



BEKEN WiFi-SOC SDK

API Reference



Better Life with Wireless
美好生活尽在无线

Version 3.0.27
Copyright © 2020



Version History

Version	Date	Description
1.0.0	2019.04	First Release
2.0.0	2019.08	1.Modified about bootloader 2.add qspi/jpeg moudle
3.0.0	2019.09	1.Adjust document's structure 2.add BLE moudle
3.0.3	2020.10	1.Adjust whole directory structure and content basing on BEKEN SDK 3.0.3
3.0.27	2021.3.30	1.Add OTA API for freertos SDK



目录

Version History	2
1 ADC	19
1.1 ADC 简介	19
1.2 ADC Related API in	19
1.2.1 adc 通用结构体说明	19
1.2.2 创建 adc 检测线程	20
1.2.3 配置 adc 检测通道及回调函数	20
1.2.4 启动 adc	20
1.2.5 关闭 adc	20
1.3 ADC 示例代码	21
2 PWM	23
2.1 PWM 简介	23
2.2 PWM Related API	23
2.2.1 pwm 枚举类型说明	23
2.2.2 初始化 pwm	24
2.2.3 启动 pwm 功能	24
2.2.4 停止 pwm 功能	24
2.2.5 7231n 初始化 pwm	24
2.2.6 设置 2 个 pwm 组成互斥通道	25
2.2.7 调节 pwm 参数	25
2.2.8 停止 pwm 互斥功能	25
2.2.9 设置 pwm 初始电平为低电平	26
2.2.10 设置 pwm 初始电平为高电平	26
2.3 PWM 示例代码	26
2.4 操作说明	27
2.4.1 打开配置	27
2.4.2 运行现象	27
2.5 注意事项	28
3 GPIO	29



3.1 GPIO 简介	29
3.2 GPIO Related API	29
3.2.1 设置引脚模式	29
3.2.2 设置引脚电平	30
3.2.3 读取引脚电平	30
3.2.4 绑定引脚中断回调函数	30
3.2.5 使能引脚中断	30
3.2.6 脱离引脚中断回调函数	31
3.2.6 脱离引脚中断回调函数	31
3.3 GPIO 示例代码	31
3.3.1 关键说明	31
3.3.2 示例代码	32
3.4 操作说明	33
3.4.1 打开配置	33
3.4.2 运行现象	34
4 Button	35
4.1 Button 简介	35
4.2 Button Realated API	35
4.2.1 button 初始化	35
4.2.2 配置回调函数	35
4.2.3 开始 button 工作	36
4.2.4 结束 button 工作	36
4.3 Button 示例代码	36
4.3.1 关键说明	36
4.3.2 示例代码	36
4.4 操作说明	39
4.4.1 打开配置	39
4.4.2 按键测试	39
5 I2C 总线	41
5.1 I2C 简介	41
5.2 I2C Related API	41



5.3 I2C1 读写 eeprom 数据示例如下:	41
6 I2S 总线	47
6.1 I2S 简介	47
6.2 I2S Related API	47
6.2.1 i2s 通用结构体说明	47
6.2.2 i2s 模块参数设置	47
6.2.3 i2s 主从设备发送/接收数据	48
6.3 I2S 示例代码	48
6.3.1 关键说明	48
6.3.2 示例代码	49
7 通用 SPI	54
7.1 通用 SPI 简介	54
7.2 通用 SPI Related API	54
7.2.1 spi 通用结构体说明	54
7.2.2 spi 模块配置	54
7.2.3 spi 发送数据	55
7.2.4 spi 接收数据	55
7.3 通用 SPI 示例代码	55
7.3.1 关键说明	56
7.3.2 示例代码	56
7.4 操作说明	59
7.4.1 打开配置	59
7.4.2 运行现象	59
7.5 注意事项	60
8 通用 SPI FLASH 设备	61
8.1 通用 SPI FLASH 简介	61
8.2 通用 SPI FLASH Related API	61
8.2.1 控制设备	61
8.3 通用 SPI FLASH 示例代码	61
8.3.1 关键说明	61
8.3.2 示例代码	62



8.4 操作说明	64
8.4.1 运行现象	64
8.5 注意事项	65
9 通用 SPI PSRAM 设备	66
9.1 通用 SPI PSRAM 简介	66
9.2 通用 SPI PSRAM related API	66
9.3 通用 SPI PSRAM 示例代码	66
9.3.1 关键说明	66
9.3.2 示例代码	66
9.4 操作说明	68
9.4.1 打开配置	68
9.4.2 运行现象	68
9.5 注意事项	69
10 高速 SPI 从设备	69
10.1 高速 SPI 从设备简介	69
10.2 高速 SPI 从设备 Related API	69
10.3 高速 SPI 从示例代码	69
10.3.1 关键说明	69
10.3.2 示例代码	69
10.4 操作说明	72
10.4.1 打开配置	72
10.4.2 运行现象	72
10.5 注意事项	72
11 UART	73
11.1 UART 简介	73
11.2 UART Related API	73
11.2.1 uart 通用结构体说明	73
11.2.2 控制串口设备	74
11.2.3 uart 初始化	75
11.2.4 uart 发送数据	75
11.2.5 uart 接收数据	75



11.2.6 uart 接收回调函数	75
11.3 UART 示例代码	76
11.3.1 关键说明	76
11.3.2 示例代码	77
11.4 操作说明	79
11.4.1 打开配置	79
11.4.2 运行现象	80
11.5 注意事项	80
12 Airkiss	81
12.1 Airkiss 简介	81
12.2 Airkiss Related API	81
12.2.1 开始 airkiss	81
12.2.2 获取 airkiss 状态	81
12.2.3 获取 airkiss 解码结果	81
12.2.4 开始 airkiss	82
12.3 Airkiss 示例代码	82
12.3.1 关键说明	82
12.3.2 示例代码	83
12.4 操作说明	84
12.4.1 下载 Airkiss 调试工具	84
12.4.2 运行现象	84
12.5 注意事项	86
13 Qspi Dcache 模式	87
13.1 Qspi Dcache 简介	87
13.2 Qspi Dcache Related API	87
13.2.1 初始化 qspi 为 dcache 模式	87
13.2.2 启动 qspi 功能	87
13.2.3 停止 qspi 功能	88
13.3 Qspi Dcache 示例代码	88
13.4 操作说明	95
13.4.1 运行现象	95



13.5 注意事项	95
14 低功耗	96
14.1 低功耗简介	96
14.2 低功耗 Related API.....	96
14.2.1 进入低功耗模式	96
14.2.2 deep_sleep 模式	96
14.3 低功耗示例代码.....	97
14.3.1 关键说明	97
14.3.2 示例代码.....	97
14.4 操作说明	99
14.4.1 连接万用表	99
14.4.2 运行现象	100
15 Bootloader	101
15.1 Bootloader 简介	101
15.2 分区表的设置	101
15.2.1 Bootloader 分区	101
15.2.2 App 分区	101
15.2.3 Download 分区	101
15.3 L_boot.....	102
15.4 UP_boot	102
15.5 获取 bootloader.bin 文件.....	102
15.6 生成 all.bin 文件.....	102
15.7 Bootloader 示例代码	103
15.7.1 2M 分区表信息配置文件 partition_audio_2M.json 示例	103
15.7.2 UP_boot 示例.....	104
15.7.3 生成 all.bin 的配置文件 config_sample.json 示例	107
16 网络接口	108
16.1 网络接口简介	108
16.2 网络接口 Related API.....	108
16.2.1 启动网络	108
16.2.2 启动 STATION 快速连接.....	109



16.2.3 关闭网络	110
16.2.4 启动 scan.....	110
16.2.5 注册 scan 结束后的回调函数.....	110
16.2.6 scan 特定的网络.....	110
16.2.7 启动监听模式.....	110
16.2.8 关闭监听模式.....	111
16.2.9 注册监听回调函数	111
16.2.10 获取当前的网络状态	111
16.2.11 获取当前的连接状态	112
16.2.12 获取当前的信道	112
16.2.13 设置信道	113
16.3 网络接口使用示例	113
16.3.1 关键说明	113
16.3.2 代码示例	113
16.4 操作说明	123
16.4.1 启动 STATION 连接.....	123
16.4.2 启动 STATION 快速连接.....	124
16.4.3 STATION 模式获取状态	125
16.4.4 启动 AP.....	125
16.4.5 AP 模式获取状态	125
16.4.6 启动 SCAN	126
16.4.7 启动混杂包监听	127
17 BLE	128
17.1 BLE 简介	128
17.2 BLE_4.2 Related API (通用)	128
17.2.1 启动 ble 协议栈	128
17.2.2 设置 write callback.....	128
17.2.3 设置 read callback.....	128
17.2.4 设置 event callback.....	129
17.2.5 设置接收 advertising callback	129
17.2.6 获取 public key 及 private key.....	129



17.2.7 计算对称密钥.....	129
17.3 BLE_4.2 Related API (slave)	130
17.3.1 开始广播.....	130
17.3.2 关闭广播.....	130
17.3.3 发送数据	130
17.3.4 断开连接.....	131
17.4 BLE_4.2 Related API (master)	131
17.4.1 开始扫描.....	131
17.4.2 停止扫描.....	131
17.4.3 发起连接.....	131
17.4.4 断开连接.....	132
17.4.5 停止连接.....	132
17.4.6 发送数据	132
17.5 BLE_5.1 Related API	132
17.5.1 创建广播.....	132
17.5.2 开始广播	133
17.5.3 停止广播	133
17.5.4 设置广播数据.....	133
17.5.5 更新广播数据.....	133
17.5.6 删除广播	134
17.5.7 创建扫描.....	134
17.5.8 设置扫描请求数据	134
17.5.9 更新连接参数.....	134
17.5.10 断开连接.....	134
17.5.11 发送 notification value.....	135
17.5.12 发送 indification value	135
17.6 BLE 结构体说明.....	135
17.6.1 广播参数.....	135
17.6.2 扫描参数.....	136
17.6.3 ecdh 加密算法结果返回	136
17.6.4 gapc 连接参数	136



17.7 BLE 示例代码.....	137
17.7.1 关键说明	137
17.7.2 示例代码.....	138
17.8 操作说明	148
17.8.1 启动 SCAN	148
18 OTA.....	150
18.1 OTA 简介	150
18.2 OTA Related API	150
18.2.1 fal 初始化	150
18.2.2 远程下载固件	150
18.2.3 freertos 远程下载固件	151
18.3 OTA 示例代码	151
18.3 操作说明	152
18.3.1 生成 rbl 升级文件	152
18.3.2 搭建本地 HTTP Server 环境	152
18.3.3 运行现象	153
19 OS 接口.....	155
19.1 OS 接口简介	155
19.2 OS Related APIs	155
19.2.1 OS 结构体说明	155
19.2.2 创建一个新的线程	156
19.2.3 删除一个使用结束的线程	157
19.2.4 使当前线程挂起，等待另一个线程终止	157
19.2.5 使一个线程挂起一段时间	157
19.2.6 初始化一个信号量	157
19.2.7 发出信号量	157
19.2.8 获取一个信号量，并提供超时机制	158
19.2.9 销毁一个信号量	158
19.2.10 初始化一个互斥锁	158
19.2.11 获得一个互斥锁	158
19.2.12 释放一个互斥锁	158



19.2.13 销毁一个互斥锁	159
19.2.14 初始化一个消息队列	159
19.2.15 将一个数据对象推入消息队列	159
19.2.16 从消息队列中取出一个数据对象	159
19.2.17 销毁一个消息队列	160
19.2.18 查询一个队列是否为空	160
19.2.19 查询一个队列是否已满	160
19.2.20 初始化一个时钟，并传入回调函数	160
19.2.21 启动一个时钟	161
19.2.22 停止一个时钟	161
19.2.23 重新加载一个过期的时钟	161
19.2.24 销毁一个时钟	161
19.2.25 获取一个时钟是否正在运行	161
19.3 RTOS 示例代码	162
19.3.1 关键说明	162
19.3.2 示例代码	163
20 Audio	177
20.1 Audio 简介	177
20.2 Audio Related API	177
20.2.1 audio 通用结构体说明	177
20.2.2 audio 设备初始化	177
20.2.3 打开 mic	178
20.2.4 设置 mic 数据采集通道	178
20.2.5 设置 mic 采样率	178
20.2.6 采集 mic 声音数据	178
20.2.7 关闭 mic	179
20.2.8 打开 audio 设备	179
20.2.9 设置 dac 采样率	179
20.2.10 设置 dac 音量	179
20.2.11 获取 dac 数据，播放音频	179
20.2.12 关闭 dac	180



20.3 Audio 示例代码.....	180
20.3.1 关键说明	180
20.3.2 示例代码.....	180
21 List Player.....	184
21.1 List Player 简介	184
21.2 List Player Related API.....	184
21.2.1 初始化播放器	185
21.2.2 获取当前播放曲目的 handle	185
21.2.3 获取当前播放曲目在列表中的索引	185
21.2.4 获取当前播放器的状态	185
21.2.5 获取当前播放曲目的播放位置	185
21.2.6 获取当前播放列表	186
21.2.7 查询是否有播放器及列表存在	186
21.2.8 播放指定列表	186
21.2.9 切换播放至指定列表	186
21.2.10 播放当前列表指定索引曲目	186
21.2.11 播放指定曲目	187
21.2.12 播放器停止	187
21.2.13 暂停播放	187
21.2.14 播放恢复	187
21.2.15 播放上一曲	187
21.2.16 播放下一曲	188
21.2.17 注销当前播放列表	188
21.2.18 注销并释放当前播放列表	188
21.2.19 设置播放器模式	188
21.2.20 建立播放列表	188
21.2.21 删除播放列表	189
21.2.22 回调注册	189
21.2.23 获取播放列表中曲目数	189
21.2.24 获取播放列表中最近播放曲目索引	189
21.2.25 获取播放列表中最近播放曲目	189



21.2.26 曲目添加	190
21.2.27 曲目删除	190
21.2.28 指定索引曲目删除	190
21.2.29 获取指定索引曲目	190
21.2.30 获取指定曲目索引	191
21.3 List Player 示例代码	191
21.3.1 关键说明	191
21.3.2 示例代码	191
21.4 操作说明	194
21.4.1 打开配置	194
21.4.2 运行现象	194
22 混音	196
22.1 混音简介	196
22.2 混音 Related API	196
22.2.1 混音初始化	196
22.2.2 暂停背景音播放	196
22.2.3 重新播放背景音	196
22.3 混音示例代码	197
22.3.1 关键说明	197
22.3.2 示例代码	197
22.4 操作说明	198
22.4.1 打开配置	198
22.4.2 运行现象	198
23 Vad	199
23.1 Vad 自动语音检测简介	199
23.2 Vad Related API	199
23.2.1 进入 vad 检测模式	199
23.2.2 获取帧的长度	199
23.2.3 vad 入口函数	199
23.2.4 关闭 vad	200
23.3 Vad 示例代码	200



23.4 操作说明	203
23.4.1 打开配置	203
23.4.2 运行现象	203
24 AMR 编码器	205
24.1 AMR 编码器简介	205
24.2 AMR 编码器 Related API	205
24.2.1 AMR-NB 编码器初始化	205
24.2.2 AMR-NB 编码	205
24.2.3 释放 AMR-NB 编码	206
24.3 AMR 编码器示例代码	206
24.3.1 关键说明	206
24.3.2 示例代码	206
24.4 操作说明	214
24.4.1 下载 AMR Player 工具	214
24.4.2 网络调试助手设置	214
24.4.3 打开配置	214
24.4.4 运行现象	215
25 Opus 编码器	216
25.1 Opus 编码器简介	216
25.2 Opus 编码器 Related API	216
25.2.1 创建 opus 编码器	216
25.2.2 返回 opus 编码器所需内存的大小	216
25.2.3 修改 opus 编码器的复杂度	217
25.2.4 获取 opus 编码器的比特率	217
25.2.5 获取 opus 编码器的最终状态	217
25.2.6 opus 编码	217
25.2.7 释放 opus 编码器对象	218
25.3 Opus 编码器示例代码	218
25.3.1 关键说明	218
25.3.2 示例代码	218
25.4 操作说明	226



25.4.1 下载 Cool Edit Pro 工具	226
25.4.2 网络调试助手设置	226
25.4.3 打开配置	226
25.4.4 运行现象	227
26 EasyFlash	228
26.1 EasyFlash 简介	228
26.2 EasyFlash Related API	228
26.2.1 easyflash 初始化	228
26.2.2 获得 easyflash 环境变量	228
26.2.3 将数据写入到环境变量中	228
26.2.4 保存数据到 flash	229
26.3 EasyFlash 示例代码	229
26.3.1 关键说明	229
26.3.2 示例代码	229
26.4 操作说明	232
26.4.1 打开配置	232
26.4.2 运行现象	232
27 Voice Changer	234
27.1 Voice Changer 简介	234
27.2 Voice Changer Related API	234
27.2.1 voice changer 初始化	234
27.2.2 退出 voice changer	234
27.2.3 开始 voice changer	234
27.2.4 停止 voice changer	235
27.2.5 设置 voice changer 变声功能标志	235
27.2.6 voice changer 获取 mic 数据	235
27.2.7 设置消耗数据的长度	235
27.2.8 处理数据	235
27.3 Voice Changer 示例代码	236
27.3.1 关键说明	236
27.3.2 示例代码	236



27.4 操作说明	244
27.4.1 运行现象	244
28 声波配网	245
28.1 声波配网简介	245
28.2 声波配网 Related API	245
28.2.1 声波配网开始	245
28.2.2 用户提前终止声波配网	246
28.2.3 获取版本	246
28.3 声波配网示例代码	246
28.4 操作说明	249
28.4.1 打开配置	249
28.4.2 运行现象	249
29 图像传输	251
29.1 图像传输简介	251
29.2 图像传输 Related API	251
29.2.1 打开 video_transfer	252
29.2.2 关闭 video_transfer	252
29.2.3 设置摄像头的参数	252
29.2.4 打开获取 jpeg 帧的功能	253
29.2.5 关闭获取 jpeg 帧的功能	253
29.2.6 获取 jpeg 帧的数据	253
29.3 图像传输的示例代码	253
29.3.1 关键说明	253
29.3.2 示例代码	254
29.4 操作说明	258
29.4.1 下载 PC 调试工具	259
29.4.2 启动 softap	259
29.4.3 UDP 传输测试	259
29.4.4 TCP 传输测试	260





1 ADC

1.1 ADC简介

BEKEN WiFi Soc 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, bk7251 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需要以对应的wifi soc gpio映射表为准，请注意查看相应芯片手册。

1.2 ADC Related API in

RTT OS中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

Freertos中开启和关闭为freertos标准接口，ADC相关接口如下:

函数	描述
saradc_config_param_init()	adc参数初始化
ddev_open()	打开adc检测功能
ddev_close();	关闭adc检测功能

1.2.1 adc通用结构体说明

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

1.2.2 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

1.2.3 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

1.2.4 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

1.2.5 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示例代码

ADC示例代码参考test\adc_test.c，本示例仅在rtt os中实现，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 <rtdef.h>
#include "saradc_intf.h"
#include "sys_ctrl_pub.h"

#define CONFIG_ADC_TEST
#ifndef 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

BEKEN WiFi Soc具有6路PWM输出，每一路的周期及占空比都可以单独配置，bk7251 channel 如下。

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: 此表格是bk7251为例， wifi 系列 soc 中 pwm通道对应GPIO需要以对应的wifi soc gpio映射表为准，请注意查看相应芯片手册。

2.2 PWM Related API

7231u/7251 pwm相关接口参考beken378\func\user_driver\BkDriverPwm.h，相关接口如下：

函数	描述
<code>bk_pwm_initialize()</code>	PWM初始化
<code>bk_pwm_start()</code>	启动PWM功能
<code>bk_pwm_stop()</code>	停止PWM功能

7231n pwm api 如下：

函数	描述
<code>bk_pwm_initialize()</code>	PWM初始化
<code>bk_pwm_start()</code>	启动PWM功能
<code>bk_pwm_stop()</code>	停止PWM功能
<code>bk_pwm_group_initialize()</code>	开启2组互斥的pwm channel
<code>bk_pwm_update_param()</code>	更新pwm channel 频率和占空比
<code>bk_pwm_group_mode_disable()</code>	停止pwm互斥功能
<code>bk_pwm_initlevel_set_low()</code>	设置pwm channel 初始电平为低电平
<code>bk_pwm_initlevel_set_high()</code>	设置pwm channel 初始电平为高电平

2.2.1 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

2.2.2 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

2.2.3 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

2.2.4 停止pwm功能

```
OSStatus 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

bk7231n pwm api 说明

2.2.5 7231n初始化pwm

```
OSStatus bk_pwm_initialize(bk_pwm_t pwm, uint32_t frequency, uint32_t  
duty_cycle1,uint32_t duty_cycle2,uint32_t duty_cycle3);
```



参数	描述
pwm	选择的pwm通道: 0 ~ 5
frequency	pwm 频率
duty_cycle1	pwm的占空比
duty_cycle2	pwm第二次电平翻转
duty_cycle3	pwm第三次电平翻转
返回	0: 成功; -1: 错误

2.2.6 设置2个pwm组成互斥通道

```
OSStatus bk_pwm_group_initialize(bk_pwm_t pwm1, bk_pwm_t pwm2, uint32_t frequency,  
uint32_t duty_cycle1, uint32_t duty_cycle2, uint32_t dead_band);
```

参数	描述
pwm1	pwm互斥通道1
pwm2	pwm互斥通道2
frequency	2个互斥pwm频率
duty_cycle1	互斥pwm的占空比
duty_cycle2	pwm翻转时间,默认一般为0
dead_band	pwm互斥死区时间
返回	0: 成功; -1: 错误

2.2.7 调节pwm参数

```
OSStatus bk_pwm_update_param(bk_pwm_t pwm, uint32_t frequency, uint32_t duty_cycle1,  
uint32_t duty_cycle2, uint32_t duty_cycle3);
```

参数	描述
pwm	pwm通道
frequency	2个互斥pwm频率
duty_cycle1	互斥pwm的占空比
duty_cycle2	pwm第2次翻转时间,默认一般为0
duty_cycle3	pwm第3次翻转时间,默认一般为0
返回	0: 成功; -1: 错误

2.2.8 停止pwm互斥功能

```
OSStatus bk_pwm_group_mode_disable(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

2.2.9 设置pwm初始电平为低电平

```
OSStatus bk_pwm_initlvl_set_low(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

2.2.10 设置pwm初始电平为高电平

```
OSStatus bk_pwm_initlvl_set_high(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示例代码

rtt os中示例代码参考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
#endif      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
```

2.4 操作说明

2.4.1 打开配置

bk7251示例代码参考test\pwm_test.c, 打开宏定义: CONFIG_PWM_TEST, 编译完成后, 将固件下载至设备。

2.4.2 运行现象

串口输入命令 :pwm_test <channel> <duty_cycle> <cycle>即可在对应的gpio上检测到波形。分别输入channel:1~5的命令, 即可用逻辑分析仪看到相应的gpio输出周期性的方波。如下图所示



图2.4.2-1: 5个pwm channel同时输出波形图

2.5 注意事项

- pwm channel0 已经被cpu用来做timer，所以其pwm功能不能使用。
- pwm输出的时候，其时钟源选择的是26M。



3 GPIO

3.1 GPIO简介

BEKEN wifi soc 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）。

3.2 GPIO Related API

RTT OS中gpio接口位于\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

Freertos中ADC相关接口如下：

函数	描述
<code>BkGpioInitialize ()</code>	gpio初始化
<code>BkGpioOutputHigh ()</code>	gpio输出高电平
<code>BkGpioOutputLow ()</code>	gpio输出低电平
<code>BKGpioOp ()</code>	设置gpio模式
<code>BkGpioInputGet ()</code>	获取gpio输入值
<code>BkGpioEnableIRQ ()</code>	使能gpio中断
<code>BkGpioDisableIRQ()</code>	关闭gpio中断

3.2.1 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



3.2.2 设置引脚电平

设置引脚输出电平的函数如下所示:

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

参数	描述
pin	引脚编号
value	电平逻辑值, 可取2种宏定义值之一: PIN_LOW 低电平, PIN_HIGH 高电平
返回	空

3.2.3 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

3.2.4 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

3.2.5 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

3.2.6 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。

3.2.6 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

3.3 GPIO示例代码

示例代码的主要步骤如下：

1. 初始化LED控制引脚，输出高电平，点亮LED2和LED3。
2. 初始化S3和S4按键，设置下降沿方式触发中断并使能。

3.3.1 关键说明

- **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

3.3.2 示例代码

```
/*
 * 程序清单： 这是一个PIN 设备使用例程 本示例仅供7251 rtt os参考
 * 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>
#include "test_config.h"

#ifndef GPIO_DEMO
#define LED_PIN_NUM 24
#define LED1_PIN_NUM 26
#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_HIGH);
    /* 按键0引脚为输入模式*/
    rt_pin_mode(KEY0_PIN_NUM , PIN_MODE_INPUT_PULLUP);
}
```

```
/* 绑定中断， 下降沿模式， 回调函数名为beep_on */
rt_pin_attach_irq(KEY0_PIN_NUM, PIN_IRQ_MODE_FALLING, led_on, RT_NULL);
/* 使能中断*/
rt_pin_irq_enable(KEY0_PIN_NUM, PIN_IRQ_ENABLE);
/* 按键1引脚为输入模式*/
rt_pin_mode(KEY1_PIN_NUM, PIN_MODE_INPUT_PULLUP);
rt_pin_attach_irq(KEY1_PIN_NUM, PIN_IRQ_MODE_FALLING, led_off, RT_NULL);
rt_pin_irq_enable(KEY1_PIN_NUM, PIN_IRQ_ENABLE);
rt_pin_mode(LED1_PIN_NUM, PIN_MODE_OUTPUT);
/* 默认低电平*/
rt_pin_write(LED1_PIN_NUM, PIN_HIGH);
}
/* 导出到msh 命令列表中*/
MSH_CMD_EXPORT(pin_led_sample, pin led sample);
#endif
```

3.4 操作说明

GPIO设备示例代码位于\test\gpio_demo.c，SDK默认没有打开此功能，需要打开功能后测试，在调试串口输入触发命令使能GPIO Demo，然后操作S3和S4按键控制LED灯，按下S3按键，LED2熄灭，按下S4按键，LED2重新点亮。

3.4.1 打开配置

打开宏定义：GPIO_DEMO，编译下载后，调试串口输入pin_led_sample，LED2和LED3点亮。



图3.4.1-1

3.4.2 运行现象

按下S3按键，LED2熄灭：



图3.4.1-2

4 Button

4.1 Button简介

按键功能包含有按键长按，短按，双击等功能。

4.2 Button Realated API

button相关接口函数参考beken378/func/key/multi_button.h, 相关接口如下:

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

4.2.1 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
BUTTON_S* handle	按键句柄
uint8_t(*pin_level)	读取HAL gpio
uint8_t active_level	gpio level
void *user_data	用户数据
返回	空

4.2.2 配置回调函数

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

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



4.2.3 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

4.2.4 结束button工作

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

参数	描述
BUTTON_S* handle	button 句柄
返回	空

4.3 Button示例代码

4.3.1 关键说明

- **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;
```

4.3.2 示例代码

```
/*  
 * 程序清单: 这是一个按键使用例程 仅供在RTT OS参考  
 * 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
#ifndef 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
```

4.4 操作说明

Button示例代码参考\samples\key\button_test.c，使能后支持按键的短按、双击、长按功能。

4.4.1 打开配置

打开宏定义：BUTTON_TEST，编译下载后，调试串口输入button_test 3，使能S3按键，设备log如下：

```
---gpio3 as button : test start---
```

```
msh />button_short_press
```

4.4.2 按键测试

- 短按测试

连续三次短按S3按键(大于300ms小于1s)，设备log如下：

```
---gpio3 as button : test start---
```

```
msh />button_short_press
```

```
button_short_press
```

```
button_short_press
```

```
button_short_press
```

- 长按测试

长按S3按键(大于1s)，设备log如下：

```
msh />button_test 3
```

```
---gpio3 as button : test start---
```

```
msh />button_short_press
```

```
button_short_press
```

```
button_short_press
```

```
button_short_press
```

```
button_long_press_hold
```

```
button_long_press_hold
```

```
button_long_press_hold
```



button_long_press_hold

- 双击测试

双击S3按键，设备log如下：

```
msh />button_test 3
---gpio3 as button : test start---
msh />button_short_press
button_short_press
button_short_press
button_short_press
button_long_press_hold
button_long_press_hold
button_long_press_hold
button_long_press_hold
button_double_press
button_double_press
button_double_press
```



5 I2C总线

5.1 I2C 简介

BEKEN wifi soc设有I2C模块,i2c1只能在master模式下工作，i2c2 在master和slave模式下都可以传输数据。

5.2 I2C Related API

I2c1/i2c2已经关联到 RT-thread/Freertos操作系统的标准设备操作函数集了，所以直接调用RT-thread/Freertos标准设备操作接口进行操作

i2c2_msg:

TxMode	发送模式
WkMode	工作模式
InnerAddr	slave 片上地址
SendAddr	master发送地址
CurrentNum	当前数据
AllDataNum	传输数据总和
*pData	传输的数据
Slave_addr	从设备地址
AddrFlag	地址标志
TransDone	传输完成标志
ack_check	ack 标志

i2c_op_st:

slave_id	从设备id
op_addr	操作地址
slave_addr	从设备地址
mode	传输模式

5.3 I2C1读写eeprom数据示例如下：

示例代码位于\test\i2c_test.c。打开宏定义：I2C_TEST，开启i2c功能测试。

```
/*
* 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_eeprom
* 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eprom。
*/
#include "include.h"
#include <rtthread.h>
#include <rthw.h>
#include <rtdevice.h>
#include <string.h>
#include "icu_pub.h"
#include "i2c_pub.h"
#include "drv_model_pub.h"
#include "target_util_pub.h"
#include "test_config.h"

#ifndef I2C_TEST

#define I2C_EEPROM_DEBUG
#define I2C_EEPROM_DEBUG
#define I2C_EEPROM_PRT      os_printf
#define I2C_EEPROM_WARN     warning_prf
#define I2C_EEPROM_FATAL   fatal_prf
#else
#define I2C_EEPROM_PRT      null_prf
#define I2C_EEPROM_WARN     null_prf
#define I2C_EEPROM_FATAL   os_printf
#endif

#define I2C1          0
#define I2C2          1
#define I2C_DEV_ID    I2C1
#define I2C_SALVE_ID  0x21

static DD_HANDLE i2c_hdl;

static void i2c_device_init()
{
    unsigned int oflag,status;
    oflag = 0;
```



```
#if I2C_DEV_ID
    i2c_hdl = ddev_open(I2C2_DEV_NAME, &status, oflag);
#else
    i2c_hdl = ddev_open(I2C1_DEV_NAME, &status, oflag);
#endif
}

static void i2c_device_deinit()
{
    ddev_close(i2c_hdl);
}

/******************
 * I2C1_write_eeprom
 * Description: I2C1 write FT24C02 eeprom
 * Parameters: op_addr: operate address
 *             pData: data point
 *             len: data len
 * return:     unsigned long
 * error:      none
 */
static unsigned long I2C_write_eeprom(unsigned char op_addr, unsigned char *pData, unsigned char len)
{
    unsigned char i;
    unsigned int status;
    I2C_OP_ST i2c_op;

    I2C_EEPROM_PRT("---- I2C1_write_eeprom start ----\r\n");

    i2c_op.op_addr = op_addr;
    i2c_op.salve_id = I2C_SALVE_ID;

    do
    {
        status = ddev_write(i2c_hdl, pData, len, (unsigned long)&i2c_op);
```



```
        } while (status != 0);

        I2C_EEPROM_PRT("---- I2C1_write_eeprom over ----\r\n");
        return 0;
    }

/*****************/
/* I2C1_read_eeprom
 * Description: I2C1 read FT24C02 eeprom
 * Parameters:  op_addr: operate address
 *              pData: data point
 *              len: data len
 * return:      unsigned long
 * error:       none
 */
static unsigned long I2C_read_eeprom(unsigned char op_addr, unsigned char *pData, unsigned char len)
{
    unsigned char i;
    DD_HANDLE i2c_hdl;
    unsigned int status;
    I2C_OP_ST i2c_op;

    I2C_EEPROM_PRT("---- I2C1_read_eeprom start ----\r\n");

    i2c_op.op_addr = op_addr;
    i2c_op.salve_id = I2C_SALVE_ID;
    do
    {
        status = ddev_read(i2c_hdl, pData, len, (unsigned long)&i2c_op);
    } while (status != 0);

    for (i=0; i<8; i++)
    {
        I2C_EEPROM_PRT("pData[%d] = 0x%x\r\n", i, pData[i]);
    }

    I2C_EEPROM_PRT("---- I2C1_read_eeprom over ----\r\n");
```



```
        return status;
    }

/*****
 * I2C1_test_eeprom
 * Description: I2C1 test FT24C02 eeprom
 * Parameters:  none
 * return:      unsigned long
 * error:       none
 */
static unsigned long i2c_test_eeprom(void)
{
    int i, j;
    DD_HANDLE i2c_hdl;
    unsigned char pReadData[8];
    unsigned char pWriteData[8];

    i2c_device_init();
    I2C_EEPROM_PRT("----- I2C1_test_eeprom start -----\\r\\n");

    for (j=0; j<100; j++)
    {
        delay_ms(100);

        for (i=0; i<8; i++)
        {
            pWriteData[i] = (i << 2) + 0x01 + j;
        }
        I2C_write_eeprom(0x00+j*8, pWriteData, 8);

        delay_ms(100);

        memset(pReadData, 0, 8);
        I2C_read_eeprom(0x00+j*8, pReadData, 8);

        if (memcmp(pReadData, pWriteData, 8) == 0)
        {
            os_printf("I2C_test_eeprom: memcmp %d ok!\\r\\n", j);
        }
    }
}
```



```
else
{
    I2C_EEPROM_FATAL("I2C_test_eeprom: memcmp %d error!\r\n", j);
    for (i=0; i<8; i++)
    {
        I2C_EEPROM_FATAL("pReadData[%d]=0x%x, pWriteData[%d]=0x%x\r\n",
                          i, pReadData[i], i, pWriteData[i]);
    }
}

I2C_EEPROM_PRT("----- i2c_test_eeprom over ----- \r\n");
i2c_device_deinit();
return 0;
}

MSH_CMD_EXPORT(i2c_test_eeprom,i2c_test_eeprom);

#endif
```



6 I2S总线

6.1 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和Right justified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。

6.2 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

6.2.1 i2s通用结构体说明

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

6.2.2 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

6.2.3 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

6.3 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。

6.3.1 关键说明

- 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）
```

6.3.2 示例代码

```
/*
 * 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，示例仅供在rtt os中参考。
 * 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 receive */
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], 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],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);
```



7 通用SPI

7.1 通用SPI简介

BEKEN wifi soc 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) 时针极性（CPOL）和时针相位（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）。

7.2 通用SPI Related API

通用SPI的驱动只适配了RT-thread操作系统的标准设备操作函数，Freertos 中没有此类的api接口，所以在RT-thread中的相关接口参考
rt-thread/components\drivers\include\drivers\spi.h。

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

7.2.1 spi通用结构体说明

`rt_spi_device:`

<code>parent</code>	spi device对象
<code>bus</code>	spi bus句柄
<code>config</code>	spi 模式配置的结构体

7.2.2 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
<code>device</code>	SPI设备接口的指针
<code>cfg</code>	SPI配置结构体，见如下说明
返回	RT_EOK(0): 成功； 其他：出错



参数类型

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

7.2.3 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。

7.2.4 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。

7.3 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。

7.3.1 关键说明

- **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

7.3.2 示例代码

```
/*
 * 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 */
#ifndef ((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!\r\n", "gspi");
        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("gspi_test master/slave    tx/rx    rate    len\r\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\r\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_recv(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("gspi_test master/slave    tx/rx  rate  len\r\n");
}
}

MSH_CMD_EXPORT(gspi_test, gspi_test);
```

7.4 操作说明

7.4.1 打开配置

sys_config.h, general_spi_test.c文件中，设置如下：

#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

7.4.2 运行现象

烧录完成后，接到在串口工具中，发送cmd命令。
gspi_test master tx 1000000 128, 1M速率发送128字节数据

```
gspi_test master tx 1000000 128
cfg:8, 0x04, 1000000
[SPI]:data_width = 8
[SPI]:max_hz = 1000000, mode:0x04
max_hz = 1000000
config spi clk source 26MHz
div = 12
spi_clk = 1000000
source_clk = 26000000
target frequency = 1000000, actual frequency = 1000000
[CTRL]:0x00c30c3f
spi init tx_len:128
00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f,10,11,12,13,14,15,16,17,18,19,1a,1b,1c,1d,1e,1f,
20,21,22,23,24,25,26,27,28,29,2a,2b,2c,2d,2e,2f,30,31,32,33,34,35,36,37,38,39,3a,3b,3c,3d,3e,3f,
40,41,42,43,44,45,46,47,48,49,4a,4b,4c,4d,4e,4f,50,51,52,53,54,55,56,57,58,59,5a,5b,5c,5d,5e,5f,
60,61,62,63,64,65,66,67,68,69,6a,6b,6c,6d,6e,6f,70,71,72,73,74,75,76,77,78,79,7a,7b,7c,7d,7e,7f.
```

图7.4.1

接着发送gspi_test master rx 1000000 128, 1M速率接收128字节数据

```
nsh />
nsh />gspi_test master rx 1000000 128
cfg:8, 0x04, 1000000
[SPI]:data_width = 8
[SPI]:max_hz = 1000000, mode:0x04
max_hz = 1000000
config spi clk source 26MHz
div = 12
spi_clk = 1000000
source_clk = 26000000
target frequency = 1000000, actual frequency = 1000000
[CTRL]:0x00c30c3f
spi init rx_len:128
rx ret:128
00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f,10,11,12,13,14,15,16,17,18,19,1a,1b,1c,1d,1e,1f,
20,21,22,23,24,25,26,27,28,29,2a,2b,2c,2d,2e,2f,30,31,32,33,34,35,36,37,38,39,3a,3b,3c,3d,3e,3f,
40,41,42,43,44,45,46,47,48,49,4a,4b,4c,4d,4e,4f,50,51,52,53,54,55,56,57,58,59,5a,5b,5c,5d,5e,5f,
60,61,62,63,64,65,66,67,68,69,6a,6b,6c,6d,6e,6f,70,71,72,73,74,75,76,77,78,79,7a,7b,7c,7d,7e,7f,
```

图7.4.2

7.5 注意事项

- SPI的最高速率为30M，超过这个之后，会自动被限制在30M。
- Rate参数，单位不是M，比如1M，需要写1 000 000。



8 通用SPI FLASH设备

8.1 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。

8.2 通用SPI FLASH Related API

通用SPI的驱动只适配了RT-thread操作系统的标准设备操作函数，Freertos中没有此类的api接口。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

8.2.1 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

8.3 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。

8.3.1 关键说明

- **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



8.3.2 示例代码

```
/*
 * 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"
#ifndef 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(gsipi_flash_test, gsipi_flash_test);
#endif // BEKEN_USING_SPI_FLASH
```

8.4 操作说明

初始FLASH为空，然后写入数据，读出数据，读到的数据与写入数据相同，最后擦除数据。

8.4.1 运行现象

在串口输入gsipi_flash_test即可启动该项功能，设备log如下：



图8.4.1-1

8.5 注意事项

- 需要外接**FLASH**模块进行。
 - 擦除后重新读取内容，如果全是0xFF，则说明已经擦除成功。



9 通用SPI PSRAM设备

9.1 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。

9.2 通用SPI PSRAM related API

通用SPI的驱动只适配了RT-thread操作系统的标准设备操作函数，Freertos中没有此类的api接口。

9.3 通用SPI PSRAM示例代码

9.3.1 关键说明

- SPI PSRAM宏定义：

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

9.3.2 示例代码

```
/*
 * 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"

#ifndef 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
```



```
#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;
    }
    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
```

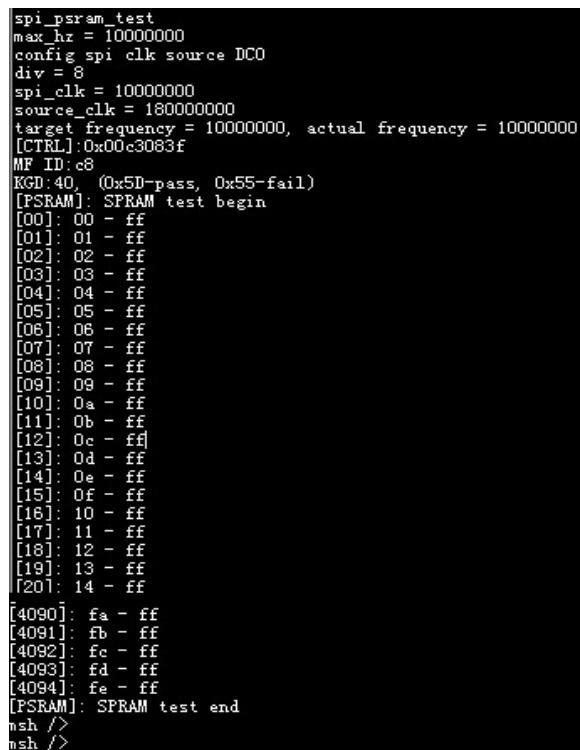
9.4 操作说明

9.4.1 打开配置

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

9.4.2 运行现象

在串口输入spi_psram_test启动此项功能，设备log如下图：



spi_psram_test
max_hz = 10000000
config spi clk source DCO
div = 8
spi_clk = 10000000
source_clk = 180000000
target frequency = 10000000, actual frequency = 10000000
[CTRLL:0x00c3083f
MF ID:c8
KGD:40, (0x5D-pass, 0x55-fail)
[PSRAM]: SPRAM test begin
[00]: 00 - ff
[01]: 01 - ff
[02]: 02 - ff
[03]: 03 - ff
[04]: 04 - ff
[05]: 05 - ff
[06]: 06 - ff
[07]: 07 - ff
[08]: 08 - ff
[09]: 09 - ff
[10]: 0a - ff
[11]: 0b - ff
[12]: 0c - ff
[13]: 0d - ff
[14]: 0e - ff
[15]: 0f - ff
[16]: 10 - ff
[17]: 11 - ff
[18]: 12 - ff
[19]: 13 - ff
[20]: 14 - ff
[4090]: fa - ff
[4091]: fb - ff
[4092]: fc - ff
[4093]: fd - ff
[4094]: fe - ff
[PSRAM]: SPRAM test end
nsh />
nsh />

图9.4.2-1



9.5 注意事项

- 需要外接SRAM模块进行。

10 高速SPI从设备

10.1 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。

10.2 高速SPI从设备Related API

通用SPI的驱动只适配了RT-thread操作系统的标准设备操作函数，Freertos中没有此类的api接口。

10.3 高速SPI从示例代码

10.3.1 关键说明

- 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

10.3.2 示例代码

```
/*
* 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)
#ifndef BEKEN_USING_SPI_HSLAVE

/*SPI HS需要关联BEKEN_USING_SPI_HSLAVE、 CFG_USE_HSLAVE_SPI 二个功能宏*/
#ifndef CFG_USE_HSLAVE_SPI
#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
```

10.4 操作说明

10.4.1 打开配置

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

10.4.2 运行现象

编译运行后，在调试串口输入spi hs test tx 100，接收100字节数据

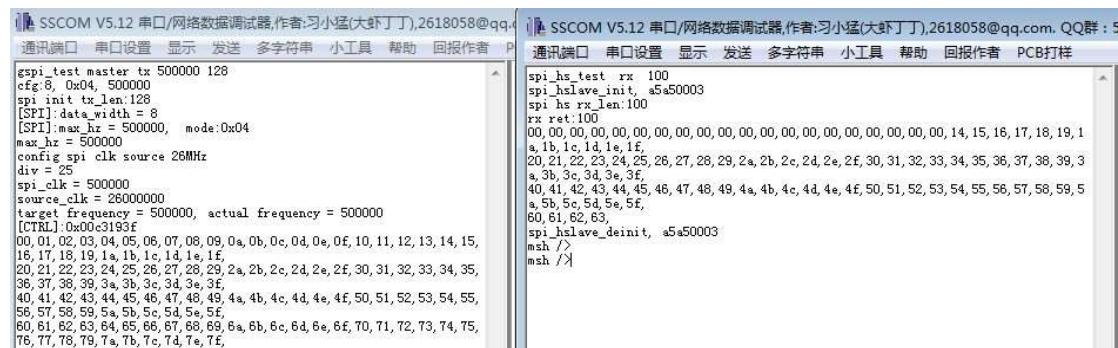


图 10.4.2-1

10.5 注意事项

- 速率最高为50M。



11 UART

11.1 UART简介

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

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

11.2 UART Related API

Uart 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

Freertos中的uart接口位于/beken378/func/user_driver目录下，相关api接口如下：

函数	描述
<code>bk_uart_initialize ()</code>	uart初始化
<code>bk_uart_send()</code>	uart发送数据
<code>bk_uart_recv()</code>	uart接收数据
<code>bk_uart_set_rx_callback()</code>	接收回调函数

11.2.1 uart通用结构体说明

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
<code>rt_uint32_t baud_rate</code>	波特率设置：一般为115200
<code>rt_uint32_t data_bits</code>	数据位：一般为8bit
<code>rt_uint32_t stop_bits</code>	停止位：一般为1
<code>rt_uint32_t parity</code>	奇偶校验位：无校验位
<code>rt_uint32_t bit_order</code>	大小端：一般为小端



<code>rt_uint32_t invert</code>	模式转化: 不转换
<code>rt_uint32_t bufsz</code>	接收buffer大小
<code>rt_uint32_t reserved</code>	保留

bk_uart_t:

结构体类型	成员
<code>BK_UART_1</code>	Uart1
<code>BK_UART_2</code>	uart2

bk_uart_config_t:

结构体类型	成员
<code>data_width</code>	速率
<code>parity</code>	校验位
<code>stop_bits</code>	停止位
<code>flow_control</code>	流控
<code>flags</code>	标志位

11.2.2 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	<code>RT_EOK</code> 函数执行成功 <code>-RT_ENOSYS</code> 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。

Freertos中的uart 相关接口如下：



11.2.3 uart 初始化

```
OSStatus bk_uart_initialize( bk_uart_t uart, const bk_uart_config_t *config, ring_buffer_t  
*optional_rx_buffer );
```

参数	描述
uart	串口设备号
config	串口设置结构体
optional_rx_buffer	串口操作的buffer
返回	0: 函数执行成功 非0: 执行失败

11.2.4 uart 发送数据

```
OSStatus bk_uart_send( bk_uart_t uart, const void *data, uint32_t size );
```

参数	描述
uart	串口设备号
data	串口发送的数据
size	串口发送数据大小
返回	0: 函数执行成功 非0: 执行失败

11.2.5 uart 接收数据

```
OSStatus bk_uart_recv( bk_uart_t uart, const void *data, uint32_t size );
```

参数	描述
uart	串口设备号
data	串口接收的数据
size	串口接收数据大小
返回	0: 函数执行成功 非0: 执行失败

11.2.6 uart 接收回调函数

```
bk_uart_set_rx_callback(bk_uart_t uart, uart_callback callback, void *param);
```



参数	描述
uart	串口设备号
callback	串口接收回调函数
param	接收参数
返回	0: 函数执行成功 非0: 执行失败

11.3 UART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。

11.3.1 关键说明

• 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

设备配置宏定义：

#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

11.3.2 示例代码

```
#include <rtthread.h>
#include "test_config.h"
#include <rtdevice.h>
#ifndef UART_DEMO
#define SAMPLE_UART_NAME "uart1"
/* 用于接收消息的信号量 */
static struct rt_semaphore rx_sem;
static rt_device_t serial;
```



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

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

static int uart_sample(int argc, char *argv[])
{
    rt_err_t ret = RT_EOK;
    char uart_name[RT_NAME_MAX];
    char str[] = "hello BK72xx!\r\n";
    struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT;
    if (argc == 2) {
        rt_strncpy(uart_name, argv[1], RT_NAME_MAX);
    }
    else {
        rt_strncpy(uart_name, SAMPLE_UART_NAME, RT_NAME_MAX);
    }
    serial = rt_device_find(uart_name);
```



```
if (!serial) {
    rt_kprintf("find %s failed!\n", uart_name);
    return RT_ERROR;
}

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);

/* 设置配置参数 */
config.baud_rate = BAUD_RATE_115200;
config.data_bits = DATA_BITS_8;
config.stop_bits = STOP_BITS_1;
config.parity = PARITY_NONE;
config.bufsz = 2048;      /*can not change buffer size, must be 2048*/
/* 打开设备后才可修改串口配置参数 */
rt_device_control(serial, RT_DEVICE_CTRL_CONFIG, &config);
rt_device_write(serial, 0, str, (sizeof(str) - 1));
rt_thread_t thread = rt_thread_create("serial", serial_thread_entry , RT_NULL,1024, 25, 10);
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);
#endif
```

11.4 操作说明

uart示例代码位于\test\uart_demo.c，修改配置信息后，可测试uart1的通信功能。

11.4.1 打开配置

打开宏定义：UART_DEMO，重新编译完成后，将固件下载至设备。

11.4.2 运行现象

- 硬件连接

串口转USB模块一端连接串口UART1，另一端插入PC。

- 运行

调试串口输入uart_sample, UART1会发送hello BK72xx!, PC上串口助手收到数据后，发送0x40给设备UATRT1,然后设备回复0x41,运行情况如下图所示：

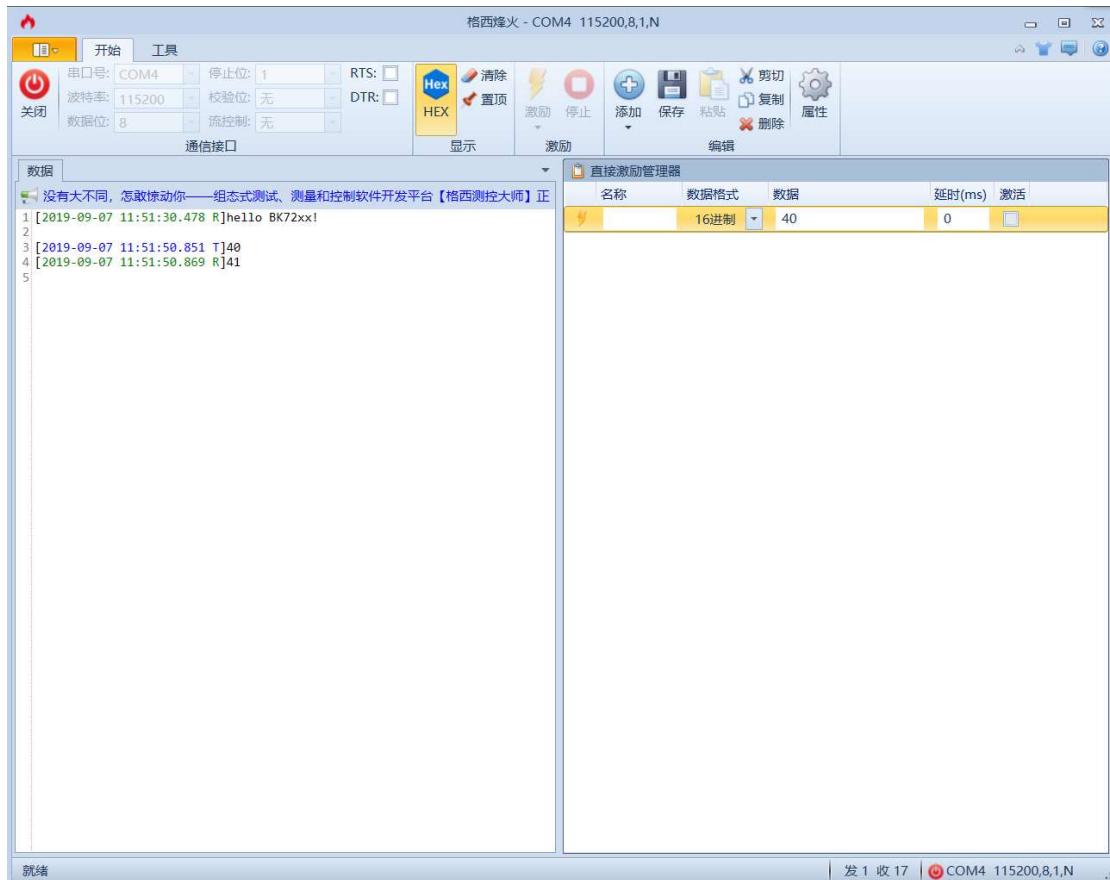


图11.4.2-1

11.5 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。



12 Airkiss

12.1 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。

12.2 Airkiss Related API

RTT OS 中airkiss相关接口参考\samples\airkiss\airkiss.h，相关接口如下：

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

Freertos中airkiss相关接口如下：

函数	描述
airkiss_process ()	开始airkiss

12.2.1 开始airkiss

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

参数	描述
void	空
返回	0:成功；其他：失败

12.2.2 获取airkiss状态

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

参数	描述
void	空
返回	airkiss状态

12.2.3 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

Freertos中airkiss 说明:

12.2.4 开始airkiss

```
u32 airkiss_process(u8 start);
```

参数	描述
<code>start</code>	1:启动 ; 0: 停止
返回	0:成功; 其他: 失败

`airkiss_result_t:`

<code>char *pwd</code>	wifi密码
<code>char *ssid</code>	wifi ssid
<code>unsigned char pwd_length</code>	wifi密码长度
<code>unsigned char ssid_length</code>	wifi ssid长度
<code>unsigned char random</code>	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
<code>unsigned char reserved</code>	保留值

12.3 Airkiss示例代码

12.3.1 关键说明

- **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;
```



12.3.2 示例代码

```
/*
 * 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_mngt.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[])
{
    if(g_cfg_done_sem == RT_NULL)
    {
        if(1 == airkiss())
        {
            rt_kprintf("airkiss start\r\n");
            rt_thread_delay(rt_tick_from_millisecond(1000));
            while(g_cfg_done_sem)
            {
                uint32_t res;
                res = airkiss_get_status();
                if(res == AIRKISS_STATUS_COMPLETE)
                {
                    airkiss_result_t *result;
                    result = airkiss_result_get();
                    rt_kprintf("---ssid:%s , key:%s---\r\n", result->ssid,result->pwd);
                    break;
                }
            }
            rt_thread_delay(rt_tick_from_millisecond(100));
        }
    }
}
```

```
    }
}

else
    rt_kprintf("airkiss fail\r\n");
}

#endif FINSH_USING_MSH
#include "finsh.h"

MSH_CMD_EXPORT(start_airkiss, start_arksss);
#endif
```

12.4 操作说明

示例代码位于\samples\airkiss目录下，SDK默认已经打开此功能，需要在调试串口输入触发命令使设备进入Airkiss配网模式，然后操作APP进行配网。

12.4.1 下载Airkiss调试工具

微信官方Airkiss调试工具：[下载地址](#)

进入下载页面后，下载下图所示工具：

WiFi设备

AirKiss技术简介：[下载](#)

AirKiss调试工具：[下载](#)

AirLink调试工具：[下载](#)

图12.4.1-1

12.4.2 运行现象

- 设备触发配网

编译下载运行后，在调试串口输入命令start_airkiss，程序运行日志如下所示：

```
airkiss start
AirKiss version: airkiss-2.0.0-25360(Dec 17 2015 17:20:50);arm-none-eabi/gcc-4.9.3;ARM
[DRV_WLAN]set monitor callback
Drv_start monitor
Soft_AP_start
[saap]MM_RESET_REQ
[saap]ME_CONFIG_REQ
[saap]ME_CHANNEL_REQ
[saap]MM_START_REQ
apm start with vif:0
-----beacon_int_set:100 TU
update_ongoing_1_bcn_update
Indicate after monitor mode
Switch channel 2
Switch channel 3
Switch channel 4
Switch channel 5
Switch channel 6
Switch channel 7
Switch channel 8
Switch channel 9
Switch channel 10
Switch channel 11
Switch channel 12
Switch channel 13
Switch channel 1
Switch channel 2
Switch channel 3
Switch channel 4
Switch channel 5
Switch channel 6
Switch channel 7
Switch channel 8
Switch channel 9
Switch channel 10
Switch channel 11
Switch channel 12
Switch channel 13
Switch channel 1
Switch channel 2
Switch channel 3
Switch channel 4
Switch channel 5
Switch channel 6
Switch channel 7
Switch channel 8
Switch channel 9
Switch channel 10
Switch channel 11
Switch channel 12
Switch channel 13
Switch channel 1
```

图12.4.2-1

• APP配网

打开调试APP，填入手机连接路由器的密码，点击发送，如下图所示：



图12.4.2-2



图12.4.2-3

• 配网完成

设备收到APP下发的路由器ssid和key后，显示日志如下：

```
Switch channel 9
Lock channel in 9
---vbat voltage:3084---
---vbat voltage:3086---
airkiss_get_result() ok!
ssid = wifi-team
pwd = stm32f215
, ssid_length = 9
pwd_length = 9
random = 0x39
[DRV_WLAN]stop monitor
[DRV_WLAN]set monitor callback
[DRV_WLAN]drivers\wlan\drv_wlan.c L922 beken_wlan_control cmd: case WIFI_INIT!
[wifi_connect]: fast connect
psk = 4ebe695d2af768a2ebd94fcfd165daaf7806fedef956d54ea87e07b32316a813
[sa_sta]MM_RESET_REQ
[sa_sta]ME_CONFIG_REQ
[sa_sta]ME_CHAN_CONFIG_REQ
[sa_st--ssid: , key:
msh />a:wifi-team , kebssid c0-31-0e-c7-91-4c
security2cipher 2 3 24 8 security=6
cipher2security 2 3 24 8
hapd_intf_add_vif.type:2, s:0, id:0
wpa_dInit
wpa_supplicant_req_scan
Setting scan request: 0.100000 sec
MANUAL_SCAN_REQ
wpa_supplicant_scan
Cancelling scan request
wpa_driver_associate
scan_start_req_handler
sm_auth_send:1
sm_auth_handler
sm_assoc_rsp_handler
rc_init: station_id=0 format_mod=2 pre_type=0 short_gi=1 max_bw=0
rc_init: nss_max=0 mcs_max=7 r_idx_min=0 r_idx_max=3 no_samples=10
```

图12.4.2-4

12.5 注意事项

- 手机需要连接2.4G的路由器。



13 Qspi Dcache模式

13.1 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。

13.2 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
<code>bk_qspi_dcache_initialize()</code>	qspi dcache初始化
<code>bk_qspi_start()</code>	启动qspi功能
<code>bk_qspi_stop()</code>	停止qspi功能

13.2.1 初始化qspi为dcache模式

```
OSStatus bk_qspi_dcache_initialize(qspi_dcache_drv_desc *qspi_config);
```

参数	描述
<code>qspi_config</code>	qspi的参数配置
返回	0: 成功; -1: 错误

参数类型

`qspi_dcache_drv_desc`

<code>mode</code>	qspi模式
<code>clk_set</code>	时钟源选择分频系数设置
<code>wr_command</code>	写入数据命令
<code>rd_command</code>	读取数据命令
<code>wr_dummy_size</code>	写数据大小
<code>rd_dummy_size</code>	读数据大小

13.2.2 启动qspi功能

```
OSStatus bk_qspi_start(void);
```

参数	描述
<code>void</code>	空
返回	0: 成功; -1: 错误



13.2.3 停止qspi功能

```
OSStatus bk_qspi_stop(void);
```

参数	描述
void	空
返回	0: 成功; -1: 错误

13.3 Qspi Dcache示例代码

```
/*
 * 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 <rtdef.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"

#ifndef 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_RDDData1 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDDData1[0]));
if(p_RDDData1 == RT_NULL)
{
    rt_kprintf("p_RDDData1 malloc failed\r\n");
}

p_RDDData2 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDDData2[0]));
if(p_RDDData2 == RT_NULL)
{
    rt_kprintf("p_RDDData2 malloc failed\r\n");
}

p_RDDData3 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDDData3[0]));
if(p_RDDData3 == RT_NULL)
{
    rt_kprintf("p_RDDData3 malloc failed\r\n");
}

p_RDDData4 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDDData4[0]));
if(p_RDDData4 == RT_NULL)
{
    rt_kprintf("p_RDDData4 malloc failed\r\n");
}

p_RDDData5 = rt_malloc(QSPI_TEST_LENGTH * sizeof(p_RDDData5[0]));
if(p_RDDData5 == RT_NULL)
{
    rt_kprintf("p_RDDData5 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_RDDData1, 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_RDDData[%d]=0x%lx\r\n", i, *(p_WRData1 +
i), i, *(p_RDDData1 + i));
    }
}

if(memcmp(p_WRData2, p_RDDData2, 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_RDDData[%d]=0x%lx\r\n", i, *(p_WRData2 +
i), i, *(p_RDDData2 + i));
    }
}

if(memcmp(p_WRData3, p_RDDData3, 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_RDDData[%d]=0x%lx\r\n", i, *(p_WRData3 +
i), i, *(p_RDDData3 + i));
}

}

if(memcmp(p_WRData4, p_RDDData4, 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_RDDData[%d]=0x%lx\r\n", i, *(p_WRData4 +
i), i, *(p_RDDData4 + i));
    }
}

if(memcmp(p_WRData5, p_RDDData5, 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_RDDData[%d]=0x%lx\r\n", i, *(p_WRData5 +
i), i, *(p_RDDData5 + 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_RDDData1 != RT_NULL)
{
    rt_free(p_RDDData1);
    p_RDDData1= RT_NULL;
}

if(p_RDDData2 != RT_NULL)
{
    rt_free(p_RDDData2);
    p_RDDData2= RT_NULL;
}

if(p_RDDData3 != RT_NULL)
{
    rt_free(p_RDDData3);
    p_RDDData3= RT_NULL;
}

if(p_RDDData4 != RT_NULL)
{
```

```
rt_free(p_RDDData4);
p_RDDData4= RT_NULL;
}

if(p_RDDData5 != RT_NULL)
{
    rt_free(p_RDDData5);
    p_RDDData5= 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
```

13.4 操作说明

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。使用qspi dcache模式需要将芯片外接psram。

13.4.1 运行现象

输入命令：qspi_test 0，可以看到log打印存取数据和读取数据都能成功，log如下：

```
msh />
msh />qspi_test 0
[qspi_test]:test_qspi_dcache_write_read_data
[qspi_test]:qspi read data 1 pass
[qspi_test]:qspi read data 2 pass
[qspi_test]:qspi read data 3 pass
[qspi_test]:qspi read data 4 pass
[qspi_test]:qspi read data 5 pass
msh />
msh />
```

图13.4.1-1

13.5 注意事项

- 测试中需要外接psram芯片。



14 低功耗

14.1 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。

14.2 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
<code>bk_wlan_enter_powersave()</code>	低功耗模式
<code>bk_enter_deep_sleep_mode()</code>	deep_sleep模式

14.2.1 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
<code>struct rt_wlan_device *device</code>	wlan设备句柄
<code>level</code>	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

14.2.2 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
<code>PS_DEEP_CTRL_PARAM *deep_param</code>	进入deep_sleep之前的参数设置
返回	空

参数类型
<code>PS_DEEP_CTRL_PARAM:</code>
<code>PS_DEEP_WAKEUP WAY deep_wkway</code>
<code>UINT32 gpio_index_map</code>



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

14.3 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deep_sleep测试, 示例代码如下:

14.3.1 关键说明

- 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;
```

- 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

14.3.2 示例代码

```
/*
 * 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);
```

14.4 操作说明

14.4.1 连接万用表

测量低功耗模式下的电流，需要将电源接到vbat引脚上以及串联万用表，如

图：

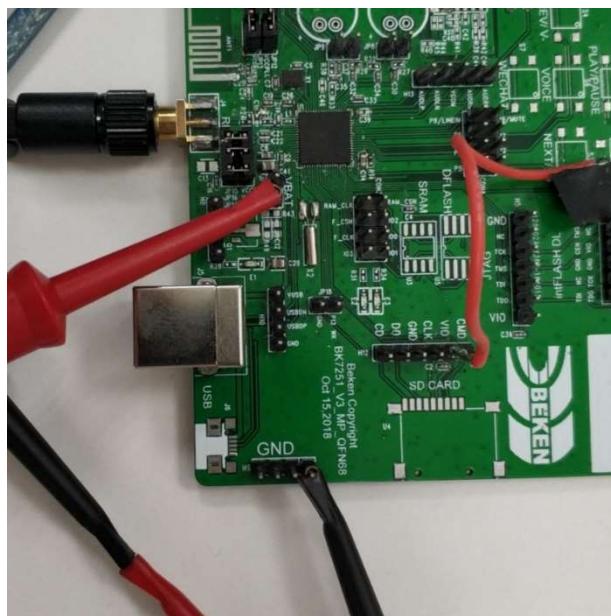


图14.4.1-1

14.4.2 运行现象

- mcu睡眠，rf睡眠示例

设备上电后，输入命令：`wifi w0 join wifiname password` 连接网络，， 输入命令 `pm_level level` 进入低功耗模式。其中，level值表示含义：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，可以看到电流表上显示芯片的电流值会发生变化。

- Deep Sleep模式

设备上电后，输入命令：`sleep_mode 1c 0 1c 0 10 0/1/2` 进入 `deep_sleep` 模式，`deep_wkway`选择唤醒模式，0:gpio唤醒，1: rtc唤醒，2: gpio与rtc都可唤醒Deep Sleep模式，具体可参考结构体类型说明。进入Deep Sleep模式下，电流可以达到8uA左右。

15 Bootloader

15.1 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。

15.2 分区表的设置

15.2.1 Bootloader分区

以 bk7251为例，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，对于不同系列soc的分区以实际芯片设置地址为准。

15.2.2 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，对于不同系列soc的分区以实际芯片设置地址为准。

15.2.3 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，对于不同系列soc的分区以实际芯片设置地址为准。

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

15.3 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。

15.4 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。

15.5 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。

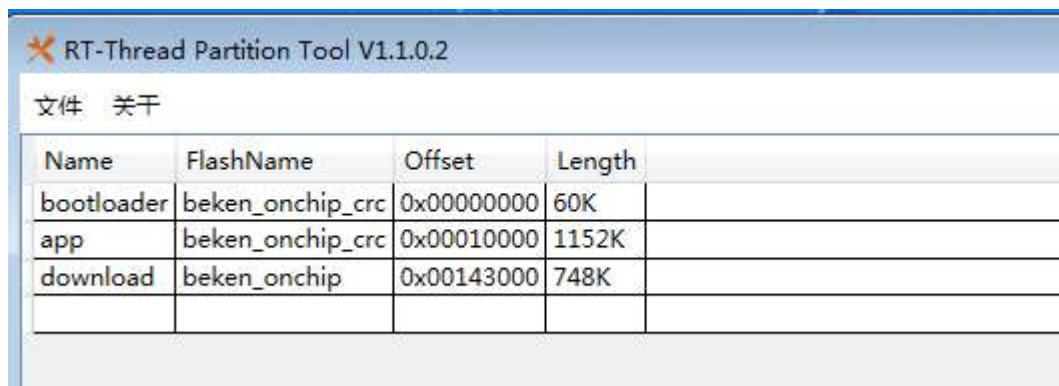


图15.5-1

15.6 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能完整的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

15.7 Bootloader示例代码

15.7.1 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"
        }
    ]
}
```

15.7.2 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 (!strcmp(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;
}
```

15.7.3 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"
        }
    ]
}
```



16 网络接口

16.1 网络接口简介

BKEN wifi soc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
<code>bk_wlan_start()</code>	启动网络，包括STATION和AP
<code>bk_wlan_start_sta_adv()</code>	启动STATION快速连接
<code>bk_wlan_stop()</code>	关闭网络，包括STATION和AP
<code>bk_wlan_start_scan()</code>	启动scan
<code>bk_wlan_scan_ap_reg_cb()</code>	注册scan结束后的回调函数
<code>bk_wlan_start_assign_scan()</code>	scan特定的网络
<code>bk_wlan_start_monitor()</code>	启动监听模式
<code>bk_wlan_stop_monitor()</code>	关闭监听模式
<code>bk_wlan_register_monitor_cb()</code>	注册监听回调函数
<code>bk_wlan_get_ip_status()</code>	获取当前的网络状态
<code>bk_wlan_get_link_status()</code>	获取当前的连接状态
<code>bk_wlan_get_channel()</code>	获取当前的信道
<code>bk_wlan_set_channel()</code>	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
<code>inNetworkInitPara</code>	传入需要配置信息
返回	kNoErr: 成功；其他：失败
参数类型	

**network_InitTypeDef_st:**

char	wifi_mode	WiFi模式
char	wifi_ssid[33]	需要连接或建立的网络SSID
char	wifi_key[64]	需要连接或建立的网络密码
char	local_ip_addr[16]	静态IP地址, 在DHCP关闭时有效
char	net_mask[16]	静态子网掩码, 在DHCP关闭时有效
char	gateway_ip_addr[16]	静态网关地址, 在DHCP关闭时有效
char	dns_server_ip_addr[16]	静态DNS地址, 在DHCP关闭时有效
char	dhcp_mode	DHCP模式
char	reserved[32]	保留
Int	wifi_retry_interval	重连间隔, 单位是毫秒

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

参数类型

network_InitTypeDef_adv_st:

apinfo_adv_t	ap_info	需要快速连接的网络信息
char	key[64]	需要快速连接的网络密码
Int	key_len	网络密码长度
char	local_ip_addr[16]	静态IP地址, 在DHCP关闭时有效
char	net_mask[16]	静态子网掩码, 在DHCP关闭时有效
char	gateway_ip_addr[16]	静态网关地址, 在DHCP关闭时有效
char	dns_server_ip_addr[16]	静态DNS地址, 在DHCP关闭时有效
char	dhcp_mode	DHCP模式
char	reserved[32]	保留
int	wifi_retry_interval	重连时间, 单位是毫秒

apinfo_adv_st:

char	ssid[32]	需要快速连接的网络信息
char	bssid[6]	需要快速连接的网络密码
uint8_t	channel	网络密码长度
wlan_sec_type_t	local_ip_addr[16]	静态IP地址, 在DHCP关闭时有效 typedef uint8_t wlan_sec_type_t



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: <pre>typedef void (*monitor_data_cb_t)(uint8_t *data, int len, hal_wifi_link_info_t *info);</pre>
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。
inInterface	需要获取网络状态的模式。
返回	kNoErr: 成功; 其他: 失败

参数类型

IPStatusTypeDef:

uint8_t	dhcp	获取的DHCP模式
char	ip[16]	获取的IP地址
char	gate[16]	获取的网关IP地址
char	mask[16]	获取的子网掩码
char	dns[16]	DNS服务IP地址
char	mac[16]	获取的mac地址
char	broadcastip[16]	获取的广播IP地址

```
#define WiFi_Interface wlanInterfaceTypeDef
```



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

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

参数类型

LinkStatusTypeDef:

msg_sta_states	conn_state	当前连接状态
int	wifi_strength	当前的信号强度
uint8_t	ssid[32]	当前网络的SSID
uint8_t	bssid[6]	当前网络的BSSID
int	channel	当前网络的信道
wlan_sec_type_t	security	当前网络的加密方式
		Typedef uint8_t wlan_sec_type_t

```
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;
```

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 网络接口使用示例

16.3.1 关键说明

- **DHCP宏定义说明**

#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*/

16.3.2 代码示例

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->scannu_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->scannu_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;

}

#endif 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_scanner_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_st ap_info;
    char ssid[33] = {0};
    #if CFG_IEEE80211N
        bk_printf("sta: %d, softap: %d, b/g/n\r\n",sta_ip_is_start(),uap_ip_is_start());
    #else
        bk_printf("sta: %d, softap: %d, b/g\r\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\r\n");
                break;
            case SECURITY_TYPE_WEP :
                bk_printf("WEP\r\n");
                break;
            case SECURITY_TYPE_WPA_TKIP:
                bk_printf("WPA_TKIP\r\n");
                break;
        }
    }
}
```



```
        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;
    }
}

/*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);
    bkprintf("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:
            bkprintf("OPEN\r\n");
            break;
        case SECURITY_TYPE_WEP :
            bkprintf("WEP\r\n");
            break;
        case SECURITY_TYPE_WPA_TKIP:
            bkprintf("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_recv_ret;
    airkiss_recv_ret = airkiss_recv(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。
```



```
*/\n\nint wifi_demo(int argc, char **argv)\n{\n    char *oob_ssid = NULL;\n    char *connect_key;\n\n    if (strcmp(argv[1], "sta") == 0)\n    {\n        os_printf("sta_Command\\r\\n");\n        if (argc == 3)\n        {\n            oob_ssid = argv[2];\n            connect_key = "1";\n        }\n        else if (argc == 4)\n        {\n            oob_ssid = argv[2];\n            connect_key = argv[3];\n        }\n        else\n        {\n            os_printf("parameter invalid\\r\\n");\n            return -1;\n        }\n        if(oob_ssid)\n        {\n            demo_sta_app_init(oob_ssid, connect_key);\n        }\n        return 0;\n    }\n    if(strcmp(argv[1], "adv") == 0)\n    {\n        os_printf("sta_adv_Command\\r\\n");\n        if (argc == 3)\n        {\n            oob_ssid = argv[1];\n            connect_key = "1";\n        }\n    }\n}
```



```
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);
```

16.4 操作说明

本节的示例代码均位于\beken378\demo\ieee802_11_demo.c,系统默认已经打开此功能，设备上电后，在调试串口输入相应指令即可运行不同程序。

16.4.1 启动STATION连接

设备上电后，调试串口输入wifi_demo sta your_ssid your_key,设备开始连接路由器。

```
wifi_demo sta your_ssid your_key
sta_Command
ssid: your_ssid key: your_key
rl_sta_start
[sa_sta]MM_RESET_REQ
[sa_sta]ME_CONFIG_REQ
```



```
[sa_sta]ME_CHAN_CONFIG_REQ
[sa_sta]MM_START_REQ
hapd_intf_add_vif,type:2, s:0, id:0
    [wlan_connect]:start tick = 0, connect done tick = 22379, total = 22379
    [wlan_connect]:start tick = 0, connect done tick = 22385, total = 22385
[WLAN_MGNT]wlan sta connected evenew dtim period:2
nt callback
IP UP: 192.168.44.27
[ip_up]:start tick = 0, ip_up tick = 25797, total = 25797
```

16.4.2 启动STATION快速连接

设备上电后，在调试串口输入wifi_demo adv your_ssid your_key，设备开始连接路由器，设备log如下：

```
wifi_demo adv your_ssid your_key
sta_adv_Command
[sa_sta]MM_RESET_REQ
[sa_sta]ME_CONFIG_REQ
[sa_sta]ME_CHAN_CONFIG_REQ
[sa_sta]MM_START_REQ
bssid 48-ee-0c-48-93-12
security2cipher 2 3 24 8 security=6
cipher2security 2 3 24 8
-----SM_CONNECT_IND_ok
wpa_driver_assoc_cb
Cancelling scan request
hapd_intf_add_key CCMP
add sta_mgmt_get_sta
sta:1, vif:0, key:0
sta_mgmt_add_key
add hw key idx:25
add TKIP
add is_broadcast_ether_addr
sta:255, vif:0, key:1
add hw key idx:1
ctrl_port_hdl:1
[wlan_connect]:start tick = 0, connect done tick = 31898, total = 31898
[wlan_connect]:start tick = 0, connect done tick = 31904, total = 31904
[WLAN_MGNT]wlan sta connected event callback
sta_ip_start
```

```
configuring interface wlan (with DHCP client)
dhcp_check_status_init_timer
IP UP: 192.168.44.49
[ip_up]:start tick = 0, ip_up tick = 35292, total = 35292
```

16.4.3 STATION模式获取状态

- 获取网络状态

连接路由器，方法参考16.4.1小节，然后串口输入wifi_demo status net获取设备当前网络状态，设备log如下：

```
msh />
msh />wifi_demo status net
dhcp=0 ip=192.168.44.52 gate=192.168.44.119 mask=255.255.255.0 mac=c8:47:8c:2f:4b:d2
```

图16.4.3-1

- 获取连接状态

连接路由器，方法参考16.4.1小节，然后串口输入wifi_demo status link获取设备当前连接状态，设备log如下：

```
msh />
msh />wifi_demo status link
sta: 1, softap: 0, b/g/n
sta:rssi=-65,ssid=wifi-team,bssid=c0:3f:0e:c7:91:4c ,channel=11,cipher_type:MIXED
```

图16.4.3-2

16.4.4 启动AP

设备上电后，在调试串口输入wifi_demo softap beken 12345678，设备开始连接路由器，设备log如下：

```
wifi_demo softap beken 12345678
SOFTAP_COMMAND

ssid:beken key:12345678
rl_ap_start
Soft_AP_start
[saap]MM_RESET_REQ
[saap]ME_CONFIG_REQ
[saap]ME_CHAN_CONFIG_REQ
[saap]MM_START_REQ
hapd_intf_add_vif,type:3, s:0, id:0
apm start with vif:0
-----beacon_int_set:100 TU
update_ongoing_1_bcn_update
vif_idx:0, ch_idx:0, bmc_idx:2
update_ongoing_1_bcn_update
hapd_intf_add_key CCMP
add is_broadcast_ether_addr
sta:255, vif:0, key:1
add hw key idx:1
uap_ip_start
```

图16.4.4-1

16.4.5 AP模式获取状态

- 获取网络状态

连接路由器，方法参考16.4.4小节，然后串口输入wifi_demo status net获取设备当前网络状态，设备log如下：

```
msh />
msh />wifi_demo status net
dhcp=0 ip=20.240.159.229 gate=20.240.159.229 mask=20.240.159.229 mac=00:00:00:00:00:00
```

图16.4.5-1

- 获取连接状态

连接路由器，方法参考16.4.4小节，然后串口输入wifi_demo status link获取设备当前连接状态，设备log如下：

```
msh />wifi_demo status link
sta: 0, softap: 1, b/g/n
softap:ssid=beken,channel=11,dhcp=1,cipher_type:CCMP
ip=192.168.0.1.gate=255.255.255.255.mask=255.255.255.0.dns=0.0.0.0
```

图16.4.5-2

16.4.6 启动SCAN

- 扫描WIFI热点

设备上电后，在调试串口输入wifi_demo scan，设备开始扫描附近WIFI热点，设备log如下：

```
msh />wifi_demo scan
[sa_sta]MM_RESET_REQ
[sa_sta]ME_CONFIG_REQ
[sa_sta]ME_CHAN_CONFIG_REQ
[sa_sta]MM_START_REQ
scan_start_req_handler
msh />ethernetif_input no netif found 255
Got ap count: 13
Xiaomi_E21E, RSSI=210
labast, RSSI=206
HUAWEI-EZ7HKY, RSSI=204
FAST_5AC4, RSSI=200
ssid-tcj, RSSI=197
B-LINK_F11566, RSSI=196
antbang_195F4C, RSSI=193
bk7252_smart, RSSI=193
wifi-team, RSSI=192
bekен_airport, RSSI=189
Honor Magic 2, RSSI=189
Bekencorp-Guest, RSSI=187
Bekencorp-WIFI, RSSI=177
Get ap end.....
```

图16.4.6-1

- 扫描指定WIFI热点

设备上电后，在调试串口输入wifi_demo scan Bekencorp-WIFI，设备开始扫描Bekencorp-WIFI，设备log如下：

```
msh />wifi_demo scan Bekencorp-WIFI
scan for ssid:Bekencorp-WIFI
[sa_sta]MM_RESET_REQ
[sa_sta]ME_CONFIG_REQ
[sa_sta]ME_CHAN_CONFIG_REQ
[sa_sta]MM_START_REQ
scan_start_req_handler
msh />Got ap count: 1
Bekencorp-WIFI, RSSI=186
Get ap end.....
```

图16.4.6-2



16.4.7 启动混杂包监听

设备上电后，在调试串口输入wifi_demo monitor start，设备开始监听混杂包，输入wifi_demo monitor stop，停止监听，设备log如下：

```
wifi_demo adv your_ssid your_key
msh />wifi_demo monitor
parameter invalid
parameter invalid
msh />wifi_demo monitor start
net_wlan_add_netif not vif idx found
Soft_AP_start
[saap]MM_RESET_REQ
[saap]ME_CONFIG_REQ
[saap]ME_CHAN_CONFIG_REQ
[saap]MM_START_REQ
apm start with vif:0
-----beacon_int_set:100 TU
update_ongoing_1_bcn_update
hal_machw_enter_monitor_mode
msh />len:136
len:260
len:166
len:173
len:225
len:136
len:270
len:260
len:166
len:270
len:173
len:136
len:225
len:260
len:225
wifi_demo monitor stop
msh />
```



17 BLE

17.1 BLE简介

BK7251支持ble_4.2和ble_5.1协议，有master和slave两种模式独立工作，若处于master模式，想要在slave模式工作需要先退出master模式，同理，若想要在master模式工作，需要先退出slave模式。

17.2 BLE_4.2 Related API（通用）

17.2.1 启动ble协议栈

```
void ble_activate(char *ble_name);
```

参数	描述
ble_name	可以传入NULL
返回	无

17.2.2 设置write callback

```
void ble_set_write_cb(ble_write_cb_t func);
```

参数	描述
func	write操作的callback函数。函数定义： typedef void (*ble_write_cb_t)(uint16_t char_id, uint8_t *buf, uint8_t len);
返回	无

17.2.3 设置read callback

```
void ble_set_read_cb(ble_read_cb_t func);
```

参数	描述
func	read操作的callback函数。函数定义： typedef uint8_t (*ble_read_cb_t)(uint16_t char_id, uint8_t *buf, uint8_t len);
返回	无



17.2.4 设置event callback

```
void ble_set_event_cb(ble_event_cb_t func);
```

参数	描述
func	事件处理的callback函数。函数定义: typedef void (*ble_event_cb_t)(ble_event_t event, void *param);
返回	无

17.2.5 设置接收 advertising callback

```
void ble_set_recv_adv_cb(ble_recv_adv_cb_t func);
```

参数	描述
func	接收广播包的callback函数。函数定义: typedef void (*ble_recv_adv_cb_t)(uint8_t *buf, uint8_t len);
返回	无

17.2.6 获取public key及private key

```
void appm_get_key(void);
```

参数	描述
void	无
返回	无

17.2.7 计算对称密钥

```
void appm_calc_psk(uint8_t *p_pub_key);
```

参数	描述
p_pub_key	接收到对端的public key
返回	无



17.3 BLE_4.2 Related API (slave)

17.3.1 开始广播

```
ble_err_t appm_start_advertising(void);
```

参数	描述
void	无
返回	ERR_SUCCESS: 成功; 其它: 失败

17.3.2 关闭广播

```
ble_err_t appm_stop_advertising(void);
```

参数	描述
void	无
返回	ERR_SUCCESS: 成功; 其它: 失败

17.3.3 发送数据

```
void ayla_wifi_send_statu_ntf_value(uint32_t len,uint8_t *buf,uint16_t seq_num);
```

参数	描述
len	数据长度 (byte)
buf	数据指针
seq_num	顺序 (传入0xFF即可)
返回	无

```
void ayla_wifi_send_scre_ntf_value(uint32_t len,uint8_t *buf,uint16_t seq_num);
```

参数	描述
len	数据长度 (byte)
buf	数据指针
seq_num	顺序 (传入0xFF即可)
返回	无

Note: 发送数据针对不同的characteristic拥有不同的接口，此处只列举了ayla service上的两个接口，其它characteristic对应的接口仅函数名不同，参数相同



17.3.4 断开连接

```
void appm_disconnect(uint8_t reason);
```

参数	描述
reason	断链原因(一般为0x13)
返回	无

17.4 BLE_4.2 Related API (master)

17.4.1 开始扫描

```
ble_err_t appm_start_scanning(void);
```

参数	描述
void	无
返回	ERR_SUCCESS: 成功; 其它: 失败

Note: 开始扫描后，会将扫描结果在串口打印，包括mac地址，id等信息

17.4.2 停止扫描

```
ble_err_t appm_stop_scanning(void);
```

参数	描述
void	无
返回	ERR_SUCCESS: 成功; 其它: 失败

17.4.3 发起连接

```
ble_err_t appm_start_connenct_by_id(uint8_t id);
```

参数	描述
id	连接id(scan过程中获取)
返回	ERR_SUCCESS: 成功; 其它: 失败

```
ble_err_t appm_start_connenct_by_addr(uint8_t* bdaddr);
```

参数	描述
bdaddr	对端mac地址(scan过程中获取)
返回	ERR_SUCCESS: 成功; 其它: 失败



17.4.4 断开连接

```
ble_err_t appm_disconnect_link(void);
```

参数	描述
void	无
返回	ERR_SUCCESS: 成功; 其它: 失败

17.4.5 停止连接

```
ble_err_t appm_stop_connencting(void);
```

参数	描述
void	无
返回	ERR_SUCCESS: 成功; 其它: 失败

17.4.6 发送数据

```
ble_err_t appm_write_data_by_uuid(uint16_t uuid,uint8_t len,uint8_t *data);
```

参数	描述
uuid	Characteristic id
len	数据长度(byte)
data	数据指针
返回	ERR_SUCCESS: 成功; 其它: 失败

17.5 BLE_5.1 Related API

目前BLE_5.1只支持slave模式。

17.5.1 创建广播

```
void ble_appm_create_advertising(unsigned char chnl_map,uint32_t intv_min,uint32_t intv_max);
```

参数	描述
chnl_map	无
intv_min	最小时间隙
intv_max	最大时间隙
返回	空



17.5.2 开始广播

```
void ble_appm_start_advertising(unsigned char adv_actv_idx,uint16 duration);
```

参数	描述
adv_actv_idx	广播id
duration	广播间隙
返回	空

17.5.3 停止广播

```
void ble_appm_stop_advertising(unsigned char adv_actv_idx);
```

参数	描述
adv_actv_idx	广播id
返回	空

17.5.4 设置广播数据

```
int ble_appm_set_adv_data(unsigned char adv_actv_idx,unsigned char* adv_buff, unsigned char adv_len);
```

参数	描述
adv_actv_idx	广播id
adv_buff	广播数据
adv_len	广播数据长度
返回	0: 成功; 其它: 失败

17.5.5 更新广播数据

```
int ble_appm_update_adv_data(unsigned char adv_actv_idx,unsigned char* adv_buff, unsigned char adv_len);
```

参数	描述
adv_actv_idx	广播id
adv_buff	广播数据
adv_len	广播数据长度
返回	0: 成功; 其它: 失败



17.5.6 删除广播

```
void ble_appm_delete_advertising(unsigned char adv_actv_idx);
```

参数	描述
adv_actv_idx	广播id
返回	空

17.5.7 创建扫描

```
void ble_appm_create_scanning(void);
```

参数	描述
void	空
返回	0: 成功; 其它: 失败

17.5.8 设置扫描请求数据

```
int ble_appm_set_scan_rsp_data(unsigned char adv_actv_idx,unsigned char* scan_buff,  
unsigned char scan_len);
```

参数	描述
adv_actv_idx	扫描id
scan_buff	扫描数据
scan_len	扫描数据长度
返回	0: 成功; 其它: 失败

17.5.9 更新连接参数

```
void ble_appm_update_param(unsigned char conidx,struct gapc_conn_param  
*conn_param);
```

参数	描述
conidx	连接id
conn_param	连接数据
返回	空

17.5.10 断开连接

```
void ble_appm_create_scanning(void);
```



参数	描述
void	空
返回	空

17.5.11 发送notification value

```
ble_err_t bk_ble_send_ntf_value(uint32_t len, uint8_t *buf, uint16_t prf_id, uint16_t att_idx);
```

参数	描述
len	长度
buf	数据buff
prf_id	profile id
att_idx	attribute id
返回	空

17.5.12 发送indification value

```
ble_err_t bk_ble_send_ntf_value(uint32_t len, uint8_t *buf, uint16_t prf_id, uint16_t att_idx);
```

参数	描述
len	长度
buf	数据buff
prf_id	profile id
att_idx	attribute id
返回	空

17.6 BLE结构体说明

17.6.1 广播参数

adv_info_t	
advData	Advertising数据(最大长度为31byte)
advDataLen	Advertising数据长度
respData	Response数据(最大长度为31byte)
respDataLen	Response数据长度
channel_map	Channel map(默认0x7: 37, 38, 39 channel都发送)
interval_min	最小interval(单位: 625us)
interval_max	最大interval(单位: 625us)

Note: 调用发送广播接口前需要先进行广播参数配置，广播参数需要配置到全



局结构adv_info_t adv_info中， 默认channel_map是0x7， interval_min为160， interval_max为160， 若对adv_info执行了memset操作， 需要重新配置 channel_map, interval_min, interval_max， 否则会广播参数错误

17.6.2 扫描参数

scan_info_t

filter_en	是否filter(默认disable)
channel_map	Channel map(默认0x7: 37, 38, 39 channel都发送)
interval	扫描interval(默认100, 单位625us)
window	扫描window(默认20, 单位625us)

Note: 调用开始扫描接口前需要先配置扫描参数，扫描参数配置到全局结构体 scan_info_t scan_info中， 默认filter_en为disable, channel map为0x7, interval 为100, window为20， 若对scan_info执行了memset操作， 需要重新配置参数，否则会扫描参数错误。

17.6.3 ecdh加密算法结果返回

ble_get_key_ind

pri_key	Private key
pri_len	Private key length
pub_key_x	Public key X
pub_x_len	Public key X length
pub_key_y	Public key Y
pub_y_len	Public key Y length

ble_gen_dh_key_ind

result	计算结果
len	数据长度

17.6.4 gapc连接参数

gapc_conn_param

intv_min	最小间隙
intv_max	最大时间间隙
latency	收包间隔
time_out	超时时间



17.7 BLE示例代码

17.7.1 关键说明

- **BLE枚举类型说明**

```
typedef enum
{
    BLE_STACK_OK,           协议栈初始化成功
    BLE_STACK_FAIL,         协议栈初始化失败
    BLE_CONNECT,            连接成功
    BLE_DISCONNECT,          断开连接
    BLE_MTU_CHANGE,         更改MTU size
    BLE_CFG_NOTIFY,          设置notify(disable, enable)
    BLE_CFG_INDICATE,        设置indicate(disable, enable)
    BLE_TX_DONE,             发送数据完成
    BLE_GEN_DH_KEY,          计算密钥成功
    BLE_GET_KEY              获取public key及private key
} ble_event_t;

typedef enum
{
    ERR_SUCCESS = 0,
    ERR_STACK_FAIL,
    ERR_MEM_FAIL,
    ERR_INVALID_ADV_DATA,
    ERR_ADV_FAIL,
    ERR_STOP_ADV_FAIL,
    ERR_GATT_INDICATE_FAIL,
    ERR_GATT_NOTIFY_FAIL,
    ERR_SCAN_FAIL,
    ERR_STOP_SCAN_FAIL,
    ERR_CONN_FAIL,
    ERR_STOP_CONN_FAIL,
    ERR_DISCONN_FAIL,
    ERR_READ_FAIL,
    ERR_WRITE_FAIL,
    ERR_REQ_RF,
    /* Add more BLE error code hereafter */
} ble_err_t; //函数返回类型
```



17.7.2 示例代码

```
/*
 * 程序清单： 这是一个简单ble的使用例程。
 * 命令调用格式： ble active/start_adv/stop_adv/start_scan 等
 * 程序功能： 通过ble命令实现开启协议栈，广播，扫描，连接等功能。
 */

#include <rtthread.h>
#include <finsh.h>
#include "common.h"
#include "ble_pub.h"
#include "app_sdp.h"
#include "param_config.h"
#include "app_task.h"

#define BLE_DEBUG    1
#if BLE_DEBUG
#define BLE_DBG(...)    rt_kprintf("[BLE]"),rt_kprintf(__VA_ARGS__)
#else
#define BLE_DBG(...)
#endif

#define BUILD_UINT16(loByte, hiByte) \
    ((uint16_t)((loByte) & 0x00FF) + (((hiByte) & 0x00FF) << 8))

int _atoi(char *str)
{
    int value = 0;
    while(*str>='0' && *str<='9')
    {
        value *= 10;
        value += *str - '0';
        str++;
    }
    return value;
}

static void ble_usage(void)
{
```



```
rt_kprintf("ble help           - Help information\n");
rt_kprintf("ble dut            - Enter ble dut mode\n");
rt_kprintf("ble active          - Active ble to config network\n");
rt_kprintf("ble start_adv [channel_map=7] [interval_max=160] [interval_min=160]\n");
rt_kprintf("                         - Start advertising as a slave device\n");
rt_kprintf("ble stop_adv         - Stop advertising as a slave device\n");
rt_kprintf("ble send param value\n");
rt_kprintf("                         - Send value to master\n");
rt_kprintf("ble start_scan [channel_map=7] [window=20] [interval=100]\n");
rt_kprintf("                         - Start scanning as a host device\n");
rt_kprintf("ble stop_scan        - Stop scan as a host device\n");
rt_kprintf("ble conn_id [idx=0]   - Connect with index as a host device\n");
rt_kprintf("ble conn_addr addr   - Connect with address as a host device\n");
rt_kprintf("ble disc_master      - Disconnect as a host device\n");
rt_kprintf("ble write uuid value\n");
rt_kprintf("                         - Write value to slave\n");
}

void ble_write_callback(uint16_t char_id, uint8_t *data, uint8_t len)
{
    rt_kprintf("%s char_id=0x%04x, len=%d\n", __FUNCTION__, char_id, len);
}

uint8_t ble_read_callback(uint16_t char_id, uint8_t *data, uint8_t len)
{
    rt_kprintf("%s char_id=0x%04x, len=%d\n", __FUNCTION__, char_id, len);
    data[0] = 'a';
    data[1] = 'b';
    return 2;
}

void ble_event_callback(ble_event_t event, void *param)
{
    switch(event)
    {
        case BLE_STACK_OK:
            rt_kprintf("STACK INIT OK\r\n");
            break;
        case BLE_STACK_FAIL:
```



```
rt_kprintf("STACK INIT FAIL\r\n");
break;
case BLE_CONNECT:
    rt_kprintf("BLE CONNECT\r\n");
break;
case BLE_DISCONNECT:
{
    rt_kprintf("BLE DISCONNECT\r\n");
}
break;
case BLE_MTU_CHANGE:
    rt_kprintf("BLE_MTU_CHANGE:%d\r\n", *(uint16_t *)param);
break;
case BLE_CFG_NOTIFY:
    rt_kprintf("BLE_CFG_NOTIFY:%d\r\n", *(uint16_t *)param);
break;
case BLE_CFG_INDICATE:
    rt_kprintf("BLE_CFG_INDICATE:%d\r\n", *(uint16_t *)param);
break;
case BLE_TX_DONE:
    rt_kprintf("BLE_TX_DONE\r\n");
break;
case BLE_GEN_DH_KEY:
{
    rt_kprintf("BLE_GEN_DH_KEY\r\n");
    rt_kprintf("key_len:%d\r\n", ((struct ble_gen_dh_key_ind *)param)->len);
    for(int i = 0; i < ((struct ble_gen_dh_key_ind *)param)->len; i++)
    {
        rt_kprintf("%02x ", ((struct ble_gen_dh_key_ind *)param)->result[i]);
    }
    rt_kprintf("\r\n");
}
break;
case BLE_GET_KEY:
{
    rt_kprintf("BLE_GET_KEY\r\n");
    rt_kprintf("pri_len:%d\r\n", ((struct ble_get_key_ind *)param)->pri_len);
    for(int i = 0; i < ((struct ble_get_key_ind *)param)->pri_len; i++)
    {
```



```
        rt_kprintf("%02x ", ((struct ble_get_key_ind *)param)->pri_key[i]);  
    }  
    rt_kprintf("\r\n");  
}  
break;  
default:  
    rt_kprintf("UNKNOW EVENT\r\n");  
    break;  
}  
}  
  
void ble_recv_adv_callback(uint8_t *buf, uint8_t len)  
{  
#if (BLE_APP_CLIENT)  
    uint8_t find = 0;  
  
    find = appm_adv_data_decode(len, buf, NULL, 0);  
  
    if(find)  
    {  
        extern int *scan_check_result;  
        if(scan_check_result)  
        {  
            *scan_check_result = 2;  
            bk_printf("scan_check_result\r\n");  
        }  
    }  
#endif  
}  
  
void ble_adv(ble_adv_param_t *adv_param)  
{  
    uint8_t mac[6];  
    char ble_name[20];  
    uint8_t adv_idx, adv_name_len;  
  
    wifi_get_mac_address((char *)mac, CONFIG_ROLE_STA);  
    adv_name_len = rt_snprintf(ble_name, sizeof(ble_name), "bk-%02x%02x", mac[4], mac[5]);
```



```
memset(&adv_info, 0x00, sizeof(adv_info));

if(adv_param)
{
    adv_info.channel_map = adv_param->channel_map;
    adv_info.interval_min = adv_param->interval_min;
    adv_info.interval_max = adv_param->interval_max;
}

adv_idx = 0;
adv_info.advData[adv_idx] = 0x02; adv_idx++;
adv_info.advData[adv_idx] = 0x01; adv_idx++;
adv_info.advData[adv_idx] = 0x06; adv_idx++;

adv_info.advData[adv_idx] = adv_name_len + 1; adv_idx +=1;
adv_info.advData[adv_idx] = 0x09; adv_idx +=1; //name
memcpy(&adv_info.advData[adv_idx], ble_name, adv_name_len); adv_idx +=adv_name_len;

adv_info.advDataLen = adv_idx;

adv_idx = 0;

adv_info.respData[adv_idx] = adv_name_len + 1; adv_idx +=1;
adv_info.respData[adv_idx] = 0x08; adv_idx +=1; //name
memcpy(&adv_info.respData[adv_idx], ble_name, adv_name_len); adv_idx +=adv_name_len;
adv_info.respDataLen = adv_idx;

if (ERR_SUCCESS != appm_start_advertising())
{
    rt_kprintf("ERROR\r\n");
}

int ble(int argc, char **argv)
{
    uint8_t mac[6];
    char ble_name[20];
    ble_adv_param_t adv_param;
```



```
if ((argc < 2) || (strcmp(argv[1], "help") == 0))
{
    ble_usage();
    return 0;
}

if (strcmp(argv[1], "active") == 0)
{
    ble_activate(NULL);
    ble_set_write_cb(ble_write_callback);
    ble_set_read_cb(ble_read_callback);
    ble_set_event_cb(ble_event_callback);
    ble_set_recv_adv_cb(ble_recv_adv_callback);
}

else if(strcmp(argv[1], "start_adv") == 0)
{
    /* ble start_adv channel interval_max interval_min */
    if (argc > 4)
    {
        adv_param.interval_min = _atoi(argv[4]);
    }
    else
    {
        adv_param.interval_min = 160;
    }
    if (argc > 3)
    {
        adv_param.interval_max = _atoi(argv[3]);
    }
    else
    {
        adv_param.interval_max = 160;
    }
    if (argc > 2)
    {
        adv_param.channel_map = _atoi(argv[2]);
    }
    else
    {
```



```
adv_param.channel_map = 7;
}

rt_kprintf("channel_map=%d,interval=[%d,%d]\n", adv_param.channel_map,
adv_param.interval_min,adv_param.interval_max);

ble_adv(&adv_param);

}

else if(strcmp(argv[1], "stop_adv") == 0)
{
    if(ERR_SUCCESS != appm_stop_advertising())
    {
        rt_kprintf("ERROR\r\n");
    }
}

else if(strcmp(argv[1], "send") == 0)
{
    uint8_t len;
    uint8_t write_buffer[20];
    if(argc != 4)
    {
        ble_usage();
        return 0;
    }

    len = strlen(argv[3]);
    if(len % 2 != 0)
    {
        rt_kprintf("ERROR\r\n");
        return 0;
    }
    hexstr2bin(argv[3], write_buffer, len/2);

    if(strcmp(argv[2], "ayla_statu") == 0)
    {
#if (BLE_APP_AYLA_WIFI)
        ayla_wifi_send_statu_ntf_value(len/2, write_buffer, 0xff);
#else
        rt_kprintf("unvalid param\r\n");
#endif
    }
}
```



```
}

else if(strcmp(argv[2], "ayla_scre") == 0)
{
#if (BLE_APP_AYLA_WIFI)
    ayla_wifi_send_scre_ntf_value(len/2, write_buffer, 0xff);
#else
    rt_kprintf("unvalid param\r\n");
#endif

}
else
{
    rt_kprintf("unvalid param\r\n");
}

}

#endif (BLE_APP_CLIENT)

else if(strcmp(argv[1], "start_scan") == 0)
{
    /* ble start_scan channel window interval */
    if (argc > 4)
    {
        scan_info.interval = _atoi(argv[4]);
    }
    else
    {
        scan_info.interval = 100;
    }
    if (argc > 3)
    {
        scan_info.window = _atoi(argv[3]);
    }
    else
    {
        scan_info.window = 20;
    }
    if (argc > 2)
    {
        scan_info.channel_map = _atoi(argv[2]);
    }
    else
```

```
{  
    scan_info.channel_map = 7;  
}  
rt_kprintf("channel_map=%d,window=%d,interval=%d\n", scan_info.channel_map,  
scan_info.window,scan_info.interval);  
if(ERR_SUCCESS != appm_start_scanning())  
{  
    rt_kprintf("ERROR\r\n");  
}  
}  
else if(strcmp(argv[1], "stop_scan") == 0)  
{  
    if(ERR_SUCCESS != appm_stop_scanning())  
{  
        rt_kprintf("ERROR\r\n");  
    }  
}  
else if(strcmp(argv[1], "disc_master") == 0)  
{  
    if(ERR_SUCCESS != appm_disconnect_link())  
{  
        rt_kprintf("ERROR\r\n");  
    }  
}  
else if(strcmp(argv[1], "disc_slave") == 0)  
{  
    appm_disconnect(0x13);  
}  
else if (strcmp(argv[1], "conn_id") == 0)  
{  
    int idx = 0;  
    //char to int  
    if (argc > 2)  
    {  
        idx = _atoi(argv[2]);  
    }  
    rt_kprintf("start connect, idx = %d\r\n", idx);  
    if(ERR_SUCCESS != appm_start_connenct_by_id(idx))  
    {  
    }
```



```
rt_kprintf("ERROR\r\n");
}

}

else if(strcmp(argv[1], "conn_addr") == 0)
{
    uint8_t mac_addr[6];
    if (argc < 3)
    {
        ble_usage();
        return 0;
    }
    hexstr2bin(argv[2], mac_addr, 6);
    rt_kprintf("start connect\r\n");
    if(ERR_SUCCESS != appm_start_connecct_by_addr(mac_addr))
    {
        rt_kprintf("ERROR\r\n");
    }
}

else if(strcmp(argv[1], "stop_conn") == 0)
{
    if(ERR_SUCCESS != appm_stop_connecnting())
    {
        rt_kprintf("ERROR\r\n");
    }
}

else if(strcmp(argv[1], "write") == 0)
{
    uint8_t len;
    uint8_t hex_uuid[2];
    uint8_t write_buffer[20];
    uint16_t uuid;

    hexstr2bin(argv[2], hex_uuid, 2);
    uuid = BUILD_UINT16(hex_uuid[1], hex_uuid[0]);
    len = strlen(argv[3]);
    if(len % 2 != 0)
    {
        rt_kprintf("ERROR\r\n");
        return 0;
    }
}
```



```

    }

    hexstr2bin(argv[3], write_buffer, len/2);
    rt_kprintf("write uuid = 0x%x\r\n", uuid);
    rt_kprintf("write len = %d\r\n", len/2);
    if(!appm_write_data_by_uuid(uuid, len/2, write_buffer))
    {
        rt_kprintf("ERROR\r\n");
    }
}

#endif

else if (strcmp(argv[1], "get_key") == 0)
{
    appm_get_key();
}

else if (strcmp(argv[1], "calc_key") == 0)
{
    uint8_t pub_peer_key[64] =
{0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0,1,2,3,4,
,5,6,7,8,9,0,1,2,3};

    appm_calc_psk(pub_peer_key);
}

return 0;
}

MSH CMD EXPORT(ble, ble command);

```

17.8 操作说明

17.8.1 启动SCAN

设备上电后，调试串口输入ble active启动BLE协议栈,然后输入ble start_scan,开始扫描BLE设备，设备log如下：

```
msh />ble active  
ble start no ble name  
ble name:BK7231BT-01, c9:47:8c:63:58:d6  
----rw_main task init----  
----rw_main  start----  
gapm cmp evt handler operation = 0x1, status = 0x0
```



```
gapm_cmp_evt_handler operation = 0x3, status = 0x0
app_ayla_gen_add_gen
ayla_gen_env->start_hdl = 0x7gapm_cmp_evt_handler operation = 0x1b, status = 0x0
app_ayla_wifi_add_wifi
ayla_wifi_env->start_hdl = 0x14gapm_cmp_evt_handler operation = 0x1b, status = 0x0
app_ayla_conn_add_conn
ayla_conn_env->start_hdl = 0x21gapm_cmp_evt_handler operation = 0x1b, status = 0x0
msh />
msh />ble start_scan
channel_map=7,window=20,interval=100
LLD_ADV_HDL=0x8027
msh />

adv number = 0
adv type = 3 :ADV_NONCONN_UNDIR
adv rssi = -68
adv addr = d7:25:1e:f8:28:3f
adv addr type = 01
adv number = 1
adv type = 3 :ADV_NONCONN_UNDIR
adv rssi = -72
adv addr = 53:21:eb:a4:2c:2a
adv addr type = 01
adv number = 2
```

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 rbl固件支持压缩和加密，ota固件制作使用rt_ota_packaging_tool工具。freertos和rt-thread类似，采用http协议从服务器下载ota rbl格式固件，然后烧录到download分区中，但freertos API不同。

18.2 OTA Related API

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

<code>fal_init()</code>	rt fal初始化
<code>http_ota_fw_download()</code>	rt-thread远程下载固件
<code>http_ota_download ()</code>	freertos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



18.2.3 freertos远程下载固件

从服务器下载ota固件，然后烧录到download分区中，函数如下所示：

```
int http_ota_download(const char *uri);
```

参数	描述
url	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]);
```

18.3 操作说明

18.3.1 生成rbl升级文件

使用“rt_ota_packaging_tool”，可以生成rbl文件，配置如下：



图18.3.1-1

其中：压缩算法一定要选gzip，加密算法选AES256，加密密匙与加密IV应与boot中的对应。

```
static const uint8_t iv_table[16 + 1] = "0123456789ABCDEF";
static const uint8_t key_table[32 + 1] =
"0123456789ABCDEF0123456789ABCDEF";
```

18.3.2 搭建本地HTTP Server环境

如果没有HTTP服务器，可以先搭建本地HTTP Server环境进行http_ota的功

能验证。打开“MyWebServer3621.exe”，界面如下所示：



图 18.3.2-1

点击浏览按钮，选择存放3.4.1生成的rbl文件的路径，修改端口为8080，默认是80，然后点击启动按钮，



图 18.3.2-2



图 18.3.2-3

url为http://local_ip:8080/rtthread.rtl,浏览器输入后可以下载到rtthread.rbl说明环境搭建成功。

18.3.3 运行现象

- **连接路由器**

设备上电后，调试串口输入wifi_demo sta your_ssid your_key，设备开始连接路由器。



```
wifi_demo sta your_ssid your_key
sta_Command
ssid: your_ssid key: your_key
rl_sta_start
[sa_sta]MM_RESET_REQ
[sa_sta]ME_CONFIG_REQ
[sa_sta]ME_CHAN_CONFIG_REQ
[sa_sta]MM_START_REQ
hapd_intf_add_vif,type:2, s:0, id:0
[wlan_connect]:start tick = 0, connect done tick = 22379, total = 22379
[wlan_connect]:start tick = 0, connect done tick = 22385, total = 22385
[WLAN_MGNT]wlan sta connected evenew dtim period:2
nt callback
IP UP: 192.168.44.27
[ip_up]:start tick = 0, ip_up tick = 25797, total = 25797
```

• OTA升级

调试串口输入http_ota http://local_ip:8080/rtthread.rtl，设备log如下所示：

```
http_ota http://local_ip:8080/rtthread.rtl
current firmware name: app, version: 2M.1220, timestamp: 1568000867
dl_part->name : download
dl_part->flash : beken_onchip
dl_part->offset: 0x00143000
dl_part->len : 665600
[I/HTTP_OTA] OTA file size is (624560)
[I/HTTP_OTA] OTA file raw size 755580 bytes.
[I/HTTP_OTA] OTA file describe partition name app, version: 2M.1221, timestamp: 1568000867.
```

```
[I/HTTP_OTA] FLASH_PROTECT_HALF.
```

Download:

```
[=====]
=====] 100%
```

```
[I/OTA] Verify 'download' partition(fw ver: 2M.1221, timestamp: 1568000867) success.
```

```
[I/HTTP_OTA] FLASH_UNPROTECT_LAST_BLOCK.
```

reboot system

.....



19 OS接口

19.1 OS接口简介

RTT/Freertos接口提供OS的操作API，包括线程，互斥锁，时钟，信号量的操作。

19.2 OS Related APIs

OS相关接口如下所示：

函数	描述
<code>rtos_create_thread()</code>	创建一个新的线程
<code>rtos_delete_thread()</code>	删除一个使用结束的线程
<code>rtos_thread_join()</code>	使当前线程挂起，等待另一个线程终止
<code>rtos_thread_sleep()</code>	使一个线程挂起一段时间，时间单位是：秒
<code>rtos_init_semaphore()</code>	初始化一个信号量，并提供一个最大数
<code>rtos_set_semaphore()</code>	发出信号量
<code>rtos_get_semaphore()</code>	获取一个信号量，并提供超时机制
<code>rtos_deinit_semaphore()</code>	销毁一个信号量
<code>rtos_init_mutex()</code>	初始化一个互斥锁
<code>rtos_lock_mutex()</code>	获得一个互斥锁
<code>rtos_unlock_mutex()</code>	释放一个互斥锁
<code>rtos_deinit_mutex()</code>	销毁一个互斥锁
<code>rtos_init_queue()</code>	初始化一个消息队列
<code>rtos_push_to_queue()</code>	将一个数据对象推入消息队列
<code>rtos_pop_from_queue()</code>	从消息队列中取出一个数据对象
<code>rtos_deinit_queue()</code>	销毁一个消息队列
<code>rtos_is_queue_empty()</code>	查询一个队列是否为空
<code>rtos_is_queue_full()</code>	查询一个队列是否已满
<code>rtos_init_timer()</code>	初始化一个时钟，并传入回调函数
<code>bk_rtos_start_timer()</code>	启动一个时钟
<code>rtos_stop_timer()</code>	停止一个时钟
<code>rtos_reload_timer()</code>	重新加载一个过期的时钟
<code>rtos_deinit_timer()</code>	销毁一个时钟
<code>rtos_is_timer_running()</code>	获取一个时钟是否正在运行

19.2.1 OS结构体说明

`beken_timer_t:`

<code>handle</code>	<code>rtos_init_timer</code> 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

19.2.2 创建一个新的线程

```
OSStatus 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



19.2.3 删除一个使用结束的线程

```
OSStatus 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

19.2.4 使当前线程挂起，等待另一个线程终止

```
OSStatus 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

19.2.5 使一个线程挂起一段时间

```
void 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

19.2.6 初始化一个信号量

```
OSStatus 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

19.2.7 发出信号量

```
int 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



19.2.8 获取一个信号量，并提供超时机制

```
OSStatus 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

19.2.9 销毁一个信号量

```
OSStatus 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

19.2.10 初始化一个互斥锁

```
OSStatus 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

19.2.11 获得一个互斥锁

```
OSStatus 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

19.2.12 释放一个互斥锁

```
OSStatus 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



19.2.13 销毁一个互斥锁

```
OSStatus 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

19.2.14 初始化一个消息队列

```
OSStatus 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

19.2.15 将一个数据对象推入消息队列

```
OSStatus 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

19.2.16 从消息队列中取出一个数据对象

```
OSStatus 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

19.2.17 销毁一个消息队列

```
OSStatus 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

19.2.18 查询一个队列是否为空

```
BOOL 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

19.2.19 查询一个队列是否已满

```
BOOL 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

19.2.20 初始化一个时钟，并传入回调函数

```
OSStatus 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

19.2.21 启动一个时钟

```
OSStatus 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

19.2.22 停止一个时钟

```
OSStatus 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

19.2.23 重新加载一个过期的时钟

```
OSStatus 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

19.2.24 销毁一个时钟

```
OSStatus 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

19.2.25 获取一个时钟是否正在运行

```
BOOL 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

19.3 RTOS示例代码

本次只提供RTOS启动线程示例代码,具体参考os_demo.c。

19.3.1 关键说明

- 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	10000000000UUL
#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)

- 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;
```

19.3.2 示例代码

```
#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 )
```



```
{\n    /* Create a new thread, and this thread will delete its self and clean its resource */\n    err = bk_rtos_create_thread( &t_handler,\n                                BEKEN_APPLICATION_PRIORITY,\n                                "Thread 0",\n                                thread_0,\n                                0x400,\n                                0);\n\n    if(err != kNoErr)\n    {\n        os_printf("ERROR: Unable to start the thread 1.\r\n");\n    }\n\n    /* wait thread 0 delete it's self */\n    bk_rtos_thread_join( &t_handler );\n}\n}\n\n/*线程2的入口函数，打印一句log。*/\nstatic void thread_2( beken_thread_arg_t arg )\n{\n    (void)( arg );\n\n    while ( 1 )\n    {\n        os_printf( "This is thread 2\r\n" );\n        bk_rtos_delay_milliseconds((TickType_t)600);\n    }\n}\n\n/*该示例函数将建立两个线程*/\nstatic int thread_demo_start( void )\n{\n    OSStatus err = kNoErr;\n\n    /*定义两个线程的句柄*/\n    beken_thread_t t_handler1 = NULL, t_handler2 = NULL;\n\n    os_printf("\r\n\r\noperating system thread demo.....\r\n");\n\n    /*创建第一个线程，入口函数为thread_1，没有参数。*/\n    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。*/\n\n/*设置信号量的入口函数，设置信号量后等待500毫秒。*/\nstatic void set_semaphore_thread( beken_thread_arg_t arg )\n{\n    while ( 1 )\n    {\n        os_printf( "release semaphore!\r\n" );\n        bk_rtos_set_semaphore( &os_sem );\n        bk_rtos_delay_milliseconds( 500 );\n    }\n\n    exit:\n    if(os_sem)\n    {\n        bk_rtos_deinit_semaphore(&os_sem);\n    }\n    bk_rtos_delete_thread( NULL );\n}\n\n/*获取信号量的线程入口函数，在获得信号量后打印log。*/\nstatic void get_semaphore_thread( beken_thread_arg_t arg )\n{\n    OSStatus err;\n\n    while(1)\n    {\n        err = bk_rtos_get_semaphore(&os_sem, BEKEN_NEVER_TIMEOUT);\n        if(err == kNoErr)\n        {\n            os_printf("Get_Sem Succend!\r\n");\n        }\n        else\n        {\n            os_printf("Get_Sem Err:%d\r\n", err);\n            goto exit;\n        }\n    }\n}
```



```
    }

}

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_msg(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。*/
        err = mutex_printf_msg(strprt);
        if(err != kNoErr)
        {
            os_printf( "%s printf_msg error!\r\n", taskname);
            goto exit;
        }
        bk_rtos_delay_milliseconds( rd );
    }

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

/*使用互斥量的示例入口函数*/
static int mutex_demo_start( void )
```



```
{  
    OSStatus err = kNoErr;  
    /*初始化定义的全局互斥量os_mutex*/  
    beken_thread_t t_handler1 = NULL, t_handler2 = NULL;  
    err = bk_rtos_init_mutex( &os_mutex );  
    /*检查是否初始化成功*/  
    if(err != kNoErr)  
    {  
        os_printf( "rtos_init_mutex err: %d\r\n", err );  
        goto exit;  
    }  
    /*创建线程1，用于发送"my name is thread1"。 */  
    err = bk_rtos_create_thread(&t_handler1,  
                                BEKEN_APPLICATION_PRIORITY,  
                                "sender1",  
                                os_mutex_sender_thread,  
                                0x400,  
                                "my name is thread1");  
    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 );

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)
    {
#if OS_MUTEX_DEMO
        mutex_demo_start();
#endif
    }
    else if(strcmp(argv[1], "queue") == 0)
    {
#if OS_QUEUE_DEMO
        queue_demo_start();
#endif
    }
    else if(strcmp(argv[1], "semaphore") == 0)
}
```



```
{  
#if OS_SEM_DEMO  
    sem_demo_start();  
#endif  
}  
else if(strcmp(argv[1], "timer") == 0)  
{  
#if OS_TIMER_DEMO  
    timer_demo_start();  
#endif  
}  
else  
{  
    os_printf("os demo %s dosn't support.\n", argv[1]);  
}  
}  
MSH_CMD_EXPORT(os_demo, os_demo command);
```



20 Audio

20.1 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。

20.2 Audio Related API

audio相关接口参考\drivers\audio\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

20.2.1 audio通用结构体说明

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

20.2.2 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

20.2.3 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

20.2.4 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

20.2.5 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

20.2.6 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



20.2.7 关闭mic

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

参数	描述
void	空
返回	空

20.2.8 打开audio设备

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

参数	描述
void	空
返回	空

20.2.9 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

20.2.10 设置dac音量

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

参数	描述
volume	音量大小
返回	空

20.2.11 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



20.2.12 关闭dac

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

参数	描述
void	空
返回	空

20.3 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。

20.3.1 关键说明

- **Audio宏定义**

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

20.3.2 示例代码

```
/*  
 * 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*/\n\nwhile(1)\n{\n    buffer = (uint16_t *)audio_device_get_buffer(RT_NULL);\n    if(dac_wr_len >= mic_read_len)\n    {\n        audio_device_put_buffer(buffer);\n        break;\n    }\n\n    memcpy(buffer,test_buf+dac_wr_len,READ_SIZE);\n    dac_wr_len += READ_SIZE;\n\n    audio_device_write((uint8_t *)buffer, READ_SIZE); /*dac播放数据*/\n}\n\naudio_device_close(); /*关闭dac设备*/\n\nif(test_buf)\n    sdram_free(test_buf); /*释放ram内存*/\n\n#if CONFIG_SOUND_MIXER\n    mixer_replay();\n#endif\n}\n\nMSH_CMD_EXPORT(record_and_play, record play);\n#endif
```



21 List Player

21.1 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。

21.2 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
<code>list_player_init()</code>	初始化播放器
<code>list_player_current_item()</code>	获取当前播放曲目的handle
<code>list_player_current_index()</code>	获取当前播放曲目在列表中的索引
<code>list_player_current_state()</code>	获取播放器当前状态
<code>list_player_current_position()</code>	获取当前播放曲目的播放位置
<code>list_player_current_items()</code>	获取当前播放列表
<code>list_player_is_exist()</code>	查询是否有播放器及列表存在
<code>list_player_play()</code>	播放指定列表
<code>list_player_switch()</code>	切换播放列表
<code>list_player_play_index()</code>	播放指定列表中指定索引曲目
<code>list_player_play_item()</code>	播放指定列表中指定handle曲目
<code>list_player_stop()</code>	停止播放
<code>list_player_pause()</code>	暂停播放
<code>list_player_resume()</code>	返回播放
<code>list_player_prev()</code>	播放上一曲
<code>list_player_next()</code>	播放下一曲
<code>list_player_detach_items()</code>	删除播放列表
<code>list_player_set_mode()</code>	设置播放器模式
<code>list_player_items_create()</code>	产生播放列表
<code>list_player_items_delete()</code>	删除播放列表
<code>list_player_items_empty()</code>	清空播放列表
<code>list_player_set_table_handler()</code>	播放列表完成时的回调函数
<code>list_player_items_get_num()</code>	播放列表中歌曲数目
<code>list_player_items_get_index()</code>	播放列表中当前的播放曲目索引
<code>list_player_items_get_item()</code>	播放列表中当前的播放曲目
<code>list_player_item_add()</code>	添加曲目到指定播放列表
<code>list_player_item_del()</code>	从指定播放列表删除曲目
<code>list_player_item_del_by_index()</code>	从指定播放列表删除指定索引曲目
<code>list_player_item_get()</code>	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

21.2.1 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

21.2.2 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

21.2.3 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

21.2.4 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

21.2.5 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

21.2.6 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

21.2.7 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

21.2.8 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

21.2.9 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

21.2.10 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

21.2.11 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

21.2.12 播放器停止

```
void list_player_stop(void)
```

参数	描述
void	空
返回	空

21.2.13 暂停播放

```
void list_player_pause(void)
```

参数	描述
void	空
返回	空

21.2.14 播放恢复

```
void list_player_resume(void)
```

参数	描述
void	空
返回	空

21.2.15 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

21.2.16 播放下一曲

```
void list_player_next(void)
```

参数	描述
void	空
返回	空

21.2.17 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

21.2.18 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

21.2.19 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

21.2.20 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

21.2.21 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

21.2.22 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

21.2.23 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

21.2.24 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

21.2.25 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

21.2.26 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

21.2.27 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

21.2.28 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

21.2.29 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

21.2.30 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

21.3 List Player示例代码

21.3.1 关键说明

- 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

21.3.2 示例代码

```
/*
 * 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 position = 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 = position;
    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);
```

21.4 操作说明

List Player示例代码参考\samples\Player\list_player_demo.c，开启List Player功能重新编译，下载运行后，需要首先连接网络然后在进行功能测试。

21.4.1 打开配置

打开宏定义：LIST_PLAY_TEST，开启list player的功能测试，编译后下载到设备。

21.4.2 运行现象

- 设备连接路由器

调试串口输入wifi_demo sta your_ssid your_key,设备开始连接路由器。

```
wifi_demo sta your_ssid your_key
sta_Command
ssid: your_ssid key: your_key
rl_sta_start
[sa_sta]MM_RESET_REQ
[sa_sta]ME_CONFIG_REQ
[sa_sta]ME_CHAN_CONFIG_REQ
[sa_sta]MM_START_REQ
hapd_intf_add_vif,type:2, s:0, id:0
[wlan_connect]:start tick = 0, connect done tick = 22379, total = 22379
[wlan_connect]:start tick = 0, connect done tick = 22385, total = 22385
[WLAN_MGMT]wlan sta connected evenew dtim period:2
nt callback
IP UP: 192.168.44.27
[ip_up]:start tick = 0, ip_up tick = 25797, total = 25797
```

- 开始播放

调试串口输入lister_play,用耳机接在Audio Out接口上即可听到声音，设备log信息如下所示：

```
lister_play
[icodec]:close sound device
audio_device_closed
close req, abort fetch data
web session: http://bernard.coding.me/channel/music/later.mp3
    Mime type: application/octet-stream
    position: 0
Content_length: 5411316
[icodec]:open sound device
audio_device_opened

====set fade in flag====
```

图21.4.3-1

调试串口输入lister_prev, lister_next可进行歌曲切换，输入lister_prev时，设备log信息如下所示：

```
msh />lister_prev
[icodec]:close sound device
audio_device_closed
web session: http://bernard.coding.me/channel/music/congcong.mp3
    Mime type: audio/mpeg
    position: 0
Content_length: 3857711
[icodec]:open sound device
audio_device_opened

====set fade in flag====

msh />
```

图21.4.3-2

22 混音

22.1 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。

22.2 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

22.2.1 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

22.2.2 暂停背景音播放

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

参数	描述
<code>void</code>	无
返回	<code>void</code>

22.2.3 重新播放背景音

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

参数	描述



void	无
返回	void

22.3 混音示例代码

22.3.1 关键说明

- 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

22.3.2 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);
```

22.4 操作说明

22.4.1 打开配置

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，设备需要播放云端音乐，所以必须开启list player的功能。

22.4.2 运行现象

- 使用配网命令，将设备连网成功，并且混音播放音乐

连网成功后，输入播放云端音乐的命令：list_player

<http://appfile.tuling123.com/media/audio/20180524/7983d929d01a4c8c99d28cf6ff2446ad.mp3>, audio out接口接入耳机，可以听到云端播放的音乐；demo板line in 接口需要接入播放音乐的播放设备作为背景音乐，这样可以同时听到云端播放的音乐和其他设备播放的音乐。设备连接如图：

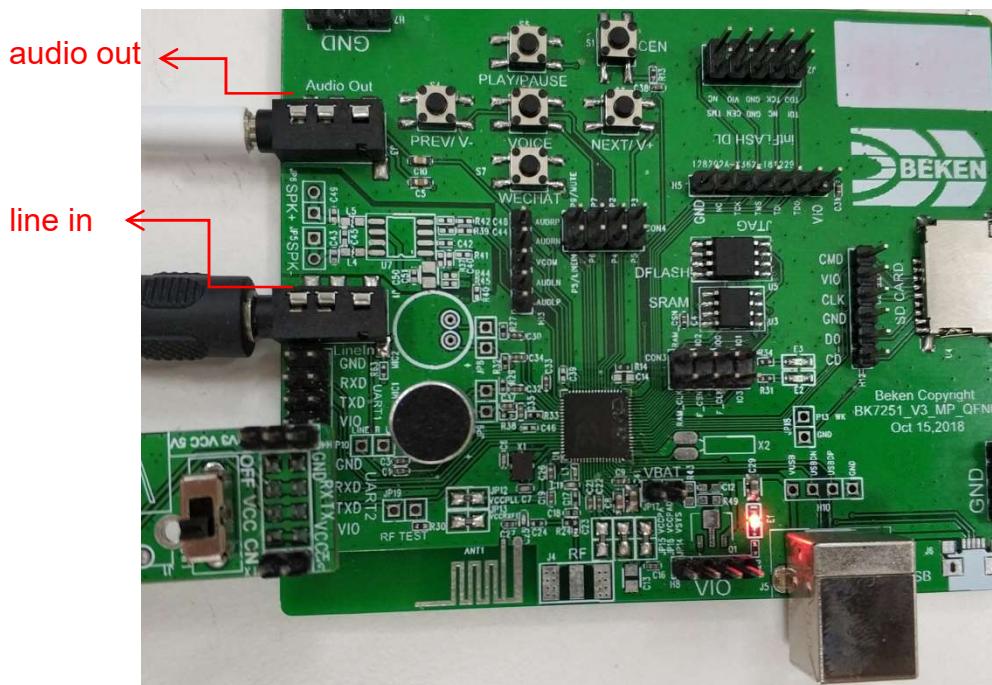


图22.4.2-1

•发送命令停止混音

输入命令：mixer_set_value 1 停止背景音乐的播放，mixer_set_value 0 播放背景音乐。



23 Vad

23.1 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。

23.2 Vad Related API

vad相关接口参考\beken378\func\vad.h，相关接口如下：

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

23.2.1 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
<code>void</code>	空
返回	0：成功； 其他：错误

23.2.2 获取帧的长度

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

参数	描述
<code>void</code>	空
返回	WB_FRAME_LEN：帧的长度

23.2.3 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
<code>buffer</code>	测试buffer



len	测试buffer长度
返回	vad_flag

23.2.4 关闭vad

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

参数	描述
void	空
返回	空

23.3 Vad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
#ifndef 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
```

23.4 操作说明

Vad示例代码参考\test\mic_record.c，具体使用方式如下：

23.4.1 打开配置

打开宏定义：MICPHONE_TEST，开启list player的功能测试，编译后下载到设备。

23.4.2 运行现象

上电后，在调试串口输入record_and_play，可听到识别的芯片中的声音，同时串口Log如下所示：

```
msh >/>record_and_play
adc-buf:00900cc8, adc-buf-len:5120, ch:1
audio_device_mic_opened
adc-buf:00900cc8, adc-buf-len:5120, ch:1
set adc channel 1
audio_device_mic_set_channel:1
set adc sample rate 16000
audio_device_mic_set_rate:16000
```



```
mic_read_len is 61440
audio_device_mic_closed
[icodec]:open sound device
audio_device_opened
```

```
====set fade in flag====
```

```
[icodec]:close sound device
audio_device_closed
msh />
```



24 AMR编码器

24.1 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。

24.2 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
amrnb_encoder_init()	amr编码初始化
amrnb_encoder_encode()	amr编码
amrnb_encoder_deinit()	退出amr编码

24.2.1 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
amrnb	AMR-NB编码器的指针
dtx	0: 连续传输数据 1: 不连续传输数据
pmalloc	malloc函数指针
pfree	free函数指针
返回	RT_EOK: 成功; 其他: 失败

24.2.2 AMR-NB编码

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

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



24.2.3 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。

24.3.1 关键说明

- **AMR编码器宏定义**

定义AMR编码器每帧中数据的大小

```
#define AMRNBNB_ENCODER_SAMPLES_PER_FRAME (160)
```

定义AMR编码器最大帧的大小

```
#define AMRNBNB_ENCODER_MAX_FRAME_SIZE (32)
```

- **AMR编码器枚举类型说明**

AMR编码速率枚举类型:

```
enum Mode {  
    AMRNBNB_MODE_MR475 = 0, /* 4.75 kbps */  
    AMRNBNB_MODE_MR515, /* 5.15 kbps */  
    AMRNBNB_MODE_MR59, /* 5.90 kbps */  
    AMRNBNB_MODE_MR67, /* 6.70 kbps */  
    AMRNBNB_MODE_MR74, /* 7.40 kbps */  
    AMRNBNB_MODE_MR795, /* 7.95 kbps */  
    AMRNBNB_MODE_MR102, /* 10.2 kbps */  
    AMRNBNB_MODE_MR122, /* 12.2 kbps */  
    AMRNBNB_MODE_MRDTX, /* DTX */  
    AMRNBNB_MODE_N_MODES /* Not Used */  
};
```

24.3.2 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 "amrnrb_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 = amrnbc_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);
{
    amrnrb_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_recv(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);
```

24.4 操作说明

24.4.1 下载AMR Player工具

AMR Player工具: [下载地址](#)

24.4.2 网络调试助手设置

本示例需要借助PC端工具网络调试助手和AMR Player工具，其中AMR Player用来播放amr声音文件。调试助手设置如下图



图24.4.2-1

24.4.3 打开配置

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。



24.4.4 运行现象

输入命令: record_amr_tcp start 采样率 网络地址 网络端口号
开始录音并且网络调试助手接收编码的数据流, 将数据流改为amr格式用AMR Player工具播放amr文件。



25 Opus编码器

25.1 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。

25.2 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

25.2.1 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

25.2.2 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



25.2.3 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

25.2.4 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

25.2.5 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

25.2.6 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

25.2.7 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

25.3 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。

25.3.1 关键说明

• 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
```

25.3.2 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(&(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_recv(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();
}
}

FINSH_FUNCTION_EXPORT_ALIAS(record_opus_tcp, __cmd_record_opus_tcp, record opus tcp);
```

25.4 操作说明

25.4.1 下载Cool Edit Pro工具

Cool Edit Pror工具: [下载地址](#)

25.4.2 网络调试助手设置

本示例需要借助PC端工具网络调试助手和Cool Edit Pro工具，其中Cool Edit Pro用来播放opus声音文件。调试助手设置如下图



图25.4.2-1

25.4.3 打开配置

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。



24.4.4 运行现象

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opus工具转换成pcm文件，通过cool edit pro 播放生成的pcm格式文件。

26 EasyFlash

26.1 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。

26.2 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
easyflash_init()	easyflash初始化
ef_get_env()	获得easyflash环境变量
ef_set_env()	写数据到easyflash中
ef_save_env()	保存数据到flash

26.2.1 easyflash初始化

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

参数	描述
void	空
返回	0: 成功; 其他: 失败

26.2.2 获得easyflash环境变量

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

参数	描述
key	环境变量名字
返回	value:变量地址

26.2.3 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
key	环境变量名字



value	要写入的值
返回	0: 成功; 其他: 失败

26.2.4 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

26.3 EasyFlash示例代码

26.3.1 关键说明

- **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

26.3.2 示例代码

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

#ifndef EASY_FLASH_TEST
```



```
static void easy_flash_set(char *key, char *value)
{
    EfErrCode result = EF_NO_ERR;
    easyflash_init();           /*初始化 */
    result = ef_set_env(key, value);      /*将要写入的数据存放到 easy flash 环境变量 */
    if(result != EF_NO_ERR)
    {
        rt_kprintf("easy_flash set error\r\n");
        return;
    }
    result = ef_save_env();          /*保存数据 */
    if(result != EF_NO_ERR)
    {
        rt_kprintf("easy_flash save error\r\n");
        return;
    }
    rt_kprintf("---Flash Write over \r\n");
}

static void easy_flash_get(char *key, char *value) /*读取easy flash 写入的数据*/
{
    easyflash_init();

    value = ef_get_env(key);      /*获取easy flash存入的数据*/
    if( value )
    {
        rt_kprintf("%s\r\n",value);
    }
    else
    {
        rt_kprintf("easy_flash get error\r\n");
    }
    return ;
}

static void easy_flash_erase(char *key) /*读取easy flash 写入的数据*/
{
    EfErrCode result = EF_NO_ERR;
    char value = 0;
```



```
easyflash_init()           /*初始化*/
result = ef_set_env(key, &value);      /*将要写入的数据存放到 easy flash 环境变量 */
if(result != EF_NO_ERR)
{
    rt_kprintf("easy_flash erase error\r\n");
}
else
{
    rt_kprintf("easy_flash erase success\r\n");
}
return;
}

static int easy_flash(uint8_t argc, char **argv)
{
    char *key = NULL;
    char *value = NULL;

    if (strcmp(argv[1], "set") == 0)
    {
        os_printf("easyflash set command\r\n");
        if (argc == 4)
        {
            key = argv[2];
            value = argv[3];
        }
        else
        {
            os_printf("parameter invalid\r\n");
            return -1;
        }
        easy_flash_set(key, value);
    }
    return 0;
}else if (strcmp(argv[1], "get") == 0)
{
    os_printf("easyflash get command\r\n");
    if (argc == 3)
    {
        key = argv[2];
        easy_flash_get(key, value);
    }
}
```

```
}

else

{

    os_printf("parameter invalid\r\n");

    return -1;

}

return 0;

}else if (strcmp(argv[1], "erase") == 0)

{

    os_printf("easyflash erase command\r\n");

    if (argc == 3)

    {

        key = argv[2];

        easy_flash_erase(key);

    }

    else

    {

        os_printf("parameter invalid\r\n");

        return -1;

    }

    return 0;

}

}

MSH_CMD_EXPORT(easy_flash, easy_flash_command: easy_flash <set/get/erase> <key> [value]);

#endif
```

26.4 操作说明

EasyFlash示例代码参考\test\ easyflash_test.c，使能后支持参数的读、写、擦除功能。

26.4.1 打开配置

打开宏定义：EASY_FLASH_TEST，编译后重新下载到设备。

26.4.2 运行现象

- 写入



调试串口输入**easy_flash set test 111111111111**, 其中key为“test”, value为“111111111111”, 设备log如下:

```
msh />easy_flash set test 111111111111
easyflash set command
[Flash]EasyFlash V3.0.4 already initialize.
[Flash]Erased ENV OK.
[Flash]Saved ENV OK.
---Flash Write over
```

• 读取

调试串口输入**easy_flash get test**, 读取key为“test”的数据, 设备log如下:

```
msh />easy_flash get test
easyflash get command
[Flash]EasyFlash V3.0.4 already initialize.
111111111111
```

• 擦除

调试串口输入**easy_flash erase test**, 擦除key为“test”的所有数据, 设备log如下:

```
msh />easy_flash erase test
easyflash erase command
[Flash]EasyFlash V3.0.4 already initialize.
easy_flash erase success
```

调试串口输入**easy_flash get test**, 再次读取key为“test”的数据, 以此来验证擦除操作是否成功, 擦除操作成功, 重新读取报错, 设备log如下:

```
msh />easy_flash get test
easyflash get command
[Flash]EasyFlash V3.0.4 already initialize.
easy_flash get error
```

27 Voice Changer

27.1 Voice Changer简介

Voice changer支持变声功能，能将声音转换成其他的声音特性。

27.2 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

27.2.1 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

27.2.2 退出voice changer

```
void voice_changer_exit(void);
```

参数	描述
void	空
返回	无

27.2.3 开始voice changer

```
void voice_changer_start(void);
```



参数	描述
void	空
返回	无

27.2.4 停止voice changer

```
void voice_changer_stop(void);
```

参数	描述
void	空
返回	无

27.2.5 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

27.2.6 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

27.2.7 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

27.2.8 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

27.3 Voice Changer示例代码

27.3.1 关键说明

- **Voice Changer宏定义**

#define CONFIG_VOICE_CHANGER	使用voice changer必须开启
------------------------------	---------------------

- **Voice Changer枚举类型**

```
typedef enum {
    VC_STOP,      //停止
    VC_FIRST,     //第一个
    VC_START,     //开始
} VC_STA;
```

27.3.2 示例代码

```
/*
* 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 acnnumber_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bufffer*/
{
    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);

    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;

#ifndef 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;
    }
#endif 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_THREAD_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;  
    }  
#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_THREADT_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_THREAD_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
```

27.4 操作说明

Voice changer示例代码参考\components\voice_changer\voice_changer_task.c。输入命令： voice_changer_sample launch/shutoff/next。

27.4.1 运行现象

输入命令： voice_changer_sample launch，对mic发声，可以明显发现自己的声音被拉长；输入命令： voice_changer_sample shutoff，对mic发声，可以明显发现自己的声音被拉短；输入命令： voice_config_stop 停止变声功能。



28 声波配网

28.1 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。

28.2 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

28.2.1 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

参数类型	
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



28.2.2 用户提前终止声波配网

```
void voice_config_stop(void)
```

参数	描述
void	无
返回	无

28.2.3 获取版本

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

参数	描述
void	无
返回	版本号

28.3 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\r\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;
}

#endif 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);
```

28.4 操作说明

28.4.1 打开配置

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。

28.4.2 运行现象

- 使用**voice_tools**生成.wav声音文件

运行**voice_tools.exe**，在cmd输入命令：

voice_tools "tp link" "passwd" "openid" wifi.wav 生成**wifi.wav**文件。运行cmd命令如下：

```
D:\111DDD\tools\voice_tool>voice_tools " " " " " "openid" wifi
4.wav
Shanghai Real Thread Electronic Technology Co.,Ltd.
voice config tools. V2.0.1
build Feb 15 2019 10:45:01

ssid[9]:
password[9]:
custom[6]: openid
raw data: data: 1A AC 77 69 66 69 2D 74 65 61 6D 00 73 74 6D 33 32 66 32 31 35
00 6F 70 65 6E 69 64
```

图28.4.2-1

- 输入命令:**voice_config**

用手机或者其他工具播放**wifi.wav**声音文件，**demo**板获取声音数据，连接生成.wav文件的网络。运行**log**如下：

```
voice_config
msh /]
msh />[voice] voice config version: 2.0.0
adc_buf:00900cc8, adc_buf_len:5120, ch:1
set adc sample_rate 16000
ssid len=9, [      ]
passwd L=9, [      ]
custom L=3, [      ]
[DRV_WLAN]drivers\wlan\drv_wlan.c L922 beken_wlan_control cmd: case
WIFI_INIT!
[Flash]ENV isn't initialize OK.
[wifi_connect]: read ap_info is empty
[wifi_connect]: normal connect
_wifi_easyjoin: ssid:wifi-team key:stm32f215
rl_sta_start
[sa_sta]MM_RESET_REQ
[sa_sta]ME_CONFIG_REQ
[sa_sta]ME_CHAN_CONFIG_REQ
[sa_sta]MM_START_REQ
hapd_intf_add_vif, type:2, s:0, id:0
wpa_dInit
wpa_supplicant_req_scan
Setting scan request: 0.100000 sec
MANUAL_SCAN_REQ
wpa_supplicant_scan
wpa_drv_scan
wpa_send_scan_req
scan_start_req_handler
wpa_driver_scan_cb
wpa_get_scan_rst:1
```

图28.4.2-2

网络连接成功log如下：

```
-----SM_CONNECT_IND_ok
wpa_driver_assoc_cb
Cancelling scan request
hapd_intf_add_key CCMP
add sta_mngt_get_st
sta:0, vif:0, key:0
sta_mngt_add_key
add hw key idx:24
add TKIP
add is_broadcast_ether_addr
sta:255, vif:0, key:1
add hw key idx:1
ctrl_port_hdl:1
[wlan_connect]: start tick = 8176, connect done tick = 12325, total = 4149
[wlan_connect]: start tick = 8176, connect done tick = 12331, total = 4155
[WLAN_MGMT]wlan sta connected event callback
sta_ip_start

configuring interface wlan (with DHCP client)
dhcp_check_status_init_timer

new dtim period:2
IP UP: 19
[ip_up]: start tick = 8176, ip_up tick = 15288, total = 7112
[Flash]ENV isn't initialize OK.
write new profile to flash 0x001FF000 72 byte!
[Flash]ENV isn't initialize OK.
-[31:22m[E/NTP]: ERROR select the socket timeout(10s)-[0m
```

图28.4.2-3

29 图像传输

29.1 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；
- d) 图像传输结构框图如下所示：

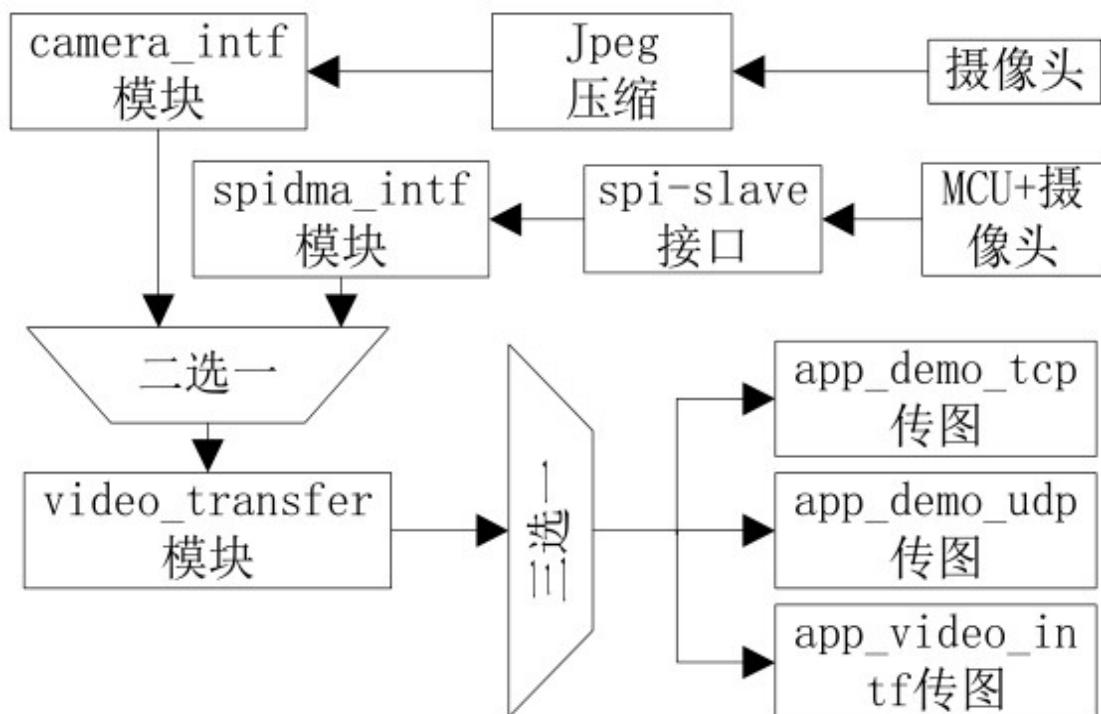


图29.1-1

如上图，图像的输入，可以选择bk7251自带的jpeg+摄像头模块，也可以通过spi-slave接口，外接MCU+摄像头模块。两种方式只能选择其中的一种。如下图，通过sys_config.h里的宏进行选择。

```
#define CFG_USE_SPIDMA          0  
#define CFG_USE_CAMERA_INTF      1
```

29.2 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



<code>video_transfer_set_video_param()</code>	使用DCMI接口时，设置摄像头的参数
<code>video_buffer_open()</code>	打开获取jpeg帧数据功能
<code>video_buffer_close()</code>	关闭获取jpeg帧数据功能
<code>video_buffer_read_frame()</code>	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

29.2.1 打开video_transfer

```
int video_transfer_init(TVIDEO_SETUP_DESC_PTR setup_cfg);
```

参数	描述
<code>setup_cfg</code>	TVIDEO_SETUP_DESC_PTR类型的结构体
返回	kNoErr: 成功；其他：失败

参数类型

`TVIDEO_SETUP_DESC_PTR :`

<code>UINT32 send_type</code>	一般为TVIDEO_SND_TYPE枚举类型，模块里会根据 <code>send_type</code> 决定每个图像数据包的大小。
<code>send_func</code>	图像数据的发送函数，模块转发图像数据包时，通过 <code>send_func</code> 发送。
<code>start_cb</code>	模块打开spi或camera_intf后，回调此函数，用于指示传图开始。
<code>end_cb</code>	模块关闭spi或camera_intf前，回调此函数，用于指示传图结束。
<code>pkt_header_size</code>	若要在图像数据包里加入“头信息”， <code>pkt_header_size</code> 用于指示“头信息”的大小，注意 <code>pkt_header_size</code> 的值必须是4的整数倍。如果不使用“头信息”，设成0即可。
<code>add_pkt_header</code>	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

29.2.2 关闭video_transfer

```
int video_transfer_deinit(void);
```

参数	描述
<code>void</code>	无
返回	kNoErr: 成功；其他：失败

29.2.3 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 seccond)，见FPS_TYPE的定义
返回	0: 成功; 1: 失败

29.2.4 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

29.2.5 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

29.2.6 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

29.3 图像传输的示例代码

29.3.1 关键说明

- 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

- 图像传输枚举类型说明

```
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;          /*通过其他接口上传*/

typedef enum {
    QVGA_320_240 = 0,
    VGA_640_480,
    PPI_MAX
} PPI_TYPE;                /*分辨率的枚举*/

typedef enum {
    TYPE_5FPS = 0,
    TYPE_10FPS,
    TYPE_20FPS,
    FPS_MAX
} FPS_TYPE;                /*帧率的枚举*/
```

29.3.2 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");
#ifndef (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_romote_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_CONECTED);
}

/*指示停止传图 */
static void app_demo_udp_app_disconnected(void)
{
    app_demo_softap_send_msg(DMSG_APP_DISCONECTED);
}

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 buf\n");  
            return;  
        }  
        my_len = video_buffer_read_frame(mybuf, my_len);  
        rt_kprintf("frame_len: %d\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);
```

29.4 操作说明

图像传输示例代码参考\beken378\app\app_demo文件夹下，流程如下图：

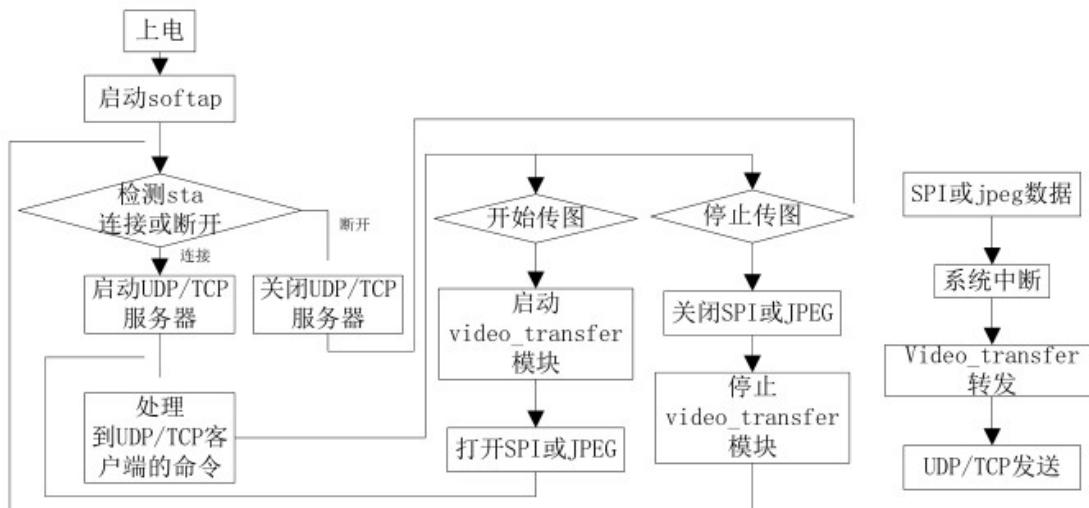


图29.4-1

如上图，上电后，会启动softap（启动参数可配置），然后监听station设备来连接（最多连接一个station）。

若检测到station连接成功，就会启动UDP或TCP服务器，等待station端发起UDP或TCP连接（UDP是无连接的，这里时UDP自定义的命令）。处理相关station发过来的数据。

若为打开图传，则打开video_transfer模块，video_transfer模块里会打开spi或jpeg。从此开始，spi接口或jpeg里的图像数据会通过系统中断的方式触发video_transfer的转发函数，最终通过UDP或TCP方式发送。

若为关闭图传，则关闭video_transfer模块，该模块里会关闭spi接口或jpeg模块，图像数据不再接收了。

若检测到station断线，会先关闭UDP或TCP服务，并关闭video_transfer模块。

29.4.1 下载PC调试工具

调试工具在SDK根目录tool\beken_wifi_camera文件夹中。

29.4.2 启动softap

调试串口输入video_demo，开启softap，PC机(带wifi功能的笔记本)，找到BK_WIFI_00000的ssid，softap发现station连接成功后，会启动UDP和TCP服务传输图像，设备log下：

```
hapd_intf_sta_add:1, vif:0
rc_init: station_id=0 format_mod=0 pre_type=0 short_gi=0 max_bw=0
rc_init: nss_max=0 mcs_max=255 r_idx_min=0 r_idx_max=11 no_samples=10
sta_idx:0, pm_state:0
RW_EVT_AP_CONNECTED-(mac=3c:f0:11:46:29:b3 )
app_demo_udp_init
app_demo_udp_main entry
app_demo_tcp_init
app_demo_tcp_main entry
wifi connected!
```

29.4.3 UDP传输测试

运行BK_Wifi_Camera.exe，设置locate_ip和remote_ip后，勾选By UDP选项框，点击Play/Stop按钮，会实时显示摄像头采集的图像。

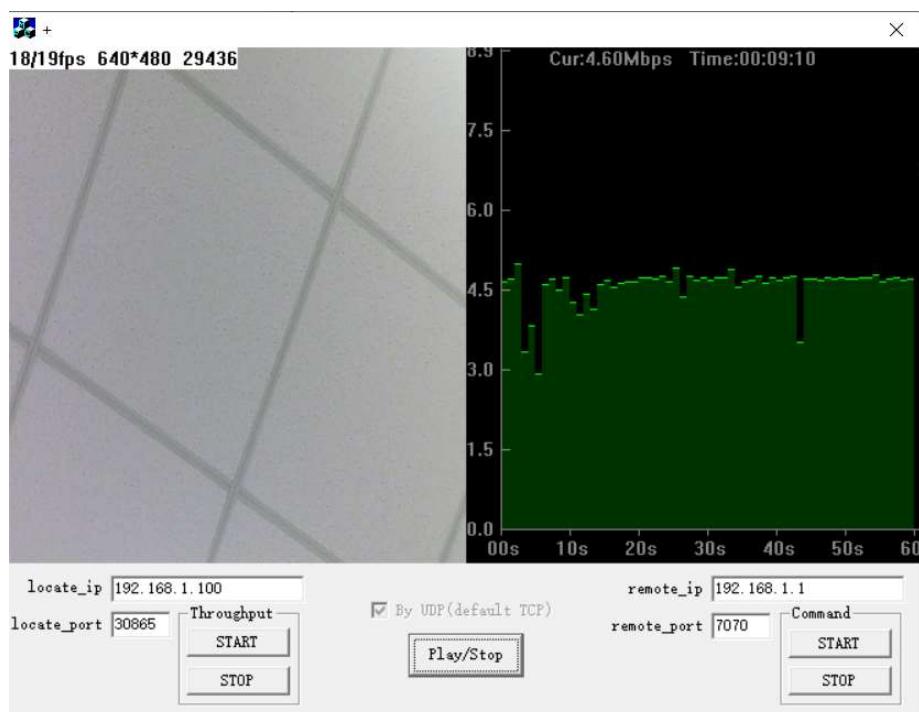


图29.4.3-1

29.4.4 TCP传输测试

运行BK_Wifi_Camera.exe，设置locate_ip和remote_ip后，不勾选ByUDP选项框，点击Play/Stop按钮，会实时显示摄像头采集的图像。

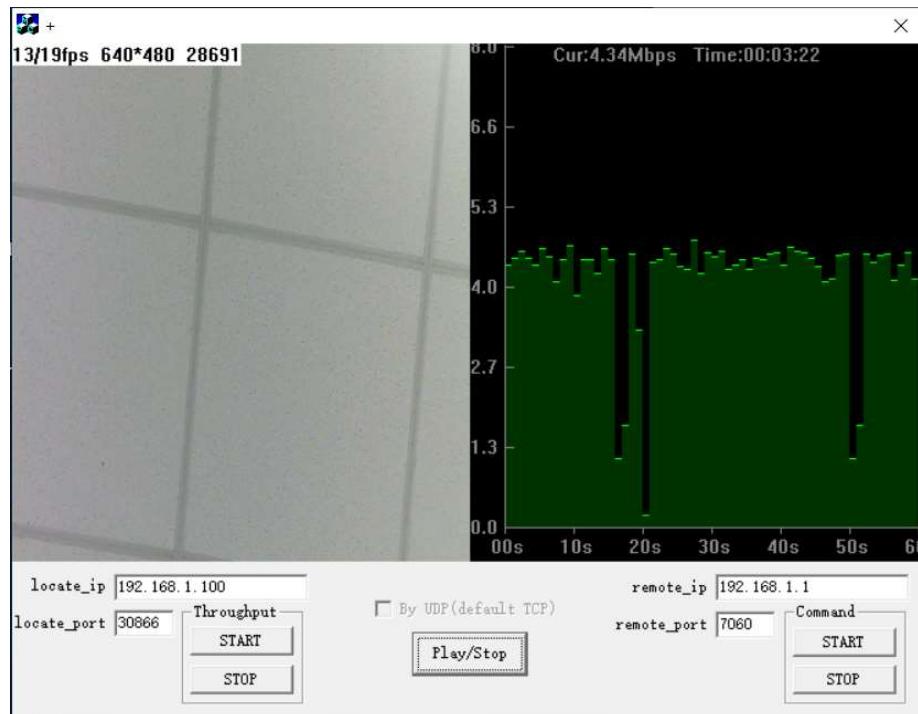


图29.4.4-1