfud_flash_def.h` 中的 SFUD_FLASH_CHIP_TABLE 的 flash 参数是否填写错误

4.3.4 flash 无法挂载

问题解决: 关掉 `sd flash0` 的挂载点，重启设备查看能否挂载成功

第 5 章

exfat 应用手册

5.1 exfat 介绍

exFAT (Extended File Allocation Table File System，扩展 FAT，也称作 FAT64，即扩展文件分配表) 是 Microsoft 在 Windows Embeded 5.0 以上 (包括 Windows CE 5.0、6.0、Windows Mobile5、6、6.1) 中引入的一种适合于闪存的文件系统，为了解决 FAT32 等不支持 4G 及其更大的文件而推出的一种格式。

5.2 exfat 使用说明

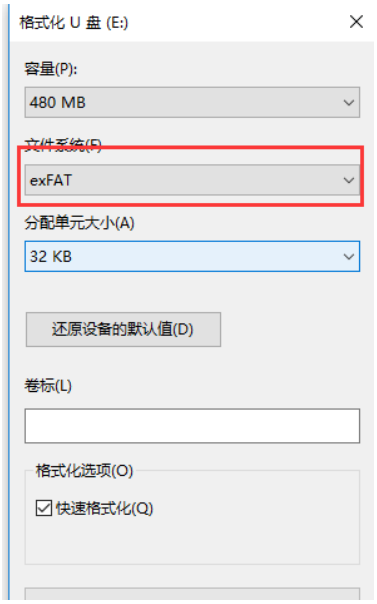
5.2.1 开启 exfat 功能

rtconfig.h 中添加 exfat 的宏定义

```
#define RT_DFS_ELM_USE_EXFAT
```

5.2.2 格式化 sd card

如果 sd card 不是 exfat 格式，需要将 sd card 格式化为 exfat 格式



5.2.3 挂载 sd card

在 main 函数添加挂载 sd card 的代码

```
#if 1
    /* mount sd card fat partition 1 as root directory */
    if(dfs_mount("sd0", "/sd", "elm", 0, 0) == 0)
        rt_kprintf("SD File System initialized!\n");
    else
        rt_kprintf("SD File System initialzation failed!\n");
#endif
```

编译，OTA 升级。

5.3 运行结果

OTA 升级成功后，重启设备.

MSH 中执行如下命令，将看见 sd card 中的文件

```
msh />cd sd

msh /sd>ls
Directory /sd:
System Volume Information<DIR>
儿歌                <DIR>
故事                <DIR>
国学                <DIR>
英语                <DIR>
应用                <DIR>
```

注: 支持的 sd card 最大容量为 32G, 本次测试使用的 sd card 为 32G.

第 6 章

文件系统中文支持 + 长文件名应用手册

文档主要针对文件系统存在中文文件/文件夹时，提供对中文文件/文件夹的支持。

当 TF 卡中有中文名时，如果不开启中文长文件名，将无法访问这些中文名的文件。要支持中文长文件名，FLASH 需要增加 180KB。

为了节省 FLASH 空间，可以把这个 180KB 的转换表放到 TF 卡中，并以 8.3 的格式存放。这样访问这个转换表时并不需要转换。

6.1 使用说明

6.1.1 开启中文支持和长文件名功能

在 rtconfig.h 中添加如下宏：

```
#define RT_DFS_ELM_USE_LFN_3
#define RT_DFS_ELM_USE_LFN 3
#define RT_DFS_ELM_MAX_LFN 255
#define RT_DFS_ELM_CODE_PAGE_FILE
```

注意：如果存在 define RT_DFS_ELM_CODE_PAGE 936 或者 define RT_DFS_ELM_CODE_PAGE 437 这两个宏定义，暂时注释掉

定义了 RT_DFS_ELM_CODE_PAGE_FILE 之后，会加载 rt-thread\components\dfs\filesystems\elmfat\option\ccfile.c 这个文件，在 ccfile.c 中，定义了 TF 卡存放转换表的路径，也可以自己修改。拷贝 gbk2uni.tbl 和 uni2gbk.tbl 到 TF 卡对应的目录。然后注释掉以下代码 INIT_APP_EXPORT(ff_convert_init); (ccfile.c 27 行).

转换表存放路径

```
#ifndef GBK2UNI_FILE
#define GBK2UNI_FILE "/sd/resource/gbk2uni.tbl"
#endif

#ifndef UNI2GBK_FILE
#define UNI2GBK_FILE "/sd/resource/uni2gbk.tbl"
```

```
#endif
```

转换表下载地址 https://github.com/aozima/stm32_radio/tree/master/trunk/stm32radio/resource

6.1.2 初始化转换表

在 sd 卡挂载成功后，通过 ff_convert_init 初始化转换表

```
#if 1
    /* mount sd card fat partition 1 as root directory */
    if(dfs_mount("sd0", "/sd", "elm", 0, 0) == 0)
    {
         rt_kprintf("SD File System initialized!\n");
         ff_convert_init();
    }
    else
        rt_kprintf("SD File System initialzation failed!\n");
#endif
```

ff_convert_init() 函数必须放到 sd 卡初始化之后，否则 ff_convert_init 函数失败。

修改成功后，编译，下载到开发板。

6.2 运行结果

```
msh /sd/test>ls
Directory /sd/test:
故事                32
```

第 7 章

UART 设备

7.1 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。

UART 串口的特点是将数据一位一位地顺序传送，只要 2 根传输线就可以实现双向通信，一根线发送数据的同时用另一根线接收数据。UART 串口通信有几个重要的参数，分别是波特率、起始位、数据位、停止位和奇偶检验位，对于两个使用 UART 串口通信的端口，这些参数必须匹配，否则通信将无法正常完成。UART 串口传输的数据格式如下图所示：

图 7.1: *串口传输数据格式*

- 起始位：表示数据传输的开始，电平逻辑为 “0” 。
- 数据位：可能值有 5、6、7、8、9，表示传输这几个 bit 位数据。一般取值为 8，因为一个 ASCII 字符值为 8 位。
- 奇偶校验位：用于接收方对接收到的数据进行校验，校验 “1” 的位数为偶数 (偶校验) 或奇数 (奇校验)，以此来校验数据传送的正确性，使用时不需要此位也可以。
- 停止位：表示一帧数据的结束。电平逻辑为 “1”。
- 波特率：串口通信时的速率，它用单位时间内传输的二进制代码的有效位 (bit) 数来表示，其单位为每秒比特数 bit/s(bps)。常见的波特率值有 4800、9600、14400、38400、115200 等，数值越大数据传输的越快，波特率为 115200 表示每秒钟传输 115200 位数据。

7.2 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()	根据串口设备名称查找设备获取设备句柄
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

7.2.1 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

一般情况下，注册到系统的串口设备名称为 uart0，uart1 等，使用示例如下所示：

```
#define SAMPLE_UART_NAME       "uart2"  /* 串口设备名称 */
static rt_device_t serial;              /* 串口设备句柄 */
/* 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);
```

7.2.2 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 参数支持下列参数 (可以采用或的方式支持多种参数)：

```
#define RT_DEVICE_FLAG_STREAM       0x040     /* 流模式       */
/* 接收模式参数 */
#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 */
/* 发送模式参数 */
#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，在通信频繁或有大段信息要传输时非常有用。

若串口要使用 DMA 接收模式，oflags 取值 RT_DEVICE_FLAG_DMA_RX。

!!! note “注意事项” * RT_DEVICE_FLAG_STREAM：流模式用于向串口终端输出字符串：当输出的字符是 `"\n"`（对应 16 进制值为 0x0A）时，自动在前面输出一个 `"\r"`（对应 16 进制值为 0x0D）做分行。

流模式 RT_DEVICE_FLAG_STREAM 可以和接收发送模式参数使用或 “|” 运算符一起使用。

以中断接收及轮询发送模式打开串口设备使用示例如下所示：

```
#define SAMPLE_UART_NAME       "uart2"  /* 串口设备名称 */
static rt_device_t serial;              /* 串口设备句柄 */
/* 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);

/* 以中断接收及轮询发送模式打开串口设备 */
rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);
```

7.2.3 控制串口设备

通过命令控制字，应用程序可以对串口设备进行配置，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，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;          /* 波特率 */
    rt_uint32_t data_bits    :4;    /* 数据位 */
    rt_uint32_t stop_bits    :2;    /* 停止位 */
    rt_uint32_t parity       :2;    /* 奇偶校验位 */
    rt_uint32_t bit_order    :1;    /* 高位在前或者低位在前 */
    rt_uint32_t invert       :1;    /* 模式 */
    rt_uint32_t bufsz        :16;   /* 接收数据缓冲区大小 */
    rt_uint32_t reserved     :4;    /* 保留位 */
};
```

- RT-Thread 提供的默认宏配置如下：

```
#define RT_SERIAL_CONFIG_DEFAULT           \
{                                          \
    BAUD_RATE_115200, /* 115200 bits/s */  \
    DATA_BITS_8,      /* 8 databits */     \
    STOP_BITS_1,      /* 1 stopbit */      \
    PARITY_NONE,      /* No parity  */     \
    BIT_ORDER_LSB,    /* LSB first sent */ \
    NRZ_NORMAL,       /* Normal mode */    \
    RT_SERIAL_RB_BUFSZ, /* Buffer size */  \
    0                                      \
}
```

RT-Thread 提供的配置参数可取值为如下宏定义：

```
/* 波特率可取值 */
#define BAUD_RATE_2400                  2400
#define BAUD_RATE_4800                  4800
#define BAUD_RATE_9600                  9600
#define BAUD_RATE_19200                 19200
#define BAUD_RATE_38400                 38400
#define BAUD_RATE_57600                 57600
#define BAUD_RATE_115200                115200
#define BAUD_RATE_230400                230400
#define BAUD_RATE_460800                460800
#define BAUD_RATE_921600                921600
#define BAUD_RATE_2000000               2000000
#define BAUD_RATE_3000000               3000000
/* 数据位可取值 */
#define DATA_BITS_5                     5
#define DATA_BITS_6                     6
#define DATA_BITS_7                     7
#define DATA_BITS_8                     8
#define DATA_BITS_9                     9
/* 停止位可取值 */
#define STOP_BITS_1                     0
#define STOP_BITS_2                     1
#define STOP_BITS_3                     2
#define STOP_BITS_4                     3
/* 极性位可取值 */
#define PARITY_NONE                     0
#define PARITY_ODD                      1
#define PARITY_EVEN                     2
/* 高低位顺序可取值 */
#define BIT_ORDER_LSB                   0
#define BIT_ORDER_MSB                   1
/* 模式可取值 */
#define NRZ_NORMAL                      0   /* normal mode */
#define NRZ_INVERTED                    1   /* inverted mode */
/* 接收数据缓冲区默认大小 */
#define RT_SERIAL_RB_BUFSZ              64
```

接收缓冲区

当串口使用中断接收模式打开时，串口驱动框架会根据 RT_SERIAL_RB_BUFSZ 大小开辟一块缓冲区用于保存接收到的数据，底层驱动接收到一个数据, 都会在中断服务程序里面将数据放入缓冲区。

!!! note “注意事项” 接收数据缓冲区大小默认 64 字节。若一次性数据接收字节数很多，没有及时读取数据，那么缓冲区的数据将会被新接收到的数据覆盖，造成数据丢失，建议调大缓冲区。

配置串口硬件参数如数据位、校验位、停止位等的示例程序如下：

```
#define SAMPLE_UART_NAME       "uart2"  /* 串口设备名称 */
static rt_device_t serial;              /* 串口设备句柄 */
```

```
struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT; /* 配置参数 */
/* 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);

/* 以中断接收及轮询发送模式打开串口设备 */
rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);

config.baud_rate = BAUD_RATE_115200;
config.data_bits = DATA_BITS_8;
config.stop_bits = STOP_BITS_2;
config.parity = PARITY_NONE;
/* 打开设备后才可修改串口配置参数 */
rt_device_control(serial, RT_DEVICE_CTRL_CONFIG, &config);
```

7.2.4 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
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 errno 来判断错误状态

调用这个函数，会把缓冲区 buffer 中的数据写入到设备 dev 中，写入数据的大小是 size。

向串口写入数据示例程序如下所示：

```
#define SAMPLE_UART_NAME       "uart2"  /* 串口设备名称 */
static rt_device_t serial;              /* 串口设备句柄 */
char str[] = "hello RT-Thread!\r\n";
struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT; /* 配置参数 */
/* 查找串口设备 */
serial = rt_device_find(SAMPLE_UART_NAME);

/* 以中断接收及轮询发送模式打开串口设备 */
rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);
```

```
/* 发送字符串 */
rt_device_write(serial, 0, str, (sizeof(str) - 1));
```

7.2.5 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。

7.2.6 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。

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。使用示例如下所示：

```
#define SAMPLE_UART_NAME       "uart2"  /* 串口设备名称 */
static rt_device_t serial;              /* 串口设备句柄 */
static struct rt_semaphore rx_sem;     /* 用于接收消息的信号量 */

/* 接收数据回调函数 */
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    /* 串口接收到数据后产生中断，调用此回调函数，然后发送接收信号量 */
    rt_sem_release(&rx_sem);

    return RT_EOK;
}

static int uart_sample(int argc, char *argv[])
{
    serial = rt_device_find(SAMPLE_UART_NAME);

    /* 以中断接收及轮询发送模式打开串口设备 */
    rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);
    /* 初始化信号量 */
    rt_sem_init(&rx_sem, "rx_sem", 0, RT_IPC_FLAG_FIFO);

    /* 设置接收回调函数 */
    rt_device_set_rx_indicate(serial, uart_input);
}
```

7.2.7 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
    );
```

参数	描述
dev	设备句柄
pos	读取数据偏移量，此参数串口未使用
buffer	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

参数	描述
0	需要读取当前线程的 errno 来判断错误状态

读取数据偏移量 pos 针对字符设备无效，此参数主要用于块设备中。

串口使用中断接收模式并配合接收回调函数的使用示例如下所示：

```
static rt_device_t serial;                /* 串口设备句柄 */
static struct rt_semaphore rx_sem;     /* 用于接收消息的信号量 */

/* 接收数据的线程 */
static void serial_thread_entry(void *parameter)
{
    char ch;

    while (1)
    {
        /* 从串口读取一个字节的数据，没有读取到则等待接收信号量 */
        while (rt_device_read(serial, -1, &ch, 1) != 1)
        {
            /* 阻塞等待接收信号量，等到信号量后再次读取数据 */
            rt_sem_take(&rx_sem, RT_WAITING_FOREVER);
        }
        /* 读取到的数据通过串口错位输出 */
        ch = ch + 1;
        rt_device_write(serial, 0, &ch, 1);
    }
}
```

7.2.8 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

关闭设备接口和打开设备接口需配对使用，打开一次设备对应要关闭一次设备，这样设备才会被完全关闭，否则设备仍处于未关闭状态。

7.3 串口设备使用示例

串口设备的具体使用方式可以参考如下示例代码，示例代码的主要步骤如下：

1. 首先查找串口设置获取设备句柄。
2. 初始化回调函数发送使用的信号量，然后以读写及中断接收方式打开串口设备。
3. 设置串口设备的接收回调函数，之后发送字符串，并创建读取数据线程。

- 读取数据线程会尝试读取一个字符数据，如果没有数据则会挂起并等待信号量，当串口设备接收到数据时会触发中断并调用接收回调函数，此函数会发送信号量唤醒线程，此时线程会马上读取接收到的数据。

```
/*
 * 程序清单：这是一个 串口 设备使用例程
 * 例程导出了 uart_sample 命令到控制终端
 * 命令调用格式：uart_sample uart2
 * 命令解释：命令第二个参数是要使用的串口设备名称，为空则使用默认的串口设备
 * 程序功能：通过串口输出字符串"hello RT-Thread!"，然后错位输出输入的字符
*/

#include <rtthread.h>

#define SAMPLE_UART_NAME       "uart2"

/* 用于接收消息的信号量 */
static struct rt_semaphore rx_sem;
static rt_device_t serial;

/* 接收数据回调函数 */
static rt_err_t uart_input(rt_device_t dev, rt_size_t size)
{
    /* 串口接收到数据后产生中断，调用此回调函数，然后发送接收信号量 */
    rt_sem_release(&rx_sem);

    return RT_EOK;
}

static void serial_thread_entry(void *parameter)
{
    char ch;

    while (1)
    {
```

```
        /* 从串口读取一个字节的数据，没有读取到则等待接收信号量 */
        while (rt_device_read(serial, -1, &ch, 1) != 1)
        {
            /* 阻塞等待接收信号量，等到信号量后再次读取数据 */
            rt_sem_take(&rx_sem, RT_WAITING_FOREVER);
        }
        /* 读取到的数据通过串口错位输出 */
        ch = ch + 1;
        rt_device_write(serial, 0, &ch, 1);
    }
}

static int uart_sample(int argc, char *argv[])
{
    rt_err_t ret = RT_EOK;
    char uart_name[RT_NAME_MAX];
    char str[] = "hello RT-Thread!\r\n";

    if (argc == 2)
    {
        rt_strncpy(uart_name, argv[1], RT_NAME_MAX);
    }
    else
    {
        rt_strncpy(uart_name, SAMPLE_UART_NAME, RT_NAME_MAX);
    }

    /* 查找系统中的串口设备 */
    serial = rt_device_find(uart_name);
    if (!serial)
    {
        rt_kprintf("find %s failed!\n", uart_name);
        return RT_ERROR;
    }

    /* 初始化信号量 */
    rt_sem_init(&rx_sem, "rx_sem", 0, RT_IPC_FLAG_FIFO);
    /* 以中断接收及轮询发送模式打开串口设备 */
    rt_device_open(serial, RT_DEVICE_FLAG_INT_RX);
    /* 设置接收回调函数 */
    rt_device_set_rx_indicate(serial, uart_input);
    /* 发送字符串 */
    rt_device_write(serial, 0, str, (sizeof(str) - 1));

    /* 创建 serial 线程 */
    rt_thread_t thread = rt_thread_create("serial", serial_thread_entry, RT_NULL,
        1024, 25, 10);
    /* 创建成功则启动线程 */
    if (thread != RT_NULL)
```

```
    {
        rt_thread_startup(thread);
    }
    else
    {
        ret = RT_ERROR;
    }

    return ret;
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(uart_sample, uart device sample);
```

第 8 章

PIN 设备

8.1 引脚简介

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，一般有下图所示 5 种中断触发模式：

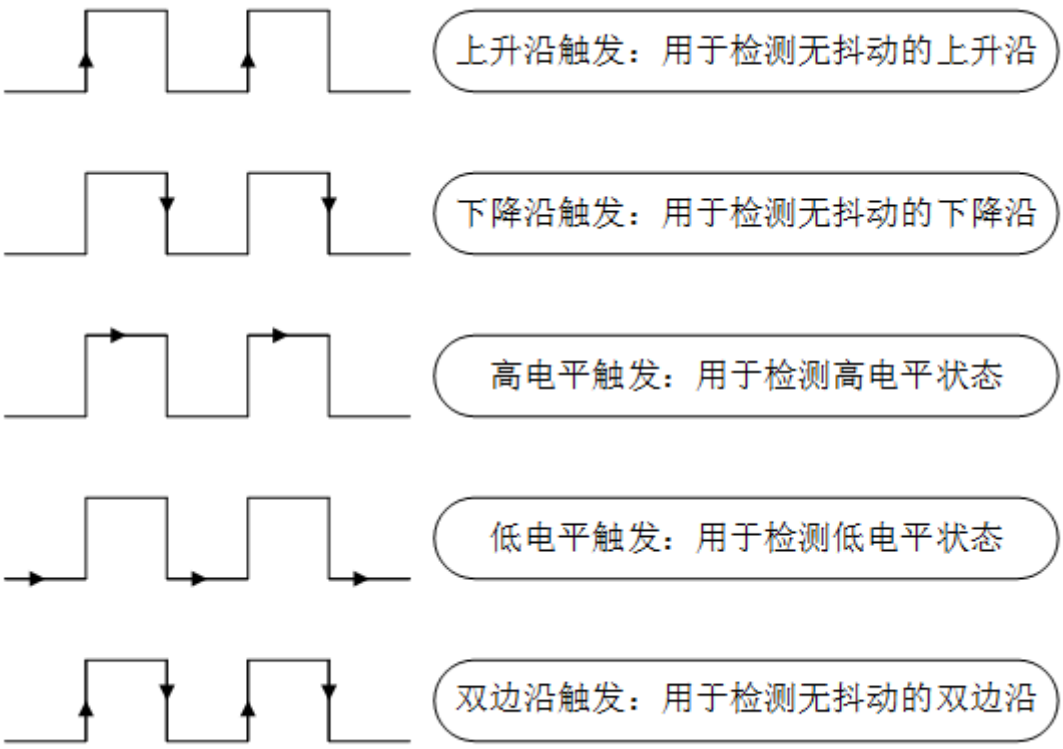


图 8.1: *5 种中断触发模式*

- 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。

8.2 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

8.2.1 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

目前 RT-Thread 支持的引脚工作模式可取如所示的 5 种宏定义值之一，每种模式对应的芯片实际支持的模式需参考 PIN 设备驱动程序的具体实现：

```
#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 */
```

使用示例如下所示：

```
#define LED_PIN_NUM            24
```

```
/* LED引脚为输出模式 */
rt_pin_mode(LED_PIN_NUM, PIN_MODE_OUTPUT);
```

8.2.2 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

使用示例如下所示：

```
#define LED_PIN_NUM             24

/* LED引脚为输出模式 */
rt_pin_mode(LED_PIN_NUM, PIN_MODE_OUTPUT);
/* 设置低电平 */
rt_pin_write(LED_PIN_NUM, PIN_LOW);
```

8.2.3 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

使用示例如下所示：

```
#define LED_PIN_NUM             24
int status;
```

```
/* LED引脚为输出模式 */
rt_pin_mode(LED_PIN_NUM, PIN_MODE_OUTPUT);
/* 设置低电平 */
rt_pin_write(LED_PIN_NUM, PIN_LOW);

status = rt_pin_read(LED_PIN_NUM);
```

8.2.4 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

中断触发模式 mode 可取如下 5 种宏定义值之一：

```
#define PIN_IRQ_MODE_RISING 0x00          /* 上升沿触发 */
#define PIN_IRQ_MODE_FALLING 0x01         /* 下降沿触发 */
#define PIN_IRQ_MODE_RISING_FALLING 0x02 /* 边沿触发（上升沿和下降沿都触发）*/
#define PIN_IRQ_MODE_HIGH_LEVEL 0x03      /* 高电平触发 */
#define PIN_IRQ_MODE_LOW_LEVEL 0x04       /* 低电平触发 */
```

使用示例如下所示：

```
#define KEY0_PIN_NUM            55  /* PD8 */
/* 中断回调函数 */
void led_on(void *args)
{
    rt_kprintf("turn on led!\n");

    rt_pin_write(LED_PIN_NUM, PIN_HIGH);
}
static void led_beep_sample(void)
{
```

```
    /* 按键0引脚为输入模式 */
    rt_pin_mode(KEY0_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，上升沿模式，回调函数名为led_on */
    rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, led_on, RT_NULL);
}
```

8.2.5 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

使用示例如下所示：

```
#define KEY0_PIN_NUM            55  /* PD8 */
/* 中断回调函数 */
void led_on(void *args)
{
    rt_kprintf("turn on led!\n");

    rt_pin_write(LED_PIN_NUM, PIN_HIGH);
}
static void pin_led_sample(void)
{
    /* 按键0引脚为输入模式 */
    rt_pin_mode(KEY0_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，上升沿模式，回调函数名为led_on */
    rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, led_on, RT_NULL);
    /* 使能中断 */
    rt_pin_irq_enable(KEY0_PIN_NUM, PIN_IRQ_ENABLE);
}
```

8.2.6 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

引脚脱离了中断回调函数以后，中断并没有关闭，还可以调用绑定中断回调函数再次绑定其他回调函数。

```
#define KEY0_PIN_NUM            55
/* 中断回调函数 */
void beep_on(void *args)
{
    rt_kprintf("turn on led!\n");

    rt_pin_write(LED_PIN_NUM, PIN_HIGH);
}
static void pin_led_sample(void)
{
    /* 按键0引脚为输入模式 */
    rt_pin_mode(KEY0_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，上升沿模式，回调函数名为led_on */
    rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, led_on, RT_NULL);
    /* 使能中断 */
    rt_pin_irq_enable(KEY0_PIN_NUM, PIN_IRQ_ENABLE);
    /* 脱离中断回调函数 */
    rt_pin_detach_irq(KEY0_PIN_NUM);
}
```

8.3 PIN 设备使用示例

PIN 设备的具体使用方式可以参考如下示例代码，示例代码的主要步骤如下：

1. 设置 led 对应引脚为输出模式，并给一个默认的低电平状态。
2. 设置按键 0 和按键 1 对应引脚为输入模式，然后绑定中断回调函数并使能中断。
3. 按下按键 0 led 开始亮，按下按键 1 led 关闭。

```
/*
 * 程序清单：这是一个 PIN 设备使用例程
```

```
 * 例程导出了 pin_led_sample 命令到控制终端
 * 命令调用格式：pin_led_sample
 * 程序功能：通过按键控制led对应引脚的电平状态控制led
*/

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


#define LED_PIN_NUM            24

#define KEY0_PIN_NUM           55

#define KEY1_PIN_NUM           56

void led_on(void *args)
{
    rt_kprintf("turn on led!\n");

    rt_pin_write(LED_PIN_NUM, PIN_HIGH);
}

void led_off(void *args)
{
    rt_kprintf("turn off led!\n");

    rt_pin_write(LED_PIN_NUM, PIN_LOW);
}

static void pin_led_sample(void)
{
    /* led引脚为输出模式 */
    rt_pin_mode(LED_PIN_NUM, PIN_MODE_OUTPUT);
    /* 默认低电平 */
    rt_pin_write(LED_PIN_NUM, PIN_LOW);

    /* 按键0引脚为输入模式 */
    rt_pin_mode(KEY0_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，下降沿模式，回调函数名为beep_on */
    rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, led_on, RT_NULL);
    /* 使能中断 */
    rt_pin_irq_enable(KEY0_PIN_NUM, PIN_IRQ_ENABLE);

    /* 按键1引脚为输入模式 */
    rt_pin_mode(KEY1_PIN_NUM, PIN_MODE_INPUT_PULLUP);
    /* 绑定中断，下降沿模式，回调函数名为led_off */
    rt_pin_attach_irq(KEY1_PIN_NUM, PIN_IRQ_MODE_FALLING, led_off, RT_NULL);
    /* 使能中断 */
    rt_pin_irq_enable(KEY1_PIN_NUM, PIN_IRQ_ENABLE);
```

```
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(pin_led_sample, pin led sample);
```

文档中的 KEY0_PIN_NUM 根据实际的硬件有所改变

第 9 章

I2C 总线设备

9.1 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。如下图所示：

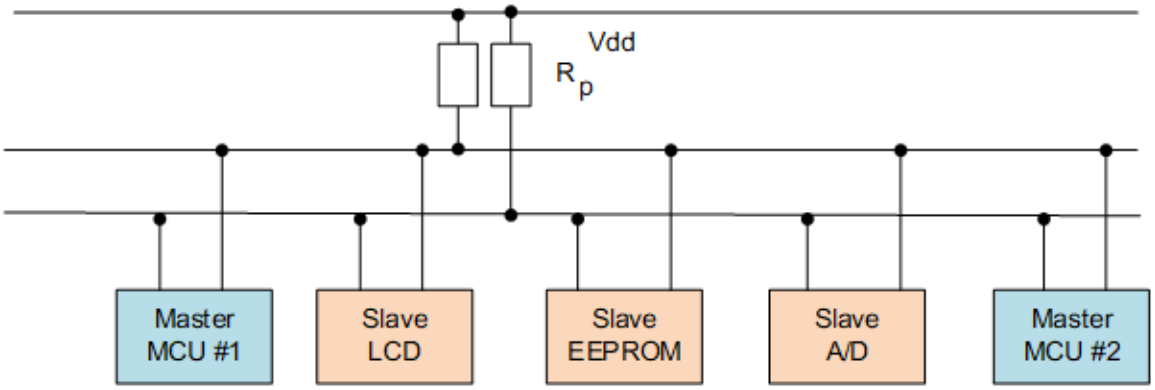


图 9.1: *I2C 总线主从设备连接方式*

如下图所示为 I2C 总线主要的数据传输格式：

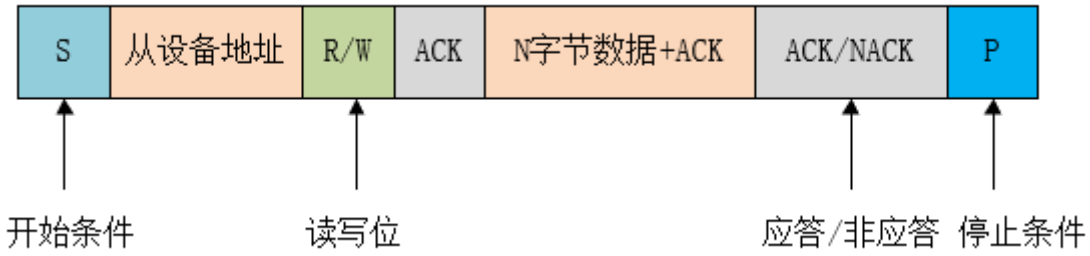


图 9.2: *I2C 总线数据传输格式*

当总线空闲时，SDA 和 SCL 都处于高电平状态，当主机要和某个从机通讯时，会先发送一个开始条件，然后发送从机地址和读写控制位，接下来传输数据（主机发送或者接收数据），数据传输结束时主机会发送停止条件。传输的每个字节为 8 位，高位在前，低位在后。数据传输过程中的不同名词详解如下所示：

- **开始条件：** SCL 为高电平时，主机将 SDA 拉低，表示数据传输即将开始。
- **从机地址：** 主机发送的第一个字节为从机地址，高 7 位为地址，最低位为 R/W 读写控制位，1 表示读操作，0 表示写操作。一般从机地址有 7 位地址模式和 10 位地址模式两种，如果是 10 位地址模式，第一个字节的头 7 位是 11110XX 的组合，其中最后两位（XX）是 10 位地址的两个最高位，第二个字节为 10 位从机地址的剩下 8 位，如下图所示：

图 9.3: *7 位地址和 10 位地址格式*

- **应答信号：** 每传输完成一个字节的数据，接收方就需要回复一个 ACK（acknowledge）。写数据时由从机发送 ACK，读数据时由主机发送 ACK。当主机读到最后一个字节数据时，可发送 NACK（Not acknowledge）然后跟停止条件。
- **数据：** 从机地址发送完后可能会发送一些指令，依从机而定，然后开始传输数据，由主机或者从机发送，每个数据为 8 位，数据的字节数没有限制。
- **重复开始条件：** 在一次通信过程中，主机可能需要和不同的从机传输数据或者需要切换读写操作时，主机可以再发送一个开始条件。
- **停止条件：** 在 SDA 为低电平时，主机将 SCL 拉高并保持高电平，然后在将 SDA 拉高，表示传输结束。

9.2 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

9.2.1 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

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：

```
#define I2C_BUS_NAME      "i2c"  /* I2C总线设备名称 */
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(I2C_BUS_NAME);
```

9.2.2 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

参数	描述
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。

!!! note “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。

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。

!!! note “注意事项” 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。

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。

```
#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 */
```

使用示例如下所示：

```
#define I2C_BUS_NAME      "i2c"  /* I2C总线设备名称 */
#define I2C_ADDR               0x38   /* 从机地址 */
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);

/* 读寄存器数据 */
static rt_err_t read_regs(struct rt_i2c_bus_device *bus, rt_uint8_t len, rt_uint8_t
    *buf)
{
    struct rt_i2c_msg msgs;
```

```
    msgs.addr = AHT10_ADDR;     /* 从机地址 */
    msgs.flags = RT_I2C_RD;     /* 读标志 */
    msgs.buf = buf;             /* 读写数据缓冲区指针 */
    msgs.len = len;             /* 读写数据字节数 */

    /* 调用I2C设备接口传输数据 */
    if (rt_i2c_transfer(bus, &msgs, 1) == 1)
    {
        return RT_EOK;
    }
    else
    {
        return -RT_ERROR;
    }
}
```

9.3 I2C 总线设备使用示例

I2C 设备的具体使用方式可以参考如下示例代码

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

static int i2c_test(int argc, char *argv)
{
    const char *i2c_bus_device_name = "i2c";
    struct rt_i2c_bus_device *i2c_device;
    struct rt_i2c_msg msgs[2];
    rt_uint8_t buffer1[2];
    rt_uint8_t buffer2[3];
    rt_size_t i, ret;

    i2c_device = rt_i2c_bus_device_find(i2c_bus_device_name);
    if (i2c_device == RT_NULL)
    {
        rt_kprintf("i2c bus device %s not found!\n", i2c_bus_device_name);
        return -RT_ENOSYS;
    }
    else
    {
        rt_kprintf("find i2c success\n");
    }

    //step 1: read out.
    buffer1[0] = 5;
    msgs[0].addr = 0x10;
    msgs[0].flags = RT_I2C_WR; /* Write to slave */
```

```
    msgs[0].buf = buffer1;      /* eeprom offset. */
    msgs[0].len = 1;

    msgs[1].addr = 0x10;
    msgs[1].flags = RT_I2C_RD; /* Read from slave */
    msgs[1].buf = buffer2;
    msgs[1].len = sizeof(buffer2);

    ret = rt_i2c_transfer(i2c_device, msgs, 1);
    rt_kprintf("rt_i2c_transfer ret:%d\n", ret);

    buffer1[0] = 3;
    msgs[1].len = 1;
    memset(msgs[1].buf, 0x99, msgs[1].len);

    for(i=0; i<msgs[1].len; i++)
    {
        rt_kprintf("%02X ", buffer2[i]);
    }
    rt_kprintf("\n");

    ret = rt_i2c_transfer(i2c_device, msgs, 2);
    rt_kprintf("rt_i2c_transfer ret:%d\n", ret);

    for(i=0; i<msgs[1].len; i++)
    {
        rt_kprintf("%02X ", buffer2[i]);
    }
    rt_kprintf("\n");

    return 0;
}
MSH_CMD_EXPORT(i2c_test, i2c_test);
```

第 10 章

PWM 设备

10.1 PWM 简介

PWM(Pulse Width Modulation , 脉冲宽度调制) 是一种对模拟信号电平进行数字编码的方法，通过不同频率的脉冲使用方波的占空比用来对一个具体模拟信号的电平进行编码，使输出端得到一系列幅值相等的脉冲，用这些脉冲来代替所需要波形的设备。

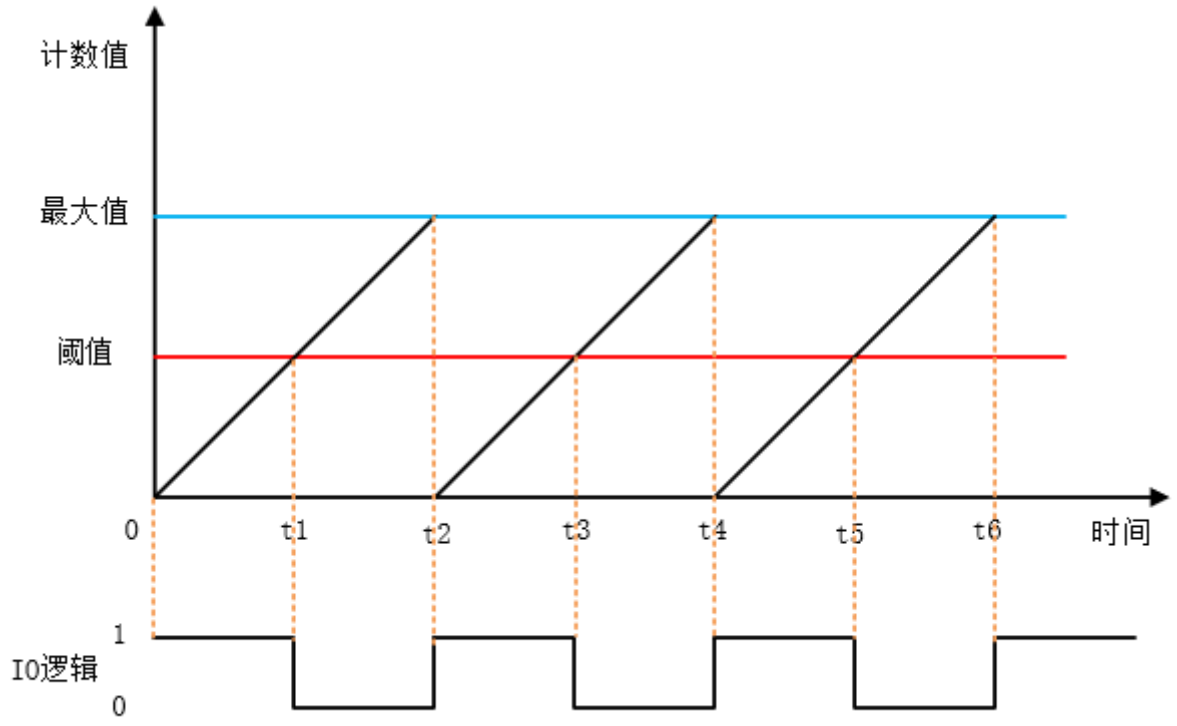


图 10.1: *PWM 原理图*

上图是一个简单的 PWM 原理示意图，假定定时器工作模式为向上计数，当计数值小于阈值时，则输出一种电平状态，比如高电平，当计数值大于阈值时则输出相反的电平状态，比如低电平。当计数值达到最大值是，计数器从 0 开始重新计数，又回到最初的电平状态。高电平持续时间（脉冲宽度）和周期时间的比值就是占空比，范围为 0~100%。上图高电平的持续时间刚好是周期时间的一半，所以占空比为 50%。

一个比较常用的 pwm 控制情景就是用来调节灯或者屏幕的亮度，根据占空比的不同，就可以完成亮度的调节。PWM 调节亮度并不是持续发光的，而是在不停地点亮、熄灭屏幕。当亮、灭交替够快时，肉眼就会认为一直在亮。在亮、灭的过程中，灭的状态持续时间越长，屏幕给肉眼的观感就是亮度越低。亮的时间越长，灭的时间就相应减少，屏幕就会变亮。

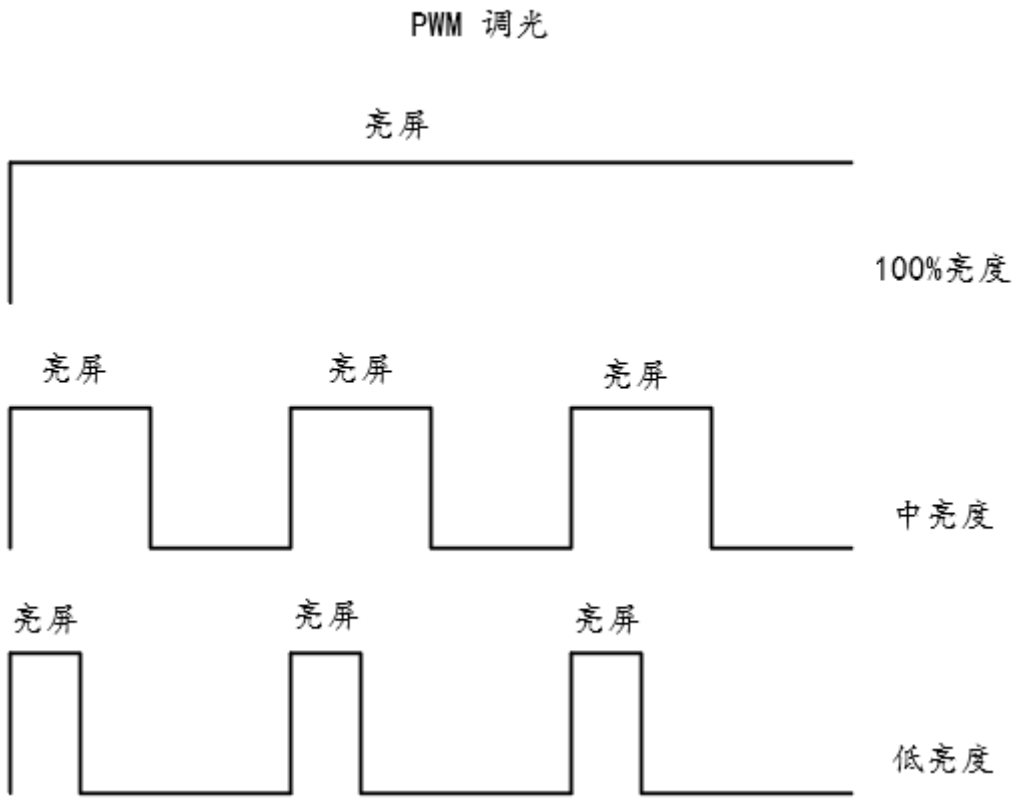


图 10.2: *PWM 调节亮度*

10.2 访问 PWM 设备

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

函数	描述
rt_device_find()	根据 PWM 设备名称查找设备获取设备句柄
rt_pwm_set()	设置 PWM 周期和脉冲宽度
rt_pwm_enable()	使能 PWM 设备
rt_pwm_disable()	关闭 PWM 设备

10.2.1 查找 PWM 设备

应用程序根据 PWM 设备名称获取设备句柄，进而可以操作 PWM 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	没有找到设备

一般情况下，注册到系统的 PWM 设备名称为 pwm ，使用示例如下所示：

```
#define PWM_DEV_NAME        "pwm"    /* PWM 设备名称 */
struct rt_device_pwm *pwm_dev;       /* PWM 设备句柄 */
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
```

10.2.2 设置 PWM 周期和脉冲宽度

通过如下函数设置 PWM 周期和占空比：

```
rt_err_t rt_pwm_set(struct rt_device_pwm *device,
                    int channel,
                    rt_uint32_t period,
                    rt_uint32_t pulse);
```

参数	描述
device	PWM 设备句柄
channel	PWM 通道
period	PWM 周期时间 (单位纳秒 ns)
pulse	PWM 脉冲宽度时间 (单位纳秒 ns)
返回	——
RT_EOK	成功
-RT_EIO	device 为空
-RT_ENOSYS	设备操作方法为空
其他错误码	执行失败

PWM 的输出频率由周期时间 period 决定，例如周期时间为 0.5ms (毫秒)，则 period 值为 500000ns（纳秒），输出频率为 2KHz，占空比为 pulse / period，pulse 值不能超过 period。

使用示例如下所示：

```
#define PWM_DEV_NAME        "pwm"  /* PWM设备名称 */
#define PWM_DEV_CHANNEL     4       /* PWM通道 */
struct rt_device_pwm *pwm_dev;      /* PWM设备句柄 */
rt_uint32_t period, pulse;

period = 500000;    /* 周期为0.5ms，单位为纳秒ns */
pulse = 0;          /* PWM脉冲宽度值，单位为纳秒ns */
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
/* 设置PWM周期和脉冲宽度 */
rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
```

10.2.3 使能 PWM 设备

设置好 PWM 周期和脉冲宽度后就可以通过如下函数使能 PWM 设备:

```
rt_err_t rt_pwm_enable(struct rt_device_pwm *device, int channel);
```

参数	描述
device	PWM 设备句柄
channel	PWM 通道
返回	——
RT_EOK	设备使能成功
-RT_ENOSYS	设备操作方法为空
其他错误码	设备使能失败

使用示例如下所示：

```
#define PWM_DEV_NAME        "pwm"  /* PWM设备名称 */
#define PWM_DEV_CHANNEL     4       /* PWM通道 */
struct rt_device_pwm *pwm_dev;      /* PWM设备句柄 */
rt_uint32_t period, pulse;

period = 500000;    /* 周期为0.5ms，单位为纳秒ns */
pulse = 0;          /* PWM脉冲宽度值，单位为纳秒ns */
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
/* 设置PWM周期和脉冲宽度 */
rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
/* 使能设备 */
rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL);
```

10.2.4 关闭 PWM 设备通道

通过如下函数关闭 PWM 设备对应通道。

```
rt_err_t rt_pwm_disable(struct rt_device_pwm *device, int channel);
```

参数	描述
device	PWM 设备句柄
channel	PWM 通道
返回	——
RT_EOK	设备关闭成功
-RT_EIO	设备句柄为空
其他错误码	设备关闭失败

使用示例如下所示：

```
#define PWM_DEV_NAME        "pwm"  /* PWM设备名称 */
#define PWM_DEV_CHANNEL     4       /* PWM通道 */
struct rt_device_pwm *pwm_dev;      /* PWM设备句柄 */
rt_uint32_t period, pulse;

period = 500000;    /* 周期为0.5ms，单位为纳秒ns */
pulse = 0;          /* PWM脉冲宽度值，单位为纳秒ns */
/* 查找设备 */
pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
/* 设置PWM周期和脉冲宽度 */
rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
/* 使能设备 */
rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL);
/* 关闭设备通道 */
rt_pwm_disable(pwm_dev,PWM_DEV_CHANNEL);
```

10.3 FinSH 命令

设置 PWM 设备的某个通道的周期和占空比可使用命令`pwm_set pwm 1 500000 5000`，第一个参数为命令，第二个参数为 PWM 设备名称，第 3 个参数为 PWM 通道，第 4 个参数为周期（单位纳秒），第 5 个参数为脉冲宽度（单位纳秒）。

```
msh />pwm_set pwm 1 500000 5000
msh />
```

使能 PWM 设备的某个通道可使用命令`pwm_enable pwm 1`，第一个参数为命令，第二个参数为 PWM 设备名称，第 3 个参数为 PWM 通道。

```
msh />pwm_enable pwm1 1
msh />
```

关闭 PWM 设备的某个通道可使用命令pwm_disable pwm 1，第一个参数为命令，第二个参数为 PWM 设备名称，第 3 个参数为 PWM 通道。

```
msh />pwm_disable pwm 1
msh />
```

10.4 PWM 设备使用示例

PWM 设备的具体使用方式可以参考如下示例代码，示例代码的主要步骤如下：

1. 查找 PWM 设备获取设备句柄。
2. 设置 PWM 周期和脉冲宽度。
3. 使能 PWM 设备。
4. while 循环里每 50 毫秒修改一次脉冲宽度。

- 将 PWM 通道对应引脚和 LED 对应引脚相连，可以看到 LED 不停的由暗变到亮，然后又从亮变到暗。

```
/*
 * 程序清单：这是一个 PWM 设备使用例程
 * 例程导出了 pwm_led_sample 命令到控制终端
 * 命令调用格式：pwm_led_sample
 * 程序功能：通过 PWM 设备控制 LED 灯的亮度，可以看到LED不停的由暗变到亮，然后又从亮
     变到暗。
*/

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

#define LED_PIN_NUM         57      /* LED PIN脚编号 */
#define PWM_DEV_NAME        "pwm"   /* PWM设备名称 */
#define PWM_DEV_CHANNEL     4       /* PWM通道 */

struct rt_device_pwm *pwm_dev;      /* PWM设备句柄 */

static int pwm_led_sample(int argc, char *argv[])
{
    rt_uint32_t period, pulse, dir;

    period = 500000;    /* 周期为0.5ms，单位为纳秒ns */
    dir = 1;            /* PWM脉冲宽度值的增减方向 */
```

```
    pulse = 0;           /* PWM脉冲宽度值，单位为纳秒ns */

    /* 设置LED引脚脚模式为输出 */
    rt_pin_mode(LED_PIN_NUM, PIN_MODE_OUTPUT);
    /* 拉高LED引脚 */
    rt_pin_write(LED_PIN_NUM, PIN_HIGH);

    /* 查找设备 */
    pwm_dev = (struct rt_device_pwm *)rt_device_find(PWM_DEV_NAME);
    if (pwm_dev == RT_NULL)
    {
        rt_kprintf("pwm sample run failed! can't find %s device!\n", PWM_DEV_NAME);
        return RT_ERROR;
    }

    /* 设置PWM周期和脉冲宽度默认值 */
    rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
    /* 使能设备 */
    rt_pwm_enable(pwm_dev, PWM_DEV_CHANNEL);

    while (1)
    {
        rt_thread_mdelay(50);
        if (dir)
        {
            pulse += 5000;      /* 从0值开始每次增加5000ns */
        }
        else
        {
            pulse -= 5000;      /* 从最大值开始每次减少5000ns */
        }
        if (pulse >= period)
        {
            dir = 0;
        }
        if (0 == pulse)
        {
            dir = 1;
        }

        /* 设置PWM周期和脉冲宽度 */
        rt_pwm_set(pwm_dev, PWM_DEV_CHANNEL, period, pulse);
    }
}
/* 导出到 msh 命令列表中 */
MSH_CMD_EXPORT(pwm_led_sample, pwm sample);
```

第 11 章

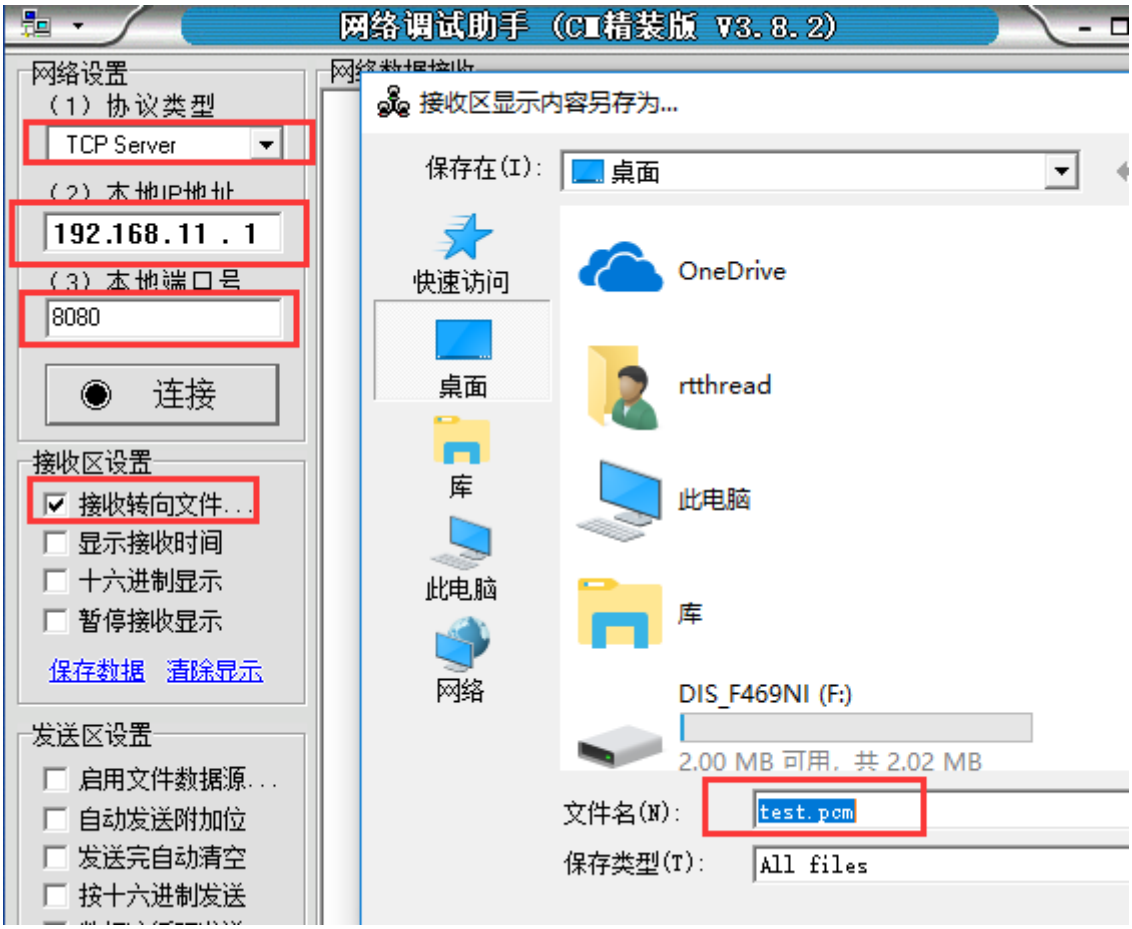
RECORD 设备

11.1 RECORD 使用

设备采集 pcm 数据，windows 开启 tcpserver，设备通过 tcp 将采集的 pcm 数据发送的 windows 上，从而将 pcm 数据保存成文件，在 Windows 播放这个文件，从而验证 record 是否正常工作

11.1.1 windows 开启 tcpserver

如下图所示，开启网络调试助手，选择协议类型，ip 地址，端口号。



11.1.2 设备开启录音功能

设备联网成功, msh 输入如下命令

```
record_pcm_tcp start 16000 192.168.11.1 8080
```

参数	描述
record_pcm_tcp	采集 pcm 数据命令
start	开启录音
16000	采样率
192.168.11.1	tcpserver ip
8080	tcpserver port

接下来就可以对着麦克风录音了

11.1.3 结束录音

msh 输入如下命令

```
record_pcm_tcp stop
```

11.2 示例代码位置

```
test\record_pcm_tcp.c
```

第 12 章

WLAN 设备

12.1 WLAN 设备介绍

随着物联网快速发展，越来越多的嵌入式设备上搭载了 WIFI 无线网络设备。为了能够管理 WIFI 网络设备，RT-Thread 引入了 WLAN 设备管理框架。这套框架具备控制和管理 WIFI 的众多功能，为开发者使用 WIFI 设备提供许多便利。

12.2 WLAN API 介绍

12.2.1 初始化 WLAN 设备

参数	描述
device	wlan 设备
mode	wifi 工作模式 (WIFI_STATION/WIFI_AP)

```
int rt_wlan_init(struct rt_wlan_device *device, rt_wlan_mode_t mode)
```

12.2.2 初始化 wifi 信息

参数	描述
info	保存 wifi 信息的结构体
mode	wifi 工作模式 (WIFI_STATION/WIFI_AP)
security	wifi 加密方式
ssid	wifi ssid

```
void rt_wlan_info_init(struct rt_wlan_info *info, rt_wlan_mode_t mode,
    rt_wlan_security_t security, char *ssid)
```

12.2.3 wifi 连接

参数	描述
device	wlan 设备
info	保存 wifi 信息的结构体
password	wifi 密码

```
int rt_wlan_connect(struct rt_wlan_device *device, struct rt_wlan_info *info, char *
    password);
```

12.2.4 wifi 断开连接

参数	描述
device	wlan 设备

```
int rt_wlan_disconnect(struct rt_wlan_device *device)
```

12.2.5 反初始化 wifi 信息

参数	描述
info	保存 wifi 信息的结构体

```
void rt_wlan_info_deinit(struct rt_wlan_info *info)
```

12.2.6 wifi 扫描

参数	描述
device	wlan 设备
scan_result	wifi 扫描结果

参数	描述

```
int rt_wlan_scan(struct rt_wlan_device *device, struct rt_wlan_scan_result **
    scan_result)
```

12.2.7 释放 wifi 扫描结果

参数	描述
scan_result	wifi 扫描结果

```
void rt_wlan_release_scan_result(struct rt_wlan_scan_result **scan_result)
```

12.2.8 获取 wifi 信号强度

参数	描述
device	wlan 设备
返回值	wifi 信号强度

```
int rt_wlan_get_rssi(struct rt_wlan_device *device)
```

12.3 MSH 命令

12.3.1 wifi 连接

```
wifi w0 join ssid passwd
```

```
msh />wifi w0 join rtthread 12345678
[DRV_WLAN]drivers\wlan\drv_wlan.c L900 beken_wlan_control cmd: case WIFI_INIT!
fast_connect ap info crc failed, crc1:FFFFFFFF crc2:0
[wlan_fast_connect_info_read] channel out of range!
[wifi_connect]: read ap_info is empty
[wifi_connect]: normal connect
_wifi_easyjoin: ssid:rtthread key:12345678
rl_sta_start
........
ctrl_port_hdl:1
```

```
[wlan_connect]:start tick =  14044, connect done tick = 18203, total = 4159
[wlan_connect]:start tick =  14044, connect done tick = 18209, total = 4165
[WLAN_MGNT]wlan sta connected event callback
sta_ip_start

configuring interface mlan (with DHCP client)
dhcp_check_status_init_timer

new dtim period:2
IP UP: 192.168.43.24
[ip_up]:start tick =  14044, ip_up tick = 21727, total = 7683
write new profile to flash 0x001FF000 72 byte!
```

12.3.2 wifi 断开连接

```
wifi w0 down
```

```
sta_ip_down
netif_set_link_down !, vif_idx:0
sm_disconnect_process
SM_DISCONNECT_IND
[WLAN_MGNT]wlan sta disconnected event callback
del hw key idx:1
Cancelling scan request
wpa/host apd remove_if:0
```

12.3.3 wifi 扫描

```
wifi w0 scan
```

```
[DRV_WLAN]drivers\wlan\drv_wlan.c L907 beken_wlan_control cmd: case WIFI_SCAN!
scan_start_req_handler
[DRV_WLAN]release scan done semaphore

ssid: rtthread                          security: WPA2-AES
```

12.3.4 获取 wifi 状态

```
wifi w0 status
```

```
wifi w0 status
Wi-Fi AP: rtthread
MAC Addr: 00:00:00:00:00:00
 Channel: 0
```

```
DataRate: 0Mbps
    RSSI: 976376
```

12.3.5 获取 wifi 信号强度

```
wifi w0 rssi
```

```
msh />wifi w0 rssi
rssi=1026108
```

第 13 章

声波配网

13.1 声波配网介绍

声波配网，即通过手机发出声波，将 ssid、password 等信息传给设备的一种配网方式。适用于没有触屏或触屏较小不易于信息输入，但是拥有麦克风的智能设备，如智能音箱等。

13.2 声波配网使用

13.2.1 设备开启声波配网功能

在 rtconfig.h 中添加宏定义 `SAMPLE_USING_VOICE_CONFIG`, 开启声波配网测试代码。测试代码位置 `samples\voice_config\voice_config.c`

msh 中输入命令 `voice_config` 开启声波，手机打开声波配网小程序，输入 ssid 和 passwd 进行配网

声波配网小程序如下:

串口工具打印如下信息表示接收完成

```
msh />voice_config
msh />[voice] voice config version: 2.0.0
adc-buf:00415c08, adc-buf-len:5120, ch:1
set adc sample rate 16000
ssid len=8, [rtt test]
```

```
passwd L=6, [123456]
custom L=6, [custom]
```

13.2.2 结束声波配网

msh 输入如下命令, 结束声波配网

```
voice_config stop
```

13.2.3 音频文件生成

在不使用小程序的情况下，需要自己手动生成声波测试文件。操作如下:

在 windows 使用 voice_tools.exe 工具生成音频文件, voice_tools.exe 在 tools\voice_config 路径下

在 windows 下打开 cmd 终端，使用如下命令生成音频文件

```
voice_tools.exe "test" "test" "custom" test.wav
```

参数	描述
voice_tools.exe	工具命令
“test”	wifi ssid
“test”	wifi password
“custom”	自定义数据
test.wav	生成的音频文件

注意事项：

如果 custom 为空, 使用空字符串代替

```
voice_tools.exe "test" "test" "" test.wav
```

ssid 最长为 32 bit, 长度超过 32 位，默认截取到 32 位。

将生成的音频拷贝到手机播放即可。

13.2.4 API 说明

13.2.4.1 获取版本号

```
extern const char *voice_config_version(void);
```

13.2.4.2 声波配网开始函数

参数	描述
device	录音设备
sample_rate	采样率 (16000)
timeout	超时时间
result	声波识别结果
返回	–
0	成功
其他	失败

```
struct voice_config_result
{
    uint32_t ssid_len;   /* max 32byte */
    uint32_t passwd_len; /* max 63byte */
    uint32_t custom_len; /* max 16byte */
    char ssid[32+1];
    char passwd[63+1];
    char custom[16+1];
};
```

```
extern int voice_config_work(void *device, uint32_t sample_rate, uint32_t timeout,
    struct voice_config_result *result);
```

13.2.4.3 声波配网停止函数

```
extern void voice_config_stop(void);
```

13.2.4.4 获取录音数据函数

该函数在voice_config_work内部调用，用户只需要实现该函数即可。

参数	描述
device	录音设备
buffer	录音数据 buffer
size	录音数据长度
返回	实际获取的录音数据长度

```
extern int voice_read(void *device, void *buffer, int size);
```

13.2.4.5 申请内存

该函数在voice_config_work内部调用，用户只需要实现该函数即可。

参数	描述
size	要申请的内存大小
返回	内存地址

```
extern void *voice_malloc(int size);
```

13.2.4.6 释放内存

该函数在voice_config_work内部调用，用户只需要实现该函数即可。

参数	描述
mem	要释放的内存指针

```
extern void voice_free(void *mem);
```

第 14 章

airkiss 配网

14.1 airkiss 配网介绍

AirKiss 是微信硬件平台提供的一种 WIFI 设备快速入网配置技术，要使用微信客户端的方式配置设备入网，需要设备支持 AirKiss 技术。

14.2 airkiss 使用

14.2.1 设备开启 airkiss 配网功能

在 rtconfig.h 中添加宏定义 `RT_USING_AIRKISS`, 开启 airkiss 配网功能。

airkiss 代码路径: `samples\airkiss` 和 `samples\airkiss_lan` , 其中 airkiss 文件夹是 airkiss 的适配层，主要对接 airkiss 的接口，用户不需要关心，airkiss_lan 是应用层的代码，用户需要适当修改。

关闭 WiFi 连接，msh 中输入 `airkiss` 开启 airkiss 配网。

日志如下

```
msh />airkiss
msh />Airkiss version: airkiss-2.0.0-25360(Dec 17 2015 17:20:50);arm-none-eabi/gcc
    -4.9.3;ARM
[DRV_WLAN]set monitor callback
[DRV_WLAN]start monitor
Soft_AP_start
[saap]MM_RESET_REQ
[saap]ME_CONFIG_REQ
[saap]ME_CHAN_CONFIG_REQ
[saap]MM_START_REQ
apm start with vif:0
------beacon_int_set:100 TU
set_active param 0
[msg]APM_STOP_CFM
update_ongoing_1_bcn_update
```

```
hal_machw_enter_monitor_mode
Switch channel 2
Switch channel 3
Switch channel 4
Switch channel 5
Switch channel 6
Switch channel 7
Switch channel 8
Switch channel 9
Switch channel 10
Switch channel 11
Switch channel 12
Switch channel 13
Switch channel 14
Switch channel 1
Switch channel 2
Switch channel 3
Switch channel 4
```

14.2.2 微信公众号配网

手机扫描下图的二维码，填写 WiFi ssid 和 passwd，开始配网。

连接成功的打印如下:

```
Lock channel in 1
AIRKISS_STATUS_COMPLETE
airkiss_get_result() ok!
 ssid = realthread pwd = 88888888, ssid_length = 10 pwd_length = 11, random = 0x6
AIRKISS_STATUS_COMPLETE
AIRKISS_STATUS_COMPLETE
AIRKISS_STATUS_COMPLETE
AIRKISS_STATUS_COMPLETE
AIRKISS_STATUS_COMPLETE
AIRKISS_STATUS_COMPLETE
AIRKISS_STATUS_COMPLETE
AIRKISS_STATUS_COMPLETE
```

14.2.3 修改 airkiss 配置

用户想要使用自己的公众号配网时，需要修改`samples\airkiss_lan\weixin_config_custom.c` 文件，将`WEIXIN_DEVICE_TYPE`和`WEIXIN_PRODUCT_ID`修改为用户自己的产品 id 和原始 id。`samples\airkiss_lan\`

airkiss_lan_raw.c中的DEVICE_ID替换为用户自己的设备 id。

以上配置主要用于设备端联网成功后，向公众号发送设备信息，公众号使用设备信息进行绑定。

第 15 章

内存泄漏调试指南

15.1 memtrace 介绍

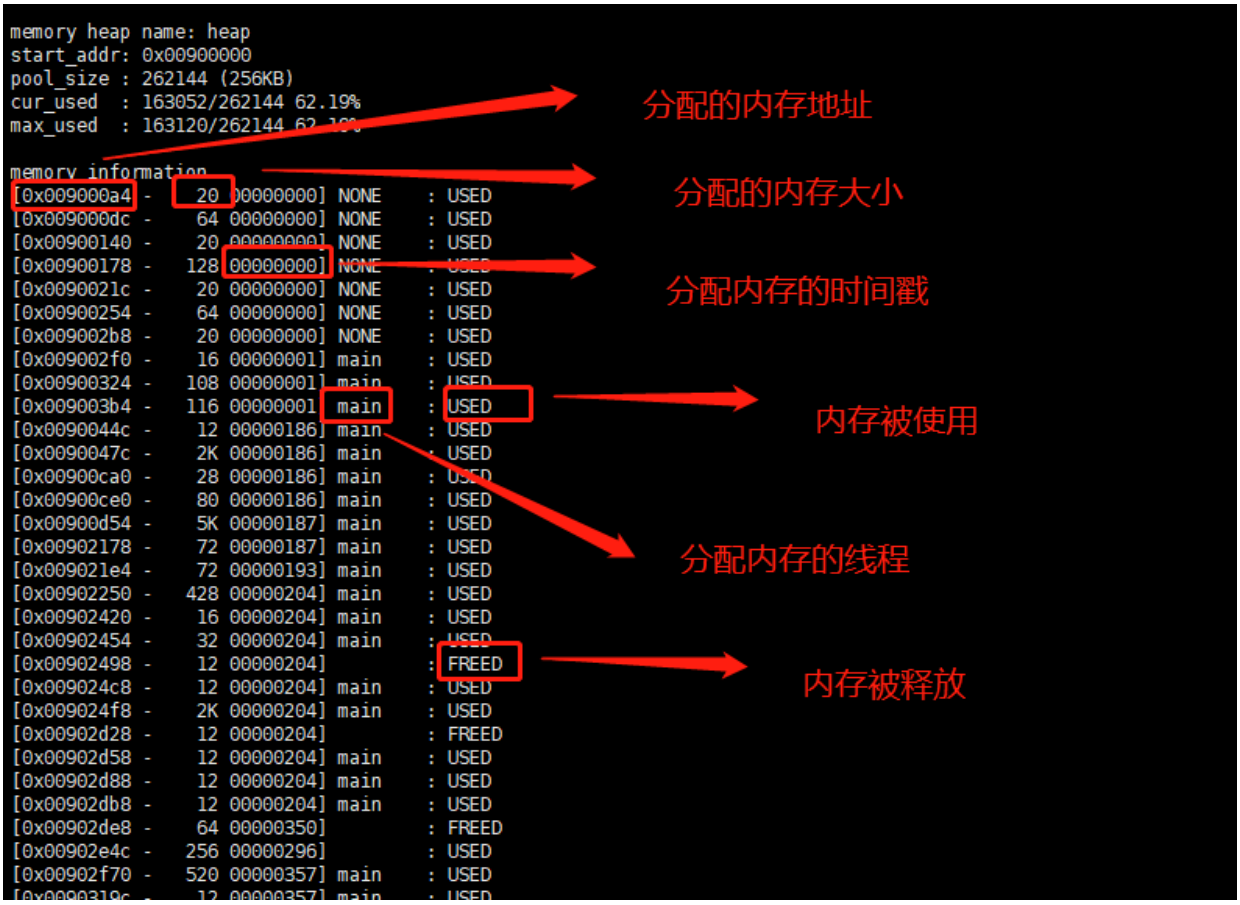
memtrace 是 RT-Thread 开发的用于调试内存泄漏的命令行工具，使用 memtrace 可以实时查看内存创建的时间戳和是否有未释放的内存。

15.2 设备开启 memtrace

在 rtconfig.h 中添加宏定义 `RT_USING_MEMTRACE`, 开启 memtrace 功能。

15.3 memtrace 使用

msh 中输入命令 `memtrace` ， 日志如下:



一般在系统中会存在两部分常驻内存：

1. 设备启动时分配的一些内存，这部分内存一般是系统全局缓存或者不释放的句柄等等。
2. 设备当前时间段内存在的内存，这部分一般是一些中间变量，随着系统的运行会被反复的创建和释放，所以当一些使用动态使用内存的逻辑频繁调用时，会在当前时间节点中存在一些内存

一般在系统中会存在的异常内存（释放泄漏）：

1. 随着设备的开启，不同的时间段内出现周期性相同线程且内存大小一样的 n 个内存，一般这种都是分配后没有释放导致，这种最终会导致系统没有内存使用。
2. 被释放的内存和被使用的内存块在地址上交替出现，这种情况一般也是由于没有释放导致，但是它和第一种情况略微有点不同，这种一般不是频繁周期性的执行逻辑，而是一次性逻辑里或者超低周期性或者随机性执行的逻辑出现的。虽然这种情况一般不会导致系统内存逐步变小，但是它会导致内存碎片的出现，被释放的内存无法整理成大内存，例如：最终会出现内存还有 20kbytes，结果分配不到 15Kbytes。