

BK7251 RTOS SDK

API Reference



Better Life with Wireless
美好生活尽在无线

Version 1.0.0
Copyright © 2019

Release Notes

Version	Date	Description
V1.0	2019.04	First Release
V2.0	2019.08	1.Modified about bootloader 2.add qspi/jpeg moudle

目录

Release Notes	2
1 ADC	14
1.1 ADC 简介	14
1.2 ADC Related API	14
1.2.1 创建 ADC 检测线程	14
1.2.2 配置 ADC 检测通道及回调函数	14
1.2.3 启动 ADC	15
1.2.4 关闭 ADC	15
1.3 ADC 结构体说明	15
1.4 ADC 示例代码	15
2 PWM	18
2.1 PWM 简介	18
2.2 PWM Related API	18
2.2.1 初始化 pwm	18
2.2.2 启动 pwm 功能	18
2.2.3 停止 pwm 功能	19
2.3 PWM 枚举类型说明	19
2.4 PWM 示例	19
3 Audio	21
3.1 Audio 简介	21
3.2 Audio Related API	21
3.2.1 audio 设备初始化	21
3.2.2 打开 mic	21
3.2.3 设置 mic 数据采集通道	22
3.2.4 设置 mic 采样率	22
3.2.5 采集 mic 声音数据	22
3.2.6 关闭 mic	22
3.2.7 打开 audio 设备	22
3.2.8 设置 dac 采样率	23
3.2.9 设置 dac 音量	23

3.2.10 获取 dac 数据，播放音频	23
3.2.11 关闭 dac	23
3.3 Audio 结构体说明	23
3.4 Audio 宏定义	24
3.5 Audio 示例代码	24
4 Airkiss	28
4.1 Airkiss 简介	28
4.2 Airkiss Related API	28
4.2.1 开始 airkiss	28
4.2.2 获取 airkiss 状态	28
4.2.3 获取 airkiss 解码结果	28
4.3 Airkiss 结构体说明	29
4.4 Airkiss 枚举类型说明	29
4.5 Airkiss 示例代码	29
5 声波配网	31
5.1 声波配网简介	31
5.2 声波配网 Related API	31
5.2.1 声波配网开始	31
5.2.2 用户提前终止声波配网	31
5.2.3 获取版本号	31
5.3 声波配网结构体说明	32
5.4 声波配网宏定义	32
5.5 声波配网示例代码	32
6 Button	36
6.1 Button 简介	36
6.2 Button Related API	36
6.2.1 button 初始化	36
6.2.2 配置回调函数	36
6.2.3 开始 button 工作	36
6.2.4 结束 button 工作	37
6.3 Button 枚举类型	37

6.4 Button 示例代码.....	37
7 I2C 总线.....	40
7.1 I2C 简介.....	40
7.2 I2C Related API	40
7.2.1 寻找总线获取设备句柄.....	40
7.2.2 对从设备的读写数据.....	40
7.3 I2C 结构体说明.....	40
7.4 I2C 宏定义.....	41
7.5 模拟 I2C 总线示例代码.....	41
8 I2S 总线.....	45
8.1 I2S 简介.....	45
8.2 I2S Related API.....	45
8.2.1 i2s 模块参数设置.....	45
8.2.2 i2s 主从设备发送/接收数据.....	45
8.3 I2S 结构体说明.....	46
8.4 I2S 宏定义.....	46
8.5 I2S 示例代码.....	46
9 通用 SPI.....	52
9.1 通用 SPI 简介.....	52
9.2 通用 SPI Related API	52
9.2.1 spi 模块配置.....	52
9.2.2 spi 发送数据.....	52
9.2.3 spi 接收数据.....	53
9.3 通用 SPI 结构体说明.....	53
9.4 通用 SPI 宏定义.....	53
9.5 通用 SPI 示例代码.....	54
10 通用 SPI FLASH 设备.....	58
10.1 通用 SPI FLASH 简介.....	58
10.2 通用 SPI FLASH Related API	58
10.2.1 控制设备.....	58
10.3 通用 SPI FLASH 宏定义.....	58

10.4 通用 SPI FLASH 示例代码.....	58
11 通用 SPI PSRAM 设备	62
11.1 通用 SPI PSRAM 简介	62
11.2 通用 SPI PSRAM related API	62
11.3 通用 SPI PSRAM 结构体说明	62
11.4 通用 SPI PSRAM 宏定义	62
11.5 通用 SPI PSRAM 示例代码	62
12 高速 SPI 从设备	65
12.1 高速 SPI 从设备简介	65
12.2 高速 SPI 从设备 Related API	65
12.3 高速 SPI 从设备结构体说明.....	65
12.4 高速 SPI 从设备宏定义.....	65
12.5 高速 SPI 从示例代码.....	65
13 GPIO	69
13.1 GPIO 简介	69
13.2 GPIO Related API.....	69
13.2.1 设置引脚模式	69
13.2.2 设置引脚电平	69
13.2.3 读取引脚电平	70
13.2.4 绑定引脚中断回调函数	70
13.2.5 使能引脚中断	70
13.2.6 脱离引脚中断回调函数	70
13.3 GPIO 宏定义	71
13.4 GPIO 示例代码	71
14 UART	73
14.1 UART 简介	73
14.2 UART Related API.....	73
14.2.1 控制串口设备	73
14.3 UART 结构体说明	73
14.4 UART 宏定义	74
14.5 UART 示例代码	75
15 List Player.....	78

15.1 List Player 简介	78
15.2 List Player Related API	78
15.2.1 初始化播放器	79
15.2.2 获取当前播放曲目的 handle	79
15.2.3 获取当前播放曲目在列表中的索引	79
15.2.4 获取当前播放器的状态	79
15.2.5 获取当前播放曲目的播放位置	79
15.2.6 获取当前播放列表	80
15.2.7 查询是否有播放器及列表存在	80
15.2.8 播放指定列表	80
15.2.9 切换播放至指定列表	80
15.2.10 播放当前列表指定索引曲目	80
15.2.11 播放指定曲目	81
15.2.12 播放器停止	81
15.2.13 暂停播放	81
15.2.14 播放恢复	81
15.2.15 播放上一曲	81
15.2.16 播放下一曲	82
15.2.17 注销当前播放列表	82
15.2.18 注销并释放当前播放列表	82
15.2.19 设置播放器模式	82
15.2.20 建立播放列表	82
15.2.21 删除播放列表	83
15.2.22 回调注册	83
15.2.23 获取播放列表中曲目数	83
15.2.24 获取播放列表中最近播放曲目索引	83
15.2.25 获取播放列表中最近播放曲目	83
15.2.26 曲目添加	84
15.2.27 曲目删除	84
15.2.28 指定索引曲目删除	84
15.2.29 获取指定索引曲目	84

15.2.30 获取指定曲目索引	84
15.3 List Player 宏定义	85
15.4 List Player 示例代码	85
16 网络接口	88
16.1 网络接口简介	88
16.2 网络接口 Related API	88
16.2.1 启动网络	88
16.2.2 启动 STATION 快速连接	88
16.2.3 关闭网络	89
16.2.4 启动 scan	89
16.2.5 注册 scan 结束后的回调函数	89
16.2.6 scan 特定的网络	89
16.2.7 启动监听模式	89
16.2.8 关闭监听模式	90
16.2.9 注册监听回调函数	90
16.2.10 获取当前的网络状态	90
16.2.11 获取当前的连接状态	90
16.2.12 获取当前的信道	91
16.2.13 设置信道	91
16.3 网络接口结构体说明	91
16.4 网络接口枚举类型说明	92
16.5 网络接口宏定义说明	93
16.6 网络接口使用示例	93
17 RTOS 接口	104
17.1 RTOS 接口简介	104
17.2 RTOS Related APIs	104
17.2.1 创建一个新的线程	104
17.2.2 删除一个使用结束的线程	105
17.2.3 使当前线程挂起，等待另一个线程终止	105
17.2.4 使一个线程挂起一段时间	105
17.2.5 初始化一个信号量	105

17.2.6	发出信号量	106
17.2.7	获取一个信号量，并提供超时机制	106
17.2.8	销毁一个信号量	106
17.2.9	初始化一个互斥锁	106
17.2.10	获得一个互斥锁	106
17.2.11	释放一个互斥锁	107
17.2.12	销毁一个互斥锁	107
17.2.13	初始化一个消息队列	107
17.2.14	将一个数据对象推入消息队列	107
17.2.15	从消息队列中取出一个数据对象	108
17.2.16	销毁一个消息队列	108
17.2.17	查询一个队列是否为空	108
17.2.18	查询一个队列是否已满	108
17.2.19	初始化一个时钟，并传入回调函数	109
17.2.20	启动一个时钟	109
17.2.21	停止一个时钟	109
17.2.22	重新加载一个过期的时钟	109
17.2.23	销毁一个时钟	109
17.2.24	获取一个时钟是否正在运行	110
17.3	RTOS 结构体说明	110
17.4	RTOS 枚举类型说明	110
17.5	RTOS 宏定义说明	111
17.6	RTOS 示例代码	111
18	OTA	125
18.1	OTA 简介	125
18.2	OTA Related API	125
18.2.1	fal 初始化	125
18.2.2	远程下载固件	125
18.3	OTA 示例代码	125
19	低功耗	127
19.1	低功耗简介	127

19.2 低功耗 Related API.....	127
19.2.1 进入低功耗模式.....	127
19.2.2 deep_sleep 模式.....	127
19.3 低功耗结构体说明.....	127
19.4 低功耗枚举型说明.....	128
19.5 低功耗宏定义.....	128
19.6 低功耗示例代码.....	128
20 Bootloader.....	131
20.1 Bootloader 简介.....	131
20.2 分区表的设置.....	131
20.2.1 Bootloader 分区.....	131
20.2.2 App 分区.....	131
20.2.3 Download 分区.....	131
20.3 L_boot.....	132
20.4 UP_boot.....	132
20.5 获取 bootloader.bin 文件.....	132
20.6 生成 all.bin 文件.....	132
20.7 Bootloader 示例代码.....	133
20.7.1 2M 分区表信息配置文件 partition_audio_2M.json 示例.....	133
20.7.2 UP_boot 示例.....	133
20.7.3 生成 all.bin 的配置文件 config_sample.json 示例.....	137
21 混音.....	138
21.1 混音简介.....	138
21.2 混音 Related API.....	138
21.2.1 混音初始化.....	138
21.2.2 暂停背景音播放.....	138
21.2.3 重新播放背景音.....	138
21.3 混音宏定义.....	139
21.4 混音示例代码.....	139
22 Vad.....	140
22.1 Vad 自动语音检测简介.....	140

22.2 Vad Related API	140
22.2.1 进入 vad 检测模式	140
22.2.2 获取帧的长度	140
22.2.3 vad 入口函数	140
22.2.4 关闭 vad	141
22.3 Vad 示例代码	141
23 AMR 编码器	145
23.1 AMR 编码器简介	145
23.2 AMR 编码器 Related API	145
23.2.1 AMR-NB 编码器初始化	145
23.2.2 AMR-NB 编码	145
23.2.3 释放 AMR-NB 编码	146
24.3 AMR 编码器宏定义	146
23.4 AMR 编码器枚举类型说明	146
23.5 AMR 编码器示例代码	146
24 Opus 编码器	154
24.1 Opus 编码器简介	154
24.2 Opus 编码器 Related API	154
24.2.1 创建 opus 编码器	154
24.2.2 返回 opus 编码器所需内存的大小	154
24.2.3 修改 opus 编码器的复杂度	155
24.2.4 获取 opus 编码器的比特率	155
24.2.5 获取 opus 编码器的最终状态	155
24.2.6 opus 编码	155
24.2.7 释放 opus 编码器对象	156
24.3 Opus 编码器宏定义	156
24.4 Opus 编码器示例代码	156
25 EasyFlash	164
25.1 EasyFlash 简介	164
25.2 EasyFlash Related API	164
25.2.1 easyflash 初始化	164

25.2.2 获得 easyflash 环境变量	164
25.2.3 将数据写入到环境变量中	164
25.2.4 保存数据到 flash	165
25.3 EasyFlash 宏定义	165
25.4 EasyFlash 示例代码	165
26 Voice Changer	167
26.1 Voice Changer 简介	167
26.2 Voice Changer Related API	167
26.2.1 voice changer 初始化	167
26.2.2 退出 voice changer	167
26.2.3 开始 voice changer	167
26.2.4 停止 voice changer	168
26.2.5 设置 voice changer 变声功能标志	168
26.2.6 voice changer 获取 mic 数据	168
26.2.7 设置消耗数据的长度	168
26.2.8 处理数据	168
26.3 Voice Changer 宏定义	169
26.4 Voice Changer 枚举类型说明	169
26.5 Voice Changer 示例代码	169
27 图像传输	178
27.1 图像传输简介	178
27.2 图像传输 Related API	178
27.2.1 打开 video_transfer	178
27.2.2 关闭 video_transfer	178
27.2.3 设置摄像头的参数	178
27.2.4 打开获取 jpeg 帧的功能	179
27.2.5 关闭获取 jpeg 帧的功能	179
27.2.6 获取 jpeg 帧的数据	179
27.3 图像传输的结构体说明	179
27.4 图像传输的宏定义	180
27.5 图像传输枚举类型说明	180

27.6 图像传输的示例代码	181
28 Qspi Dcache 模式	185
28.1 Qspi Dcache 简介	185
28.2 Qspi Dcache Related API.....	185
28.2.1 初始化 qspi 为 dcache 模式.....	185
28.2.2 启动 qspi 功能	185
28.2.3 停止 qspi 功能	185
28.3 Qspi Dcache 结构体说明	186
28.4 Qspi Dcache 示例	186

1 ADC

1.1 ADC简介

BK7251具有多路通用ADC检测模块，输出精度为10-16bit，可以支持单步及连续等操作模式。电压检测范围为0 ~ 2.4v, ADC channel如下：

通道	描述
0	检测vbat引脚电压，读取值为vbat电压值的1/2
1	检测gpio4引脚电压
2	检测gpio5引脚电压
3	检测gpio23引脚电压(与JTAG引脚复用)
4	检测gpio2引脚电压
5	检测gpio3引脚电压
6	检测gpio12引脚电压
7	检测gpio13引脚电压

Note: 其中ADC通道3与JTAG复用，如果需要用到ADC channel 3需要将JTAG功能关闭，ADC通道对应GPIO需要以上表格为主。

1.2 ADC Related API

ADC相关接口参考\beken378\func\saradc_intf.h，相关接口如下：

函数	描述
saradc_work_create()	创建adc检测线程
adc_obj_init()	配置adc通道以及回调函数
adc_obj_start()	启动adc检测功能
adc_obj_stop()	关闭adc检测功能

1.2.1 创建ADC检测线程

```
void saradc_work_create(UINT32 scan_interval_ms);
```

参数	描述
scan_interval_ms	adc扫描间隔
返回	无

1.2.2 配置ADC检测通道及回调函数

```
void adc_obj_init(ADC_OBJ* handle, adc_obj_callback cb, UINT32 channel, void *user_data);
```

参数	描述
handle	adc检测通道的结构体，参数包括通道号，回调函数等

adc_obj_callback_cb	读取电压值后的回调函数，用来对读取结果进行处理。该回调函数有两个参数：1、new_mv 读取的电压值，单位为mv；2、user_data:对应adc_obj_init()传递的参数user_data
channel	电压检测通道，0-7
user_data	用户数据
返回	无

1.2.3 启动ADC

```
int adc_obj_start(ADC_OBJ* handle);
```

参数	描述
handle	adc检测通道的结构体
返回	0：成功；-1：失败

1.2.4 关闭ADC

```
int adc_obj_stop(ADC_OBJ* handle);
```

参数	描述
handle	adc检测通道的结构体
返回	0：成功；-1：失败

1.3 ADC结构体说明

ADC_OBJ：adc对象的结构体说明

user_data	用户数据
channel	adc通道
cb	回调函数
next	指向下一个adc对象的结构体

1.4 ADC示例代码

ADC示例代码参考test\adc_test.c，打开宏定义：CONFIG_ADC_TEST后，即可看到电池电量的打印信息，在串口输入命令adc_channel_test <start> <channel>，并在对应channel的引脚上接上输入电压，即可看到通道channel的电量打印。对于通道0，检测的电压值需要乘2，才是电池实际电压值。

```
/*
* 程序清单： 这是一个简单ADC程序使用例程，打开宏定义CONFIG_ADC_TEST，输入命令可以看到电压打印信息。
```

* 命令调用格式： 开始检测： adc_channel_test start channel

结束检测： adc_channel_test stop

* 程序功能： 输入开始检测命令后，可以看到打印对应adc通道测量的电压值。

*/

```
#include "include.h"
```

```
#include "arm_arch.h"
```

```
#include "error.h"
```

```
#include "include.h"
```

```
#include <rthw.h>
```

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

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

```
#include <stdint.h>
```

```
#include <stdlib.h>
```

```
#include <finsh.h>
```

```
#include <rtddef.h>
```

```
#include "saradc_intf.h"
```

```
#include "sys_ctrl_pub.h"
```

```
#define CONFIG_ADC_TEST
```

```
#ifdef CONFIG_ADC_TEST
```

```
static ADC_OBJ test_adc;
```

```
/*channel 1 - 7*/
```

```
static void adc_detect_callback(int new_mv, void *user_data)
```

```
{
```

```
    static int cnt = 0;
```

```
    test_adc.user_data = (void*)new_mv;
```

```
    if(cnt++ >= 100)
```

```
    {
```

```
        cnt = 0;
```

```
        rt_kprintf("adc channel%d voltage:%d,%x\r\n",test_adc.channel,new_mv,test_adc.user_data);
```

```
    }
```

```
}
```

```
void adc_channel_test(int argc,char *argv[])
```

```
{
```

```
    int channel;
```



```

if (strcmp(argv[1], "start") == 0)
{
    if(argc == 3)
    {
        channel = atoi(argv[2]);
        rt_kprintf("---adc channel:%d---\r\n",channel);
        saradc_work_create(20);
        adc_obj_init(&test_adc, adc_detect_callback, channel, &test_adc);
        adc_obj_start(&test_adc);
    }
    else
    {
        rt_kprintf("input param error\r\n");
    }
}
if(strcmp(argv[1], "stop") == 0)
{
    adc_obj_stop(&test_adc);
}
}

MSH_CMD_EXPORT(adc_channel_test,adc test);
#endif

```

2 PWM

2.1 PWM简介

BK7251具有6路PWM输出，每一路的周期及占空比都可以单独配置。

通道	描述
0	对应gpio6引脚
1	对应gpio7引脚
2	对应gpio8引脚
3	对应gpio9引脚
4	对应gpio24引脚
5	对应gpio26引脚

Note: 由于目前SDK中pwm0的定时器已经被做为系统时钟使用，所以通道0的pwm功能不能使用。

2.2 PWM Related API

pwm相关接口参考beken378\func\user_driver\BkDriverPwm.h，相关接口如下：

函数	描述
bk_pwm_initialize()	PWM初始化
bk_pwm_start()	启动PWM功能
bk_pwm_stop()	停止PWM功能

2.2.1 初始化pwm

```
OSStatus bk_pwm_initialize(bk_pwm_t pwm, uint32_t cycle, uint32_t duty_cycle);
```

参数	描述
pwm	选择的pwm通道：0 ~ 5
cycle	设置pwm的方波周期
duty_cycle	设置pwm的占空值
返回	0：成功；-1：错误

2.2.2 启动pwm功能

```
OSStatus bk_pwm_start(bk_pwm_t pwm);
```

参数	描述
pwm	选择的pwm通道：0 ~ 5
返回	0：成功；-1：错误

2.2.3 停止pwm功能

OStatus bk_pwm_stop(bk_pwm_t pwm);

参数	描述
pwm	选择的pwm通道： 0~5
返回	0： 成功； -1： 错误

2.3 PWM枚举类型说明

bk_pwm_t:

BK_PWM_0	pwm0
BK_PWM_1	pwm1
BK_PWM_2	pwm2
BK_PWM_3	pwm3
BK_PWM_4	pwm4
BK_PWM_5	pwm5

2.4 PWM示例

示例代码参考test\pwm_test.c，打开宏定义：CONFIG_PWM_TEST，代码运行后，串口输入命令：pwm_test <channel> <duty_cycle> <cycle>即可在对应的pin脚上检测到波形。

```
/*
 * 程序清单： 这是一个简单PWM使用例程，打开宏定义#define CONFIG_PWM_TEST，开启测功能。
 * 命令调用格式： pwm_test 1 8000 16000
 * 程序功能： 输入命令可以检测到对应的PWM通道上输出PWM波形。
 */
#include "rtos_pub.h"
#include "BkDriverPwm.h"
#include "pwm_pub.h"
#include "error.h"
#include <stdint.h>
#include <stdlib.h>
#include <finsh.h>

#define CONFIG_PWM_TEST
#ifdef CONFIG_PWM_TEST

static void pwm_test(int argc,char *argv[])
{
```

```
UINT32 channel,duty_cycle,cycle;
if(argc != 4)
    return;
channel  = atoi(argv[1]);
duty_cycle  = atoi(argv[2]);
cycle      = atoi(argv[3]);
if(cycle < duty_cycle)
{
    rt_kprintf("pwm param error: end < duty\r\n");
    return;
}
rt_kprintf("---pwm %d test--- \r\n",channel);
bk_pwm_initialize(channel, cycle, duty_cycle);    /*pwm 模块初始化，设置对应通道的占空比*/
bk_pwm_start(channel);                          /*启动pwm */

rt_thread_delay(100);

bk_pwm_stop(channel);                          /*关闭pwm */
rt_kprintf("---pwm test stop---\r\n");
}

MSH_CMD_EXPORT(pwm_test,pwm test);
#endif
```

3 Audio

3.1 Audio简介

BK7251芯片具有audio播放以及录音功能，利用麦克风采集音频数据，通过dac输出声音。

3.2 Audio Related API

audio相关接口参考\driver\audio_device.h，相关接口如下：

函数	描述
audio_device_init()	找到sound 和 mic设备，将这两个设备挂在总线上
audio_device_mic_open()	打开mic
audio_device_mic_set_channel()	设置adc通道
audio_device_mic_set_rate()	设置adc采样率
audio_device_mic_read()	mic采集声音数据
audio_device_mic_close()	关闭mic设备
audio_device_open()	打开dac
audio_device_set_rate()	设置dac采样率
audio_device_set_volume()	设置dac音量 0 ~ 16
audio_device_write()	audio播放音频数据
audio_device_close()	关闭dac

3.2.1 audio设备初始化

初始化audio设备包括需找设备“sound”和“mic”句柄，以及分配内存。由于设备已经在开机的时候自动注册了这两个设备，所以只需要在初始化的时候找到这个设备，拿到设备句柄即可。

```
int audio_device_init(void)
```

参数	描述
void	空
返回	RT_EOK: 成功; 错误码: 失败

3.2.2 打开mic

打开 mic device,设置成只读模式。

```
void audio_device_mic_open(void);
```

参数	描述
void	空

返回	空
----	---

3.2.3 设置mic数据采集通道

```
void audio_device_mic_set_channel(int channel);
```

参数	描述
channel	adc通道
返回	空

3.2.4 设置mic采样率

设置mic 采集数据通道,采样率分别有48k,44.1k,32k,16k,8k等。

```
void audio_device_mic_set_rate(int sample_rate);
```

参数	描述
sample_rate	adc通道
返回	空

3.2.5 采集mic声音数据

```
int audio_device_mic_read(void *buffer, int size);
```

参数	描述
buffer	mic读取的buffer
size	mic读取数据长度
返回	length:返回读取数据的长度

3.2.6 关闭mic

```
void audio_device_mic_close(void);
```

参数	描述
void	空
返回	空

3.2.7 打开audio设备

```
void audio_device_open(void);
```

参数	描述
void	空

返回	空
----	---

3.2.8 设置dac采样率

设置dac的采样率，采样率分别有48k、44.1k、32k、16k、8k等。

```
void audio_device_set_rate(int sample_rate);
```

参数	描述
sample_rate	采样率
返回	空

3.2.9 设置dac音量

```
void audio_device_set_volume(int volume);
```

参数	描述
volume	音量大小
返回	空

3.2.10 获取dac数据，播放音频

```
void audio_device_write(void *buffer, int size);
```

参数	描述
buffer	audio播放音频的数据
size	audio播放数据的长度
返回	空

3.2.11 关闭dac

```
void audio_device_close(void);
```

参数	描述
void	空
返回	空

3.3 Audio结构体说明

audio_device:

struct rt_device *snd	sound 设备的句柄
struct rt_device *mic	mic 设备的句柄
struct rt_mempool mp	memory pool

state	audio状态
void (*evt_handler)(void *parameter, int state)	事件中断处理
parameter	audio参数

3.4 Audio宏定义

#define	TEST_BUFF_LEN	60*1024	/*buffer大小
#define	READ_SIZE	1024	/*读取buffer大小

3.5 Audio示例代码

示例代码参考test\mic_record.c，打开宏定义： MICPHONE_TEST，测试录音和播放功能必须关闭混音的宏定义CONFIG_SOUND_MIXER，串口输入命令record_and_play可以听到audio输出端口播放之前录的声音。

```
/*
 * 程序清单： 这是一个录音以及audio播放使用例程
 * 命令调用格式： record_and_play
 * 程序功能： 设备录音完后可以通过audio播放。
 */

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

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "board.h"
#include "audio_device.h"

#define MICPHONE_TEST
#ifdef MICPHONE_TEST

#define TEST_BUFF_LEN 60*1024
#define READ_SIZE 1024

static uint8_t *test_buf;

void record_and_play(int argc, char *argv[])
{
    int mic_read_len = 0;
    int actual_len, i;
    int dac_wr_len=0;
```



```

uint16_t *buffer = NULL;

int vad_on;

#if CONFIG_SOUND_MIXER
    mixer_pause();
#endif

vad_on = atoi(argv[1]);

test_buf = sdram_malloc(TEST_BUFF_LEN);
if(test_buf == NULL)
{
    rt_kprintf("===not enough memory===\r\n");
    return;
}

audio_device_init();                                /*初始化 sound mic设备*/

audio_device_mic_open();                             /*打开mic设备*/
audio_device_mic_set_channel(1);                     /*设置adc通道*/
audio_device_mic_set_rate(16000);                   /*设置adc采样率*/

if (vad_on)
{
    rt_kprintf("Vad is ON !!!!!!!\r\n"); /*进入vad检测*/
    wb_vad_enter();
}

while(1)
{
    if (vad_on)
        rt_thread_delay(5);
    else
        rt_thread_delay(20);

    int chunk_size = wb_vad_get_frame_len();//320
    char *val = NULL;

    if(mic_read_len > TEST_BUFF_LEN - READ_SIZE)

```

```

        break;

    if (!vad_on)
    {
        actual_len = audio_device_mic_read(test_buf+mic_read_len,READ_SIZE);
    }
    else
    {
        /*mic 采集声音数据*/
        actual_len = audio_device_mic_read(test_buf+mic_read_len,chunk_size);
        if(wb_vad_entry(test_buf+mic_read_len, actual_len))
        {
            rt_kprintf("Vad Detected !!!!!!!\r\n");          /*检测到声音*/
            break;
        }
    }

    mic_read_len += actual_len;
}

if (vad_on)
{
    wb_vad_deinit();          /*关闭vad检测*/
}

rt_kprintf("mic_read_len is %d\r\n", mic_read_len);
audio_device_mic_close();          /*关闭mic设备*/

audio_device_open();          /*打开dac设备*/
audio_device_set_rate(8000);      /*设置dac采样率*/

while(1)
{
    buffer = (uint16_t *)audio_device_get_buffer(RT_NULL);
    if(dac_wr_len >= mic_read_len)
    {
        audio_device_put_buffer(buffer);
        break;
    }
}

```

```
        memcpy(buffer, test_buf + dac_wr_len, READ_SIZE);
        dac_wr_len += READ_SIZE;

        audio_device_write((uint8_t *)buffer, READ_SIZE); /*dac播放数据*/
    }
    audio_device_close(); /*关闭dac设备*/

    if(test_buf)
        sdram_free(test_buf); /*释放ram内存*/

#ifdef CONFIG_SOUND_MIXER
    mixer_replay();
#endif
}
MSH_CMD_EXPORT(record_and_play, record play);
#endif
```

4 Airkiss

4.1 Airkiss简介

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。除了配网之外，还包括进场发现功能，该功能是使用型号码必备的功能，用来绑定设备。

4.2 Airkiss Related API

Airkiss相关接口参考\samples\airkiss\airkiss.h，相关接口如下：

函数	描述
<code>airkiss()</code>	开始airkiss
<code>airkiss_get_status()</code>	获取airkiss状态
<code>airkiss_get_result()</code>	当airkiss_recv()返回AIRKISS_STATUS_COMPLETE后，调用此函数来获取AirKiss解码结果

4.2.1 开始airkiss

```
int airkiss(void);
```

参数	描述
<code>void</code>	空
返回	0:成功；其他：失败

4.2.2 获取airkiss状态

```
uint32_t airkiss_get_status(void);
```

参数	描述
<code>void</code>	空
返回	airkiss状态

4.2.3 获取airkiss解码结果

```
int airkiss_get_result(airkiss_context_t *context, airkiss_result_t *result);
```

参数	描述
<code>airkiss_context_t* context,</code>	为airkiss库分配的内存
<code>airkiss_result_t *result</code>	Airkiss解码后的结果
返回	0:成功；其他：失败

4.3 Airkiss结构体说明

airkiss_result_t: airkiss解码成功后的结果

char *pwd	wifi密码
char *ssid	wifi ssid
unsigned char pwd_length	wifi密码长度
unsigned char ssid_length	wifi ssid长度
unsigned char random	随机值, 根据AirKiss协议, 当wifi连接成功后, 需要通过udp向10000端口广播这个随机值, 这样AirKiss发送端(微信客户端或者AirKissDebugger)就能知道AirKiss已配置成功
unsigned char reserved	保留值

4.4 Airkiss枚举类型说明

airkiss_status_t: airkiss状态枚举类型

```
typedef enum
{
    /* 解码正常, 无需特殊处理, 继续调用airkiss_recv()直到解码成功 */
    AIRKISS_STATUS_CONTINUE = 0,

    /* wifi信道已经锁定, 上层应该立即停止切换信道 */
    AIRKISS_STATUS_CHANNEL_LOCKED = 1,

    /* 解码成功, 可以调用airkiss_get_result()取得结果 */
    AIRKISS_STATUS_COMPLETE = 2
} airkiss_status_t;
```

4.5 Airkiss示例代码

示例代码参考\samples\airkiss\airkiss.c, 创建了airkiss配网的线程, airkiss需要在微信平台中连网锁定信道。输入命令start_airkiss, 在微信平台连网, log信息可以显示出是否配网成功。

```
/*
 * 程序清单: 这是一个airkiss配网使用例程
 * 命令调用格式: start_airkiss
 * 程序功能: 通过微信平台给设备配网
 */

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

```

#include <rthw.h>
#include <wlan_dev.h>
#include <wlan_mgnt.h>
#include "airkiss.h"
#include "bk_rtos_pub.h"
#include <stdio.h>
#include <sys/socket.h>
#include "error.h"

int start_airkiss(int argc, char *argv[])
{
    airkiss_result_t *result;
    if(1 == airkiss())
        rt_kprintf("airkiss start\r\n");
    else
        rt_kprintf("airkiss fail\r\n");

    while(airkiss_get_status() != AIRKISS_STATUS_COMPLETE)
    {
        rt_thread_delay(rt_tick_from_millisecond(100));
    }

    result = airkiss_result_get();

    rt_kprintf("---ssid:%s , key:%s---\r\n", result->ssid,result->pwd);
}

#ifdef FINSH_USING_MSH
#include "finsh.h"

MSH_CMD_EXPORT(start_airkiss, start_arksss);
#endif

```

5 声波配网

5.1 声波配网简介

通过voice_tools工具生成16bit, 48kHz, 1个channel的wav/pcm格式的文件, BK7251芯片可以通过识别此类格式的文件来连接网络。

5.2 声波配网 Related API

声波配网相关接口参考samples\voice_config\include\voice_config.h, 相关接口如下:

函数	描述
voice_config_work()	打开设备
voice_config_stop()	用户提前终止声波配网
voice_config_version()	获取声波配网版本号

5.2.1 声波配网开始

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

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

5.2.2 用户提前终止声波配网

```
void voice_config_stop(void)
```

参数	描述
void	无
返回	无

5.2.3 获取版本号

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

参数	描述
void	无
返回	版本号

5.3 声波配网结构体说明

voice_config_result:	
uint32_t ssid_len	网络id长度
uint32_t passwd_len	网络密码长度
uint32_t custom_len	用户自定义数据的长度
char ssid[32+1]	ssid数组
char passwd[63+1]	密码数组
char custom[16+1]	用户自定义的数据

5.4 声波配网宏定义

#define	SAMPLE_USING_VOICE_CONFIG	开启声波配网
---------	---------------------------	--------

5.5 声波配网示例代码

声波配网示例代码参考\test\samples\voice_config\voice_config.c。打开宏定义：VOICE_CONFIG_TEST，开启声波配网测试。

```
/*
 * 程序清单： 这是一个声波配网使用例程，声波配网需要用工具生成一个声音文件，在输入命令之后
 * 让demo板来获取声音，等待demo板配网，配网成功之后会有一系列的打印信息。
 * 命令调用格式： voice_config
 * 程序功能： 手机上播放声音（声音需要voice_tools生成），demo板通过识别手机播放的声音可以连
 * 上网络
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdint.h>

#include <rtthread.h>
#include <rtdevice.h>
#include <rthw.h>
#include "bk_rtos_pub.h"
#include "error.h"
#include "voice_config.h"
```



```

/***** voice config start *****/
static unsigned char voice_config_ssid[32 + 1] = {0};
static unsigned char voice_config_password[64 + 1] = {0};
int voice_config(int argc, char *argv[])
{
    if (tid)
    {
        rt_kprintf("voice config already init.\n");
        return -1;
    }

    tid = rt_thread_create("voice_config",
                           cmd_voice_config_thread,
                           RT_NULL,
                           1024 * 6,
                           20,
                           10);

    if (tid != RT_NULL)
    {
        rt_thread_startup(tid);
    }

    return 0;
}

static rt_thread_t tid = RT_NULL;
static void cmd_voice_config_thread(void *parameter)
{
    rt_device_t device = 0;
    struct voice_config_result result={0};
    int res;

    DEBUG_PRINTF("voice config version: %s\r\n", voice_config_version());    //get voice config
    version

    /* open audio device and set tx done call back */
    device = rt_device_find("mic");
    if (device == RT_NULL)
    {
        DEBUG_PRINTF("audio device not found!\r\n");
        goto _err;
    }

```

```

    }
    codec_device_lock();
    res = rt_device_open(device, RT_DEVICE_OFLAG_RDWR);

    /* set samplerate */
    if (RT_EOK == res)
    {
        int SamplesPerSec = SAMPLE_RATE;
        if (rt_device_control(device, CODEC_CMD_SAMPLERATE, &SamplesPerSec)
            != RT_EOK)
        {
            rt_kprintf("[record] audio device doesn't support this sample rate: %d\n",
                SamplesPerSec);
            goto _err;
        }
    }
    else
    {
        goto _err;
    }

    rt_device_write(device, 0, 0, 100); // start to record
    res = voice_config_work(device, SAMPLE_RATE, 1000 * 60 * 1, &result); //start voice
    configure
    if(res == 0)
    {
        rt_kprintf("ssid len=%d, [%s]\n", result.ssid_len, result.ssid);
        rt_kprintf("passwd L=%d, [%s]\n", result.passwd_len, result.passwd);
        rt_kprintf("custom L=%d, [%s]\n", result.custom_len, result.custom);

        station_connect(result.ssid,result.passwd); //connect station
    }
    else
    {
        rt_kprintf("voice_config res:%d\n", res);
    }

_err:
    if (device)
    {
        rt_device_close(device); //close device
    }

```

```
        codec_device_unlock();
    }
    tid = RT_NULL;

    return;
}

#ifdef FINSH_USING_MSH
#include "finsh.h"
/*命令形式*/
MSH_CMD_EXPORT(voice_config, start voice config);
MSH_CMD_EXPORT(voice_config_stop, stop voice config);
```

6 Button

6.1 Button简介

按键功能包含有按键长按，短按，双击等功能。

6.2 Button Realated API

button相关接口函数参考samples\key\multi_button.h，相关接口如下：

函数	描述
<code>button_init()</code>	按键初始化
<code>button_attach()</code>	配置回调函数
<code>button_start()</code>	开始按键工作，将handle 加入到工作清单
<code>button_stops()</code>	结束按键工作

6.2.1 button初始化

```
void button_init(BUTTON_S* handle, uint8_t(*pin_level)(), uint8_t active_level,void *user_data);
```

参数	描述
<code>BUTTON_S* handle</code>	按键句柄
<code>uint8_t(*pin_level)</code>	读取HAL gpio
<code>uint8_t active_level</code>	gpio level
<code>void *user_data</code>	用户数据
返回	空

6.2.2 配置回调函数

```
void button_attach(BUTTON_S* handle, PRESS_EVT event, btn_callback cb);
```

参数	描述
<code>BUTTON_S* handle</code>	button 句柄
<code>PRESS_EVT event</code>	触发事件类型
<code>btn_callback cb</code>	回调函数
返回	空

6.2.3 开始button工作

```
int button_start(BUTTON_S* handle);
```

参数	描述
----	----

BUTTON_S* handle	button 句柄
返回	0: 成功; 其他: 失败

6.2.4 结束button工作

```
void button_stop(BUTTON_S* handle);
```

参数	描述
BUTTON_S* handle	button 句柄
返回	空

6.3 Button枚举类型

触发按键的事件类型:

```
typedef enum {  
    PRESS_DOWN = 0,      //按键按下  
    PRESS_UP,            //不按  
    PRESS_REPEAT,        //重复按  
    SINGLE_CLICK,        //单击  
    DOUBLE_CLICK,        //双击  
    LONG_PRESS_START,    //长按开始  
    LONG_PRESS_HOLD,     //保持长按  
    NUMBER_OF_EVENT,     //按键事件数量  
    NONE_PRESS           //没有按按键  
}PRESS_EVT;
```

6.4 Button示例代码

Button示例代码参考\samples\key\button_test.c。打开宏定义:
BUTTON_TEST, 开启按键功能测试。

```
/*  
 * 程序清单: 这是一个按键使用例程  
 * 命令调用格式: button_test gpio, 需要用到哪一个gpio作为按键就输入该gpio的序号  
 * 程序功能: 通过短按, 长按, 双击相应的按键, 会在串口打印出相应的状态。  
 */  
  
#include "include.h"  
#include "typedef.h"  
#include "arm_arch.h"  
#include "gpio_pub.h"  
#include "gpio_pub.h"
```

```

#include "uart_pub.h"
#include "multi_button.h"
#include "bk_rtos_pub.h"
#include "error.h"
#include "sys_ctrl_pub.h"

#define BUTTON_TEST
#ifdef  BUTTON_TEST

#define TEST_BUTTON 4
static beken_timer_t g_key_timer;

static void button_short_press(void *param)
{
    rt_kprintf("button_short_press\r\n");
}

static void button_double_press(void *param)
{
    rt_kprintf("button_double_press\r\n");
}

static void button_long_press_hold(void *param)
{
    rt_kprintf("button_long_press_hold\r\n");
}

static uint8_t key_get_gpio_level(BUTTON_S*handle)
{
    return bk_gpio_input((uint32_t)handle->user_data);
}

BUTTON_S gpio_button_test[2];

void button_test(int argc,char *argv[])
{
    OSStatus result;

    int gpio ;
    if(argc != 2)
    {
        rt_kprintf("---! !param error---\r\n");
    }
}

```

```

else
{
    gpio = atoi(argv[1]);

    rt_kprintf("---gpio%d as button : test start---n",gpio);

    if((gpio >=40)|| (gpio >= 40))
    {
        rt_kprintf("---! ! !gpio error---\r\n");
        return;
    }

    /*gpio key config:input && pull up*/
    gpio_config(gpio,GMODE_INPUT_PULLUP);

    button_init(&gpio_button_test[0], key_get_gpio_level, 0,(void*)gpio);    /*初始化按键*/

    /*配置按键事件的回调函数*/
    button_attach(&gpio_button_test[0], SINGLE_CLICK,button_short_press);
    button_attach(&gpio_button_test[0], DOUBLE_CLICK,button_double_press);
    button_attach(&gpio_button_test[0], LONG_PRESS_HOLD,button_long_press_hold);

    button_start(&gpio_button_test[0]);    /*开始按键检测*/
    result = bk_rtos_init_timer(&g_key_timer,    /*初始化按键状态检测时钟*/
                                TICKS_INTERVAL,
                                button_ticks,
                                (void *)0);

    ASSERT(kNoErr == result);

    result = bk_rtos_start_timer(&g_key_timer);    /*开启时钟*/
    ASSERT(kNoErr == result);
}
}

MSH_CMD_EXPORT(button_test,button test);

// eof
#endif

```

7 I2C总线

7.1 I2C简介

BK7251芯片上设有I2C模块，但是在RT_thread 实时操作系统中带有模拟I2C总线的驱动文件，而且已经关联到 RT-thread操作系统的标准设备操作函数集了。底层驱动文件利用MCU的GPIO模拟I2C总线时序，并不是用MCU的硬件I2C接口，I2C接口两个信号线SCL,SDA分别对应gpio2,gpio3。

7.2 I2C Related API

I2C相关接口参考\rt-thread\components\drivers\include\drivers\i2c.h

函数	描述
rt_i2c_bus_device_find()	寻找总线
rt_i2c_transfer()	读/写数据

7.2.1 寻找总线获取设备句柄

```
struct rt_i2c_bus_device *rt_i2c_bus_device_find (const char *bus_name);
```

参数	描述
const char *bus_name	总线名称
返回	i2c总线句柄

7.2.2 对从设备的读写数据

```
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
struct rt_i2c_bus_device *bus	获取的总线句柄，根据注册的总线句柄进行一系列的底层驱动操作
struct rt_i2c_msg msgs[]	包含从设备地址，读/写标志，读/写buffer等
rt_uint32_t num	传输数据次数
返回	RT_EOK: 成功；其他：失败

7.3 I2C结构体说明

rt_i2c_bus_device:

rt_device parent	parent
rt_i2c_bus_device_ops* ops	i2c操作

rt_uint16_t address	从设备地址
rt_uint16_t flags	读写标志位
rt_mutex lock	锁定结构体
rt_uint32_t timeout	超时标志
rt_uint32_t retries	重复次数
void *priv	设备私有数据

rt_i2c_msg:	
rt_uint16_t addr	地址
rt_uint16_t flags	读写标志
rt_uint16_t len	buffer 长度
rt_uint8_t *buf	传送数据的buffer

7.4 I2C宏定义

#define	RT_USING_I2C	使用MCU的I2C设备
#define	RT_USING_I2C_BITOPS	MCU的GPIO模拟I2C总线
#define	BEKEN_USING_IIC	使用I2C驱动

7.5 模拟I2C总线示例代码

软件模拟I2C总线设备示例代码位于\test\i2c_rtt_test.c。打开宏定义：
I2C_RTT_TEST，开启i2c功能测试。

Note: 芯片中没有作为I2C从设备的器件，所以在测试其准确性的时候需要
外挂一个EEPROM。

```

/*
 * 程序清单： 这是一个简单的I2C驱动程序使用例程，从设备使用的是地址为 0x55的EEPROM
 * 例程写出了I2C总线中主设备对从设备的读取操作，
 * 命令调用格式： i2c_test_rtt
 * 程序功能： 7251通过I2C总线对EEPROM的读写控制，来写入或者读取EEPROM的数据，测试过程中
               需要外挂一个eeprom。
 */
#include <rtthread.h>
#include <rtdevice.h>
#include "finsh.h"
#include <rthw.h>
#include <string.h>
#include <time.h>
#include <drv_iic.h>
#define eeprom_addr 0x50; /* 1010A2A1A0 -R/W */

```

```

/*****I2C sample*****/
static int i2c_test_rtt(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;
    rt_uint8_t ret1;

    ret1 = iic_bus_attach( ); /* gpio init and add bus */
    rt_kprintf("iic_bus_attach ret:%d\n", ret1);
    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] = 0;
    msgs[0].addr = eeprom_addr;
    msgs[0].flags = RT_I2C_WR; /* write to slave */
    msgs[0].buf = buffer1; /* eeprom offset. */
    msgs[0].len = 1;

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

    if ( rt_i2c_transfer(i2c_device, msgs, 2) !=2 ) /* write or read data */
    {
        rt_kprintf("--read eeprom fail--\r\n");
    }
}

```

```

else
{
    rt_kprintf("---read eeprom sucess---\r\n");
}
for(i=0; i<sizeof(buffer2); i++)
{
    rt_kprintf("%02X ", buffer2[i]);
}

rt_thread_delay(rt_tick_from_millisecond(50));
rt_kprintf("\r\n---read test over---\r\n");
/*****step 2: write back. *****/

for(i=0; i<sizeof(buffer2); i++)
{
    buffer2[i] = buffer2[i]+5 ;
}
msgs[0].addr = eeprom_addr;
msgs[0].flags = RT_I2C_WR;          /* write to slave */
msgs[0].buf = buffer1;              /* eeprom offset. */
msgs[0].len = 1;

msgs[1].addr = eeprom_addr;
msgs[1].flags = RT_I2C_WR;
msgs[1].len = sizeof(buffer2);
if ( rt_i2c_transfer(i2c_device, msgs, 2) !=2 )
{
    rt_kprintf("---write eeprom fail---\r\n");
}
else

{
    rt_kprintf("---write eeprom sucess---\r\n");
}

rt_thread_delay(rt_tick_from_millisecond(50));

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

```

```

    }

    rt_kprintf("\r\n ---write test over---\r\n");

/*****step 3: read out.*****/

    buffer1[0] = 0;
    msgs[0].addr = eeprom_addr;
    msgs[0].flags = RT_I2C_WR;          /* write to slave */
    msgs[0].buf = buffer1;              /* eeprom offset. */
    msgs[0].len = 1;

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

    if ( rt_i2c_transfer(i2c_device, msgs, 2) !=2 )
    {
        rt_kprintf("---re-read eeprom fail---\r\n");
    }
    else
    {
        rt_kprintf("---re-read eeprom sucess---\r\n");
    }

    rt_thread_delay(rt_tick_from_millisecond(50));

    for(i=0; i<msgs[1].len; i++)
    {
        rt_kprintf("%02X ", buffer2[i]);
    }
    rt_kprintf("\r\n ---re-read test over---\r\n");
    return 0;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
FINSH_FUNCTION_EXPORT_ALIAS(i2c_test_rtt, __cmd_i2c_test_rtt, i2c test rtt cm);
#endif

```

8 I2S总线

8.1 I2S简介

BK7251芯片上设有I2S模块，I2S(Inter—IC Sound)总线是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准，该总线专责于音频设备之间的数据传输，广泛应用于各种多媒体系统。I2S模块包含四根信号线，分别是I2S_CLK, I2S_SYNC,I2S_DIN,I2S_DOUT，对应gpio分别为gpio2, gpio3, gpio4, gpio5, I2S模块可分为I2S、Left justified、Right justified和2B+D等模式。

I2S_CLK:串行时钟信号,也称作BCLK,对应数字音频的每一位数，I2S_SYNC:采样率，I2S_DIN,I2S_DOUT分别为数据的输入和输出。

8.2 I2S Related API

I2S相关接口参考\rt-thread\components\drivers\include\drivers\i2s.h。

函数	描述
i2s_configure()	I2s模块初始化设置
i2s_transfer()	主/从设备发送接收数据

8.2.1 i2s模块参数设置

```
i2s_configure(UINT32 fifo_level, UINT32 sample_rate, UINT32 bits_per_sample, UINT32 mode);
```

参数	描述
fifo_level	配置寄存器中的读写数据fifo水位
sample_rate	配置i2s模块采样率
bits_per_sample	位宽（每个声道的bit数）
mode	配置模式
返回	I2S_SUCCESS: 成功；其他：失败

8.2.2 i2s主从设备发送/接收数据

```
UINT32 i2s_transfer(UINT32 *i2s_send_buf , UINT32 *i2s_recv_buf, UINT32 count, UINT32 param );
```

参数	描述
i2s_send_buf	发送数据buffer
i2s_recv_buf	接收数据buffer
count	发送数据总长度
param	1: 主 0: 从
返回	0: 成功； 其他：失败

8.3 I2S结构体说明

i2s_trans_t:	
p_tx_buf	发送数据buffer
*p_rx_buf;	接收buffer
trans_done	传送数据完成标志位
tx_remain_data_cnt;	发送剩余数据
rx_remain_data_cnt	接收剩余数据
i2s_message:	
send_buf	发送数据buffer
send_len	发送长度
recv_buf	接收数据buffer
recv_len	接收长度

8.4 I2S宏定义

#define	RT_USING_I2S	使用MCU的I2S模块
#define	BEKEN_USING_IIS	使用I2S驱动

I2S工作模式的宏定义:

#define	I2S_MODE	(0 << 0)	I2S模式
#define	I2S_LEFT_JUSTIFIED	(1 << 0)	左对齐模式
#define	I2S_RIGHT_JUSTIFIED	(2 << 0)	右对齐模式
#define	I2S_RESERVE	(3 << 0)	保留
#define	I2S_SHORT_FRAME_SYNC	(4 << 0)	短帧同步
#define	I2S_LONG_FRAME_SYNC	(5 << 0)	长帧同步
#define	I2S_NORMAL_2B_D	(6 << 0)	正常2B+D模式
#define	I2S_DELAY_2B_D	(7 << 0)	延后2B+D模式
#define	I2S_LRCK_NO_TURN	(0 << 3)	lrck不反转
#define	I2S_SCK_NO_TURN	(0 << 4)	sck不反转
#define	I2S_MSB_FIRST	(0 << 5)	MSB先发送
#define	I2S_SYNC_LENGTH_BIT	(8)	Sync长度（仅在长帧同步模式下有效）
#define	I2S_PCM_DATA_LENGTH_BIT	(12)	D的长度（仅在2B+D模式下有效）

8.5 I2S示例代码

示例代码参考\test\i2s_test.c。打开宏定义：I2S_TEST，开启i2s功能测试。

```
/*
```

- * 程序清单： 这是一个简单的I2S驱动程序使用例程，两块demo板一个为主，一个为从设备，
- * 例程写出了i2s总线中主从设备接收/发送数据的操作，
- * 命令调用格式： i2s_test master/slave rate bit_length
- * 程序功能： 主从设备分别接收和发送数据，测试能否正常接受/发送数据，频率位宽是否能够达到要求。

*/

```
#include "include.h"
```

```
#include "arm_arch.h"
```

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

```
#include <rthw.h>
```

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

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#include <stdint.h>
```

```
#include <stdlib.h>
```

```
#include "typedef.h"
```

```
#include "icu_pub.h"
```

```
#include "i2s.h"
```

```
#include "i2s_pub.h"
```

```
#include "sys_ctrl_pub.h"
```

```
#include "drv_model_pub.h"
```

```
#include "mem_pub.h"
```

```
#include "sys_config.h"
```

```
#include "error.h"
```

```
#include "rtos_pub.h"
```

```
#define I2S_DATA_LEN      0x100
```

```
extern UINT32 i2s_configure(UINT32 fifo_level, UINT32 sample_rate, UINT32 bits_per_sample,
UINT32 mode);
```

```
volatile i2s_trans_t i2s_trans;
```

```
i2s_level_t i2s_fifo_level;
```

```
int i2s_test(int argc, char** argv)
```

```
{
```

```

struct rt_device *i2s_device;
struct i2s_message msg;
uint32 i,rate,bit_length ;
uint32 i2s_mode = 0;
if(argc != 4)
{
    rt_kprintf("---cmd error--\r\n");
    return RT_ERROR;
}
rate      = atoi(argv[2]);
bit_length = atoi(argv[3]);

msg.recv_len = I2S_DATA_LEN;
msg.send_len = I2S_DATA_LEN;

msg.recv_buf = rt_malloc(I2S_DATA_LEN * sizeof(msg.recv_buf[0]));
if(msg.recv_buf == RT_NULL)
{
    rt_kprintf("msg.recv_buf malloc failed\r\n");
}
//rt_kprintf("msg.recv_buf=%x\r\n",msg.recv_buf);

msg.send_buf = rt_malloc(I2S_DATA_LEN * sizeof(msg.send_buf[0]));
if(msg.send_buf == RT_NULL)
{
    rt_kprintf("msg.send_buf malloc failed\r\n");
}
//rt_kprintf("msg.send_buf=%x\r\n",msg.send_buf);

/* find device*/
i2s_device = rt_device_find("i2s");
if(i2s_device == RT_NULL)
{
    rt_kprintf("---i2s device find failed---\r\n ");
    return 0 ;
}

/* init device*/
if(rt_device_init( i2s_device) != RT_EOK)
{

```



```

    rt_kprintf(" --i2s device init failed---\r\n ");
    return 0;
}

/* open audio , set fifo level set sample rate/datawidth */
i2s_mode = i2s_mode| I2S_MODE| I2S_LRCK_NO_TURN| I2S_SCK_NO_TURN|
I2S_MSB_FIRST| (0<<I2S_SYNC_LENGTH_BIT)| (0<<I2S_PCM_DATA_LENGTH_BIT);

/* write and recieve */
if(strcmp(argv[1], "master") == 0)
{
    rt_kprintf("---i2s_master_test_start---\r\n");

    if(msg.send_buf == NULL)

    {
        rt_kprintf("---msg.send_buf error --\r\n ");
        return 0;
    }

    for(i=0; i<I2S_DATA_LEN; i++)
    {
        msg.send_buf[i]= ((i+1)<<24) | ((i+1)<<16) | ((i+1)<<8) | ((i+1)<<0);
    }
    i2s_configure(FIFO_LEVEL_32, rate, bit_length, i2s_mode);
    i2s_transfer(msg.send_buf, msg.recv_buf, I2S_DATA_LEN, MASTER);

    for(i=0; i<I2S_DATA_LEN; i++)
    {
        rt_kprintf("msg.send_buf[%d]=0x%x ---msg.recv_buf[%d]=0x%x \r\n", i,
        msg.send_buf[i], i, msg.recv_buf[i]);
    }
    rt_kprintf("---i2s_master_test_over---\r\n");
}

else if(strcmp(argv[1], "slave") == 0)                                     //slave
{
    rt_kprintf("---i2s_slave_test_start---\r\n");

    if(msg.send_buf == NULL)

```

```

    {
        rt_kprintf("---msg.send_buf error --\r\n ");
        return 0;
    }

    for(i=0; i<I2S_DATA_LEN; i++)
    {
        msg.send_buf[i]= ((i+1)<<24) | ((i+1)<<16) | ((i+1)<<8) | ((i+1)<<0) |0x80808080;
    }

    i2s_configure(FIFO_LEVEL_32, rate, bit_length, i2s_mode);
    i2s_transfer(msg.send_buf, msg.recv_buf, I2S_DATA_LEN, SLAVE);

    for(i=0; i<I2S_DATA_LEN; i++)
    {
        rt_kprintf("msg.send_buf[%d]=0x%x , msg.recv_buf[%d]=0x%x \r\n", i, msg.send_buf[i],
            msg.recv_buf[i]);
    }

    rt_kprintf("---i2s_slave_test_over---\r\n");
}
else
{
    rt_kprintf("---no test command--\r\n");
}

i2s_trans.p_rx_buf = RT_NULL;
i2s_trans.p_tx_buf = RT_NULL;

if(msg.send_buf != RT_NULL)
{
    os_free(msg.send_buf);
    msg.send_buf= RT_NULL;
}

if(msg.recv_buf != RT_NULL)
{
    os_free(msg.recv_buf);
}

```

```
        msg.recv_buf= RT_NULL;
    }

    rt_kprintf("---i2s_test_over---\r\n");
    return 0;
}

MSH_CMD_EXPORT(i2s_test, i2s_test);
```

9 通用SPI

9.1 通用SPI简介

BK7251有硬件spi模块，特性如下：

- a) 数据交换长度可配，常以byte为单位，MSB先发，LSB后发；
- b) 支持主机模式，时钟可配置，最大速率30MHZ；
- c) 支持从机模式，能承受的最大速率10MHZ；
- d) 时钟极性（CPOL）和时钟相位（CPHA）可配置；
- e) 支持四线全双工（MOSI、MISO、CSN、CLK）和三线半双工（DATA、CS、CLK）。

9.2 通用SPI Related API

通用SPI的驱动,已经关联到 RT-thread操作系统的标准设备操作函数集了,所以直接调用RT-thread标准设备操作接口进行操作,相关接口参考\rt-thread\components\drivers\include\drivers\spi.h。

函数	描述
rt_spi_configure()	配置spi设备
rt_spi_send()	通过spi接口发送数据（从模式可能会挂起）
rt_spi_recv()	接口spi接口接收数据（从模式可能会挂起）

9.2.1 spi模块配置

在使用SPI接口前，需配置SPI接口：

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

参数	描述
device	SPI设备接口的指针
cfg	SPI配置结构体，见如下说明
返回	RT_EOK(0): 成功; 其他: 出错

9.2.2 spi发送数据

```
rt_inline rt_size_t rt_spi_send(struct rt_spi_device *device,  
                                const void *send_buf,  
                                rt_size_t length);
```

参数	描述
device	SPI设备接口的指针
send_buf	要发送的数据指针

length	要发送的数据长度
返回	此次已发送的字节数

主模式下，发送完所有数据后，立即返回。从模式下，可能会挂起，直到与之通信的SPI主发起spi时序，并且所有数据都发。

9.2.3 spi接收数据

rt_inline rt_size_t rt_spi_recv(struct rt_spi_device *device,
void *recv_buf,
rt_size_t length);

参数	描述
device	SPI设备接口的指针
recv_buf	存放接收的数据指针
length	要接收的数据长度
返回	此次已接收的字节数

主模式下，读SPI总线上的数据后，立即返回。从模式下，可能会挂起，直到与之通信的SPI主发起spi时序，收到非0长度的数据就会返回。

9.3 通用SPI结构体说明

rt_spi_device:

parent	spi device对象
bus	spi bus句柄
config	spi 模式配置的结构体

rt_spi_configuration:

mode	spi工作模式
data_width	发送/接收 数据位宽
reserved	保留
max_hz	spi速率配置，仅master有效

9.4 通用SPI宏定义

#define	RT_USING_SPI	开启SPI模式
#define	CFG_USE_SPI_MASTER	开启master
#define	CFG_USE_SPI_SLAVE	开启slave
spi工作模式:		
#define	RT_SPI_CPHA	(1<<0) sck第二个边沿采样数据
#define	RT_SPI_CPOL	(1<<1) sck空闲时处于高电平
#define	RT_SPI_LSB	(0<<2) 0-LSB

#define	RT_SPI_MSB	(1<<2)	1-MSB
#define	RT_SPI_MASTER	(0<<3)	master模式
#define	RT_SPI_SLAVE	(1<<3)	slave模式
#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_4	(RT_SPI_CPOL RT_SPI_CPHA)	CPOL = 1, CPHA = 1
#define	RT_SPI_MODE_MASK	(RT_SPI_CPHA RT_SPI_CPOL RT_SPI_MSB RT_SPI_SLAVE)	所有bit位为1

9.5 通用SPI示例代码

示例代码参考\test\general_spi_test.c。打开宏定义: GENERAL_SPI_TEST, 开启通用spi功能测试。

```

/*
 * 程序清单： 这是通用spi的使用例程，使用前确保函数 rt_hw_spi_device_init在系统初始化时调用。
 * 命令调用格式： gspi_test master/slave tx/rx rate len
 * 程序功能： 配置spi接口为主/从，传输速率rate，完成发送/接收
 */

#include <rtthread.h>
#include <rthw.h>
#include <rtdevice.h>
#include <stdio.h>
#include <string.h>
#include "sys_config.h"

#define SPI_BAUDRATE      (10 * 1000 * 1000)
#define SPI_TX_BUF_LEN    (32)
#define SPI_RX_BUF_LEN    (32)

/*依赖 CFG_USE_SPI_MASTER 和 CFG_USE_SPI_SLAVE两个宏，位于sys_config.h中 */
#if ((CFG_USE_SPI_MASTER) &&(CFG_USE_SPI_SLAVE))
int gspi_test(int argc, char** argv)
{
    struct rt_spi_device *spi_device;
    struct rt_spi_configuration cfg;
    /*找到设备*/
    spi_device = (struct rt_spi_device *)rt_device_find("gspi");
    if (spi_device == RT_NULL)

```

```

{
    rt_kprintf("spi device %s not found!\n", "gsapi");
    return -RT_ENOSYS;
}

cfg.data_width = 8;
if(strcmp(argv[1], "master") == 0)
{
    /*设置成 主模式、MSB、CPOL = 0, CPHA = 0*/
    cfg.mode = RT_SPI_MODE_0 | RT_SPI_MSB | RT_SPI_MASTER;
}
else if(strcmp(argv[1], "slave") == 0)
{
    /*设置成 从模式、MSB、CPOL = 0, CPHA = 0*/
    cfg.mode = RT_SPI_MODE_0 | RT_SPI_MSB | RT_SPI_SLAVE;
}
else
{
    rt_kprintf("gsapi_test master/slave  tx/rx  rate  len\n");
    return -RT_ENOSYS;
}

/* SPI Interface with Clock Speeds Up to 30 MHz */
if(argc == 5)
    cfg.max_hz = atoi(argv[3]);
else
    cfg.max_hz = SPI_BAUDRATE;

rt_kprintf("cfg:%d, 0x%02x, %d\n", cfg.data_width, cfg.mode, cfg.max_hz);
/*配置设备*/
rt_spi_configure(spi_device, &cfg);
if(strcmp(argv[2], "tx") == 0)
{
    rt_uint8_t *buf;
    int tx_len;
    if(argc < 4)
        tx_len = SPI_TX_BUF_LEN;
    else
        tx_len = atoi(argv[4]);
    rt_kprintf("spi init tx_len:%d\n", tx_len);
    buf = rt_malloc(tx_len * sizeof(rt_uint8_t));
    if(buf)
    {

```

```

        rt_memset(buf, 0, tx_len);
        for(int i=0; i<tx_len; i++)
        {
            buf[i] = i & 0xff;
        }
        /*发送数据， 从模式可能会挂起*/
        rt_spi_send(spi_device, buf, tx_len);
        for(int i=0; i<tx_len; i++)
        {
            rt_kprintf("%02x,", buf[i]);
            if((i+1)%32 == 0)
                rt_kprintf("\r\n");
        }
        rt_kprintf("\r\n");
        rt_free(buf);
    }
}

else if(strcmp(argv[2], "rx") == 0)
{
    rt_uint8_t *buf;
    int rx_len;
    if(argc < 4)
        rx_len = SPI_RX_BUF_LEN;
    else
        rx_len = atoi(argv[4]);
    rt_kprintf("spi init rx_len:%d\n", rx_len);
    buf = rt_malloc(rx_len * sizeof(rt_uint8_t));
    if(buf)
    {
        rt_memset(buf, 0, rx_len);
        /*接收数据， 从模式可能会挂起*/
        rx_len = rt_spi_rcv(spi_device, buf, rx_len);
        rt_kprintf("rx ret:%d\r\n", rx_len);
        for(int i=0; i<rx_len; i++)
        {
            rt_kprintf("%02x,", buf[i]);
            if((i+1)%32 == 0)
                rt_kprintf("\r\n");
        }
        rt_kprintf("\r\n");
    }
}

```



```
        rt_free(buf);
    }
}
else
{
    rt_kprintf("gsapi_test master/slave  tx/rx  rate  len\r\n");
}
}
MSH_CMD_EXPORT(gsapi_test, gsapi_test);
```

10 通用SPI FLASH设备

10.1 通用SPI FLASH简介

SPI FLASH设备是一个外挂的标准flash，特点如下：

- a) 使用SPI四线主模式；
- b) 最高访问速度达30MHz。

10.2 通用SPI FLASH Related API

通用SPI FLASH的驱动，已经关联到 RT-thread操作系统的标准设备操作函数集了，所以直接调用RT-thread标准设备操作接口进行操作。相关接口如下：

函数	描述
<code>rt_device_control()</code>	其他spi flash的操作，如：擦除指定位置，去/加写保护等

10.2.1 控制设备

需要对设备进行其他操作：

```
rt_err_t rt_device_control(rt_device_t dev, int cmd, void *arg);
```

参数	描述
<code>dev</code>	SPI FLASH设备的指针
<code>cmd</code>	设备定义的操作命令，具体见下面说明。
<code>arg</code>	设备定义的操作命令参数，具体见下面说明。
返回	出错信息：RT_EOK(0)：成功；其他：出错

10.3 通用SPI FLASH宏定义

flash工作命令：

<code>#define</code>	<code>BK_SPI_FLASH_ERASE_CMD</code>	擦除命令
<code>#define</code>	<code>BK_SPI_FLASH_PROTECT_CMD</code>	flash加写保护
<code>#define</code>	<code>BK_SPI_FLASH_UNPROTECT_CMD</code>	flash去写保护

10.4 通用SPI FLASH示例代码

示例代码参考\test\general_spi_flash_test.c。打开宏定义：
`SPI_FLASH_TEST`，开启通用spi flash功能测试。

```
/*
 * 程序清单： 这是spi flash的使用例程，使用前确保函数 rt_spi_flash_hw_init(void)函数在系统初始化
              自动调用。
 * 命令调用格式： gspi_flash_test
 * 程序功能： 测试 spi flash 读写数据的功能
```

```

*/
#include <rtthread.h>
#include <rthw.h>
#include <rtdevice.h>
#include <stdio.h>
#include <string.h>
#include "sys_config.h"
#ifdef BEKEN_USING_SPI_FLASH

/*SPI FLASH 需要关联BEKEN_USING_SPI_FLASH、 CFG_USE_SPI_MASTER 、
CFG_USE_SPI_MST_FLASH 三个功能宏*/
#if ((CFG_USE_SPI_MASTER == 0) || (CFG_USE_SPI_MST_FLASH == 0))
#error "test gspi psram need 'CFG_USE_SPI_MASTER' and 'CFG_USE_SPI_MST_FLASH'"
#endif

#include "drv_spi_flash.h"
#define FTEST_BUF_SIZE      1024
#define FTEST_BASE          0x40
#define FTEST_ADDR          0x100000
void gspi_flash_test(int argc, char** argv)
{
    struct rt_device *flash;
    /*找到设备*/
    flash = rt_device_find("spi_flash");
    if (flash == NULL)
    {
        rt_kprintf("psram not found \n");
        return;
    }
    /*初始化设备 */
    if (rt_device_init(flash) != RT_EOK)
    {
        return;
    }
    /*打开设备*/
    if (rt_device_open(flash, 0) != RT_EOK)
    {
        return;
    }
    uint8_t buffer[FTEST_BUF_SIZE], *ptr;

```

```

int i;
rt_kprintf("[SPIFLASH]: SPIFLASH test begin\n");
rt_memset(buffer, 0, FTEST_BUF_SIZE);
/*先读一次 */
rt_device_read(flash, FTEST_ADDR, buffer, FTEST_BUF_SIZE);
/*打印读到的数据 */
ptr = buffer;
rt_kprintf("flash data:%x\r\n", FTEST_ADDR);
for(i=0; i<FTEST_BUF_SIZE; i++)
{
    rt_kprintf("0x%02x,", ptr[i]);
    if((i+1)%16 == 0)
        rt_kprintf("\r\n");
}
rt_kprintf("\r\n");

/*初始化将写数据，数据来源于代码的 FTEST_BASE开始的地方 */
ptr = (uint8_t *)FTEST_BASE;
rt_kprintf("base data:%08x\r\n", ptr);
for(i=0; i<FTEST_BUF_SIZE; i++)
{
    rt_kprintf("0x%02x,", ptr[i]);
    buffer[i] = ptr[i];
    if((i+1)%16 == 0)
        rt_kprintf("\r\n");
}
rt_kprintf("\r\n");
/*写之前，先去写保护 */
rt_device_control(flash, BK_SPI_FLASH_UNPROTECT_CMD, NULL);
/*写数据 */
rt_device_write(flash, FTEST_ADDR, buffer, FTEST_BUF_SIZE);
rt_kprintf("write fin\r\n");
/*清0buffer */
rt_memset(buffer, 0, FTEST_BUF_SIZE);
/*再读回来 */
rt_device_read(flash, FTEST_ADDR, buffer, FTEST_BUF_SIZE);
rt_kprintf("read fin\r\n");
/*打印读回来的数据 */
ptr = buffer;
rt_kprintf("flash data:%x\r\n", FTEST_ADDR);

```

```
for(i=0; i<FTEST_BUF_SIZE; i++)
{
    rt_kprintf("0x%02x,", ptr[i]);
    if((i+1)%16 == 0)
        rt_kprintf("\r\n");
}
rt_kprintf("\r\n");
/*擦除flash */
rt_kprintf("earase\r\n");
BK_SPIFLASH_ERASE_ST erase_st;
erase_st.addr = FTEST_ADDR;
erase_st.size = 4 * 1024;
rt_device_control(flash, BK_SPI_FLASH_ERASE_CMD, &erase_st);
rt_kprintf("[SPIFLASH]: SPIFLASH test end\r\n");
/*加写保护 */
rt_device_control(flash, BK_SPI_FLASH_PROTECT_CMD, NULL);
/*关闭设备 */
rt_device_close(flash);
}
MSH_CMD_EXPORT(gspi_flash_test, gspi_flash_test);
#endif // BEKEN_USING_SPI_FLASH
```

11 通用SPI PSRAM设备

11.1 通用SPI PSRAM简介

SPI PSRAM设备，即需要外挂psram，是基于通用SPI模块的一个具体应用：

- a) 使用SPI四线主模式；
- b) 最高访问速度达30MHZ。

11.2 通用SPI PSRAM related API

通用SPI PSRAM的驱动，已经关联到 RT-thread操作系统的标准设备操作函数集了，所以直接调用RT-thread标准设备操作接口进行操作。

11.3 通用SPI PSRAM结构体说明

PSRAM: rt_device 类型的结构体

11.4 通用SPI PSRAM宏定义

#define	BEKEN_USING_SPI_PSRAM	开启spi psram模块
----------------	------------------------------	---------------

11.5 通用SPI PSRAM示例代码

示例代码参考\test\ general_spi_psram_test.c。打开宏定义：
SPI_PSRAM_TEST，开启通用spi psram功能测试

```
/*
 * 程序清单： 这是spi psram的使用例程，测试的时候需要外挂一个psram，使用前确保函数
   rt_spi_psram_hw_init ()会在系统初始化自动调用；
 * 命令调用格式： spi_psram_test
 * 程序功能： 测试 spi psram 读写数据的功能
 */

#include <rtthread.h>
#include <rthw.h>
#include <rtdevice.h>
#include <stdio.h>
#include <string.h>
#include "sys_config.h"

#ifdef BEKEN_USING_SPI_PSRAM
/*SPI PSRAM 需要关联BEKEN_USING_SPI_PSRAM、 CFG_USE_SPI_MASTER 、
CFG_USE_SPI_MST_PSRAM 三个功能宏*/
#if ((CFG_USE_SPI_MASTER == 0) || (CFG_USE_SPI_MST_PSRAM == 0))
#error "test gspi psram need 'CFG_USE_SPI_MASTER' and 'CFG_USE_SPI_MST_PSRAM'"
#endif
#endif
```

```

void spi_psram_test(int argc, char** argv)
{
    struct rt_device *psram;
    /*找到设备*/
    psram = rt_device_find("spi_psram");
    if (psram == NULL)
    {
        rt_kprintf("psram not found \n");
        return;
    }
    /*初始化设备*/
    if (rt_device_init(psram) != RT_EOK)
    {
        return;
    }
    /*打开设备*/
    if (rt_device_open(psram, 0) != RT_EOK)
    {
        return;
    }
    uint8_t buffer[4096];
    int i;
    rt_kprintf("[PSRAM]: SPRAM test begin\n");
    /*初始化将要写入的设备*/
    for(i = 0; i < sizeof(buffer); i++)
    {
        buffer[i] = (uint8_t)i;
    }
    /*写设备*/
    rt_device_write(psram, 0, buffer, sizeof(buffer));
    /*清0 buffer*/
    rt_memset(buffer, 0, sizeof(buffer));
    /*读设备*/
    rt_device_read(psram, 0, buffer, sizeof(buffer));
    /*比较读到的数据与写入的数据是否一致，不一致的打印出来 */
    for(i = 0; i < sizeof(buffer); i++)
    {
        if(buffer[i] != (uint8_t)i)
        {

```

```
        rt_kprintf("[%02d]: %02x - %02x\n", i, (uint8_t)i, buffer[i]);
    }
}
rt_kprintf("[PSRAM]: SPRAM test end\n");
/*关闭设备*/
rt_device_close(psram);
}
MSH_CMD_EXPORT(spi_psram_test, spi_psram_test);
#endif // BEKEN_USING_SPI_PSRAM
```


12 高速SPI从设备

12.1 高速SPI从设备简介

Highspeed spi slave设备(以下简称spi_hs)是为了解决通用spi从模式不能承受大spi clock的问题而诞生的:

- a) 支持四线全双工、三线半双工模块;
- b) 支持MSB、LSB可配置;
- c) 支持DMA传输;
- d) 承受spi clock 达50MHZ。

Note:驱动为了方便和简单, 固定了spi_hs的配置如下: 四线模式、MSB、使用DMA发送和接收。

12.2 高速SPI从设备Related API

spi_hs的驱动, 已经关联到 RT-thread操作系统的标准设备操作函数集了, 所以直接调用RT-thread标准设备操作接口进行操作。

12.3 高速SPI从设备结构体说明

spi_hs: rt_device 类型的结构体

12.4 高速SPI从设备宏定义

#define	BEKEN_USING_SPI_HSLAVE	开启高速spi 从设备
#define	SPI_TX_BUF_LEN	高速spi 从设备发送数据长度
#define	SPI_RX_BUF_LEN	高速spi 从设备接收数据长度

12.5 高速SPI从示例代码

示例参考/test/ highspeed_spi_slave_test.c。打开宏定义: SPI_HSLAVE_TEST, 开启高速spi功能的测试。

```
/*
 * 程序清单: 这是spi hs的使用例程, 使用前确保函数 rt_spi_hslave_hw_init () 在系统初始化自动调用。
 * 命令调用格式: spi_hs_test tx/rx len
 * 程序功能: 测试 spi hs 读写数据的功能
 */
#include <rtthread.h>
#include <rthw.h>
#include <rtdevice.h>
#include <stdio.h>
#include <string.h>
#include "sys_config.h"
```

```

#define SPI_TX_BUF_LEN      (512)
#define SPI_RX_BUF_LEN      (512)
#ifdef BEKEN_USING_SPI_HSLAVE

/*SPI HS需要关联BEKEN_USING_SPI_HSLAVE、 CFG_USE_HSLAVE_SPI 二个功能宏*/
#if (CFG_USE_HSLAVE_SPI == 0)
#error "spi_hs_test need 'CFG_USE_HSLAVE_SPI' and 'CFG_USE_SPI_MST_PSRAM'"
#endif

int spi_hs_test(int argc, char** argv)
{
    struct rt_device *spi_hs;
    /*找到设备*/
    spi_hs = (struct rt_device *)rt_device_find("spi_hs");
    if (spi_hs == RT_NULL)
    {
        rt_kprintf("spi device %s not found!\r\n", "spi_hs");
        return -RT_ENOSYS;
    }
    /*打开设备*/
    if (rt_device_open(spi_hs, 0) != RT_EOK)
    {
        return 0;
    }
    if(strcmp(argv[1], "tx") == 0)
    {
        rt_uint8_t *buf;
        int tx_len;
        if(argc < 3)
            tx_len = SPI_TX_BUF_LEN;
        else
            tx_len = atoi(argv[2]);
        rt_kprintf("spi hs tx_len:%d\r\n", tx_len);
        buf = rt_malloc(tx_len * sizeof(rt_uint8_t));
        if(buf)
        {
            rt_memset(buf, 0, tx_len);
            for(int i=0; i<tx_len; i++)
            {
                buf[i] = i & 0xff;
            }
        }
    }
}

```

```

        /*写数据*/
        rt_device_write(spi_hs, 0, (const void *)buf, tx_len);
        for(int i=0; i<tx_len; i++)
        {
            rt_kprintf("%02x,", buf[i]);
            if((i+1)%32 == 0)
                rt_kprintf("\r\n");
        }
        rt_kprintf("\r\n");
        rt_free(buf);
    }
}

else if(strcmp(argv[1], "rx") == 0)
{
    rt_uint8_t *buf;
    int rx_len;
    if(argc < 3)
        rx_len = SPI_RX_BUF_LEN;
    else
        rx_len = atoi(argv[2]);
    rt_kprintf("spi hs rx_len:%d\r\n", rx_len);
    buf = rt_malloc(rx_len * sizeof(rt_uint8_t));
    if(buf)
    {
        rt_memset(buf, 0, rx_len);
        /*接收数据*/
        rx_len = rt_device_read(spi_hs, 0, buf, rx_len);
        rt_kprintf("rx ret:%d\r\n", rx_len);
        for(int i=0; i<rx_len; i++)
        {
            rt_kprintf("%02x,", buf[i]);
            if((i+1)%32 == 0)
                rt_kprintf("\r\n");
        }
        rt_kprintf("\r\n");
        rt_free(buf);
    }
}

else
{

```

```
        rt_kprintf("spi_hs_test tx/rx len\r\n");
    }
    /*关闭设备*/
    rt_device_close(spi_hs);
}
MSH_CMD_EXPORT(spi_hs_test, spi_hs_test);
#endif // BEKEN_USING_SPI_HSLAVE
```

13 GPIO

13.1 GPIO简介

BK7251芯片上的引脚一般分为4类：电源、时钟、模拟/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）。

13.2 GPIO Related API

GPIO相关接口参考`rt-thread\components\drivers\include\drivers\pin.h`应用程序可通过以下API访问GPIO，相关接口如下所示：

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

13.2.1 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
<code>pin</code>	引脚编号
<code>mode</code>	引脚工作模式
返回	空

13.2.2 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
<code>pin</code>	引脚编号
<code>value</code>	电平逻辑值，可取2种宏定义值之一： <code>PIN_LOW</code> 低电平， <code>PIN_HIGH</code> 高电平
返回	空

13.2.3 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

13.2.4 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

13.2.5 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

13.2.6 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。

13.3 GPIO宏定义

目前支持的引脚工作模式可取如所示的4 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_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）
#define	PIN_IRQ_MODE_HIGH_LEVEL	0x03	高电平触发
#define	PIN_IRQ_MODE_LOW_LEVEL	0x04	低电平触发

13.4 GPIO示例代码

GPIO设备示例代码位于\test\gpio_demo.c，打开宏定义：GPIO_DEMO，开启GPIO的功能测试，PIN设备的具体使用方式可以参考如下示例代码，其中LED_PIN_NUM、LED1_PIN_NUM、LED2_PIN_NUM、KEY0_PIN_NUM、KEY1_PIN_NUM 根据实际的硬件有所改变。

```
/*
 * 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 30
#define LED1_PIN_NUM 13
#define LED2_PIN_NUM 27
#define KEY0_PIN_NUM 2
#define KEY1_PIN_NUM 3

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);
    /* 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);
    /* led 引脚为输出模式*/
    rt_pin_mode(LED1_PIN_NUM, PIN_MODE_OUTPUT);
    /* 默认低电平*/
    rt_pin_write(LED1_PIN_NUM, PIN_LOW);
    /* led 引脚为输出模式*/
    rt_pin_mode(LED2_PIN_NUM, PIN_MODE_OUTPUT);
    /* 默认低电平*/
    rt_pin_write(LED2_PIN_NUM, PIN_HIGH);
}

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

```


14 UART

14.1 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，否则通信将无法完成。

14.2 UART Related API

应用程序通过BK7251 SDK提供的I/O 设备管理接口来访问串口硬件，其API已经关联到 RT-thread操作系统的标准设备操作函数集了，所以直接调用RT-thread标准设备操作接口进行操作相关接口如下所示：

函数	描述
<code>rt_device_control()</code>	控制设备

14.2.1 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
<code>dev</code>	设备句柄
<code>cmd</code>	命令控制字，参考宏定义
<code>arg</code>	控制的参数，可取类型： <code>struct serial_configure</code>
返回	RT_EOK 函数执行成功 -RT_ENOSYS 执行失败， <code>dev</code> 为空 其他错误码 执行失败

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。

14.3 UART结构体说明

serial:rt_device类型的结构体

配置uart参数的结构体:

serial_configure:

结构体类型	成员
rt_uint32_t baud_rate	波特率设置: 一般为115200
rt_uint32_t data_bits	数据位: 一般为8bit
rt_uint32_t stop_bits	停止位: 一般为1
rt_uint32_t parity	奇偶校验位: 无校验位
rt_uint32_t bit_order	大小端: 一般为小端
rt_uint32_t invert	模式转化: 不转换
rt_uint32_t bufsz	接收buffer大小
rt_uint32_t reserved	保留

14.4 UART宏定义

BK7251SDK 提供的默认宏配置如下:

#define	BAUD_RATE_115200	115200	波特率
#define	DATA_BITS_8	8	数据位
#define	STOP_BITS_1	1	停止位
#define	PARITY_NONE	0	奇偶校验位
#define	BIT_ORDER_LSB	0	高位在前或者低位在前
#define	NRZ_NORMAL	0	模式
#define	RT_SERIAL_RB_BUFSZ	64	接收数据缓冲区大小

配置设备宏定义:

#define	RT_DEVICE_CTRL_CONFIG	0x03	配置对应的设备
---------	-----------------------	------	---------

设置波特率:

#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_203400	203400
#define	BAUD_RATE_460800	460800
#define	BAUD_RATE_921600	921600
#define	BAUD_RATE_2000000	2000000
#define	BAUD_RATE_3000000	3000000

设置数据位:

#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
设置停止位:		
#define	STOP_BITS_1	0
#define	STOP_BITS_2	1
#define	STOP_BITS_3	2
#define	STOP_BITS_4	3
设置奇偶校验位:		
#define	PARITY_NONE	0
#define	PARITY_ODD	1
#define	PARITY_EVEN	2
设置高位在前:		
#define	BIT_ORDER_LSB	0 高位在前
#define	BIT_ORDER_MSB	1 高位在后
模式选择		
#define	NRZ_NORMAL	0 normal mode
#define	NRZ_INVERTED	1 inverted mode

14.5 UART示例代码

uart示例代码位于\test\uart_demo.c，打开宏定义：UART_DEMO，开启uart的功能测试，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);
```

15 List Player

15.1 List Player简介

List Player提供列表建立与播放功能，并可以支持多个列表的切换。

15.2 List Player Related API

List Player相关接口参考\rt-thread\components\list_player.h。应用程序可通过以下APIs访问List Player，相关接口如下所示：

函数	描述
list_player_init()	初始化播放器
list_player_current_item()	获取当前播放曲目的handle
list_player_current_index()	获取当前播放曲目在列表中的索引
list_player_current_state()	获取播放器当前状态
list_player_current_position()	获取当前播放曲目的播放位置
list_player_current_items()	获取当前播放列表
list_player_is_exist()	查询是否有播放器及列表存在
list_player_play()	播放指定列表
list_player_switch()	切换播放列表
list_player_play_index()	播放指定列表中指定索引曲目
list_player_play_item()	播放指定列表中指定handle曲目
list_player_stop()	停止播放
list_player_pause()	暂停播放
list_player_resume()	返回播放
list_player_prev()	播放上一曲
list_player_next()	播放下一曲
list_player_detach_items()	删除播放列表
list_player_set_mode()	设置播放器模式
list_player_items_create()	产生播放列表
list_player_items_delete()	删除播放列表
list_player_items_empty()	清空播放列表
list_player_set_table_handler()	播放列表完成时的回调函数
list_player_items_get_num()	播放列表中歌曲数目
list_player_items_get_index()	播放列表中当前的播放曲目索引
list_player_items_get_item()	播放列表中当前的播放曲目
list_player_item_add()	添加曲目到指定播放列表
list_player_item_del()	从指定播放列表删除曲目
list_player_item_del_by_index()	从指定播放列表删除指定索引曲目
list_player_item_get()	从指定播放列表获取指定索引曲目
list_player_index_get()	从指定播放列表获取指定曲目索引

15.2.1 初始化播放器

播放器使用前需要进行初始化：

```
int list_player_init(void)
```

参数	描述
void	空
返回	0：初始化成功，-1：初始化失败

15.2.2 获取当前播放曲目的handle

```
music_item_t list_player_current_item(void)
```

参数	描述
void	空
返回	当前播放曲目的handle，如没有播放返回-1

15.2.3 获取当前播放曲目在列表中的索引

```
int list_player_current_index(void)
```

参数	描述
void	空
返回	当前播放曲目的索引，如没有播放返回-1

15.2.4 获取当前播放器的状态

```
int list_player_current_state(void)
```

参数	描述
void	空
返回	当前播放器状态，如没有播放返回-1

15.2.5 获取当前播放曲目的播放位置

```
int list_player_current_position(void)
```

参数	描述
void	空
返回	当前播放位置，如没有播放返回-1

15.2.6 获取当前播放列表

```
music_list_t list_player_current_items (void)
```

参数	描述
void	空
返回	播放器的handle;0:无播放器或播放列表

15.2.7 查询是否有播放器及列表存在

```
int list_player_is_exist (void)
```

参数	描述
void	空
返回	1: 播放器及列表都存在, 0: 无播放器或列表

15.2.8 播放指定列表

```
int list_player_play (music_list_t table)
```

参数	描述
table	指定播放列表
返回	-1: 播放器不存在, -2: 列表不存在, 0: 成功

15.2.9 切换播放至指定列表

```
int list_player_switch(music_list_t table, int index, int position, int state)
```

参数	描述
table	指定播放列表
index	播放索引
position	播放位置
state	播放状态
返回	-1: 播放器不存在, -2: 列表不存在, 0: 成功

15.2.10 播放当前列表指定索引曲目

```
int list_player_play_index(int index)
```

参数	描述
Index	指定索引
返回	0: 成功, 1: 失败

15.2.11 播放指定曲目

```
int list_player_play_item(music_item_t item)
```

参数	描述
Item	指定曲目
返回	0: 成功, 1: 失败

15.2.12 播放器停止

```
void list_player_stop(void)
```

参数	描述
void	空
返回	空

15.2.13 暂停播放

```
void list_player_pause(void)
```

参数	描述
void	空
返回	空

15.2.14 播放恢复

```
void list_player_resume(void)
```

参数	描述
void	空
返回	空

15.2.15 播放上一曲

```
void list_player_prev(void)
```

参数	描述
void	空
返回	-1: 播放器不存在, -2: 列表不存在, 0: 成功

15.2.16 播放下一曲

```
void list_player_next(void)
```

参数	描述
void	空
返回	空

15.2.17 注销当前播放列表

```
music_list_t list_player_detach(void)
```

参数	描述
void	空
返回	被注销播放列表

15.2.18 注销并释放当前播放列表

```
int list_player_empty(void)
```

参数	描述
void	空
返回	-1: 播放器不存在, 0: 成功

15.2.19 设置播放器模式

```
int list_player_set_mode(int mode)
```

参数	描述
mode	播放器模式
返回	-1: 播放器不存在, 0: 成功

15.2.20 建立播放列表

```
music_list_t list_player_items_create(void)
```

参数	描述
void	空
返回	-1: 播放器不存在, -2: 列表不存在, 0: 成功

15.2.21 删除播放列表

```
void list_player_items_delete(music_list_t table)
```

参数	描述
table	待删除列表
返回	空

15.2.22 回调注册

注册回调函数，回调在列表播放结束时执行：

```
void list_player_set_table_handler(music_list_t table, list_event_handler handler, void *arg)
```

参数	描述
table	指定播放列表
handler	回调函数指针
arg	回调函数入参指针
返回	空

15.2.23 获取播放列表中曲目数

```
int list_player_items_get_num(music_list_t table)
```

参数	描述
table	指定播放列表
返回	-1：无效列表，其它：列表中曲目数

15.2.24 获取播放列表中最近播放曲目索引

```
int list_player_items_get_index(music_list_t table)
```

参数	描述
table	指定播放列表
返回	-1：无效列表，其它：最近播放曲目索引

15.2.25 获取播放列表中最近播放曲目

```
music_item_t list_player_items_get_item(music_list_t table)
```

参数	描述
table	指定播放列表
返回	NULL：无效列表或列表已注销，其它：播放曲目

15.2.26 曲目添加

```
int list_player_item_add(music_list_t table, music_item_t item, int index)
```

参数	描述
table	指定播放列表
item	待添加曲目
index	-1: 添加到列表尾部, 0: 添加到列表头部, 其它: 添加到该索引位置

15.2.27 曲目删除

```
int list_player_item_del(music_list_t table, struct music_item *item)
```

参数	描述
table	指定播放列表
item	待删除曲目
返回	-1: 列表或曲目无效, 0: 成功

15.2.28 指定索引曲目删除

```
int list_player_item_del_by_index(music_list_t table, int index)
```

参数	描述
table	指定播放列表
index	待删除曲目索引
返回	-1: 列表或索引无效或列表为空, 0: 成功

15.2.29 获取指定索引曲目

```
music_item_t list_player_item_get(music_list_t table, int index)
```

参数	描述
table	指定列表
index	指定索引
返回	NULL: 失败, 其它: 获取的曲目

15.2.30 获取指定曲目索引

```
int list_player_index_get(music_list_t table, music_item_t item)
```

参数	描述
table	指定列表
item	指定曲目
返回	-1: 失败, 其它: 获取的索引

15.3 List Player宏定义

设置播放器模式, 可选择模式如下:

#define LISTER_NONE_MODE	(0x00)	无模式
#define LISTER_LIST_ONCE_MODE	(0x01)	列表播放
#define LISTER_SONG_ONCE_MODE	(0x02)	单曲播放
#define LISTER_LIST_REPEAT_MODE	(0x03)	列表重复播放
#define LISTER_SONG_REPEAT_MODE	(0x04)	单曲循环

15.4 List Player示例代码

List Player示例代码参考\samples\Player\list_player_demo.c, 打开宏定义: LIST_PLAY_TEST, 开启list player的功能测试。

```

/*
 * 程序清单: simple Local Player for file playing
 * 命令形式: list_player http://192.168.44.23/Kiss_The_Rain.mp3
 */

#include <rtthread.h>
#include "player.h"
#include "list_player.h"
#include "player_app.h"
#include <finsh.h>

#include <stdio.h>
#include <stdlib.h>

/* 保存列表信息的结构体 */
typedef struct play_list_struct{
    music_list_t which_playlist;
    int play_list_status;
    int play_list_position;
    int play_list_num;
    music_item_t play_list_content;
    char backup_url[128];
}play_list_struct;

```

```

/* 保存当前播放列表信息 */
play_list_struct saved_list;
static void save_current_playlist_status(void)
{
    int state = list_player_current_state();
    int num = list_player_current_index();
    int postion =list_player_current_position();
    music_list_t tmp_list =list_player_current_items();
    int list_num = list_player_items_get_num(tmp_list);
    music_item_t tmp = list_player_current_item();

    saved_list.which_playlist = tmp_list;
    saved_list.play_list_status = state;
    saved_list.play_list_num = num;
    saved_list.play_list_position = postion;
    saved_list.play_list_content = tmp;
}

/* 恢复保存的播放列表并播放*/
static void bell_list_handle(void)
{
    list_player_switch(saved_list.which_playlist,
        saved_list.play_list_num,
        saved_list.play_list_position,
        saved_list.play_list_status);
}

/* 产生播放列表，添加歌曲并播放 */
music_list_t song_list = NULL;
int list_player(int argc, char** argv)
{
    struct music_item items = {0};
    items.name = ("Stream");
    items.URL = argv[1];
    /* 产生播放列表 */
    if (!song_list)
    {
        song_list =list_player_items_create();
    }
}

```

```

    /* 设置播放器模式 */
    list_player_mode_set(LISTER_LIST_ONCE_MODE);

    /* 添加歌曲 */
    list_player_item_add(song_list, &items,-1);

    /* 播放列表中歌曲 */
    list_player_play(song_list);
    rt_kprintf("list player test\r\n");
}

/* 产生提示音列表，打断当前歌曲播放，播放提示音，提示音结束返回歌曲播放 */
music_list_t bell_list = NULL;
int bell_player(int argc, char** argv)
{
    struct music_item items = {0};
    items.name = ("Bell");

    items.URL = argv[1];

    /* 保存当前播放列表*/
    save_current_playlist_status();

    /* 产生提示音列表 */
    if (!bell_list)
    {
        bell_list =list_player_items_create();
    }

    /* 设置播放器模式 */
    list_player_mode_set(LISTER_LIST_ONCE_MODE);

    /* 添加歌曲 */
    list_player_item_add(bell_list, &items,-1);

    /* 切换播放器至新列表并播放*/
    list_player_switch(bell_list,0,0,PLAYER_STAT_PLAYING);

    /* 恢复之前的播放*/
    list_player_set_table_handler(bell_list,bell_list_handle,NULL);
}

/* 播放列表命令 */
MSH_CMD_EXPORT(list_player, list_player test);

/* 播放提示音命令 */
MSH_CMD_EXPORT(bell_player, bell_player test);

```

16 网络接口

16.1 网络接口简介

BK7251的SDK给上层应用提供的网络接口用于：1.启动STATION模式，去连接指定的网络。2.关闭STATION模式。3.启动AP模式，供其他设备连接。4.关闭AP模式。5.启动监听模式，供上层配网。6.关闭监听模式。7.获取状态，如连接状态，加密方式，当前使用的信道等等。8.设置状态，如设置信道，IP地址等等。9.启动scan，并获取scan结果。

16.2 网络接口 Related API

网络接口相关接口参考**beken378\func\include\wlan_ui_pub.h**，应用程序可通过以下APIs控制网络，相关接口如下所示：

函数	描述
bk_wlan_start()	启动网络，包括STATION和AP
bk_wlan_start_sta_adv()	启动STATION快速连接
bk_wlan_stop()	关闭网络，包括STATION和AP
bk_wlan_start_scan()	启动scan
bk_wlan_scan_ap_reg_cb()	注册scan结束后的回调函数
bk_wlan_start_assign_scan()	scan特定的网络
bk_wlan_start_monitor()	启动监听模式
bk_wlan_stop_monitor()	关闭监听模式
bk_wlan_register_monitor_cb()	注册监听回调函数
bk_wlan_get_ip_status()	获取当前的网络状态
bk_wlan_get_link_status()	获取当前的连接状态
bk_wlan_get_channel()	获取当前的信道
bk_wlan_set_channel()	设置信道

16.2.1 启动网络

上层应该获得ssid与password之后，可以启动网络。通过如下函数完成：

```
OSStatus bk_wlan_start(network_InitTypeDef_st *inNetworkInitPara);
```

参数	描述
inNetworkInitPara	传入需要配置信息
返回	kNoErr: 成功；其他：失败

16.2.2 启动STATION快速连接

```
OSStatus bk_wlan_start_sta_adv(network_InitTypeDef_adv_st *inNetworkInitParaAdv);
```


参数	描述
inNetworkInitParaAdv	需要传入的网络参数
返回	kNoErr: 成功; 其他: 失败

16.2.3 关闭网络

```
int bk_wlan_stop(char mode);
```

参数	描述
mode	需要关闭的模式，见枚举类型中关于mode的说明
返回	kNoErr: 成功; 其他: 失败

16.2.4 启动scan

```
void bk_wlan_start_scan(void);
```

参数	描述
void	无
返回	无

16.2.5 注册scan结束后的回调函数

```
void bk_wlan_scan_ap_reg_cb(FUNC_2PARAM_PTR ind_cb);
```

参数	描述
ind_cb	scan结束后回调的函数。函数定义： typedef void (*FUNC_2PARAM_PTR)(void *arg, uint8_t vif_idx);
返回	无

16.2.6 scan特定的网络

```
void bk_wlan_start_assign_scan(UINT8 **ssid_ary, UINT8 ssid_num);
```

参数	描述
ssid_ary	指定网络的SSID
ssid_num	指定网络的数量
返回	无

16.2.7 启动监听模式

```
int bk_wlan_start_monitor(void);
```

参数	描述
void	无
返回	kNoErr: 成功; 其他: 失败

16.2.8 关闭监听模式

```
int bk_wlan_stop_monitor(void);
```

参数	描述
void	无
返回	kNoErr: 成功; 其他: 失败

16.2.9 注册监听回调函数

```
void bk_wlan_register_monitor_cb(monitor_data_cb_t fn);
```

参数	描述
fn	注册的回调函数。函数定义： typedef void (*monitor_data_cb_t)(uint8_t *data, int len, hal_wifi_link_info_t *info);
返回	无

16.2.10 获取当前的网络状态

```
OSStatus bk_wlan_get_ip_status(IPStatusTypeDef *outNetpara, WiFi_Interface inInterface);
```

参数	描述
outNetpara	保存获取的网络状态。具体见该结构体说明。
inInterface	需要获取网络状态的模式。参考模式的枚举类型说明。
返回	kNoErr: 成功; 其他: 失败

16.2.11 获取当前的连接状态

```
OSStatus bk_wlan_get_link_status(LinkStatusTypeDef *outStatus);
```

参数	描述
outStatus	保存获取的连接状态。具体参考该结构体的说明。
返回	kNoErr: 成功; 其他: 失败

16.2.12 获取当前的信道

```
int bk_wlan_get_channel(void);
```

参数	描述
void	无
返回	channel

16.2.13 设置信道

```
int bk_wlan_set_channel(int channel);
```

参数	描述
channel	传入的信道数值
返回	0：成功；其他：失败

16.3 网络接口结构体说明

network_InitTypeDef_st:

wifi_mode	WiFi模式
wifi_ssid	需要连接或建立的网络SSID
wifi_key	需要连接或建立的网络密码
local_ip_addr	静态IP地址，在DHCP关闭时有效
net_mask	静态子网掩码，在DHCP关闭时有效
gateway_ip_addr	静态网关地址，在DHCP关闭时有效
dns_server_ip_addr	静态DNS地址，在DHCP关闭时有效
dhcp_mode	DHCP模式

network_InitTypeDef_adv_st:

ap_info	需要快速连接的网络信息
key	需要快速连接的网络密码
key_len	网络密码长度
local_ip_addr	静态IP地址，在DHCP关闭时有效
net_mask	静态子网掩码，在DHCP关闭时有效
gateway_ip_addr	静态网关地址，在DHCP关闭时有效
dns_server_ip_addr	静态DNS地址，在DHCP关闭时有效
dhcp_mode	DHCP模式

apinfo_adv_t:

ssid	需要快速连接的网络SSID
bssid	需要快速连接的网络BSSID

channel	需要快速连接的网络信道
security	需要快速连接的网络加密方式

IPStatusTypedef:

dhcp	获取的DHCP模式
ip	获取的IP地址
gate	获取的网关IP地址
mask	获取的子网掩码
dns	DNS服务IP地址
mac	获取的mac地址
broadcastip	获取的广播IP地址

LinkStatusTypeDef:

conn_state	当前连接状态
wifi_strength	当前的信号强度
ssid	当前网络的SSID
bssid	当前网络的BSSID
channel	当前网络的信道
security	当前网络的加密方式

16.4 网络接口枚举类型说明

网络模式说明如下所示:

```
typedef enum
{
    SOFT_AP,                /*AP模式*/
    STATION                  /*STATION模式*/
} wlanInterfaceTypedef;
```

连接状态说明如下所示:

```
typedef enum {
    MSG_IDLE = 0,           /*未任何连接状态*/
    MSG_CONNECTING,         /*正在连接中*/
    MSG_PASSWD_WRONG,       /*密码错误*/
    MSG_NO_AP_FOUND,        /*未找到要连接的网络*/
    MSG_CONN_FAIL,          /*连接失败*/
    MSG_CONN_SUCCESS,       /*连接成功*/
    MSG_GOT_IP,             /*获得IP*/
}msg_sta_states;
```

加密方式如下说明

```
enum wlan_sec_type_e
{
    SECURITY_TYPE_NONE,          /*不加密.*/
    SECURITY_TYPE_WEP,           /*WEP加密方式.*/
    SECURITY_TYPE_WPA_TKIP,      /* WPA TKIP加密方式 */
    SECURITY_TYPE_WPA_AES,       /* WPA AES加密方式 */
    SECURITY_TYPE_WPA2_TKIP,     /* WPA2 TKIP加密方式 */
    SECURITY_TYPE_WPA2_AES,      /* WPA2 AES加密方式 */
    SECURITY_TYPE_WPA2_MIXED,    /* WPA2 AES TKIP兼容加密方式 */
    SECURITY_TYPE_AUTO,          /*在普通连接中，默认使用该加密方式.*/
};
```

16.5 网络接口宏定义说明

DHCP的模式:

#define DHCP_DISABLE	(0)	/*DHCP关闭*/
#define DHCP_CLIENT	(1)	/*DHCP客户端模式*/
#define DHCP_SERVER	(2)	/* DHCP服务端模式*/

16.6 网络接口使用示例

启动一个STATION连接:

```
void demo_sta_app_init(char *oob_ssid,char *connect_key)
{
    /*定义一个结构体，用于传入参数*/
    network_InitTypeDef_st wNetConfig;
    int len;
    /*把这个结构体置空*/
    os_memset(&wNetConfig, 0x0, sizeof(network_InitTypeDef_st));

    /*检查SSID的长度，不能超过32字节*/
    len = os_strlen(oob_ssid);
    if(SSID_MAX_LEN < len)
    {
        bk_printf("ssid name more than 32 Bytes\r\n");
        return;
    }

    /*将SSID跟密码传入结构体*/
    os_strcpy((char *)wNetConfig.wifi_ssid, oob_ssid);
```

```

os_strcpy((char *)wNetConfig.wifi_key, connect_key);

/*当前为客户端模式*/
wNetConfig.wifi_mode = STATION;
/*采用DHCP CLIENT的方式获得，从路由器动态获取IP地址*/
wNetConfig.dhcp_mode = DHCP_CLIENT;
wNetConfig.wifi_retry_interval = 100;

bk_printf("ssid:%s key:%s\r\n", wNetConfig.wifi_ssid, wNetConfig.wifi_key);
/*启动WiFi连接*/
bk_wlan_start(&wNetConfig);
}

```

启动AP模式，提供其他客户端连接：

```

void demo_softap_app_init(char *ap_ssid, char *ap_key)
{
    /*定义一个结构体，用于传入参数*/
    network_InitTypeDef_adv_st wNetConfigAdv;
    int len;
    /*将结构体置空*/
    os_memset( &wNetConfigAdv, 0x0, sizeof(network_InitTypeDef_adv_st) );
    len = os_strlen(ap_ssid);
    if(SSID_MAX_LEN < len)
    {
        bk_printf("ssid name more than 32 Bytes\r\n");
        return;
    }
    /*传入要连接的ap ssid 和 ap key*/
    os_strcpy((char *)wNetConfig.wifi_ssid, ap_ssid);
    os_strcpy((char *)wNetConfig.wifi_key, ap_key);

    /*当前为ap模式*/
    wNetConfig.wifi_mode = SOFT_AP;
    /*采用DHCP SERVER模式，需要将静态地址分配为本地地址*/
    wNetConfig.dhcp_mode = DHCP_SERVER;
    wNetConfig.wifi_retry_interval = 100;
    os_strcpy((char *)wNetConfig.local_ip_addr, WLAN_DEFAULT_IP);
    os_strcpy((char *)wNetConfig.net_mask, WLAN_DEFAULT_MASK);
    os_strcpy((char *)wNetConfig.dns_server_ip_addr, WLAN_DEFAULT_IP);
}

```

```
bk_printf("ssid:%s key:%s\r\n", wNetConfig.wifi_ssid, wNetConfig.wifi_key);  
/*启动ap*/  
bk_wlan_start(&wNetConfig);}
```

启动STATION的快速连接:

```
void demo_sta_adv_app_init(char *oob_ssid,char *connect_key)  
{  
    /*定义一个结构体，用于传入参数*/  
    network_InitTypeDef_adv_st wNetConfigAdv;  
    /*将结构体置空*/  
    os_memset( &wNetConfigAdv, 0x0, sizeof(network_InitTypeDef_adv_st) );  
    /*传入要连接的SSID*/  
    os_strcpy((char*)wNetConfigAdv.ap_info.ssid, oob_ssid);  
    /*传入要连接的网络的bssid，下面这个bssid仅供参考*/  
    hwaddr_aton("12:34:56:00:00:01", wNetConfigAdv.ap_info.bssid);  
    /*要连接网络的加密方式。具体参数参考该结构体说明。*/  
    wNetConfigAdv.ap_info.security = SECURITY_TYPE_WPA2_MIXED;  
    /*要连接的网络的信道*/  
    wNetConfigAdv.ap_info.channel = 11;  
    /*要连接的网络密码以及密码长度*/  
    os_strcpy((char*)wNetConfigAdv.key, connect_key);  
    wNetConfigAdv.key_len = os_strlen(connect_key);  
    /*通过DHCP的方式获取IP地址等网络信息*/  
    wNetConfigAdv.dhcp_mode = DHCP_CLIENT;  
    wNetConfigAdv.wifi_retry_interval = 100;  
    /*启动快速连接*/  
    bk_wlan_start_sta_adv(&wNetConfigAdv);  
}
```

启动scan，并分析scan的结果:

```
/*回调函数，用于scan结束后解析scan结果*/  
static void scan_cb(void *ctxt, uint8_t param)  
{  
    /*指向scan结果的指针*/  
    struct scanu_rst_upload *scan_rst;  
    /*保存解析结果的结构体*/  
    ScanResult apList;  
    int i;  
  
    apList.ApList = NULL;
```

```

/*启动scan*/
scan_rst = sr_get_scan_results();
/*如果什么都没有scan到，返回；否则记录scan到的网络数量*/
if( scan_rst == NULL )
{
    apList.ApNum = 0;
    return;
}
else
{
    apList.ApNum = scan_rst->scanu_num;
}
if( apList.ApNum > 0 )
{
    /*申请对应的内存，用于保存scan的结果*/
    apList.ApList = (void *)os_malloc(sizeof(*apList.ApList) * apList.ApNum);
    for( i = 0; i < scan_rst->scanu_num; i++ )
    {
        /*将scan到的网络ssid与rssi记录下来*/
        os_memcpy(apList.ApList[i].ssid, scan_rst->res[i]->ssid, 32);
        apList.ApList[i].ApPower = scan_rst->res[i]->level;
    }
}
if( apList.ApList == NULL )
{
    apList.ApNum = 0;
}
/*打印scan的结果*/
bk_printf("Got ap count: %d\r\n", apList.ApNum);
for( i = 0; i < apList.ApNum; i++ )
{
    if(os_strlen(apList.ApList[i].ssid) >= SSID_MAX_LEN)
    {
        char temp_ssid[33];
        os_memset(temp_ssid, 0, 33);
        os_memcpy(temp_ssid, apList.ApList[i].ssid, 32);
        bk_printf("    %s, RSSI=%d\r\n", temp_ssid, apList.ApList[i].ApPower);
    }
    else
    {

```



```

        bk_printf("    %s, RSSI=%d\r\n", apList.ApList[i].ssid, apList.ApList[i].ApPower);
    }
}
bk_printf("Get ap end.....\r\n\r\n");

/*结束后释放申请的内存*/
if( apList.ApList != NULL )
{
    os_free(apList.ApList);
    apList.ApList = NULL;
}

#if CFG_ROLE_LAUNCH
    rl_pre_sta_set_status(RL_STATUS_STA_LAUNCHED);
#endif

    sr_release_scan_results(scan_rst);
}

void demo_scan_app_init(void)
{
    /*注册scan回调函数*/
    mhdr_scanu_reg_cb(scan_cb, 0);
    /*开始scan*/
    bk_wlan_start_scan();
}

```

连接成功后，获取连接后的网络状态

```

void demo_ip_app_init(void)
{
    /*定义一个用于保存网络状态的结构体*/
    IPStatusTypedef ipStatus;
    /*将该结构体置空*/
    os_memset(&ipStatus, 0x0, sizeof(IPStatusTypedef));
    /*获取网络状态，并保存在该结构体中*/
    bk_wlan_get_ip_status(&ipStatus, STATION);
    /*打印获取的网络状态*/
    bk_printf("dhcp=%d ip=%s gate=%s mask=%s mac=" MACSTR "\r\n",
        ipStatus.dhcp, ipStatus.ip, ipStatus.gate,
        ipStatus.mask, MAC2STR((unsigned char*)ipStatus.mac));
}

```

连接成功后，获取连接状态：

```
void demo_state_app_init(void)
{
    /*定义结构体用于保存连接状态*/
    LinkStatusTypeDef linkStatus;
    network_InitTypeDef_ap_sta ap_info;
    char ssid[33] = {0};
    #if CFG_IEEE80211N
        bk_printf("sta: %d, softap: %d, b/g/n\n", sta_ip_is_start(), uap_ip_is_start());
    #else
        bk_printf("sta: %d, softap: %d, b/g\n", sta_ip_is_start(), uap_ip_is_start());
    #endif

    /*STATION模式下的连接状态*/
    if( sta_ip_is_start() )
    {
        /*将用于保存状态的结构体置空*/
        os_memset(&linkStatus, 0x0, sizeof(LinkStatusTypeDef));
        /*获取连接状态*/
        bk_wlan_get_link_status(&linkStatus);
        /*打印连接状态*/
        os_memcpy(ssid, linkStatus.ssid, 32);

        bk_printf("sta: rssi=%d, ssid=%s, bssid=" MACSTR ", channel=%d, cipher_type:",
            linkStatus.wifi_strength, ssid, MAC2STR(linkStatus.bssid), linkStatus.channel);
        switch(bk_sta_cipher_type())
        {
            case SECURITY_TYPE_NONE:
                bk_printf("OPEN\n");
                break;
            case SECURITY_TYPE_WEP :
                bk_printf("WEP\n");
                break;
            case SECURITY_TYPE_WPA_TKIP:
                bk_printf("TKIP\n");
                break;
            case SECURITY_TYPE_WPA2_AES:
                bk_printf("CCMP\n");
                break;
        }
    }
}
```

```

        case SECURITY_TYPE_WPA2_MIXED:
            bk_printf("MIXED\r\n");
            break;
        case SECURITY_TYPE_AUTO:
            bk_printf("AUTO\r\n");
            break;
        default:
            bk_printf("Error\r\n");
            break;
    }
}

/*AP模式下的连接状态*/
if( uap_ip_is_start() )
{
    /*将用于保存连接状态的结构体置空*/
    os_memset(&ap_info, 0x0, sizeof(network_InitTypeDef_ap_st));
    /*获取连接状态*/
    bk_wlan_ap_para_info_get(&ap_info);
    /*打印出获取的连接状态值*/
    os_memcpy(ssid, ap_info.wifi_ssid, 32);
    bk_printf("softap:ssid=%s,channel=%d,dhcp=%d,cipher_type:",
    ssid, ap_info.channel, ap_info.dhcp_mode);
    switch(ap_info.security)
    {
        case SECURITY_TYPE_NONE:
            bk_printf("OPEN\r\n");
            break;
        case SECURITY_TYPE_WEP :
            bk_printf("WEP\r\n");
            break;
        case SECURITY_TYPE_WPA_TKIP:
            bk_printf("TKIP\r\n");
            break;
        case SECURITY_TYPE_WPA2_AES:
            bk_printf("CCMP\r\n");
            break;
        case SECURITY_TYPE_WPA2_MIXED:
            bk_printf("MIXED\r\n");
            break;
        case SECURITY_TYPE_AUTO:

```

```

        bk_printf("AUTO\r\n");
        break;
    default:
        bk_printf("Error\r\n");
        break;
    }
    bk_printf("ip=%s,gate=%s,mask=%s,dns=%s\r\n",
        ap_info.local_ip_addr, ap_info.gateway_ip_addr, ap_info.net_mask,
        ap_info.dns_server_ip_addr);
    }
}

```

/* monitor 回调函数*/

```
void bk_demo_monitor_cb(uint8_t *data, int len, hal_wifi_link_info_t *info)
```

```

{
    os_printf("len:%d\r\n", len);

    //Only for reference
    /*
    User can get ssid and key by prase monitor data,
    refer to the following code, which is the way airkiss
    use monitor get wifi info from data
    */

```

```
#if 0
```

```

    int airkiss_rcv_ret;
    airkiss_rcv_ret = airkiss_rcv(ak_context, data, len);

```

```
#endif
```

```

}

```

/* 程序清单： 这是一个简单网络接口程序使用例程

* 命令调用格式： wifi_demo sta oob_ssid connect_key

* 程序功能： 输入相关命令可以启动网络，连接网络等。

*/

```
int wifi_demo(int argc, char **argv)
```

```

{
    char *oob_ssid = NULL;
    char *connect_key;

```

```

if (strcmp(argv[1], "sta") == 0)
{
    os_printf("sta_Command\r\n");
    if (argc == 3)
    {
        oob_ssid = argv[2];
        connect_key = "1";
    }
    else if (argc == 4)
    {
        oob_ssid = argv[2];
        connect_key = argv[3];
    }
    else
    {
        os_printf("parameter invalid\r\n");
        return -1;
    }
    if(oob_ssid)
    {
        demo_sta_app_init(oob_ssid, connect_key);
    }
    return 0;
}

if(strcmp(argv[1], "adv") == 0)
{
    os_printf("sta_adv_Command\r\n");
    if (argc == 3)
    {
        oob_ssid = argv[1];
        connect_key = "1";
    }
    else if (argc == 4)
    {
        oob_ssid = argv[1];
        connect_key = argv[2];
    }
    else
    {

```

```

        os_printf("parameter invalid\r\n");
        return -1;
    }
    if(oob_ssid)
    {
        demo_sta_adv_app_init(oob_ssid, connect_key);
    }
    return 0;
}

if(strcmp(argv[1], "softap") == 0)
{

    os_printf("SOFTAP_COMMAND\r\n\r\n");
    if (argc == 3)
    {
        oob_ssid = argv[1];
        connect_key = "1";
    }
    else if (argc == 4)
    {
        oob_ssid = argv[1];
        connect_key = argv[2];
    }
    else
    {
        os_printf("parameter invalid\r\n");
        return -1;
    }

    if(oob_ssid)
    {
        demo_softap_app_init(oob_ssid, connect_key);
    }
    return 0;
}

if(strcmp(argv[1], "monitor") == 0)
{
    if(argc != 3)
    {
        os_printf("parameter invalid\r\n");

```

```
    }  
    if(strcmp(argv[2], "start") == 0)  
    {  
        bk_wlan_register_monitor_cb(bk_demo_monitor_cb);  
        bk_wlan_start_monitor();  
    }  
    else if(strcmp(argv[2], "stop") == 0)  
    {  
        bk_wlan_stop_monitor();  
    }  
    else  
    {  
        os_printf("parameter invalid\r\n");  
    }  
}  
return 0;  
}  
MSH_CMD_EXPORT(wifi_demo, wifi_demo command);
```

17 RTOS接口

17.1 RTOS接口简介

RTOS接口提供RTOS的操作API，包括线程，互斥锁，时钟，信号量的操作。

17.2 RTOS Related APIs

RTOS相关接口参考beken378\rtos\include\bk_rtos_pub.h，应用程序可通过以下APIs控制RTOS，相关接口如下所示：

函数	描述
bk_rtos_create_thread()	创建一个新的线程
bk_rtos_delete_thread()	删除一个使用结束的线程
bk_rtos_thread_join()	使当前线程挂起，等待另一个线程终止
bk_rtos_thread_sleep()	使一个线程挂起一段时间，时间单位是：秒
bk_rtos_init_semaphore()	初始化一个信号量，并提供一个最大数
bk_rtos_set_semaphore()	发出信号量
bk_rtos_get_semaphore()	获取一个信号量，并提供超时机制
bk_rtos_deinit_semaphore()	销毁一个信号量
bk_rtos_init_mutex()	初始化一个互斥锁
bk_rtos_lock_mutex()	获得一个互斥锁
bk_rtos_unlock_mutex()	释放一个互斥锁
bk_rtos_deinit_mutex()	销毁一个互斥锁
bk_rtos_init_queue()	初始化一个消息队列
bk_rtos_push_to_queue()	将一个数据对象推入消息队列
bk_rtos_pop_from_queue()	从消息队列中取出一个数据对象
bk_rtos_deinit_queue()	销毁一个消息队列
bk_rtos_is_queue_empty()	查询一个队列是否为空
bk_rtos_is_queue_full()	查询一个队列是否已满
bk_rtos_init_timer()	初始化一个时钟，并传入回调函数
bk_rtos_start_timer()	启动一个时钟
bk_rtos_stop_timer()	停止一个时钟
bk_rtos_reload_timer()	重新加载一个过期的时钟
bk_rtos_deinit_timer()	销毁一个时钟
bk_rtos_is_timer_running()	获取一个时钟是否正在运行

17.2.1 创建一个新的线程

```
OSStatus bk_rtos_create_thread( beken_thread_t* thread,
                                uint8_t priority,
                                const char* name,
```



```
beken_thread_function_t function,  
uint32_t stack_size,  
beken_thread_arg_t arg );
```

参数	描述
thread	beken_thread_t类型的指针，指向创建的线程句柄
priority	优先级数值越小，优先级越高。
name	线程的名字
function	线程的入口函数
stack_size	为该线程分配的堆栈大小
arg	线程入口函数的参数
返回	kNoErr: 成功；其他：失败

17.2.2 删除一个使用结束的线程

```
OSStatus bk_rtos_delete_thread( beken_thread_t* thread );
```

参数	描述
thread	beken_thread_t类型的指针，指向需要删除的线程句柄
返回	kNoErr: 成功；其他：失败

17.2.3 使当前线程挂起，等待另一个线程终止

```
OSStatus bk_rtos_thread_join(beken_thread_t* thread);
```

参数	描述
thread	beken_thread_t类型的指针，指向需要等待的线程句柄
返回	kNoErr: 成功；其他：失败

17.2.4 使一个线程挂起一段时间

```
void bk_rtos_thread_sleep(uint32_t seconds);
```

参数	描述
seconds	线程挂起的时间，单位是秒。
返回	无

17.2.5 初始化一个信号量

```
OSStatus bk_rtos_init_semaphore( beken_semaphore_t* semaphore, int maxCount );
```

参数	描述
semaphore	初始化的信号量。
返回	kNoErr: 成功; 其他: 失败

17.2.6 发出信号量

```
int bk_rtos_set_semaphore( beken_semaphore_t* semaphore );
```

参数	描述
semaphore	需要发出的信号量。
返回	kNoErr: 成功; 其他: 失败

17.2.7 获取一个信号量，并提供超时机制

```
OSStatus bk_rtos_get_semaphore( beken_semaphore_t* semaphore, uint32_t timeout_ms );
```

参数	描述
semaphore	需要获取的信号量。
timeout_ms	超时时间。
返回	kNoErr: 成功; 其他: 失败

17.2.8 销毁一个信号量

```
OSStatus bk_rtos_deinit_semaphore( beken_semaphore_t* semaphore );
```

参数	描述
semaphore	需要销毁的信号量。
返回	kNoErr: 成功; 其他: 失败

17.2.9 初始化一个互斥锁

```
OSStatus bk_rtos_init_mutex( beken_mutex_t* mutex );
```

参数	描述
mutex	指向要初始化的互斥锁的句柄的指针。
返回	kNoErr: 成功; 其他: 失败

17.2.10 获得一个互斥锁

```
OSStatus bk_rtos_lock_mutex( beken_mutex_t* mutex );
```

参数	描述
----	----

mutex	指向要获取的互斥锁的句柄的指针。
返回	kNoErr: 成功; 其他: 失败

17.2.11 释放一个互斥锁

```
OSStatus bk_rtos_unlock_mutex( beken_mutex_t* mutex );
```

参数	描述
mutex	指向要释放的互斥锁的句柄的指针。
返回	kNoErr: 成功; 其他: 失败

17.2.12 销毁一个互斥锁

```
OSStatus bk_rtos_deinit_mutex( beken_mutex_t* mutex );
```

参数	描述
mutex	指向要销毁的互斥锁的句柄的指针。
返回	kNoErr: 成功; 其他: 失败

17.2.13 初始化一个消息队列

```
OSStatus bk_rtos_init_queue( beken_queue_t* queue,
                             const char* name,
                             uint32_t message_size,
                             uint32_t number_of_messages );
```

参数	描述
queue	指向要创建的消息队列的句柄的指针。
name	队列的名字
message_size	将要进入队列对象的最大字节数
number_of_messages	队列的深度, 即队列中对象的最大数量
返回	kNoErr: 成功; 其他: 失败

17.2.14 将一个数据对象推入消息队列

```
OSStatus bk_rtos_push_to_queue( beken_queue_t* queue,
                                void* message,
                                uint32_t timeout_ms );
```

参数	描述
queue	指向要推入数据对象的消息队列的句柄的指针。

message	推入队列的对象，对象大小在队列初始化 rtos_init_queue 中已指定。
timeout_ms	返回前等待的毫秒数。
返回	kNoErr: 成功；其他：失败

17.2.15 从消息队列中取出一个数据对象

```
OSStatus bk_rtos_pop_from_queue( beken_queue_t* queue,
                                void* message,
                                uint32_t timeout_ms );
```

参数	描述
queue	指向要取出数据对象的消息队列的句柄的指针。
message	要获取的数据对象，因此必须保证此缓存区足够大，否则将导致内存崩溃。
timeout_ms	返回前等待的毫秒数。
返回	kNoErr: 成功；其他：失败

17.2.16 销毁一个消息队列

```
OSStatus bk_rtos_deinit_queue( beken_queue_t* queue );
```

参数	描述
queue	指向要销毁的消息队列的句柄的指针。
返回	kNoErr: 成功；其他：失败

17.2.17 查询一个队列是否为空

```
BOOL bk_rtos_is_queue_empty( beken_queue_t* queue );
```

参数	描述
queue	指向要查询的消息队列的句柄的指针。
返回	1: 空；0: 非空

17.2.18 查询一个队列是否已满

```
BOOL bk_rtos_is_queue_full( beken_queue_t* queue );
```

参数	描述
queue	指向要查询的消息队列的句柄的指针。
返回	1: 满；0: 不满

17.2.19 初始化一个时钟，并传入回调函数

```
OSStatus bk_rtos_init_timer( beken_timer_t *timer,
                             uint32_t time_ms,
                             timer_handler_t function,
                             void* arg );
```

参数	描述
timer	指向要创建的时钟的句柄的指针。
time_ms	时钟，单位是毫秒。
function	时钟到期后执行的回调函数
arg	回调函数的参数
返回	kNoErr: 成功; 其他: 失败

17.2.20 启动一个时钟

```
OSStatus bk_rtos_start_timer( beken_timer_t* timer );
```

参数	描述
timer	指向要启动的时钟的句柄的指针。
返回	kNoErr: 成功; 其他: 失败

17.2.21 停止一个时钟

```
OSStatus bk_rtos_stop_timer( beken_timer_t* timer );
```

参数	描述
timer	指向要停止的时钟的句柄指针。
返回	kNoErr: 成功; 其他: 失败

17.2.22 重新加载一个过期的时钟

```
OSStatus bk_rtos_reload_timer( beken_timer_t* timer );
```

参数	描述
timer	指向要加载的时钟的句柄指针。
返回	kNoErr: 成功; 其他: 失败

17.2.23 销毁一个时钟

```
OSStatus bk_rtos_deinit_timer( beken_timer_t* timer );
```

参数	描述
timer	指向要销毁的时钟的句柄指针。
返回	kNoErr: 成功; 其他: 失败

17.2.24 获取一个时钟是否正在运行

```
BOOL bk_rtos_is_timer_running( beken_timer_t* timer );
```

参数	描述
timer	指向要查询的时钟的句柄指针。
返回	1: 运行; 其他: 停止

17.3 RTOS结构体说明

beken_timer_t:	
handle	rtos_init_timer创建的时钟句柄
function	时钟回调函数
arg	回调函数的参数

beken_worker_thread_t:	
thread	指向线程的指针
event_queue	线程的事件队列

beken_timed_event_t:	
function	事件句柄函数
arg	事件句柄函数参数
timer	时钟
thread	线程

beken2_timer_t:	
handle	指向时钟的句柄指针
function	时钟事件对应的回调函数, 该函数有两个参数
left_arg	回调函数的第一个参数
right_arg	回调函数的第二个参数
beken_magic	魔术数

17.4 RTOS枚举类型说明

等待事件说明如下所示:

```
typedef enum
{
    WAIT_FOR_ANY_EVENT,    /*任何事件可唤醒*/
    WAIT_FOR_ALL_EVENTS,   /*所有事件可唤醒*/
} beken_event_flags_wait_option_t;
```

17.5 RTOS宏定义说明

执行返回值:

#define RTOS_SUCCESS	(1)	/*执行成功*/
#define RTOS_FAILURE	(0)	/*执行失败*/

RTOS优先级配置:

#define BEKEN_DEFAULT_WORKER_PRIORITY	(6)	/*默认优先级为6*/
#define BEKEN_APPLICATION_PRIORITY	(7)	/*应用优先级为7*/

RTOS时间配置:

#define kNanosecondsPerSecond	1000000000UUL
#define kMicrosecondsPerSecond	1000000UL
#define kMillisecondsPerSecond	1000
#define NANOSECONDS	1000000UL
#define MICROSECONDS	1000
#define MILLISECONDS	(1)
#define SECONDS	(1000)
#define MINUTES	(60 * SECONDS)
#define HOURS	(60 * MINUTES)
#define DAYS	(24 * HOURS)

RTOS等待配置:

#define BEKEN_NEVER_TIMEOUT	(0xFFFFFFFF)
#define BEKEN_WAIT_FOREVER	(0xFFFFFFFF)
#define BEKEN_NO_WAIT	(0)

17.6 RTOS示例代码

启动线程示例代码参考bk7251_sdk\beken378\demo\os_demo.c。

```
#include <rtthread.h>

#include "include.h"

#include "bk_rtos_pub.h"

#include "uart_pub.h"
```

```

#include "Error.h"
#include "portmacro.h"

#define OS_THREAD_DEMO      1      //打开thread示例
#define OS_MUTEX_DEMO      1      //打开mutex示例
#define OS_SEM_DEMO        1      //打开semaphore示例
#define OS_QUEUE_DEMO      1      //打开queue示例
#define OS_TIMER_DEMO      1      //打开timer示例

/*线程1的子线程的主函数，该函数打印一句log，然后退出。*/
static void thread_0( beken_thread_arg_t arg )
{
    (void)( arg );

    os_printf( "This is thread 0\r\n");
    bk_rtos_delay_milliseconds((TickType_t)1000 );

    /* Make with terminate state and IDLE thread will clean resources */
    bk_rtos_delete_thread(NULL);
}

/*线程1的入口函数，该函数创建一个子线程，入口函数是thread_0，并等到子线程退出。*/
static void thread_1( beken_thread_arg_t arg )
{
    (void)( arg );
    OSStatus err = kNoErr;
    beken_thread_t t_handler = NULL;

    while ( 1 )
    {
        /* Create a new thread, and this thread will delete its self and clean its resource */
        err = bk_rtos_create_thread( &t_handler,
                                    BEKEN_APPLICATION_PRIORITY,
                                    "Thread 0",
                                    thread_0,
                                    0x400,
                                    0);

        if(err != kNoErr)
        {
            os_printf("ERROR: Unable to start the thread 1.\r\n" );
        }
    }
}

```



```

        /* wait thread 0 delete it's self */
        bk_rtos_thread_join( &t_handler );
    }
}

/*线程2的入口函数，打印一句log。*/
static void thread_2( beken_thread_arg_t arg )
{
    (void)( arg );

    while ( 1 )
    {
        os_printf( "This is thread 2\r\n" );
        bk_rtos_delay_milliseconds((TickType_t)600);
    }
}

/*该示例函数将建立两个线程*/
static int thread_demo_start( void )
{
    OSStatus err = kNoErr;
    /*定义两个线程的句柄*/
    beken_thread_t t_handler1 = NULL, t_handler2 = NULL;

    os_printf("\r\n\r\noperating system thread demo.....\r\n" );
    /*创建第一个线程，入口函数为thread_1，没有参数。*/
    err = bk_rtos_create_thread( &t_handler1, BEKEN_APPLICATION_PRIORITY,
                                "Thread 1",
                                thread_1,
                                0x400,
                                0);

    if(err != kNoErr)
    {
        os_printf("ERROR: Unable to start the thread 1.\r\n" );
        goto exit;
    }

    /*创建第二个线程。入口函数是thread_2，没有参数。*/
    err = bk_rtos_create_thread( &t_handler2, BEKEN_APPLICATION_PRIORITY,
                                "Thread 2",
                                thread_2,

```

```

                                0x400,
                                0);

if(err != kNoErr)
{
    os_printf("ERROR: Unable to start the thread 2.\r\n" );
    goto exit;
}

exit:
/*错误处理，出错后将对应的线程删除。*/
if ( err != kNoErr )
{
    os_printf( "Thread exit with err: %d", err );

    if(t_handler1 != NULL)
    {
        bk_rtos_delete_thread(t_handler1);
    }

    if(t_handler2 != NULL)
    {
        bk_rtos_delete_thread(t_handler2);
    }
}

return err;
}

```

使用信号量示例：启动两个线程，一个用于设置信号量，一个用于获取信号量。获取成功后打印一句log。

```

static beken_semaphore_t os_sem = NULL;    /*定义一个信号量。*/

/*设置信号量的入口函数，设置信号量后等待500毫秒。*/
static void set_semaphore_thread( beken_thread_arg_t arg )
{
    while ( 1 )
    {
        os_printf( "release semaphore!\r\n" );
        bk_rtos_set_semaphore( &os_sem );
        bk_rtos_delay_milliseconds( 500 );
    }
}

```

```

    }

exit:
    if(os_sem)
    {
        bk_rtos_deinit_semaphore(&os_sem);
    }
    bk_rtos_delete_thread( NULL );
}

/*获取信号量的线程入口函数，在获得信号量后打印log。*/
static void get_semaphore_thread( beken_thread_arg_t arg )
{
    OSStatus err;

    while(1)
    {
        err = bk_rtos_get_semaphore(&os_sem, BEKEN_NEVER_TIMEOUT);
        if(err == kNoErr)
        {
            os_printf("Get_Sem Succend!\r\n");
        }
        else
        {
            os_printf("Get_Sem Err:%d\r\n", err);
            goto exit;
        }
    }

exit:
    if(os_sem)
    {
        bk_rtos_deinit_semaphore(&os_sem);
    }
    bk_rtos_delete_thread( NULL );
}

/*使用信号量的示例函数入口*/
static int sem_demo_start ( void )
{
    OSStatus err = kNoErr;

```

```

beken_thread_t t_handler1 = NULL, t_handler2 = NULL;
os_printf( "test binary semaphore\r\n" );
/*初始化信号量os_sem*/
err = bk_rtos_init_semaphore( &os_sem, 1 ); //0/1 binary semaphore || 0/N semaphore
/*检查是否初始化成功*/
if(err != kNoErr)
{
    goto exit;
}
/*创建一个线程用于获取信号量*/
err = bk_rtos_create_thread(&t_handler1,
                            BEKEN_APPLICATION_PRIORITY,
                            "get_sem",
                            get_semaphore_thread,
                            0x500,
                            0 );

if(err != kNoErr)
{
    goto exit;
}
/*创建一个线程用于设置信号量*/
err = bk_rtos_create_thread(&t_handler2,
                            BEKEN_APPLICATION_PRIORITY,
                            "set_sem",
                            set_semaphore_thread,
                            0x500,
                            0 );

if(err != kNoErr)
{
    goto exit;
}

return err;
exit:
if ( err != kNoErr )
{
    os_printf( "Thread exit with err: %d\r\n", err );
}
return err;
}

```

使用互斥量示例，两个线程同时使用相同的入口函数，打印不同的字符串。
由于互斥量的使用，打印并不会乱：

```
static beken_mutex_t os_mutex = NULL; /*定义一个互斥量*/
/*该函数用于将传入的字符串打印出来*/
static OSStatus mutex_printf(char *s)
{
    OSStatus err = kNoErr;
    if(os_mutex == NULL)
    {
        return -1;
    }
    /*打印前申请互斥量*/
    err = bk_rtos_lock_mutex(&os_mutex);
    if(err != kNoErr)
    {
        return err;
    }
    os_printf( "%s\r\n", s);
    /*打印结束后释放互斥量*/
    err = bk_rtos_unlock_mutex(&os_mutex);
    if(err != kNoErr)
    {
        return err;
    }
    return err;
}

static void os_mutex_sender_thread( beken_thread_arg_t arg )
{
    OSStatus err = kNoErr;
    char *taskname = (char *)arg;
    char strprt[100];
    int rd;
    while ( 1 )
    {
        rd = rand() & 0x1FF;
        /*组成要打印的字符串，为传入的字符串加随机数组成。*/
        sprintf(strprt, "%s , Rand:%d", taskname, rd);
        /*打印组成的字符串。*/
    }
}
```



```

if(err != kNoErr)
{
    goto exit;
}
/*创建线程2，发送"I'm is task!"*/
err = bk_rtos_create_thread(&t_handler2,
                            BEKEN_APPLICATION_PRIORITY,
                            "sender2",
                            os_mutex_sender_thread,
                            0x400,
                            "I'm is task!" );

if(err != kNoErr)
{
    goto exit;
}
exit:
if ( err != kNoErr )
{
    os_printf( "Thread exit with err: %d\r\n", err );
}
return err;
}

```

RTOS队列使用示例，结果是将推入队列的数据取出并打印出来：

```

typedef struct _msg
{
    int value;
} msg_t;                                /*定义放入队列的数据对象类型。*/
static beken_queue_t os_queue = NULL; /*定义一个全局变量，用于指向队列的指针。*/

/*本函数将数据对象从队列中取出*/
static void receiver_thread( beken_thread_arg_t arg )
{
    OSStatus err;
    msg_t received = { 0 }; /*定义一个数据对象，用于保存从队列中取出的数据。*/

    while ( 1 )
    {
        /*一直等到队列中有数据，并将其取出。*/
        err = bk_rtos_pop_from_queue( &os_queue, &received, BEKEN_NEVER_TIMEOUT);
    }
}

```

```

    /*检查返回值，确认是否正确取出。*/
    if(err == kNoErr)
    {
        os_printf( "Received data from queue:value = %d\r\n", received.value );
    }
    else
    {
        os_printf("Received data from queue failed:Err = %d\r\n", err);
        goto exit;
    }
}

exit:
    if ( err != kNoErr )
        os_printf( "Receiver exit with err: %d\r\n", err );

    bk_rtos_delete_thread( NULL );
}
/*本函数将数据对象推入队列中*/
static void sender_thread( beken_thread_arg_t arg )
{
    OSStatus err = kNoErr;

    msg_t my_message = { 0 }; /*定义一个数据对象，用于推入队列中*/

    while ( 1 )
    {
        /*将数据对象赋值*/
        my_message.value++;
        /*将数据对象推入队列中*/
        err = bk_rtos_push_to_queue(&os_queue, &my_message, BEKEN_NEVER_TIMEOUT);
        /*检查返回值，确认是否成功推出队列。*/
        if(err == kNoErr)
        {
            os_printf( "send data to queue\r\n" );
        }
        else
        {
            os_printf("send data to queue failed:Err = %d\r\n", err);
        }
    }
}

```



```

        bk_rtos_delay_milliseconds( 100 );
    }

exit:
    if ( err != kNoErr )
    {
        os_printf( "Sender exit with err: %d\r\n", err );
    }

    bk_rtos_delete_thread( NULL );
}

/*rtos队列使用示例入口函数*/
static int queue_demo_start ( void )
{
    OSStatus err = kNoErr;
    /*初始化队列os_queue。*/
    beken_thread_t t_handler1 = NULL, t_handler2 = NULL;
    err = bk_rtos_init_queue( &os_queue, "queue", sizeof(msg_t), 3 );
    /*检查是否初始化成功*/
    if(err != kNoErr)
    {
        goto exit;
    }
    /*创建一个线程，用于将数据对象推入队列。*/
    err = bk_rtos_create_thread(&t_handler1,
                                BEKEN_APPLICATION_PRIORITY,
                                "sender",
                                sender_thread,
                                0x500,
                                0 );

    if(err != kNoErr)
    {
        goto exit;
    }
    /*创建一个线程，用于将数据对象从队列中取出*/
    err = bk_rtos_create_thread(&t_handler2,
                                BEKEN_APPLICATION_PRIORITY,
                                "receiver",
                                receiver_thread,
                                0x500,
                                0 );
}

```

```

                                0 );

    if(err != kNoErr)
    {
        goto exit;
    }

exit:
    if ( err != kNoErr )
    {
        os_printf( "Thread exit with err: %d\r\n", err );
    }
    return err;
}

```

RTOS时钟使用示例：

```

beken_timer_t timer_handle, timer_handle2; /*定义两个指向时钟对象的句柄指针*/
/*本函数停止并销毁时钟*/
static void destroy_timer( void )
{
    /* 停止时钟(timer_handle) */
    bk_rtos_stop_timer( &timer_handle );
    /* 销毁时钟(timer_handle) */
    bk_rtos_deinit_timer( &timer_handle );
    /* 停止时钟(timer_handle2) */
    bk_rtos_stop_timer( &timer_handle2 );
    /* 销毁时钟(timer_handle2) */
    bk_rtos_deinit_timer( &timer_handle2 );
}
/*时钟timer_handle的回调函数，超时后将执行。打印一句log。*/
static void timer_alarm( void *arg )
{
    os_printf("I'm timer_handle1\r\n");
}
/*时钟timer_handle2的超时回调函数，打印一句log，并销毁两个时钟。*/
static void timer2_alarm( void *arg )
{
    os_printf("I'm timer_handle2,destroy timer!\r\n");

    destroy_timer();
}

```

```

/*rtos时钟示例入口函数*/
static int timer_demo_start ( void )
{
    OSStatus err = kNoErr;

    os_printf("timer demo\r\n");

    /* 初始化时钟timer_handle，超时时间为500毫秒，超时回调函数为timer_alarm。*/
    err = bk_rtos_init_timer(&timer_handle, 500, timer_alarm, 0); ///500mS
    if(kNoErr != err)
        goto exit;

    /* 初始化时钟timer_handle2，超时时间为2600毫秒，超时回调函数为timer2_alarm。*/
    err = bk_rtos_init_timer(&timer_handle2, 2600, timer2_alarm, 0); ///2.6S
    if(kNoErr != err)
        goto exit;

    /* 启动时钟(timer_handle) */
    err = bk_rtos_start_timer(&timer_handle);
    if(kNoErr != err)
        goto exit;

    /* 启动时钟 (timer_handle2) */
    err = bk_rtos_start_timer(&timer_handle2);
    if(kNoErr != err)
        goto exit;

    return err;

exit:
    if( err != kNoErr )
        os_printf( "os timer exit with err: %d", err );
    return err;
}

/*程序清单：该程序实现了创建线程，信号量，互斥量，时钟以及队列
*程序命令： os_demo  thread/mutex/queue/semaphore/timer
*/
static int os_demo(int argc, char **argv)
{
    if(strcmp(argv[1], "thread") == 0)
    {
        #if OS_THREAD_DEMO

```

```

        thread_demo_start();
#endif
    }
    else if(strcmp(argv[1], "mutex") == 0)
    {
#ifdef OS_MUTEX_DEMO
        mutex_demo_start();
#endif
    }
    else if(strcmp(argv[1], "queue") == 0)
    {
#ifdef OS_QUEUE_DEMO
        queue_demo_start();
#endif
    }
    else if(strcmp(argv[1], "semaphore") == 0)
    {
#ifdef OS_SEM_DEMO
        sem_demo_start();
#endif
    }
    else if(strcmp(argv[1], "timer") == 0)
    {
#ifdef OS_TIMER_DEMO
        timer_demo_start();
#endif
    }
    else
    {
        os_printf("os demo %s doesn't support.\n", argv[1]);
    }
}
MSH_CMD_EXPORT(os_demo, os_demo command);

```

18 OTA

18.1 OTA简介

支持网络端远程升级固件，采用http协议从服务器下载ota固件，然后烧录到download分区中，设备重启后bootloader会将ota分区的固件拷贝到app运行分区，并加载新的app分区固件。ota固件支持压缩和加密，ota固件制作使用rt_ota_paging_tool工具。

18.2 OTA Related API

OTA相关接口参考\rt-thread\samples\ota\http\http_client_ota.c，应用程序可通过以下APIs使用OTA，相关接口如下所示：

函数	描述
<code>fal_init()</code>	fal初始化
<code>http_ota_fw_download()</code>	远程下载固件

18.2.1 fal初始化

初始化所有flash设备和分区，必须在http_ota_fw_download之前被调用，函数如下所示：

```
int fal_init(void);
```

参数	描述
<code>void</code>	无
返回	总分区数目

18.2.2 远程下载固件

从服务器下载ota固件，然后烧录到download分区中，调用之前应该初始化fal使用fal_init函数，函数如下所示：

```
int http_ota_fw_download(const char *url);
```

参数	描述
<code>url</code>	http服务器上的文件地址，完整的url
返回	0

18.3 OTA示例代码

OTA示例代码参考\samples\ota\http\http_client_ota.c，具体使用方式可以参考如下示例代码：

```

/*
 * 程序清单： 这是一个ota使用例程
 * 例程导出了http_ota 命令到控制终端
 * 命令调用格式： http_ota url
 * 程序功能： 通过ota下载远程固件到download分区
 */

void http_ota(uint8_t argc, char **argv)
{
    int parts_num;
    parts_num = fal_init();    //fal初始化

    if (parts_num <= 0)
    {
        log_e("Initialize failed! Don't found the partition table.");
        return;
    }
    if (argc < 2)
    {
        rt_kprintf("using url: " HTTP_OTA_URL "\n");
        http_ota_fw_download(HTTP_OTA_URL);    //固件下载
    }
    else
    {
        http_ota_fw_download(argv[1]);
    }
}
/**
 * msh />http_ota [url]
 */
MSH_CMD_EXPORT(http_ota, OTA by http client: http_ota [url]);

```

19 低功耗

19.1 低功耗简介

BK7251低功耗模式包括了MCU睡眠，RF睡眠以及Deep Sleep睡眠模式，Deep Sleep唤醒模式包括RTC唤醒和GPIO唤醒。

19.2 低功耗 Related API

低功耗相关接口参考**beken378\func\include\wlan_ui_pub.h** 和 **manual_ps_pub.h**，相关接口如下：

函数	描述
bk_wlan_enter_powersave()	低功耗模式
bk_enter_deep_sleep_mode()	deep_sleep模式

19.2.1 进入低功耗模式

进入低功耗模式的函数如下所示：

```
int bk_wlan_enter_powersave(struct rt_wlan_device *device, int level);
```

参数	描述
struct rt_wlan_device *device	wlan设备句柄
level	0: mcu,rf都不睡眠; 1: mcu睡眠, rf不睡眠; 2: mcu不睡眠, rf睡眠 3: mcu, rf都睡眠
返回	RT_EOK(0): 成功; 其他: 出错

19.2.2 deep_sleep 模式

进入deep_sleep 模式的函数如下所示：

```
void bk_enter_deep_sleep_mode(PS_DEEP_CTRL_PARAM *deep_param);
```

参数	描述
PS_DEEP_CTRL_PARAM *deep_param	进入deep_sleep之前的参数设置
返回	空

19.3 低功耗结构体说明

PS_DEEP_CTRL_PARAM:

PS_DEEP_WAKEUP_WAY deep_wkway	唤醒模式的枚举类型
UINT32 gpio_index_map	每个bit位对应gpio0-gpio31，0: 不被设置; 1: 相应的gpio可以在deep_sleep被唤醒。
UINT32 gpio_edge_map	每个bit位对应gpio0-gpio31唤醒模式, 0: 上升沿唤醒;

	1: 下降沿唤醒, 其中gpio1为uart rx, 必须设为1。
UINT32 gpio_last_index_map	低8位bit位对应gpio32-gpio39, 0: 不被设置; 1: 相应的gpio可以在deep_sleep被唤醒。
UINT32 gpio_last_edge_map	低8位bit位对应gpio32-gpio39唤醒模式, 0: 上升沿醒; 1: 下降沿唤醒。
UINT32 sleep_time	timer唤醒模式下的唤醒时间

19.4 低功耗枚举型说明

deep_sleep模式下支持3种唤醒模式:

```
typedef enum {
    PS_DEEP_WAKEUP_GPIO = 0,    /*GPIO唤醒模式
    PS_DEEP_WAKEUP_RTC = 1,      /*RTC timer唤醒模式
    PS_DEEP_WAKEUP_GPIO_RTC = 2, /*GPIO/0RTC timer的都可唤醒模式
} PS_DEEP_WAKEUP_WAY;
```

19.5 低功耗宏定义

在进入低功耗模式必须开启宏定义: CFG_USE_MCU_PS才能进入低功耗模式。

#define	CFG_USE_MCU_PS	使用MCU的低功耗模式
#define	CFG_USE_STA_PS	使用RF的低功耗模式

19.6 低功耗示例代码

mcu睡眠, rf睡眠示例代码参考\test\test_pm.c, deep sleep模式示例代码\test\deep_sleep.c, 打开宏定义: TEST_PM, 开启mcu,rf睡眠功能测试; 打开宏定义: TEST_DEEP_SLEEP, 开启deeo_sleep测试, 示例代码如下:

```
/*
 * 程序清单: 这是一个低功耗 和deep_sleep模式的函数
 * 命令格式: 输入命令: wifi ap, 再输入命令: wifi w0 join wifiname password 连接网络, 最后输入命令 pm_level level 进入低功耗模式。
 *           测试deep sleep 模式下, 输入命令: sleep_mode 1c 0 1c 0 10 deep_wkway
 *           进入deep_sleep 模式, deep_wkway选择唤醒模式, 可参考结构体类型说明。
 * 程序功能: 实现低功耗和deep_sleep功能
 */
#include "error.h"
#include "include.h"
#include "arm_arch.h"
#include "gpio_pub.h"
#include "uart_pub.h"
```



```

#include "music_msg_pub.h"
#include "manual_ps_pub.h"

#include "co_list.h"
#include "saradc_pub.h"
#include "temp_detect_pub.h"
#include "sys_rtos.h"
#include "rtos_pub.h"
#include "saradc_intf.h"
#include "pwm_pub.h"
#include "pwm.h"
#include <stdint.h>
#include <stdlib.h>
#include <finsh.h>

/* mcu睡眠和rf睡眠模式示例*/
static int pm_level(int argc, char **argv)
{
    uint32_t level;
    if(argc != 2)
    {
        rt_kprintf("input argc is err!\n");
        return -1;
    }
    level = atoi(argv[1]);
    if(level > 3)
    {
        rt_kprintf("nonsupport level %d\n", level);
        return -1;
    }

    {
        struct rt_wlan_device *sta_device = (struct rt_wlan_device
*)rt_device_find(WIFI_DEVICE_STA_NAME);
        if (NULL != sta_device)
        {
            bk_wlan_enter_powersave(sta_device, level);
        }
    }

    return 0;
}

```

```

}

static void enter_deep_sleep_test(int argc, char **argv[])
{
    rt_thread_sleep(200);
    PS_DEEP_CTRL_PARAM deep_sleep_param;
    deep_sleep_param.deep_wkway          = 0;
    deep_sleep_param.gpio_index_map      = atoi(argv[1]);
    deep_sleep_param.gpio_edge_map       = atoi(argv[2]);
    deep_sleep_param.gpio_last_index_map = atoi(argv[3]);
    deep_sleep_param.gpio_last_edge_map  = atoi(argv[4]);
    deep_sleep_param.sleep_time          = atoi(argv[5]);
    deep_sleep_param.deep_wkway          = atoi(argv[6]);
    if(argc == 7)
    {
        rt_kprintf("enter enter_deep_sleep: 0x%0X 0x%0X 0x%0X 0x%0X %d %d\r\n",
                    deep_sleep_param.gpio_index_map,
                    deep_sleep_param.gpio_edge_map,
                    deep_sleep_param.gpio_last_index_map,
                    deep_sleep_param.gpio_last_edge_map,
                    deep_sleep_param.sleep_time,
                    deep_sleep_param.deep_wkway);
        bk_enter_deep_sleep_mode(&deep_sleep_param);
    }
    else
    {
        rt_kprintf(" argc error \r\n");
    }
}

FINSH_FUNCTION_EXPORT_ALIAS(enter_deep_sleep_test, __cmd_sleep_mode, test sleep
mode);

```

20 Bootloader

20.1 Bootloader简介

bootloader分成两级，一级为L_boot，二级为UP_boot。一级boot提供uart下载功能，二级boot实现ota功能。在使用bootloader之前，需要先根据项目的情况确定分区，并把分区信息保存到原始的bootloader.bin中。

20.2 分区表的设置

20.2.1 Bootloader分区

flash_name为beken_onchip_crc，所以offset = 0x10000是逻辑地址，该分区在FLASH中实际的物理地址为也为0,分区实际大小len = (60K*34)/32 = 65280Byte;

20.2.2 App分区

该分区为应用代码。flash_name为beken_onchip_crc，所以offset = 0x10000是逻辑地址，该分区在FLASH中实际的物理地址为: (0x10000 * 34)/32 = 0x0011000, 分区实际大小len = (1152K*34)/32 = 1224 K;

20.2.3 Download分区

该分区为OTA时下载数据存放区。flash_name为beken_onchip，所以offset = 0x143000为该分区在FLASH中实际的物理地址，分区实际大小为748K。

除了以上3个分区之外，可以根据需求添加其它分区。另外，bootloader分区的起始地址和长度不能变，app分区的起始地址不能变，但长度可以变。其它分区的起始地址和长度都可以根据方案的实际情况进行修改。

以BK7251 SDK中提供的2M分区表信息为例（partition_audio_2M.json），对其格式的解释如下：

字段	描述
name	分区名称，固件中查找分区的依据，不能重复
flash_name	所在介质名称，通常为FLASH。常用beken_onchip_crc与beken_onchip。对于前者，其offset和len字段都以逻辑地址表示，对于后者则是以物理地址表示

offset	分区起始地址，十六进制表示
len	分区长度，十进制表示

20.3 L_boot

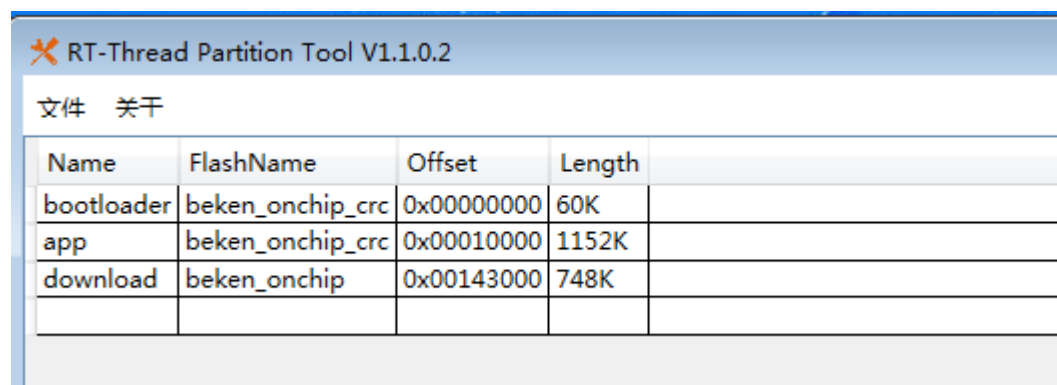
一级boot文件位于packages\boot\l_boot.bin,包含uart下载功能。一级boot应该被烧录到flash 0地址处，运行完成跳转到二级boot: CPU地址0x1F00处。

20.4 UP_boot

UP_boot必须从地址0x1F00处开始，UP_boot支持rttos的ota升级功能, ota升级功能会将download分区的rbl文件解密并解压到OS执行分区。二级boot运行完成后跳转到OS分区: CPU地址0x10000处。二级boot文件位于packages\boot\up_boot.bin。

20.5 获取bootloader.bin文件

打开rt_partition_tool软件，加载原始的bootloader.bin，然后导入分区表partition_audio_2M_sd.json，最后把分区表保存到bootloader.bin中，操作完成后该bootloader.bin即可和应用代码一起通过打包工具beken_packager生成最终的bin文件。



Name	FlashName	Offset	Length	
bootloader	beken_onchip_crc	0x00000000	60K	
app	beken_onchip_crc	0x00010000	1152K	
download	beken_onchip	0x00143000	748K	

20.6 生成all.bin文件

在得到有分区表的bootloader bin文件之后，就可以通过打包工具生成完整的bin文件。执行SDK目录\tool\beken_packager下的打包工具beken_packager.exe，即可生成完整的bin文件all_2M.1220.bin，以及串口升级所用的bin文件rtthread_uart_2M.1220.bin。

对于生成all.bin的config.json文件说明如下：

字段	描述
firmware	各分区打包输入的Bin文件
version	版本号
partition	分区名称，与bootloader.bin中对应分

	区表的名称相同
start_addr	分区起始地址，为物理地址，以十六进制表示，与分区表中对应分区的物理起始地址相同
size	分区实际大小，十进制表示，与分区表中对应分区的实际长度相同

20.7 Bootloader示例代码

20.7.1 2M分区表信息配置文件partition_audio_2M.json示例

```
{
  "part_table": [
    {
      "name": "bootloader",
      "flash_name": "beken_onchip_crc",
      "offset": "0x00000000",
      "len": "60K"
    },
    {
      "name": "app",
      "flash_name": "beken_onchip_crc",
      "offset": "0x00010000",
      "len": "1152K"
    },
    {
      "name": "download",
      "flash_name": "beken_onchip",
      "offset": "0x00143000",
      "len": "748K"
    }
  ]
}
```

20.7.2 UP_boot示例

```
/*
 * 程序清单： 这是一个二级boot使用例程
 * 程序功能： 程序实现了ota加密，解压拷贝分区等工作
 */
```

```

int ota_main(UINT32 * ex)
{
    int result = 0;
    size_t i, part_table_size;
    const struct fal_partition *dl_part = NULL;
    const struct fal_partition *part_table = NULL;
    const char *dest_part_name = NULL;

    if (rt_ota_init() >= 0)
    {
        /* verify bootloader partition
        * 1. Check if the BL partition exists
        * 2. CRC BL FW HDR
        * 3. HASH BL FW
        */
        if (rt_ota_part_fw_verify_header(fal_partition_find(RT_BK_BL_PART_NAME)) < 0)
        {
            //TODO upgrade bootloader to safe image
            // firmware HDR crc failed or hash failed. if boot verify failed, may not jump to app
            running
            #if !BOOT_OTA_DEBUG // close debug
                return -1;
            #endif
        }

        // 4. Check if the download partition exists
        dl_part = fal_partition_find(RT_BK_DL_PART_NAME);
        if (!dl_part)
        {
            log_e("download partition is not exist, please check your configuration!");
            return -1;
        }

        /* 5. Check if the target partition name is bootloader, skip ota upgrade if yes */
        dest_part_name = rt_ota_get_fw_dest_part_name(dl_part);
        if (dest_part_name && !strncmp(dest_part_name, RT_BK_BL_PART_NAME,
            strlen(RT_BK_BL_PART_NAME)))
        {
            log_e("Can not upgrade bootloader partition!");
            goto _app_check;
        }
    }
}

```

```

    }

    /* do upgrade when check upgrade OK
    * 5. CRC DL FW HDR
    * 6. Check if the dest partition exists
    * 7. CRC APP FW HDR
    * 8. Compare DL and APP HDR, containing fw version
    */
    log_d("check upgrade...");
    if ((result = rt_ota_check_upgrade()) == 1) // need to upgrade
    {
        if((rt_ota_get_fw_algo(dl_part) & RT_OTA_CRYPT_STAT_MASK) ==
RT_OTA_CRYPT_ALGO_NONE)
        {
            log_e("none encryption Not allow!");
            goto _app_check;
        }

        /* verify OTA download partition
        * 9. CRC DL FW HDR
        * 10. CRC DL FW
        */
        if (rt_ota_part_fw_verify(dl_part) == 0)
        {
            // 11. rt_ota_custom_verify
            // 12. upgrade
            set_flash_protect(NONE);
            if (rt_ota_upgrade() < 0)
            {
                log_e("OTA upgrade failed!");
                /*
                * upgrade failed, goto app check. If success, jump to app to run, otherwise
goto recovery factory firmware.
                */
                goto _app_check;
            }
            ota_erase_dl_rbl();
        }
        else
        {

```

```

        goto _app_check;
    }
}
else if (result == 0)
{
    log_d("No firmware upgrade!");
}
else if (result == -1)
{
    goto _app_check;
}
else
{
    log_e("OTA upgrade failed! Need to recovery factory firmware.");
    return -1;
}

```

_app_check:

```

part_table = fal_get_partition_table(&part_table_size);
/* verify all partition */
for (i = 0; i < part_table_size; i++)
{
    /* ignore bootloader partition and OTA download partition */
    if (!strncmp(part_table[i].name, RT_BK_APP_NAME, FAL_DEV_NAME_MAX))
    {
        // verify app firmware
        if (rt_ota_part_fw_verify_header(&part_table[i]) < 0)
        {
            // TODO upgrade to safe image
            log_e("App verify failed! Need to recovery factory firmware.");
            return -1;
        }
        else
        {
            *ex = part_table[i].offset;
            result = 0;
        }
    }
}
}

```



```
else
{
    result = -1;
}

return result;
}
```

20.7.3 生成all.bin的配置文件config_sample.json示例

```
{
  "magic": "RT-Thread",
  "version": "0.1",
  "count": 2,
  "section": [
    {
      "firmware": "bootloader.bin",
      "version": "2M.1220",
      "partition": "bootloader",
      "start_addr": "0x00000000",
      "size": "65280"
    },
    {
      "firmware": "../rtthread.bin",
      "version": "2M.1220",
      "partition": "app",
      "start_addr": "0x00011000",
      "size": "1224K"
    }
  ]
}
```

21 混音

21.1 混音简介

混音的功能是用BK7251芯片连接网络播放音乐,line in 接口接入音频作为背景音频,芯片可以同时播放两种音频数据,也可以消除line in的背景音频。

21.2 混音 Related API

mixer 相关接口参考\function\mixer.h,应用程序可通过以下APIs使用mixer功能,相关接口如下所示:

函数	描述
<code>mixer_init()</code>	混音初始化
<code>mixer_pause()</code>	暂停背景音乐,录音的时候必须暂停
<code>mixer_replay()</code>	重新播放背景音乐

21.2.1 混音初始化

混音模块初始化函数包括了audio延迟初始化, semaphore, mutex,mq的创建。

```
uint32_t mixer_init(void);
```

参数	描述
<code>void</code>	空
返回	1(MIXER_SUCCESS): 成功 0(MIXER_FAILURE) : 错误

21.2.2 暂停背景音播放

```
void mixer_pause(void);
```

参数	描述
<code>void</code>	无
返回	void

21.2.3 重新播放背景音

```
void mixer_replay(void);
```

参数	描述
<code>void</code>	无
返回	void

21.3 混音宏定义

#define	CONFIG_SOUND_MIXER	必须开启宏定义，进入混音模式
#define	MIXER_FAILURE	1: 返回失败
#define	MIXER_SUCCESS	0: 返回成功

21.4 混音示例代码

混音示例代码参考\samples\Mixer\mixer_demo.c，打开宏定义：
MIXER_DEMO，开启混音功能测试，具体使用方式可以参考如下示例代码：

```
/*
 * 程序清单： 这是一个混音使用例程，播放设备要同时播放两种音乐，一种音乐使用line in 接口接入
              其他设备播放的音乐，另一种音乐使用云端播放。
 * 命令调用格式： 配网成功之后播放云端的音乐，在输入命令： mixer_set_value 1 停止背景音乐的播
              放 命令mixer_set_value 0 播放背景音乐
 * 程序功能： 例程通过调用命令来控制背景音乐的播放与停止
 */
#include "rtconfig.h"
#if CONFIG_SOUND_MIXER
#include "mixer.h"
void mixer_set_value(int argc, char** argv)
{
    int val;
    val = atoi(argv[1]);
    if(val == 1) {
        rt_kprintf("mixer_set_value:%d pause\r\n", val);
        mixer_pause();
        /*暂停*/
    } else if(val == 0) {
        rt_kprintf("mixer_set_value:%d replay\r\n", val);
        mixer_replay();
        /*重新播放*/
    }
}
}
MSH_CMD_EXPORT(mixer_set_value, mixer_set_value test);
```

22 Vad

22.1 Vad自动语音检测简介

Vad功能是声音边界检测，检测声音的开始和结束。当芯片中有音频数据该功能就会检测到数据存在，并且打印检测到声音。

22.2 Vad Related API

vad相关接口参考\beken378\func\vad.h，相关接口如下：

函数	描述
wb_vad_enter()	进入vad检测模式
wb_vad_get_frame_len()	获取帧的长度
wb_vad_entry()	vad入口函数
wb_vad_deinit()	关闭vad模块

22.2.1 进入vad检测模式

vad检测模式包括vad初始化，buffer长度设置 。

```
int wb_vad_enter(void);
```

参数	描述
void	空
返回	0: 成功; 其他 : 错误

22.2.2 获取帧的长度

```
int wb_vad_get_frame_len(void);
```

参数	描述
void	空
返回	WB_FRAME_LEN : 帧的长度

22.2.3 vad入口函数

进入vad检测模式，函数如下：

```
int wb_vad_entry(char *buffer, int len);
```

参数	描述
buffer	测试buffer
len	测试buffer长度
返回	vad_flag

22.2.4 关闭vad

```
void wb_vad_deinit(void);
```

参数	描述
void	空
返回	空

22.3 Vad示例代码

Vad示例代码参考\test\mic_record.c，具体使用方式可以参考如下示例代码：

```
/*
 * 程序清单： 这是一个vad使用例程
 * 命令调用格式： record_and_play 1
 * 程序功能： 例程通过录音和播放功能验证vad的准确性
 */
#include <rtthread.h>
#include <rtdevice.h>
#include <finsh.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "board.h"
#include "audio_device.h"

#define MICPHONE_TEST
#ifdef MICPHONE_TEST

#define TEST_BUFF_LEN 60*1024
#define READ_SIZE 1024

static uint8_t *test_buf;

void record_and_play(int argc,char *argv[])
{
    int mic_read_len = 0;
    int actual_len,i;
    int dac_wr_len=0;
    uint16_t *buffer = NULL;
```

```

int vad_on;

#if CONFIG_SOUND_MIXER
    mixer_pause();
#endif

vad_on = atoi(argv[1]);

test_buf = sdram_malloc(TEST_BUFF_LEN);
if(test_buf == NULL)
{
    rt_kprintf("===not enough memory===\r\n");
    return;
}

audio_device_init();                                /*初始化 sound mic设备*/

audio_device_mic_open();                             /*打开mic设备*/
audio_device_mic_set_channel(1);                     /*设置adc通道*/
audio_device_mic_set_rate(16000);                   /*设置adc采样率*/

if (vad_on)
{
    rt_kprintf("Vad is ON !!!!!!!\r\n"); /*进入vad检测*/
    wb_vad_enter();
}

while(1)
{
    if (vad_on)
        rt_thread_delay(5);
    else
        rt_thread_delay(20);

    int chunk_size = wb_vad_get_frame_len();//320
    char *val = NULL;

    if(mic_read_len > TEST_BUFF_LEN - READ_SIZE)
        break;
}

```

```

if (!vad_on)
{
    actual_len = audio_device_mic_read(test_buf+mic_read_len,READ_SIZE);
}
else
{
    /*mic 采集声音数据*/
    actual_len = audio_device_mic_read(test_buf+mic_read_len,chunk_size);
    if(wb_vad_entry(test_buf+mic_read_len, actual_len))
    {
        rt_kprintf("Vad Detected !!!!!!!\r\n");          /*检测到声音*/
        break;
    }
}

mic_read_len += actual_len;
}

if (vad_on)
{
    wb_vad_deinit();          /*关闭vad检测*/
}

rt_kprintf("mic_read_len is %d\r\n", mic_read_len);
audio_device_mic_close();          /*关闭mic设备*/

audio_device_open();          /*打开dac设备*/
audio_device_set_rate(8000);          /*设置dac采样率*/

while(1)
{
    buffer = (uint16_t *)audio_device_get_buffer(RT_NULL);
    if(dac_wr_len >= mic_read_len)
    {
        audio_device_put_buffer(buffer);
        break;
    }

    memcpy(buffer,test_buf+dac_wr_len,READ_SIZE);
    dac_wr_len += READ_SIZE;
}

```

```
        audio_device_write((uint8_t *)buffer, READ_SIZE); /*dac播放数据*/
    }
    audio_device_close(); /*关闭dac设备*/

    if(test_buf)
        sdram_free(test_buf); /*释放ram内存*/

#if CONFIG_SOUND_MIXER
    mixer_replay();
#endif
}
MSH_CMD_EXPORT(record_and_play, record play);
#endif
```


23 AMR编码器

23.1 AMR编码器简介

AMR编码将接收到的语音信息编码成AMR格式的音频文件,其中编解码器所有的原文件被打包成库。

23.2 AMR编码器 Related API

AMR编码器APIs参考\components\codec\lib_amr_encode\amrnb_encode.h, 相关接口如下:

函数	描述
<code>amrnb_encoder_init()</code>	amr编码初始化
<code>amrnb_encoder_encode()</code>	amr编码
<code>amrnb_encoder_deinit()</code>	退出amr编码

23.2.1 AMR-NB编码器初始化

```
int32_t amrnb_encoder_init(void** amrnb, uint32_t dtx, void* pmalloc, void* pfree);
```

参数	描述
<code>amrnb</code>	AMR-NB编码器的指针
<code>dtx</code>	0: 连续传输数据 1: 不连续传输数据
<code>pmalloc</code>	malloc函数指针
<code>pfree</code>	free函数指针
返回	RT_EOK: 成功; 其他: 失败

23.2.2 AMR-NB编码

```
int32_t amrnb_encoder_encode(void* amrnb, uint32_t mode, const int16_t in[AMRNB_ENCODER_SAMPLES_PER_FRAME], uint8_t out[AMRNB_ENCODER_MAX_FRAME_SIZE])
```

参数	描述
<code>amrnb</code>	AMR-NB编码器的指针
<code>mode</code>	amr编码模式
<code>in</code>	输入的语音
<code>out</code>	输出的语音
返回	>0:read_byte:读取的字节数; 其他: 错误

23.2.3 释放AMR-NB编码

```
int32_t amrnb_encoder_deinit(void** amrnb);
```

参数	描述
amrnb	AMR-NB编码器的指针
返回	RT_EOK: 成功; 其他: 失败

24.3 AMR编码器宏定义

定义AMR编码器每帧中数据的大小

```
#define AMRNB_ENCODER_SAMPLES_PER_FRAME (160)
```

定义AMR编码器最大帧的大小

```
#define AMRNB_ENCODER_MAX_FRAME_SIZE (32)
```

23.4 AMR编码器枚举类型说明

AMR编码速率枚举类型:

```
enum Mode {
    AMRNB_MODE_MR475 = 0, /* 4.75 kbps */
    AMRNB_MODE_MR515,     /* 5.15 kbps */
    AMRNB_MODE_MR59,      /* 5.90 kbps */
    AMRNB_MODE_MR67,      /* 6.70 kbps */
    AMRNB_MODE_MR74,      /* 7.40 kbps */
    AMRNB_MODE_MR795,     /* 7.95 kbps */
    AMRNB_MODE_MR102,     /* 10.2 kbps */
    AMRNB_MODE_MR122,     /* 12.2 kbps */
    AMRNB_MODE_MRDTX,     /* DTX */
    AMRNB_MODE_N_MODES    /* Not Used */
};
```

23.5 AMR编码器示例代码

AMR编码器示例代码参考\test\record_amr_tcp.c, 打开宏定义:

RECORD_AMR_TCP_TEST, 开启amr编码功能测试, 具体使用方式可以参考如下示例代码:

```
/*
 * 程序清单: 这是一个amr编码器例程
 * 命令调用格式: 配网成功之后, 使用网络串口调试助手接收网络端发过来的编码数据 (注意必须使用同一个网络), 输入命令: record_amr_tcp start 采样率 网络地址 网络端口号 后开始录音并且生成amr格式的数据流。
```

停止命令: record_amr_tcp stop

* 程序功能: 例程通过调用命令将录制的音频信号转化成amr格式码流。

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

#include <drivers/audio.h>
#include <rtthread.h>
#include <sys/socket.h> /* 使用BSD socket, 需要包含sockets.h头文件 */
#include "netdb.h"
#include "record_common.h"
#include <interf_enc.h>
#include "amrnb_encoder.h"

#define RECORD_SAVE_BUF_SIZE (60 * 20 * 5)

struct record_manager
{
    struct net_worker *net;
    rt_mq_t msg;

    int action;
    struct rt_mempool mp;
    int sample_rate;
    int mp_block_size; /* sample / 50 * 2 ==> 8k:320 16k:640*/
    int mp_cnt;

    char *save_buf;
    int save_len;

    rt_sem_t ack;
};

static struct record_manager *recorder;
static short in_short[320*2]; /* NB:8K 160, WB:16K 320 */
```

```

#define AMR_MAGIC_NUMBER        "#!AMR\n"

static int record_msg_send(struct record_manager *record, void *buffer, int type, int len)
{
    int ret = RT_EOK;
    struct record_msg msg;

    msg.type = type;
    msg.arg = (uint32_t)buffer;
    msg.len = len;

    ret = rt_mq_send(record->msg, (void *)&msg, sizeof(struct record_msg));
    if (ret != RT_EOK)
        rt_kprintf("[record]:send msg failed \n");
}

static void record_thread_entry(void *parameter)                //录音以及编码功能线程的入口函数
{
    rt_device_t device = RT_NULL;
    int ret = RT_ERROR;
    uint8_t *mempool;
    rt_uint8_t *buffer;
    rt_uint32_t read_bytes = 0;
    void*amr = NULL;

    int    amr_enc_dtx = 0, frame_size, tmp;
    enum Mode amr_enc_mode = MR122;

    /* initialize mempool */
    recorder->mp_block_size = 120;
    recorder->mp_cnt = 10;
    mempool = rt_malloc(recorder->mp_block_size * recorder->mp_cnt);
    rt_mp_init(&(recorder->mp), "record_mp", mempool, recorder->mp_block_size *
recorder->mp_cnt, recorder->mp_block_size);

    /* initialize msg queue */
    recorder->msg = rt_mq_create("net_msg", sizeof(struct record_msg), 12, RT_IPC_FLAG_FIFO);

    /* initialize tcp client */
    ret = tcp_client_init(recorder->net);

```

```

if (ret != RT_EOK)
{
    return;
}

device = rt_device_find("mic");
if (!device)
{
    rt_kprintf("sound device not found \n");
    return;
}

rt_device_open(device, RT_DEVICE_OFLAG_RDONLY);

/* set samplerate */
{
    int rate = recorder->sample_rate;
    rt_device_control(device, CODEC_CMD_SAMPLERATE, (void *)&rate);
}

{
    /* Initial Encoder */
    ret = amrnb_encoder_init(&amr, amr_enc_dtx, rt_malloc, rt_free);
    if (ret != RT_EOK)
    {
        rt_kprintf("Encoder_Interface_init failed ==== %d", ret);
        return 0;
    }
    frame_size = recorder->sample_rate / 50;
    rt_kprintf("frame_size = %d \n", frame_size);
}

rt_kprintf("[record]:start record, tick %d \n", rt_tick_get());
/* write amr head */
buffer = rt_mp_alloc(&(recorder->mp), RT_WAITING_NO);
memcpy(buffer, AMR_MAGIC_NUMBER, strlen(AMR_MAGIC_NUMBER));
record_msg_send(recorder, buffer, RECORD_MSG_DATA, strlen(AMR_MAGIC_NUMBER));

while (1)
{

```

```

buffer = rt_mp_alloc(&(recorder->mp), RT_WAITING_NO);
if(!buffer)
{
    rt_kprintf("[record]: malloc memory for mempool failed \n");
    rt_thread_mdelay(20);
}
else
{
    /* read data from sound device */
    read_bytes = rt_sound_read(device, 0, in_short, frame_size * 2);
    /*encode ....*/
    {
        tmp = amrnb_encoder_encode(amr, amr_enc_mode, in_short, buffer);
    }
    record_msg_send(recorder, buffer, RECORD_MSG_DATA, tmp);
}

/* send stop cmd */
if (recorder->action == 0)
{
    int cmd;

    cmd = 0;
    record_msg_send(recorder, 0, RECORD_MSG_CMD, 1);
    /* wait ack */
    rt_kprintf("[record]:stop record, tick = %d \n", rt_tick_get());
    break;
}
}
rt_device_close(device);
rt_mp_detach(&(recorder->mp));
rt_free(mempool);
rt_mq_delete(recorder->msg);
{
    amrnb_encoder_deinit(&amr);
}
rt_kprintf("[record]:exit record thread, tick = %d \n", rt_tick_get());
}

```

static void net_transmit_thread_entry(void *parameter) //网络传输编码后的amr格式码流的入口函数

```

{
    int ret, cmd;
    struct record_msg msg;

    rt_thread_mdelay(100);
    recorder->save_len = 0;
    while(1)
    {
        if (rt_mq_rcv(recorder->msg, &msg, sizeof(struct record_msg), RT_WAITING_FOREVER)
== RT_EOK)
        {
            if(msg.type == RECORD_MSG_DATA)
            {
                memcpy(recorder->save_buf + recorder->save_len, (void *)msg.arg, msg.len);
                recorder->save_len += msg.len;
                rt_mp_free((void *)msg.arg);

                if(recorder->save_len >= RECORD_SAVE_BUF_SIZE - recorder->mp_block_size)
                {
                    /*send data*/
                    send(recorder->net->sock, recorder->save_buf, recorder->save_len, 0);
                    recorder->save_len = 0;
                }
            }
            else if(msg.type == RECORD_MSG_CMD)
            {
                cmd = *(int *)msg.arg;
                if(cmd == 0)
                {
                    /* send remain data, and send ack */
                }
            }
        }
    }
}

static int record_amr_tcp(int argc, char **argv)
{
    rt_thread_t tid = RT_NULL;
    int result;

```

```

if(recorder == RT_NULL)
{
    recorder = rt_malloc(sizeof(struct record_manager));
    if(!recorder)
    {
        rt_kprintf("[record]:malloc memory for recorder manager \n");
        return -RT_ERROR;
    }
    memset(recorder, 0, sizeof(struct record_manager));

    {
        struct net_worker *net = RT_NULL;
        net = rt_malloc(sizeof(struct net_worker));
        if(!net)
        {
            rt_kprintf("[record]:malloc memory for net worker \n");
            return -RT_ERROR;
        }
        memset(net, 0, sizeof(struct net_worker));
        recorder->net = net;

        recorder->save_buf = rt_malloc(RECORD_SAVE_BUF_SIZE);
        memset(recorder->save_buf, 0, RECORD_SAVE_BUF_SIZE);
    }

    rt_kprintf("L%d, recorder_create done \n", __LINE__);
}

rt_kprintf("L%d, record enter \n", __LINE__);
if (strcmp(argv[1], "stop") == 0)
{
    recorder->action = 0; //停止mic录音及编码
}
else if (strcmp(argv[1], "start") == 0)
{
    /* record start format samplerate url port */
    recorder->action = 1; //开始mic录音同时开始编码模式

    if(recorder->net->url)
    {

```



```

        rt_free(recorder->net->url);
        recorder->net->url = RT_NULL;
    }
    recorder->sample_rate = atoi(argv[2]);           //设置录音mic采样率
    recorder->net->url = rt_strdup(argv[3]);          //设置url地址
    recorder->net->port = atoi(argv[4]);              //设置网络连接的端口

    rt_kprintf("[record]:samplerate = %d \n", recorder->sample_rate);
    rt_kprintf("[record]:url = %s \n", recorder->net->url);
    rt_kprintf("[record]:port = %d \n", recorder->net->port);
    /* 创建一个录音的线程 */
    tid = rt_thread_create("record",
                            record_thread_entry,
                            RT_NULL,
                            1024 * 32,
                            27,
                            10);

    if (tid != RT_NULL)
        rt_thread_startup(tid);

    /* create net send thread */
    tid = rt_thread_create("net_send",
                            net_transmit_thread_entry,
                            RT_NULL,
                            1024 * 8,
                            25,
                            10);

    if (tid != RT_NULL)
        rt_thread_startup(tid);
}
else
{
    // print_record_usage();
}
}
FINSH_FUNCTION_EXPORT_ALIAS(record_amr_tcp, __cmd_record_amr_tcp, record_amr_tcp);

```

24 Opus编码器

24.1 Opus编码器简介

Opus编码将接收到的语音信息编码成opus格式的音频文件,其中编解码器所有的原文件被打包成库。

24.2 Opus编码器 Related API

opus编码器相关接口参考\components\codec\lib_opus\include\opus.h，相关接口如下：

函数	描述
<code>opus_encoder_create()</code>	创建opus编码
<code>opus_encoder_get_size()</code>	返回编码器所需内存的大小
<code>opus_encoder_set_complexity()</code>	修改编码器复杂度
<code>opus_encoder_get_bitrate()</code>	获取编码器的比特率
<code>opus_encoder_get_final_range()</code>	获取编码器最终状态
<code>opus_encode()</code>	opus编码
<code>opus_encoder_destroy()</code>	释放编码器对象

24.2.1 创建opus编码器

```
OpusEncoder *opus_encoder_create(opus_int32 Fs, int channels, int application, int *error);
```

参数	描述
Fs	输入信号的采样率（包括8k，16k）
channels	编码通道，只能为1或2
application	编码模式，由宏定义的3种编码模式
error	错误类型
返回	编码器对象的结构体

24.2.2 返回opus编码器所需内存的大小

```
int opus_encoder_get_size(int channels);
```

参数	描述
channels	通道必须为1或者2
返回	字节数

24.2.3 修改opus编码器的复杂度

```
opus_encoder_set_complexity(opus_enc, complexity);
```

参数	描述
opus_enc	opus编码器的结构体
complexity	编码器复杂度：0-10；
返回	编码器对象的结构体

24.2.4 获取opus编码器的比特率

```
opus_encoder_get_bitrate(opus_enc, bitrate_bps);
```

参数	描述
opus_enc	opus编码器的结构体
bitrate_bps	编码器的比特率
返回	编码器对象的结构体

24.2.5 获取opus编码器的最终状态

```
opus_encoder_get_final_range(opus_enc, enc_final_range);
```

参数	描述
opus_enc	opus编码器的结构体
enc_final_range	编码器最终的熵
返回	编码器对象的结构体

24.2.6 opus编码

```
opus_int32 opus_encode (OpusEncoder *st,  
                        const opus_int16 *pcm,  
                        int frame_size,  
                        unsigned char *data,  
                        opus_int32 max_data_bytes);
```

参数	描述
st	编码器对象
pcm	输入信号
size	输入音频信号每个声道的采样数量
data	输出编码结果
max_data_bytes	为输出编码结果分配内存

返回	编码长度：成功； 负数：失败
----	----------------

24.2.7 释放opus编码器对象

```
void opus_encoder_destroy(OpusEncoder *st);
```

参数	描述
st	编码器对象
返回	空

24.3 Opus编码器宏定义

三种opus编码模式的宏定义如下：

1.在给定的比特率条件下为声音信号提供最高质量，一般情况此种模式。

#define	OPUS_APPLICATION_VOIP	2048
---------	-----------------------	------

2.对大多数非语音信号在给定的比特率条件下提供最高的质量。

#define	OPUS_APPLICATION_AUDIO	2049
---------	------------------------	------

3.配置低延迟模式将为减少延迟禁用语音优化模式。

#define	OPUS_APPLICATION_RESTRICTED_LOWDELAY	2051
---------	--------------------------------------	------

24.4 Opus编码器示例代码

Opus编码器示例代码参考\test\record_opus_tcp.c，打开宏定义：RECORD_OPUS_TCP_TEST，开启opus编码功能测试，具体使用方式可以参考如下示例代码：

```
/* 程序清单： 这是一个opus编码器例程
* 命令调用格式： 配网成功之后，使用网络串口调试助手接收网络端发过来的编码数据，输入命令：
record_opus_tcp start 采样率 网络地址 网络端口号 生成opus格式的码流，修改文件名称变为
opus文件，使用工具转换成pcm文件，通过cool edit pro 播放生成的pcm格式文件。
停止命令： record_opus_tcp stop
* 程序功能： 例程通过调用命令将音频信号转化成opus格式。
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

```

#include <drivers/audio.h>

#include <rtthread.h>
#include <sys/socket.h> /* 使用BSD socket, 需要包含sockets.h头文件 */
#include "netdb.h"
#include "record_common.h"
#include <opus.h>
#define RECORD_SAVE_BUF_SIZE (60 * 20 * 5)

struct record_manager
{
    struct net_worker *net;
    rt_mq_t msg;

    int action;
    struct rt_mempool mp;
    int sample_rate;
    int mp_block_size; /* sample / 50 * 2 ==> 8k:320 16k:640*/
    int mp_cnt;

    char *save_buf;
    int save_len;

    rt_sem_t ack;
};

static struct record_manager *recorder;
static short in_short[320*2]; /* NB:8K 160, WB:16K 320 */

static int record_msg_send(struct record_manager *record, void *buffer, int type, int len)
{
    int ret = RT_EOK;
    struct record_msg msg;

    msg.type = type;
    msg.arg = (uint32_t)buffer;
    msg.len = len;

    ret = rt_mq_send(record->msg, (void *)&msg, sizeof(struct record_msg));
    if (ret != RT_EOK)

```

```

        rt_kprintf("[record]:send msg failed \n");
    }

static void record_thread_entry(void *parameter)    //录音以及编码线程的入口函数
{
    rt_device_t device = RT_NULL;
    int ret = RT_EOK;
    uint8_t *mempool;
    rt_uint8_t *buffer;
    rt_uint32_t read_bytes = 0;

    OpusEncoder *opus_enc = RT_NULL;
    int sample_rate, channels, errors, frame_size;
    int application, complexity;
    opus_int32 bitrate_bps;
    int enc_len;

    /* initialize mempool */
    recorder->mp_block_size = 120;
    recorder->mp_cnt = 10;
    mempool = rt_malloc(recorder->mp_block_size * recorder->mp_cnt);
    rt_mp_init(&(recorder->mp), "record_mp", mempool, recorder->mp_block_size *
recorder->mp_cnt, recorder->mp_block_size);

    /* initialize msg queue */
    recorder->msg = rt_mq_create("net_msg", sizeof(struct record_msg), 12, RT_IPC_FLAG_FIFO);

    /* initialize tcp client */
    ret = tcp_client_init(recorder->net);
    if (ret != RT_EOK)
    {
        return;
    }

    device = rt_device_find("mic");
    if (!device)
    {
        rt_kprintf("mic device not found \n");
        return;
    }
}

```

```

rt_device_open(device, RT_DEVICE_OFLAG_RDONLY);

/* set samplerate */
{
    int rate = recorder->sample_rate;
    rt_device_control(device, CODEC_CMD_SAMPLERATE, (void *)&rate);
}

{
    enc_len = opus_encoder_get_size(1);
    rt_kprintf("opus_encoder_get_size: 1 channel size: %d \n", enc_len);
    enc_len = opus_encoder_get_size(2);
    rt_kprintf("opus_encoder_get_size: 2 channel size: %d \n", enc_len);

    sample_rate = recorder->sample_rate;
    channels = 1;
    application = OPUS_APPLICATION_VOIP;
    complexity = 1; // 1 to 10

    opus_enc = opus_encoder_create(sample_rate, channels, application, &errors);
    if(errors != OPUS_OK)
    {
        rt_kprintf("[opus]:create opus encoder failed : %d! \n", errors);
    }

    frame_size = sample_rate / 50; // 20ms ==>
    opus_encoder_set_complexity(opus_enc, complexity);
    opus_encoder_get_bitrate(opus_enc, bitrate_bps);
    rt_kprintf("[opus]:default bitrate %d\n", bitrate_bps);
    rt_kprintf("frame_size = %d \n", frame_size);
}

rt_kprintf("[record]:start record, tick %d \n", rt_tick_get());
while (1)
{
    buffer = rt_mp_alloc(&(recorder->mp), RT_WAITING_NO);
    if(!buffer)
    {
        rt_kprintf("[record]: malloc memory for mempool failed \n");
    }
}

```

```

        rt_thread_mdelay(20);
    }
    else
    {
        /* read data from sound device */
        read_bytes = rt_sound_read(device, 0, in_short, frame_size * 2);
        /*encode ....*/
        {
            enc_len = opus_encode(opus_enc, in_short, frame_size, buffer + 8,
recorder->mp_block_size - 8);

            /* write head */
            {
                opus_uint32 enc_final_range;
                int_to_char_big_endian(enc_len, buffer);

                opus_encoder_get_final_range(opus_enc, enc_final_range);
                int_to_char_big_endian(enc_final_range, buffer+4);
            }

            enc_len += 8;
        }
        record_msg_send(recorder, buffer, RECORD_MSG_DATA, enc_len);
    }

    /* send stop cmd */
    if (recorder->action == 0)
    {
        int cmd;

        cmd = 0;
        record_msg_send(recorder, 0, RECORD_MSG_CMD, 1);
        /* wait ack */
        rt_kprintf("[record]:stop record, tick = %d \n", rt_tick_get());
        break;
    }
}

rt_device_close(device);
rt_mp_detach(&(amp;recorder->mp));
rt_free(mempool);

```



```

rt_mq_delete(recorder->msg);
{
    opus_encoder_destroy(opus_enc);
}
rt_kprintf("[record]:exit record thread, tick = %d \n", rt_tick_get());
}

static void net_transmit_thread_entry(void *parameter)    //网络传输编码后的opus码流入口函数
{
    int ret, cmd;
    struct record_msg msg;

    recorder->save_len = 0;
    while(1)
    {
        if (rt_mq_rcv(recorder->msg, &msg, sizeof(struct record_msg), RT_WAITING_FOREVER)
== RT_EOK)
        {
            if(msg.type == RECORD_MSG_DATA)
            {
                memcpy(recorder->save_buf + recorder->save_len, (void *)msg.arg, msg.len);
                recorder->save_len += msg.len;
                rt_mp_free((void *)msg.arg);

                if(recorder->save_len >= RECORD_SAVE_BUF_SIZE - recorder->mp_block_size)
                {
                    /*send data*/
                    send(recorder->net->sock, recorder->save_buf, recorder->save_len, 0);
                    recorder->save_len = 0;
                }
            }
            else if(msg.type == RECORD_MSG_CMD)
            {
                cmd = *(int *)msg.arg;
                if(cmd == 0)
                {
                    /* send remain data, and send ack */
                }
            }
        }
    }
}

```

```

    }
}
static int record_opus_tcp(int argc, char **argv)
{
    rt_thread_t tid = RT_NULL;
    int result;

    if(recorder == RT_NULL)
    {
        recorder = rt_malloc(sizeof(struct record_manager));
        if(!recorder)
        {
            rt_kprintf("[record]:malloc memory for recorder manager \n");
            return -RT_ERROR;
        }
        memset(recorder, 0, sizeof(struct record_manager));
        {
            struct net_worker *net = RT_NULL;
            net = rt_malloc(sizeof(struct net_worker));
            if(!net)
            {
                rt_kprintf("[record]:malloc memory for net worker \n");
                return -RT_ERROR;
            }
            memset(net, 0, sizeof(struct net_worker));
            recorder->net = net;
            recorder->save_buf = rt_malloc(RECORD_SAVE_BUF_SIZE);
            memset(recorder->save_buf, 0, RECORD_SAVE_BUF_SIZE);
        }

        rt_kprintf("L%d, recorder_create done \n", __LINE__);
    }

    rt_kprintf("L%d, record enter \n", __LINE__);
    if (strcmp(argv[1], "stop") == 0) //停止mic录音及编码
    {
        recorder->action = 0;
    }
    else if (strcmp(argv[1], "start") == 0) //开始mic录音及编码
    {

```

```

/* record start format samplerate url port */
recorder->action = 1;
if(recorder->net->url)
{
    rt_free(recorder->net->url);
    recorder->net->url = RT_NULL;
}
recorder->sample_rate = atoi(argv[2]);           //设置录音mic采样率
recorder->net->url = rt_strdup(argv[3]);         //设置url地址
recorder->net->port = atoi(argv[4]);             //设置网路传输端口
rt_kprintf("[record]:samplerate = %d \n", recorder->sample_rate);
rt_kprintf("[record]:url = %s \n", recorder->net->url);
rt_kprintf("[record]:port = %d \n", recorder->net->port);

/* create net send thread */
tid = rt_thread_create("record",
                        record_thread_entry,
                        RT_NULL,
                        1024 * 32,
                        27,
                        10);

if (tid != RT_NULL)
    rt_thread_startup(tid);

/* create net send thread */
tid = rt_thread_create("net_send",
                        net_transmit_thread_entry,
                        RT_NULL,
                        1024 * 8,
                        28,
                        10);

if (tid != RT_NULL)
    rt_thread_startup(tid);
}
else
{
    // print_record_usage();
}
}

FINISH_FUNCTION_EXPORT_ALIAS(record_opus_tcp, __cmd_record_opus_tcp, record opus tcp);

```

25 EasyFlash

25.1 EasyFlash简介

EasyFlash是一款开源的轻量级嵌入式Flash存储器库，能快速保存产品参数，支持写平衡和掉电保护，降低了开发者对产品参数的处理难度，也保证了产品在后期升级时拥有更好的扩展性。

25.2 EasyFlash Related API

EasyFlash相关接口参考\packages\EasyFlash\inc\easyflash.h，相关接口如下：

函数	描述
<code>easyflash_init()</code>	easyflash初始化
<code>ef_get_env()</code>	获得easyflash环境变量
<code>ef_set_env()</code>	写数据到easyflash中
<code>ef_save_env()</code>	保存数据到flash

25.2.1 easyflash初始化

```
EfErrCode easyflash_init(void);
```

参数	描述
<code>void</code>	空
返回	0: 成功; 其他: 失败

25.2.2 获得easyflash环境变量

```
char *ef_get_env(const char *key);
```

参数	描述
<code>key</code>	环境变量名字
返回	value: 变量地址

25.2.3 将数据写入到环境变量中

```
EfErrCode ef_set_env(const char *key, const char *value);
```

参数	描述
<code>key</code>	环境变量名字
<code>value</code>	要写入的值
返回	0: 成功; 其他: 失败

25.2.4 保存数据到flash

```
EfErrCode ef_save_env(void);
```

参数	描述
void	空
返回	0: 成功; 其他: 失败

25.3 EasyFlash宏定义

#define	PKG_USING_EASYFLASH		使用EasyFlash必须开启
#define	EF_START_ADDR	0x1FE000	EasyFlash起始地址为0x1FE000
#define	ENV_USER_SETTING_SIZE	1 * 1024	EasyFlash用户大小

25.4 EasyFlash示例代码

EasyFlash示例代码参考\test\easyflash_test.c，打开宏定义：
EASY_FLASH_TEST，开启EasyFlash功能测试，具体使用方式可以参考如下
示例代码：

```
/*
 * 程序清单： 这是一个easyflash用法例程
 * 命令调用格式： 输入命令： Easy_Flash_Write，写入数据到flash中
                  掉电重启后，输入命令： Easy_Flash_Read，检测读取写入的数据是否正确。
 * 程序功能： 将字符串保存到flash中，并可以将其读出。
 */
#define EASY_FLASH_TEST
#ifdef EASY_FLASH_TEST
#include "rtthread.h"
#include <dfs.h>
#include <dfs_fs.h>
#include "player.h"
#include "include.h"
#include "driver_pub.h"
#include "func_pub.h"
#include "app.h"
#include "ate_app.h"
#include "shell.h"
#include "flash.h"
#include <finsh.h>
#include "easyflash.h"
```

```

unsigned char read_buff[10*1024];
unsigned char write_buff[10*1024];

#define test_data "AABBCCDDEEFFGGHHIIJJKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ"
#define KEY "temp"

static void Easy_Flash_Write(void)
{
    int i ;

    easyflash_init();                /*初始化 */
    ef_set_env(KEY,test_data);        /*将要写入的数据存放到 easy flash 环境变量 */
    /*
    ef_save_env();                    /*保存数据 */

    rt_kprintf("---Flash Write over \r\n");
}

static void Easy_Flash_Read(void)    /*读取easy flash 写入的数据*/
{
    char *p_write_buff;

    easyflash_init();

    p_write_buff = ef_get_env(KEY);   /*获取easy flash存入的数据*/

    rt_kprintf("%s",p_write_buff);
}

MSH_CMD_EXPORT(Easy_Flash_Write,set_or_read_Easy_Flash_Write test);
MSH_CMD_EXPORT(Easy_Flash_Read, set_or_read_Easy_Flash_Read test);

#endif

```

26 Voice Changer

26.1 Voice Changer简介

Voice changer支持变声功能，能将声音变换成其他的声音特性。

26.2 Voice Changer Related API

Voice changer相关接口参考\components\voice_changer\app_voice_changer.h，相关接口如下：

函数	描述
voice_changer_initial()	变声功能初始化
voice_changer_exit()	退出变声模式
voice_changer_start()	开始变声
voice_changer_stop()	停止变声
voice_changer_set_change_flag()	设置变声功能标志
voice_changer_get_need_mic_data()	获取麦克风数据
voice_changer_set_cost_data()	设置消耗的数据长度
voice_changer_data_handle()	处理声音数据

26.2.1 voice changer初始化

```
VC_ERR voice_changer_initial(uint32_t freq);
```

参数	描述
freq	频率
返回	0：成功 其他：失败

26.2.2 退出voice changer

```
void voice_changer_exit(void);
```

参数	描述
void	空
返回	无

26.2.3 开始voice changer

```
void voice_changer_start(void);
```

参数	描述
----	----

void	空
返回	无

26.2.4 停止voice changer

```
void voice_changer_stop(void);
```

参数	描述
void	空
返回	无

26.2.5 设置voice changer变声功能标志

```
void voice_changer_set_change_flag(void);
```

参数	描述
void	空
返回	无

26.2.6 voice changer获取mic数据

```
int voice_changer_get_need_mic_data(void);
```

参数	描述
void	空
返回	剩下数据长度

26.2.7 设置消耗数据的长度

```
int voice_changer_set_cost_data(int cost_len);
```

参数	描述
cost_len	消耗的数据长度
返回	剩下数据长度

26.2.8 处理数据

```
int voice_changer_data_handle(uint8_t *mic_in, int mic_len, uint8_t **vc_out);
```

参数	描述
mic_in	mic接收的数据
mic_len	mic接收的数据长度

vc_out	变声功能处理后的数据
返回	0: 成功 其他: 失败

26.3 Voice Changer宏定义

#define	CONFIG_VOICE_CHANGER	使用voice changer必须开启
---------	----------------------	---------------------

26.4 Voice Changer枚举类型说明

voice changer枚举类型:

```
typedef enum {
    VC_STOP,      //停止
    VC_FIRST,     //第一个
    VC_START,     //开始
} VC_STA;
```

26.5 Voice Changer示例代码

Voice changer示例代码参考\components\voice_changer\voice_changer_task.c。

```
/*
 * 程序清单： 这是一个voice changer用法例程
 * 命令调用格式： 输入命令： voice_changer_sample launch/shutoff/next
 * 程序功能： 将采集到的声音做变声处理。
 */
#include <rtthread.h>
#include "vc_config.h"

#define VOICE_CHANGER_SOFT_TIMER_HANDLER      1
#define VOICE_CHANGER_THREADT_TASK_HANDLER   2

#define VOICE_CHANGER_HANDLER
VOICE_CHANGER_THREADT_TASK_HANDLER

#define VOICE_CHANGER_MIC_CFG                 1
#define VOICE_CHANGER_MIC_INIT_CFG           1

#define VOICE_CHANGER_DEFAULT_OUT_AUD        1
#define VOICE_CHANGER_AUD_INIT_CFG           1
#define VOICE_CHANGER_AUD_SINGLE_CH          1
```

```

#ifndef min
#define min(x, y)          (((x) < (y)) ? (x) : (y))
#endif

#if CONFIG_VOICE_CHANGER
#include "app_voice_changer.h"
#include "rtos_pub.h"
#include "audio_device.h"
#include "string.h"
#include "stdio.h"
#include "stdlib.h"

#define VC_BUFF_MAX_LEN    (256 * 4 * sizeof(unsigned int))
#define VC_HANDLER_INTERVAL_MS    5

beken_thread_t voice_changer_handler = NULL;
beken_timer_t vc_timer;

static void *vctimer = NULL;
static char *vcbuff = NULL;
static int g_running_flag;

#if VOICE_PCM_VC_AUD_OUTPUT_TEST
#define PCM_LENGTH          35254
extern const unsigned char acnumber_pcm[];                ///<35254
static unsigned int pc_offset = 0;
#endif

static int voice_changer_read_pcm(char*outbuf,int len)      /*读取mic数据并且存放到buffer*/
{
    int out_len = 0;
    #if VOICE_CHANGER_MIC_CFG
        out_len = audio_device_mic_read(outbuf,len);
    #endif

    #if VOICE_PCM_VC_AUD_OUTPUT_TEST
        out_len = min(len,(PCM_LENGTH - pc_offset));
        memcpy(outbuf,acnumber_pcm+pc_offset,out_len);
    #endif
}

```

```

        pc_offset += out_len;
        if(pc_offset >= PCM_LENGTH)
        {
            pc_offset = 0;
            rt_kprintf("restart\r\n");
        }
    #endif
    return out_len;
}

static int voice_changer_write_pcm(char*outbuf,int len)          /*变声数据传送到到pcm*/
{
    int input_len = 0;

    #if VOICE_CHANGER_DEFAULT_OUT_AUD
        int bufsz;
        uint16_t* aud_buf = (uint16_t *)audio_device_get_buffer(&bufsz);
        if((bufsz == 0) || (aud_buf == NULL))
        {
            if(aud_buf)
            {
                audio_device_put_buffer(aud_buf);
            }
            rt_kprintf("vc err L%d\r\n",__LINE__);
            return input_len;
        }

        input_len = min((bufsz>>1),len);
        if(len == 0)
        {
            goto exit;
        }
    #if VOICE_CHANGER_AUD_SINGLE_CH
        int16_t *src,*dst;
        int i;
        src = outbuf;
        dst = aud_buf;
        for(i=0;i<(len/2);i++)
        {
            dst[2 * i] = src[i];

```

```

        dst[2 * i + 1] = src[i];
    }

    audio_device_write((uint8_t *)aud_buf, input_len*2);
#else
    memcpy(aud_buf, outbuf, input_len);
    audio_device_write((uint8_t *)aud_buf, input_len);
#endif
#endif
return input_len;
exit:
if(aud_buf)
{
    audio_device_put_buffer(aud_buf);
}
rt_kprintf("vc L%d err\r\n", __LINE__);
return 0;
}

static int voice_changer_shutoff(void)                                /*关闭变声功能*/
{
    g_running_flag = 0;
    if (vctimer != RT_NULL)
    {
        #if VOICE_CHANGER_HANDLER == VOICE_CHANGER_SOFT_TIMER_HANDLER
            rt_timer_stop((rt_timer_t)vctimer);
        #elif VOICE_CHANGER_HANDLER == VOICE_CHANGER_THREADT_TASK_HANDLER
            bk_rtos_delete_thread(vc_handler);
        #endif
    }

    return 0;}

static int voice_changer_launch(unsigned int freq)                  /*开启变声功能：加长声音*/
{
    if(vcbuff == NULL)
    {
        vcbuff = (char*)rt_malloc(VC_BUFF_MAX_LEN);
    }
    if(vcbuff == NULL)

```

```

    {
        rt_kprintf("vcbuff == null\r\n");
        return -1;
    }
#endif

#if (VOICE_CHANGER_MIC_CFG && VOICE_CHANGER_MIC_INIT_CFG)
    audio_device_init();

    audio_device_mic_open();
    audio_device_mic_set_channel(1);
    audio_device_mic_set_rate(freq);
#endif

#if VOICE_CHANGER_DEFAULT_OUT_AUD && VOICE_CHANGER_AUD_INIT_CFG
    audio_device_init();

    audio_device_open();
    audio_device_set_rate(freq);
    audio_device_set_volume(100);
#endif

    g_running_flag = 1;
    voice_changer_initial(freq);
    if (vctimer != RT_NULL)
    {
        #if VOICE_CHANGER_HANDLER == VOICE_CHANGER_SOFT_TIMER_HANDLER
            rt_timer_start((rt_timer_t)vctimer);
            voice_changer_start();
        #elif VOICE_CHANGER_HANDLER == VOICE_CHANGER_THREAD_T_TASK_HANDLER
            rt_thread_startup((rt_thread_t)vctimer);
        #endif
        rt_kprintf("vc start\r\n");
    }

    return 0;
}

static int voice_changer_handler(void)                                /*对采集的声音数据处理*/
{
    unsigned char* vc_out;
    int vc_out_len;
    int len;

```

```

if(vcbuff == NULL)
{
    rt_kprintf("vcbuff err\r\n");
    return -1;
}

len = voice_changer_get_need_mic_data();
if(len > 0) {
    len = (len > (VC_BUFF_MAX_LEN/4))?(VC_BUFF_MAX_LEN/4) : len;
}
else if(len < 0)
{
    return -1;
}
else if(len == 0)
{
    return 0;
}

len = voice_changer_read_pcm(vcbuff,len);
if(len <= 0)
{
    rt_kprintf("origin pcm empty\r\n");
    return 0;
}

vc_out_len = voice_changer_data_handle((uint8*)vcbuff, len, &vc_out);
if(vc_out_len == 0)
{
    // no enough data for vc, so vc return 0, no need do sm_playing
    return 0;
}
else if(vc_out_len > 0)
{
    #if 1
    len = voice_changer_write_pcm((char*)vc_out,vc_out_len);
    #else
    voice_changer_write_pcm(vcbuff,len);
    len = vc_out_len;

```

```

    #endif

    if(len > 0)
    {
        voice_changer_set_cost_data(len);
    }
}

return 0;
}

static int app_voice_changer_init(void) /* 变声功能初始化*/
{
    #if VOICE_CHANGER_HANDLER == VOICE_CHANGER_SOFT_TIMER_HANDLER
        if(vctimer == NULL)
        {
            vctimer = (void*)rt_timer_create("vc",
                                              voice_changer_timer_handler,
                                              NULL,
                                              VC_HANDLER_INTERVAL_MS,
                                              RT_TIMER_FLAG_PERIODIC |
RT_TIMER_FLAG_SOFT_TIMER);
        }
    #elif VOICE_CHANGER_HANDLER == VOICE_CHANGER_THREADT_TASK_HANDLER
        if(vctimer == NULL)
        {
            vctimer = (void*)rt_thread_create("vc",
                                              voice_changer_task_handler,
                                              NULL,
                                              4*1024,
                                              15,
                                              20);

            rt_kprintf("vctimer = %p\r\n",vctimer);
        }
    #endif

    return 0;
}

INIT_APP_EXPORT(app_voice_changer_init);
static int voice_changer_sample(int argc, char *argv[])
{
    rt_err_t ret = RT_EOK;

    unsigned int freq = 16000;

```

```

if(argc == 2)
{
    if(strcmp(argv[1],"launch") == 0)
    {
        rt_kprintf("voice changer freq = %d\r\n",freq);
        voice_changer_launch(freq);
        app_voice_changer_init();
    }
    else if(strcmp(argv[1],"shutoff") == 0)
    {
        rt_kprintf("voice changer shutoff\r\n");
        voice_changer_shutoff();
    }
    else if(strcmp(argv[1],"next") == 0)
    {
        rt_kprintf("voice changer set next\r\n");
        voice_changer_set_change_flag();
    }
}
else if(argc == 3)
{
    if(strcmp(argv[1],"launch") == 0)
    {
        freq = atoi(argv[2]);
        rt_kprintf("voice changer freq = %d\r\n",freq);
        voice_changer_launch(freq);
    }
}
return ret;
}

```

```

MSH_CMD_EXPORT(voice_changer_sample,vc sample);
#endif

```


27 图像传输

27.1 图像传输简介

- a) 有高速spi-slave接口，速度高达50Mbps，可以外接其他MCU摄像头；
- b) 支持DCMI标准摄像头接口，PCLK高达24M。支持如PAS6329/6375、OV_7670、GC0328C/0308C等摄像头。
- c) 有硬件Jpeg压缩模块，目前支持最大分辨率600*800；

27.2 图像传输 Related API

图像传输相关接口参考**beken378\func\video_transfer\video_transfer.h**。

函数	描述
video_transfer_init()	打开video_transfer模块
video_transfer_deinit()	关闭video_transfer模块
video_transfer_set_video_param()	使用DCMI接口时，设置摄像头的参数
video_buffer_open()	打开获取jpeg帧数据功能
video_buffer_close()	关闭获取jpeg帧数据功能
video_buffer_read_frame()	获取一帧jpeg数据，可能会挂起，直到整张jpeg收集完，并且该jpeg长度不超过目标buf的长度，才返回

27.2.1 打开video_transfer

```
int video_transfer_init(TVIDEO_SETUP_DESC_PTR setup_cfg);
```

参数	描述
setup_cfg	TVIDEO_SETUP_DESC_PTR类型的结构体
返回	kNoErr: 成功；其他：失败

27.2.2 关闭video_transfer

```
int video_transfer_deinit(void);
```

参数	描述
void	无
返回	kNoErr: 成功；其他：失败

27.2.3 设置摄像头的参数

```
UINT32 video_transfer_set_video_param(UINT32 ppi, UINT32 fps);
```

参数	描述
----	----

ppi	分辨率（pixer per inch），见PPI_TYPE的定义。
fps	帧率（frame per second），见FPS_TYPE的定义
返回	0：成功； 1：失败

27.2.4 打开获取jpeg帧的功能

```
int video_buffer_open (void);
```

参数	描述
void	无
返回	0：成功； 1：失败

27.2.5 关闭获取jpeg帧的功能

```
int video_buffer_close (void);
```

参数	描述
void	无
返回	0：成功； 1：失败

27.2.6 获取jpeg帧的数据

```
UINT32 video_buffer_read_frame(UINT8 *buf, UINT32 buf_len);
```

参数	描述
buf	存放jpeg数据的内存首地址
buf_len	存放jpeg数据的内存长度
返回	获取的jpeg帧的长度

27.3 图像传输的结构体说明

TVIDEO_SETUP_DESC_PTR :

UINT32 send_type	一般为TVIDEO_SND_TYPE枚举类型，模块里会根据send_type决定每个图像数据包的大小。
send_func	图像数据的发送函数，模块转发图像数据包时，通过send_func发送。
start_cb	模块打开spi或camera_intf后，回调此函数，用于指示传图开始。
end_cb	模块关闭spi或camera_intf前，回调此函数，用于指示传图结束。
pkt_header_size	若要在图像数据包里加入“头信息”，pkt_header_size用于指示“头信息”的大小，注意pkt_header_size的值必须是4的整数倍。如果不使用“头信息”，设成0即可。
add_pkt_header	添加“头信息”的回调函数，该函数会在每收到一个图像数据包时，

	回调，用户需实现“头信息”的具体内容。如果不使用“头信息”，设成NULL即可
--	--

TV_HDR_PARAM_PTR:

UINT8* ptk_ptr	头信息可以从这个指针指向的内存开始填（不得超过设置的头信息大小）
UINT32 ptklen	这个数据包有多少图像数据
UINT32 frame_id	这个数据包属于本次图传的第frame_id个图像帧。
UINT32 is_eof	这个数据包是否是这个帧的帧尾，1则指示是帧尾，0则不是。
UINT32 frame_len	仅is_eof为1时有意义，指示整个帧的数据，共占frame_len个数据包。

27.4 图像传输的宏定义

#define CFG_USE_CAMERA_INTF	摄像头+jpeg 图传的开关宏
#define CFG_USE_SPIDMA	High-spi-slave 图传时spidma模块的开关宏
#define CFG_USE_HSLAVE_SPI	High-spi-slave 图传时spi接口的开关宏
#define CFG_USE_APP_DEMO_VIDEO_TRANSFER	图传demo开关宏

27.5 图像传输枚举类型说明

发送类型说明如下所示：

<pre>typedef enum { TVIDEO_SND_UDP, /*通过UDP上传*/ TVIDEO_SND_TCP, /*通过TCP上传*/ TVIDEO_SND_INTF, /*通过其他接口上传*/ } TVIDEO_SND_TYPE;</pre>	
<pre>typedef enum { QVGA_320_240 = 0, VGA_640_480, PPI_MAX } PPI_TYPE; /*分辨率的枚举*/</pre>	
<pre>typedef enum { TYPE_5FPS = 0, TYPE_10FPS, TYPE_20FPS, FPS_MAX } FPS_TYPE; /*帧率的枚举*/</pre>	

27.6 图像传输的示例代码

图像传输示例代码参考**beken378\app\app_demo**文件夹下，具体使用方式可以参考如下示例代码：

1.不使用“头信息”的示例

```
/*发送函数，什么也没有做，直接返回*/
int app_video_intf_send_packet (UINT8 *data, UINT32 len)
{
    //os_printf("voide send:%p, %p\r\n", data, len);
    return len;
}

void app_video_intf_open (void)
{
    os_printf("voide open\r\n");
    /*spi接口方式 或 camera_intf 二选一*/
    #if (CFG_USE_SPIDMA || CFG_USE_CAMERA_INTF)
        TVIDEO_SETUP_DESC_ST setup;
        /* TVIDEO_SND_INTF 指示这个传送方式为 send intf*/
        setup.send_type = TVIDEO_SND_INTF;
        setup.send_func = app_video_intf_send_packet;
        /*不需要指示 图传开始或结束 */
        setup.start_cb = NULL;
        setup.end_cb = NULL;
        /*不使用 头信息*/
        setup.pkt_header_size = 0;
        setup.add_pkt_header = NULL;

        video_transfer_init(&setup);
    #endif
}

void app_video_intf_close (void)
{
    os_printf("voide close\r\n");
    #if (CFG_USE_SPIDMA || CFG_USE_CAMERA_INTF)
        video_transfer_deinit();
    #endif
}
```

2.使用“头信息”的示例

```
/*自定义 头信息 */
```

```

typedef struct tvideo_hdr_st
{
    UINT8 id;
    UINT8 is_eof;
    UINT8 pkt_cnt;
    UINT8 size;
}HDR_ST, *HDR_PTR;

/*头信息 回调函数。这个每个数据包的前4个字节都会加入 HDR_ST的头信息*/
void app_demo_add_pkt_header(TV_HDR_PARAM_PTR param)
{
    HDR_PTR elem_tvhdr = (HDR_PTR)param->ptk_ptr;
    elem_tvhdr->id = (UINT8)param->frame_id;
    elem_tvhdr->is_eof = param->is_eof;
    elem_tvhdr->pkt_cnt = param->frame_len;
    elem_tvhdr->size = 0;
}

/*发送函数，使用udp方式发送，返回发送成功的字节数 */
int app_demo_udp_send_packet (UINT8 *data, UINT32 len)
{
    int send_byte = 0;
    if(!app_demo_udp_remote_connected)
        return 0;
    send_byte = sendto(app_demo_udp_img_fd, data, len, MSG_DONTWAIT|MSG_MORE,
        (struct sockaddr *)app_demo_remote, sizeof(struct sockaddr_in));
    if (send_byte < 0) {
        /* err */
        //APP_DEMO_UDP_PRT("send return fd:%d\r\n", send_byte);
        send_byte = 0;
    }
    return send_byte;
}

/*指示开始传图 */
static void app_demo_udp_app_connected(void)
{
    app_demo_softap_send_msg(DMSG_APP_CONNECTED);
}

/*指示停止传图 */
static void app_demo_udp_app_disconnected(void)
{
    app_demo_softap_send_msg(DMSG_APP_DISCONNECTED);
}

```

```

}

void app_video_intf_open (void)
{
    TVIDEO_SETUP_DESC_ST setup;
    setup.send_type = TVIDEO_SND_UDP;
    setup.send_func = app_demo_udp_send_packet;
    setup.start_cb = app_demo_udp_app_connected;
    setup.end_cb = app_demo_udp_app_disconnected;
    setup.pkt_header_size = sizeof(HDR_ST);
    setup.add_pkt_header = app_demo_add_pkt_header;
    video_transfer_init(&setup);
}

void app_video_intf_close (void)
{
    os_printf("voide close\r\n");
    #if (CFG_USE_SPIDMA || CFG_USE_CAMERA_INTF)
    video_transfer_deinit();
    #endif
}

```

3. 获取一帧jpeg图像以及设置摄像头参数的示例

```

/*发送串口命令 */
/*vbuf open : 打开获取一帧jpeg图像的功能 */
/*vbuf close : 关闭获取一帧jpeg图像的功能 */
/*vbuf read len_xxx: len_xxx 是读取buf的长度，读取一帧jpeg图像，并打印jpeg数据 */
/*vbuf setp ppi_xxx pfs_xxx : 分辨率ppi_xxx 的取值0、1， 帧率pfs_xxx的取值 0、1、2 */

void vbuf(int argc, char** argv)
{
    if(strcmp(argv[1], "open") == 0)
    {
        video_buffer_open();
    }
    else if(strcmp(argv[1], "read") == 0)
    {
        uint8_t *mybuf, i;
        uint32_t my_len;
        my_len = atoi(argv[2]);
        mybuf = os_malloc(my_len);
        if(mybuf == NULL)

```

```

    {
        rt_kprintf("vbuf test no buff\r\n");
        return;
    }
    my_len = video_buffer_read_frame(mybuf, my_len);
    rt_kprintf("frame_len: %d\r\n", my_len);
    if(1)
    {
        for(int i=0; i<my_len; i++)
        {
            rt_kprintf("%02x,", mybuf[i]);
            if((i+1)%32 == 0)
                rt_kprintf("\r\n");
        }
    }
    os_free(mybuf);
}
else if(strcmp(argv[1], "close") == 0)
{
    video_buffer_close();
}
else if(strcmp(argv[1], "setp") == 0)
{
    uint32_t ppi, pfs;
    ppi = atoi(argv[2]);
    pfs = atoi(argv[3]);
    video_transfer_set_video_param(ppi, pfs);
}
else
{
    rt_kprintf("vbuf open/read len/close/setp ppi pfs\r\n");
}
}
MSH_CMD_EXPORT(vbuf, vbuf);

```


28 Qspi Dcache模式

28.1 Qspi Dcache简介

BK7251的qspi dcache模式是将芯片外接的psram芯片地址映射到芯片dcache地址中，其中dcache基地址为0x03000000，通过这个基地址可以对psram进行正常的读写操作。

28.2 Qspi Dcache Related API

qspi dcache相关接口参考beken378\func\user_driver\BkDriverQspi.h，相关接口如下：

函数	描述
bk_qspi_dcache_initialize()	qspi dcache初始化
bk_qspi_start()	启动qspi功能
bk_qspi_stop ()	停止qspi功能

28.2.1 初始化qspi为dcache模式

```
OSStatus bk_qspi_dcache_initialize(qspi_dcache_drv_desc *qspi_config);
```

参数	描述
qspi_config	qspi的参数配置
返回	0：成功； -1： 错误

28.2.2 启动qspi功能

```
OSStatus bk_qspi_start(void);
```

参数	描述
void	空
返回	0：成功； -1： 错误

28.2.3 停止qspi功能

```
OSStatus bk_qspi_stop(void);
```

参数	描述
void	空
返回	0：成功； -1： 错误

28.3 Qspi Dcache结构体说明

qspi_dcache_drv_desc	
mode	qspi模式
clk_set	时钟源选择分频系数设置
wr_command	写入数据命令
rd_command	读取数据命令
wr_dummy_size	写数据大小
rd_dummy_size	读数据大小

28.4 Qspi Dcache示例

示例代码参考test\qspi_test.c，打开宏定义：QSPI_TEST，开启qspi dcache模式的测试。

Note: 测试中需要外接psram芯片。

```
/*
 * 程序清单： 这是一个简单qspi dcache模式的使用例程，打开宏定义QSPI_TEST，开启测功能。
 * 命令调用格式： qspi_test
 * 程序功能： 通过qspi模块向psram写入数据并读取数据，最后比较写入和读取的数据是否有差别。
 */
#include "error.h"
#include "include.h"
#include <rthw.h>
#include <rtthread.h>
#include <rtdevice.h>
#include <stdint.h>
#include <stdlib.h>
#include <finsh.h>
#include <rtddef.h>
#include "include.h"
#include <stdio.h>
#include <string.h>
#include "typedef.h"
#include "arm_arch.h"
#include "qspi_pub.h"
#include "BkDriverQspi.h"
#include "test_config.h"

#ifdef QSPI_TEST

#define QSPI_TEST_LENGTH      ( 0x4 * 16 )
```

```

static UINT8 DataOffset;

static void qspi_psram_dcache_test(int argc, char *argv[])
{
    UINT32 i, ret;
    UINT32 SetLineMode;
    qspi_dcache_drv_desc qspi_cfg;

    UINT32* p_WRData1;
    UINT32* p_WRData2;
    UINT32* p_WRData3;
    UINT32* p_WRData4;
    UINT32* p_WRData5;

    UINT32* p_RDData1;
    UINT32* p_RDData2;
    UINT32* p_RDData3;
    UINT32* p_RDData4;
    UINT32* p_RDData5;

    p_WRData1 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_WRData1[0]));
    if(p_WRData1 == RT_NULL)
    {
        rt_kprintf("p_WRData1 malloc failed\r\n");
    }

    p_WRData2 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_WRData2[0]));
    if(p_WRData2 == RT_NULL)
    {
        rt_kprintf("p_WRData2 malloc failed\r\n");
    }

    p_WRData3 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_WRData3[0]));
    if(p_WRData3 == RT_NULL)
    {
        rt_kprintf("p_WRData3 malloc failed\r\n");
    }

    p_WRData4 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_WRData4[0]));

```

```
if(p_WRData4 == RT_NULL)
{
    rt_kprintf("p_WRData4 malloc failed\r\n");
}

p_WRData5 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_WRData5[0]));
if(p_WRData5 == RT_NULL)
{
    rt_kprintf("p_WRData5 malloc failed\r\n");
}

p_RDData1 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDData1[0]));
if(p_RDData1 == RT_NULL)
{
    rt_kprintf("p_RDData1 malloc failed\r\n");
}

p_RDData2 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDData2[0]));
if(p_RDData2 == RT_NULL)
{
    rt_kprintf("p_RDData2 malloc failed\r\n");
}

p_RDData3 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDData3[0]));
if(p_RDData3 == RT_NULL)
{
    rt_kprintf("p_RDData3 malloc failed\r\n");
}

p_RDData4 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDData4[0]));
if(p_RDData4 == RT_NULL)
{
    rt_kprintf("p_RDData4 malloc failed\r\n");
}

p_RDData5 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDData5[0]));
if(p_RDData5 == RT_NULL)
{
    rt_kprintf("p_RDData5 malloc failed\r\n");
}
```

```

for(i=0; i<QSPI_TEST_LENGTH; i++)
{
    p_WRData1[i]= ((i+1)<<24) | ((i+1)<<16) | ((i+1)<<8) | ((i+1)<<0) |0x70707070;
    p_WRData2[i]= ((i+1)<<24) | ((i+1)<<16) | ((i+1)<<8) | ((i+1)<<0) |0x80808080;
    p_WRData3[i]= ((i+1)<<24) | ((i+1)<<16) | ((i+1)<<8) | ((i+1)<<0) |0x90909090;
    p_WRData4[i]= ((i+1)<<24) | ((i+1)<<16) | ((i+1)<<8) | ((i+1)<<0) |0xe0e0e0e0;
    p_WRData5[i]= ((i+1)<<24) | ((i+1)<<16) | ((i+1)<<8) | ((i+1)<<0) |0xf0f0f0f0;
}

if(argc == 2)
{
    rt_kprintf("[qspi_test]:test_qspi_dcache_write_read_data\r\n");

    SetLineMode = atoi(argv[1]);

    qspi_cfg.mode = SetLineMode;      // 0: 1 line mode    3: 4 line mode
    qspi_cfg.clk_set = 0x10;
    qspi_cfg.wr_command = SetLineMode ? 0x38 : 0x02;      //write
    qspi_cfg.rd_command = SetLineMode ? 0xEB : 0x03;      //read
    qspi_cfg.wr_dummy_size = 0;
    qspi_cfg.rd_dummy_size = SetLineMode ? 0x06 : 0x00;

    bk_qspi_dcache_initialize(&qspi_cfg);
    bk_qspi_start();

    bk_qspi_dcache_write_data(0x00000, p_WRData1, QSPI_TEST_LENGTH);
    bk_qspi_dcache_write_data(0x04000, p_WRData2, QSPI_TEST_LENGTH);
    bk_qspi_dcache_write_data(0x08000, p_WRData3, QSPI_TEST_LENGTH);
    bk_qspi_dcache_write_data(0x0C000, p_WRData4, QSPI_TEST_LENGTH);
    bk_qspi_dcache_write_data(0x10000, p_WRData5, QSPI_TEST_LENGTH);

    rt_thread_delay(rt_tick_from_millisecond(100));

    bk_qspi_dcache_read_data(0x00000, p_RDData1, QSPI_TEST_LENGTH);
    bk_qspi_dcache_read_data(0x04000, p_RDData2, QSPI_TEST_LENGTH);
    bk_qspi_dcache_read_data(0x08000, p_RDData3, QSPI_TEST_LENGTH);
    bk_qspi_dcache_read_data(0x0C000, p_RDData4, QSPI_TEST_LENGTH);
    bk_qspi_dcache_read_data(0x10000, p_RDData5, QSPI_TEST_LENGTH);
}

```

```

if(memcmp(p_WRData1, p_RDData1, QSPI_TEST_LENGTH*4) == 0)
{
    rt_kprintf("[qspi_test]:qspi read data 1 pass \r\n ");
}
else
{
    rt_kprintf("[qspi_test]:qspi read data 1 error !!! \r\n ");

    for (i=0; i<QSPI_TEST_LENGTH; i++)
    {
        rt_kprintf("p_WRData[%d]=0x%lx, p_RDData[%d]=0x%lx\r\n", i, *(p_WRData1 +
i), i, *(p_RDData1 + i));
    }
}

if(memcmp(p_WRData2, p_RDData2, QSPI_TEST_LENGTH*4) == 0)
{
    rt_kprintf("[qspi_test]:qspi read data 2 pass \r\n ");
}
else
{
    rt_kprintf("[qspi_test]:qspi read data 2 error !!! \r\n ");

    for (i=0; i<QSPI_TEST_LENGTH; i++)
    {
        rt_kprintf("p_WRData[%d]=0x%lx, p_RDData[%d]=0x%lx\r\n", i, *(p_WRData2 +
i), i, *(p_RDData2 + i));
    }
}

if(memcmp(p_WRData3, p_RDData3, QSPI_TEST_LENGTH*4) == 0)
{
    rt_kprintf("[qspi_test]:qspi read data 3 pass \r\n ");
}
else
{
    rt_kprintf("[qspi_test]:qspi read data 3 error !!! \r\n ");
    for (i=0; i<QSPI_TEST_LENGTH; i++)
    {

```

```

        rt_kprintf("p_WRData[%d]=0x%lx, p_RDData[%d]=0x%lx\r\n", i, *(p_WRData3 +
i), i, *(p_RDData3 + i));
    }
}
if(memcmp(p_WRData4, p_RDData4, QSPI_TEST_LENGTH*4) == 0)
{
    rt_kprintf("[qspi_test]:qspi read data 4 pass \r\n ");
}
else
{
    rt_kprintf("[qspi_test]:qspi read data 4 error !!! \r\n ");

    for (i=0; i<QSPI_TEST_LENGTH; i++)
    {
        rt_kprintf("p_WRData[%d]=0x%lx, p_RDData[%d]=0x%lx\r\n", i, *(p_WRData4 +
i), i, *(p_RDData4 + i));
    }
}

if(memcmp(p_WRData5, p_RDData5, QSPI_TEST_LENGTH*4) == 0)
{
    rt_kprintf("[qspi_test]:qspi read data 5 pass \r\n ");
}
else
{
    rt_kprintf("[qspi_test]:qspi read data 5 error !!! \r\n ");

    for (i=0; i<QSPI_TEST_LENGTH; i++)
    {
        rt_kprintf("p_WRData[%d]=0x%lx, p_RDData[%d]=0x%lx\r\n", i, *(p_WRData5 +
i), i, *(p_RDData5 + i));
    }
}

if(p_WRData1 != RT_NULL)
{
    rt_free(p_WRData1);
    p_WRData1= RT_NULL;
}

```

```
if(p_WRData2 != RT_NULL)
{
    rt_free(p_WRData2);
    p_WRData2= RT_NULL;
}
if(p_WRData3 != RT_NULL)
{
    rt_free(p_WRData3);
    p_WRData3= RT_NULL;
}
if(p_WRData4 != RT_NULL)
{
    rt_free(p_WRData4);
    p_WRData4= RT_NULL;
}
if(p_WRData5 != RT_NULL)
{
    rt_free(p_WRData5);
    p_WRData5= RT_NULL;
}

if(p_RDData1 != RT_NULL)
{
    rt_free(p_RDData1);
    p_RDData1= RT_NULL;
}
if(p_RDData2 != RT_NULL)
{
    rt_free(p_RDData2);
    p_RDData2= RT_NULL;
}
if(p_RDData3 != RT_NULL)
{
    rt_free(p_RDData3);
    p_RDData3= RT_NULL;
}
if(p_RDData4 != RT_NULL)
{
    rt_free(p_RDData4);
    p_RDData4= RT_NULL;
```



```
    }
    if(p_RDData5 != RT_NULL)
    {
        rt_free(p_RDData5);
        p_RDData5= RT_NULL;
    }
}
else
{
    rt_kprintf("[qspi_test]:argc error!!! \r\n");
}
}
```

```
FINSH_FUNCTION_EXPORT_ALIAS(qspi_psram_dcache_test, __cmd_qspi_test, test
qspi_psram_dcache mode);
#endif
```