



US282A 项目一

软件开发指南

最新版本：1.0

2015-12-22

声 明

Disclaimer

Information given in this document is provided just as a reference or example for the purpose of using Actions' products, and cannot be treated as a part of any quotation or contract for sale.

Actions products may contain design defects or errors known as anomalies or errata which may cause the products' functions to deviate from published specifications. Designers must not rely on the instructions of Actions' products marked "reserved" or "undefined". Actions reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

ACTIONS DISCLAIMS AND EXCLUDES ANY AND ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY AND ALL EXPRESS OR IMPLIED WARRANTIES OF MERCHANTABILITY, ACCURACY, SECURITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND AGAINST INFRINGEMENT OF INTELLECTUAL PROPERTY AND THE LIKE TO THE INFORMATION OF THIS DOCUMENT AND ACTIONS PRODUCTS.

IN NO EVENT SHALL ACTIONS BE LIABLE FOR ANY DIRECT, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION FOR LOSS OF DATA, PROFITS, SAVINGS OR REVENUES OF ANY KIND ARISING FROM USING THE INFORMATION OF THIS DOCUMENT AND ACTIONS PRODUCTS, REGARDLESS OF THE FORM OF ACTION, WHETHER BASED ON CONTRACT; TORT; NEGLIGENCE OF ACTIONS OR OTHERS; STRICT LIABILITY; OR OTHERWISE; WHETHER OR NOT ANY REMEDY OF BUYER IS HELD TO HAVE FAILED OF ITS ESSENTIAL PURPOSE, AND WHETHER ACTIONS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR NOT.

Actions' products are not designed, intended, authorized or warranted for use in any life support or other application where product failure could cause or contribute to personal injury or severe property damage. Any and all such uses without prior written approval of an Officer of Actions and further testing and/or modification will be fully at the risk of the customer.

Ways of obtaining information

Copies of this document and/or other Actions product literature, as well as the Terms and Conditions of Sale Agreement, may be obtained by visiting Actions' website at: <http://www.actions-semi.com> or from an authorized Actions representative.

Trademarks

The word "Actions" and the logo are the trademarks of Actions Semiconductor Co., Ltd, and Actions (Zhuhai) Technology Co., Limited is authorized to use them. Word "炬芯" is the trademark of Actions (Zhuhai)

Technology Co., Limited. Names and brands of other companies and their products that may from time to time descriptively appear in this document are the trademarks of their respective holders, no affiliation, authorization, or endorsement by such persons are claimed or implied except as may be expressly stated therein.

Rights Reserved

The provision of this document shall not be deemed to grant buyers any right in and to patent, copyright, trademark, trade secret, know how, and any other intellectual property of Actions or others.

Miscellaneous

Information contained or described herein relates only to the Actions products and as of the release date of this publication, abrogates and supersedes all previously published data and specifications relating to such products provided by Actions or by any other person purporting to distribute such information.

Actions reserves the rights to make changes to information described herein at any time without notice. Please contact your Actions sales representatives to obtain the latest information before placing your product order.

Additional Support

Additional products and company information can be obtained by visiting the Actions website at: <http://www.actions-semi.com>

支持:

如欲获得公司及产品的其它信息, 欢迎访问我公司网站: <http://www.actions-semi.com>

目 录

声 明	2
目 录	4
版本历史	21
1 引 言	22
1.1 编写目的	22
1.2 术语和缩写词	22
1.3 参考文档	22
2 阅读指引	23
3 开发环境介绍	24
3.1 搭建开发环境	24
3.1.1 Cygwin的安装	24
3.1.1.1 全新安装cygwin	24
3.1.1.2 卸载cygwin	27
3.1.2 SDE工具链的安装	27
3.1.3 多媒体固件修改工具的安装	28
3.1.4 多媒体产品量产工具的安装	28
3.1.5 量产卡固件烧写工具的安装	28
3.1.6 音效调试工具的安装	29
3.1.7 通话效果调试工具的安装	29
3.2 软件构建工具链的使用	29
3.2.1 编译与链接	30
3.2.2 打包固件	31
3.2.3 升级固件 (ATS2825)	31
3.2.4 升级固件 (ATS2823)	32
3.2.5 修改固件	37
3.2.6 修改资源文件	39
3.2.6.1 TTS资源文件	39
3.2.6.2 PCM语音文件	40
3.2.6.3 内置闹铃文件	40

3.3	软件调试手段	40
3.3.1	串口打印	40
4	SDK软件包介绍	42
4.1	Psp_rel目录介绍	42
4.1.1	Bin 目录	42
4.1.2	Cfg 目录	43
4.1.3	Include 目录	43
4.1.4	Lib 目录	43
4.1.5	Tools 目录	44
4.1.6	Usermodule 目录	44
4.2	Case目录介绍	45
4.2.1	Ap 目录	45
4.2.2	Common 目录	47
4.2.3	Cfg 目录	48
4.2.4	Drv 目录	49
4.2.5	Fwpkg 目录	50
4.2.6	Config_txt 目录	51
4.2.7	Inc 目录	53
4.2.8	Lib 目录	54
4.2.9	Tools 目录	54
5	软件基本组成介绍	55
5.1	概述	55
5.2	引导程序	56
5.3	操作系统	56
5.3.1	BANK 机制	56
5.3.2	API 机制	58
5.3.3	中断机制	60
5.3.4	VFS 机制	61
5.4	算法库	62
5.5	中间件	62
5.6	驱动程序	63
5.7	应用程序	64
5.8	其他独立程序实体	64
5.9	API接口	64

6	COMMON详解	65
6.1	概述	65
6.2	APPLIB	65
6.2.1	应用程序管理	65
6.2.1.1	应用管理	65
6.2.1.2	引擎管理	66
6.2.1.3	运行时库 Ctor.o	67
6.2.2	消息通信管理	67
6.2.2.1	概述	67
6.2.2.2	按键消息	71
6.2.2.3	系统消息	73
6.2.2.4	应用私有消息	73
6.2.2.5	蓝牙协议栈消息	74
6.2.2.6	蓝牙协议栈事件	75
6.2.3	软定时器管理	75
6.2.4	配置项解释	77
6.3	APP_RESULT_E	78
6.4	VIEW机制	79
6.4.1	概述	79
6.4.2	基本元素	79
6.4.2.1	主视图	80
6.4.2.2	子视图	81
6.4.2.3	消息视图	82
6.4.3	VIEW LOOP	83
6.4.3.1	按键消息处理	83
6.4.3.2	系统消息处理	84
6.4.3.3	蓝牙事件处理	84
6.5	应用切换	84
6.6	快捷键响应	86
6.7	按键消息预处理	86
6.8	系统消息响应	87
6.9	系统消息预处理	87
6.10	分立LED显示	88
6.11	按键音播放	89
6.12	TTS播放	90

6.13	远程控制协议 RCP.....	92
6.13.1	概述	92
6.13.2	命令包格式	93
6.13.3	RCP事件映射机制	94
6.13.4	外部接口设计	96
6.13.5	自定义协议	97
6.13.6	分段机制实现说明（该部分尚未完整实现）	97
6.13.7	用户交互说明	97
6.13.8	注意事项	98
6.14	时间日历闹钟	98
6.14.1	时间显示	98
6.14.2	日历显示及设置	99
6.14.3	闹钟显示及管理	99
6.15	供电方式与运行模式	100
6.16	电池供电充电电量测量	100
6.17	低功耗模式	102
6.18	蓝牙管理器	104
6.19	PA与外部功放.....	104
6.20	音量调节	105
6.21	数字音效调节	108
6.22	获取能量及频谱等	110
6.23	系统监控功能	111
6.24	其他功能	111
6.24.1	应用睡眠	111
6.25	COMMON工程	111
7	蓝牙框架详解	113
7.1	概述	113
7.2	蓝牙管理器	114
7.2.1	功能框图	114
7.2.2	可见性、连接、回连、设备管理、Sniff.....	115
7.2.3	蓝牙装载、卸载、S3BT	116
7.2.4	实时更新BT STATUS	116
7.2.5	生成随机地址	117
7.2.6	AVRCP控制适配层.....	117

7.2.7	HFP控制适配层	117
7.2.8	SPP/BLE数传适配层	117
7.2.9	HID控制适配层	118
7.2.10	蓝牙管理器LOOP	118
7.2.11	蓝牙管理器工作状态	118
7.3	BT STACK解析	119
7.4	AVDTP数据管道	119
7.5	SPP/BLE数据管道	119
7.6	蓝牙应用	120
8	CASE空间分配详解	121
8.1	概述	121
8.2	COMMON内存分配	123
8.2.1	所有应用共享	123
8.2.1.1	AP Manager应用	123
8.2.1.2	BT STACK应用	124
8.2.1.3	引擎应用	124
8.2.1.4	前台应用	124
8.2.2	前台应用COMMON	125
8.3	AP Manager内存分配	126
8.4	前台应用内存分配	126
8.5	前台中间件内存分配	127
8.6	引擎应用内存分配	128
8.7	后台蓝牙内存分配	128
8.8	按键驱动内存分配	128
8.9	LED驱动内存分配	130
8.10	CCD驱动内存分配	130
8.11	栈空间分配	130
8.12	内存分配复用说明	131
8.13	内存分配调整说明	132
8.14	链接脚本 *.xn 说明	132
8.14.1	输入段与输出段	132
8.14.2	链接脚本 *.xn	133

8.15	Makefile说明	135
8.16	*.map 和 *.info 解读	137
8.16.1	*.map 解读	137
8.16.2	*.info 解读	138
8.17	BANK使用考量	138
8.18	系统堆空间	138
8.19	VRAM空间分配	139
8.20	FAQ	140
8.20.1	如何增加一个BANK?	140
8.20.2	如何将相应的代码放置到对应的BANK?	141
8.20.3	如何增加常驻代码段?	141
8.20.4	如何处理空间越界问题?	142
8.20.5	如何处理栈越界?	143
9	应用管理器介绍	144
9.1	AP Manager的地位与功能	144
9.2	AP Manager的设计要点	144
9.3	AP Manager初始化	145
9.4	AP Manager主循环	145
9.5	创建应用程序的流程	146
9.6	杀死应用程序的流程	147
10	前台应用详解	148
10.1	应用程序 *.ap 文件结构	148
10.2	前台应用工程概述	150
10.2.1	工程组成	150
10.2.2	应用的一般流程	151
10.2.3	初始化流程	151
10.2.4	退出流程	152
10.2.5	主视图/场景调度	153
10.2.6	主视图主循环	154
10.2.7	链接 *.xn 脚本与Makefile脚本	155
10.3	前台应用开发要点	155
10.3.1	引擎应用的创建和杀死	155
10.3.2	与引擎应用的通信方式	155
10.3.2.1	共享查询	155

10.3.2.2	共享内存	156
10.3.2.3	消息通信	157
10.4	FAQ	157
10.4.1	如何增加一个前台应用	157
10.4.2	如何删除一个前台应用	158
10.5	开关机应用	159
10.5.1	AP config的功能设计	159
10.5.2	AP config的开机功能	159
10.5.3	AP config的关机功能	161
10.5.4	AP config的充电场景	161
10.5.5	AP config的空闲场景	162
11	引擎应用详解	163
11.1	引擎应用工程概述	163
11.1.1	工程组成	163
11.1.2	应用的一般流程	164
11.1.3	初始化流程	164
11.1.4	退出流程	165
11.1.5	业务主循环	165
11.1.6	链接 *.xn 脚本与Makefile脚本	165
11.2	FAQ	166
11.2.1	如何增加一个引擎应用	166
11.2.2	如何删除一个引擎应用	166
12	驱动程序详解	167
12.1	驱动程序 *.drv 文件结构	167
12.2	驱动程序工程概述	169
12.2.1	工程组成	169
12.2.2	驱动程序装载	169
12.2.3	驱动程序卸载	170
12.2.4	链接 *.xn 脚本与Makefile脚本	170
12.3	驱动程序API接口	171
12.4	注意事项	171
12.5	FAQ	171
12.5.1	如何增加一个驱动程序	171
12.5.2	如何删除一个驱动程序	172
13	前台中间件详解	173

13.1	前台中间件 *.al 文件结构.....	173
13.2	前台中间件工程概述	174
13.2.1	工程组成	174
13.2.2	前台中间件装载	175
13.2.3	前台中间件卸载	175
13.2.4	链接 *.xn 脚本与Makefile脚本	175
13.3	前台中间件API接口	176
13.4	FAQ	176
14	CASE驱动程序详解.....	177
14.1	KEY驱动程序	177
14.1.1	KEY的地位和功能	177
14.1.2	KEY按键检测原理	178
14.1.2.1	GPIO按键的原理	178
14.1.2.2	ON_OFF按键的原理	178
14.1.2.3	LRADC1 按键的原理	179
14.1.2.4	红外遥控器的原理	179
14.1.3	KEY驱动程序工程概述	180
14.1.3.1	源代码组成	180
14.1.3.2	KEY驱动初始化与退出	180
14.1.4	KEY按键扫描	181
14.1.5	调频管理	182
14.1.6	充电和电量检测	182
14.1.7	分立LED灯显示.....	183
14.1.8	设备热拔插检测	184
14.1.9	FAQ.....	184
14.1.9.1	如何修改物理按键设计.....	184
14.1.9.2	如何修改按键周期时间参数.....	184
14.2	LED驱动程序.....	185
14.2.1	LED的地位和功能.....	185
14.2.2	LED驱动程序的接口设计.....	185
14.2.3	LED驱动程序的实现.....	186
14.2.4	LED驱动程序的使用说明.....	186
14.2.5	FAQ.....	187
14.2.5.1	如何更换一款LED数码屏.....	187
14.3	Welcome模块	188
14.3.1	Welcome的地位和功能	188
14.3.2	Welcome的设计要点	189

14.3.3	Welcome的启动流程	190
14.3.4	注意事项	190
15	音频播放模型介绍	191
16	CASE软件开发要点	192
16.1	多线程设计与开发	192
16.1.1	多线程架构	192
16.1.1.1	应用程序、进程、线程、任务及其优先级	192
16.1.1.2	任务调度机制	193
16.1.1.3	创建子线程	194
16.1.1.4	销毁子线程	195
16.1.1.5	多线程注意事项	195
16.2	进程/线程间通信方式	195
16.2.1.1	通信方式概述	195
16.2.1.2	信号量	196
16.2.1.3	互斥锁	196
16.2.1.4	条件变量	196
16.2.1.5	消息队列	196
16.2.1.6	共享查询	196
16.2.1.7	共享内存	198
16.2.1.8	跨进程的全局变量	199
16.3	头文件依赖关系	199
16.4	多方案开发环境	200
16.5	EJTAG调试接口关闭	201
16.6	使用打印调试	201
16.7	编译工具链	201
16.7.1.1	编译优化选项	201
16.7.1.2	*.o strip debug	202
16.7.1.3	Makefile的陷阱	203
17	蓝牙推歌功能	204
17.1	应用规格	204
17.2	应用架构	204
17.3	蓝牙推歌前台设计	205
17.3.1.1	功能及设计	205
17.3.1.2	状态	206
17.3.1.3	应用结构	206

17.3.1.4	播放主视图设计	207
17.4	蓝牙推歌引擎设计	208
17.4.1.1	功能及设计	208
17.4.1.2	状态	209
17.4.1.3	应用结构	209
17.5	蓝牙推歌中间件	210
17.6	注意事项	211
18	蓝牙免提功能	212
18.1	应用规格	212
18.2	应用架构	212
18.3	前台介绍	213
18.4	引擎介绍	215
18.5	中间件及dsp	215
19	本地播歌功能	217
19.1	功能介绍	217
19.2	总体结构介绍	218
19.3	前台应用详解	219
19.3.1	总体介绍	219
19.3.2	与其他模块的同步和交互	219
19.3.3	视图及功能	220
19.3.3.1	播放状态视图	220
19.3.3.2	曲目号视图	220
19.3.3.3	数字选歌视图	220
19.3.3.4	循环模式视图	220
19.3.3.5	AB播放视图	220
19.3.4	RCP处理	220
19.3.5	获取ID3	221
19.4	引擎应用详解	221
19.4.1	引擎的总体流程	222
19.4.2	引擎的状态处理	223
19.5	文件扫描详解	223
19.6	中间件及解码部分	223
19.6.1	中间件的线程	224

19.7	卡拔出后的处理	224
19.8	格式不支持文件	224
20	音频输入功能	226
20.1	功能介绍	226
20.2	总体结构介绍	227
20.3	前台应用详解	228
20.4	引擎应用详解	229
20.5	中间件介绍	229
21	USB音箱功能.....	231
21.1	功能介绍	231
21.2	总体结构介绍	232
21.3	前台应用详解	233
21.3.1	与其他模块的同步和交互	233
21.3.2	依赖库及其接口说明	234
21.3.3	应用的业务流程	235
21.4	引擎应用详解	236
21.4.1	引擎的总体流程	237
21.4.2	引擎的调用	237
21.5	中间件介绍	238
21.6	驱动详解	238
21.6.1	总体设计	239
21.6.2	USB驱动的应用接口.....	240
21.6.3	USB驱动的配置说明.....	241
22	USB读卡器功能.....	242
22.1	功能介绍	242
22.2	总体结构介绍	242
22.3	前台应用介绍	243
22.3.1	依赖库及其接口说明	243
22.3.2	应用的业务流程	244
22.4	驱动介绍	244
22.4.1	总体设计	245
22.4.2	USB驱动的应用接口.....	245

23	FM收音机功能	247
23.1	功能介绍	247
23.2	总体结构介绍	248
23.3	前台应用详解	249
23.3.1	与其他模块的同步和交互	249
23.3.2	依赖库及其接口说明	250
23.3.3	应用的业务流程	251
23.3.4	播放视图	251
23.3.5	硬件搜台视图	252
23.3.6	软件搜台视图	253
23.4	引擎应用详解	253
23.4.1	引擎的总体流程	254
23.4.2	引擎的调用	254
23.5	中间件介绍	255
23.6	驱动详解	255
23.6.1	总体设计	257
23.6.2	FM驱动的模块划分	258
23.6.3	FM驱动的硬件接口设计	258
23.6.4	FM驱动的应用接口设计	259
23.6.5	FM驱动提供的统一接口及每个宏定义接口的介绍	259
23.6.6	FM驱动的数据流图	260
23.6.7	FM驱动的内存分配说明	260
23.6.8	FM驱动的的修改指南	261
23.6.9	FM驱动的配置说明	261
23.6.10	前台是怎样设置频率的	261
23.6.11	如何增加一个FM驱动的应用接口	262
24	API介绍	263
24.1	KERNEL API	263
24.1.1	时间管理API	263
24.1.1.1	sys_set_irq_timer1	263
24.1.1.2	sys_del_irq_timer1	263
24.1.1.3	sys_udelay	264
24.1.1.4	sys_mdelay	264
24.1.1.5	sys_get_ab_timer	264
24.1.1.6	sys_set_time	264
24.1.1.7	sys_get_time	265
24.1.1.8	sys_set_date	265

24.1.1.9	sys_get_date.....	265
24.1.1.10	sys_set_alarm_time.....	266
24.1.1.11	sys_get_alarm_time	266
24.1.1.12	sys_os_time_dly.....	266
24.1.1.13	sys_os_time_dly_resume	266
24.1.1.14	sys_sleep.....	267
24.1.1.15	sys_usleep.....	267
24.1.1.16	sys_get_delay_val.....	267
24.1.1.17	sys_us_timer_start	268
24.1.1.18	sys_us_timer_break	268
24.1.1.19	sys_reset_timer	268
24.1.2	设备驱动管理API.....	268
24.1.2.1	sys_drv_install.....	268
24.1.2.2	sys_drv_uninstall	269
24.1.2.3	sys_get_drv_install_info.....	269
24.1.2.4	sys_detect_disk	269
24.1.2.5	sys_set_drv_setting.....	270
24.1.2.6	sys_set_drv_ops.....	270
24.1.3	VFS管理 API.....	271
24.1.3.1	sys_mount_fs	271
24.1.3.2	sys_unmount_fs	271
24.1.3.3	sys_get_fat_type_after_mount.....	271
24.1.3.4	sys_format_disk.....	272
24.1.4	SDFS管理API	272
24.1.4.1	sys_sd_fopen.....	272
24.1.4.2	sys_sd_fclose	272
24.1.4.3	sys_sd_fseek	273
24.1.4.4	sys_sd_ftell	273
24.1.4.5	sys_sd_fread.....	274
24.1.4.6	sys_base_set_info	274
24.1.4.7	get_fw_info.....	274
24.1.5	VRAM读写API.....	275
24.1.5.1	sys_vm_read	275
24.1.5.2	sys_vm_write.....	275
24.1.6	中断管理API.....	276
24.1.6.1	sys_request_irq	276
24.1.6.2	sys_free_irq.....	276
24.1.6.3	sys_local_irq_save	276
24.1.6.4	sys_local_irq_restore	277
24.1.6.5	sys_set_mpu_irq	277

24.1.6.6	sys_del_mpu_irq	277
24.1.6.7	sys_request_dsp_irq	277
24.1.6.8	sys_respond_dsp_cmd	278
24.1.7	AP管理API	278
24.1.7.1	sys_exece_ap	278
24.1.7.2	sys_free_ap	278
24.1.7.3	sys_load_codec	279
24.1.7.4	sys_free_codec	279
24.1.7.5	sys_load_mmm	279
24.1.7.6	sys_free_mmm	280
24.1.7.7	sys_load_dsp_codec	280
24.1.7.8	sys_free_dsp_codec	280
24.1.8	调频管理API	281
24.1.8.1	sys_adjust_clk	281
24.1.8.2	sys_adjust_asrc_clk	281
24.1.8.3	sys_request_clkadjust	281
24.1.8.4	sys_free_clkadjust	282
24.1.8.5	sys_lock_adjust_freq	282
24.1.8.6	sys_unlock_adjust_freq	282
24.1.9	内存管理API	283
24.1.9.1	sys_malloc	283
24.1.9.2	sys_free	283
24.1.10	消息处理API	283
24.1.10.1	sys_mq_send	283
24.1.10.2	sys_mq_receive	284
24.1.10.3	sys_mq_flush	284
24.1.10.4	sys_mq_traverse	284
24.1.11	共享查询机制API	285
24.1.11.1	sys_share_query_creat	285
24.1.11.2	sys_share_query_destroy	285
24.1.11.3	sys_share_query_read	285
24.1.11.4	sys_share_query_update	286
24.1.12	共享内存机制API	286
24.1.12.1	sys_share_query_creat	286
24.1.12.2	sys_shm_destroy	287
24.1.12.3	sys_shm_mount	287
24.1.13	工作&信息配置API	287
24.1.13.1	sys_enter_high_powered	287
24.1.13.2	sys_exit_high_powered	288
24.1.13.3	sys_set_hosc_param	288

24.1.13.4	sys_set_sys_info	288
24.1.13.5	sys_get_sys_info	288
24.1.13.6	sys_random	289
24.1.13.7	sys_read_c0count	289
24.1.14	系统调试API	289
24.1.14.1	sys_cpu_monitor_start	289
24.1.14.2	sys_cpu_monitor_end	290
24.1.14.3	sys_dsp_print	290
24.1.14.4	sys_irq_print	290
24.2	LIBC API	290
24.2.1	线程管理API	291
24.2.1.1	libc_pthread_create	291
24.2.1.2	libc_pthread_exit	291
24.2.2	进程管理API	291
24.2.2.1	libc_exit	291
24.2.2.2	libc_waitpid	292
24.2.2.3	libc_get_process_struct	292
24.2.2.4	libc_free_process_struct	292
24.2.3	信号量API	293
24.2.3.1	libc_sem_init	293
24.2.3.2	libc_sem_wait	293
24.2.3.3	libc_sem_trywait	293
24.2.3.4	libc_sem_post	294
24.2.3.5	libc_sem_destroy	294
24.2.4	互斥量API	294
24.2.4.1	libc_pthread_mutex_init	295
24.2.4.2	libc_pthread_mutex_lock	295
24.2.4.3	libc_pthread_mutex_unlock	295
24.2.4.4	libc_pthread_mutex_trylock	296
24.2.4.5	libc_pthread_mutex_destroy	296
24.2.5	条件变量API	296
24.2.5.1	libc_pthread_cond_init	296
24.2.5.2	libc_pthread_cond_wait	297
24.2.5.3	libc_pthread_cond_signal	297
24.2.5.4	libc_pthread_cond_destroy	297
24.2.6	字符串操作API	298
24.2.6.1	libc_memcpy	298
24.2.6.2	libc_memset	298
24.2.6.3	libc_memcmp	298

24.2.6.4	libc_strlen	299
24.2.6.5	libc_strncat.....	299
24.2.6.6	libc_strncmp.....	299
24.2.6.7	libc_strncpy.....	300
24.2.6.8	libc_strlenuni	300
24.2.6.9	libc_strncpyuni.....	300
24.2.6.10	libc_itoa	301
24.2.7	Uart print API	301
24.2.7.1	libc_print.....	301
24.3	文件系统 API.....	302
24.3.1	目录操作接口	303
24.3.1.1	vfs_cd.....	303
24.3.1.2	vfs_dir	303
24.3.1.3	vfs_make_dir	304
24.3.2	文件操作接口	304
24.3.2.1	vfs_file_open	304
24.3.2.2	vfs_file_create.....	305
24.3.2.3	vfs_file_close	305
24.3.2.4	vfs_file_get_size	306
24.3.2.5	vfs_get_time.....	306
24.3.2.6	vfs_file_seek	307
24.3.2.7	vfs_file_tell	307
24.3.2.8	vfs_file_read	308
24.3.2.9	vfs_file_write.....	308
24.3.2.10	vfs_set_time	309
24.3.2.11	vfs_cut_file_tail	309
24.3.2.12	vfs_file_divided	310
24.3.2.13	vfs_file_insert	310
24.3.3	公共操作接口	311
24.3.3.1	vfs_file_attralter.....	311
24.3.3.2	vfs_file_dir_offset.....	311
24.3.3.3	vfs_file_dir_remove.....	312
24.3.3.4	vfs_get_err_info.....	312
24.3.3.5	vfs_file_dir_exist	313
24.3.3.6	vfs_file_rename	313
24.3.3.7	vfs_get_name	314
24.3.3.8	vfs_get_space.....	314
24.3.3.9	vfs_create_volume	315
24.3.3.10	vfs_jump_to_direntry.....	315

24.3.3.11	vfs_dir_current_entry_file	315
24.3.3.12	vfs_file_move	316
24.4	AUDIO DEVICE API.....	316
24.4.1	音频输出	317
24.4.1.1	ENABLE_PA	317
24.4.1.2	DISABLE_PA	318
24.4.1.3	SET_PA_VOLUME.....	318
24.4.1.4	GET_PA_VOLUME.....	318
24.4.1.5	ENABLE_DAC	319
24.4.1.6	DISABLE_DAC	319
24.4.1.7	SET_DAC_RATE	319
24.4.1.8	GET_DAC_RATE	320
24.4.1.9	ENABLE_AIN_OUT	320
24.4.1.10	ENABLE_AIN_OUT	320
24.4.2	音频输入	321
24.4.2.1	ENABLE_AIN.....	321
24.4.2.2	DISABLE_AIN	321
24.4.2.3	SET_AIN_GAIN	322
24.4.2.4	ENABLE_ADC	322
24.4.2.5	DISABLE_ADC	323
24.4.2.6	SET_ADC_RATE.....	323
24.4.3	ASRC配置	324
24.4.3.1	CONFIG_ASRC	324
24.4.3.2	CLOSE_ASRC	324
24.4.3.3	SET_ASRC_RATE.....	324
24.4.4	数字音效	325
24.4.4.1	SET_EFFECT_PARAM	325
24.4.4.2	GET_FEATURE_INFO.....	325

版本历史

日期	版本号	注释	作者
2015-06-11	0.1	建立初始版本	蔡李镇
2015-11-30	0.2	完成公共部分初版	蔡李镇
2015-12-7	0.3	增加蓝牙推歌功能部分，以及完善公共部分	蔡李镇
2015-12-17	0.4	完善公共部分，添加音频播放模型	蔡李镇
2015-12-22	1.0	添加本地播歌、蓝牙免提、USB 音箱、USB 读卡器等应用；完善 KEY 驱动、LED 驱动	蔡李镇、李江良、吴卓然、庄永康

1 引言

1.1 编写目的

本文档是 US282A 方案的软件开发指南。文档详细介绍了 US282A 方案的软件结构、各应用的具体业务流程、软件的开发方法等等内容。

本文档面向的读者是在炬芯的 ATS282X 系列 SOC 上，进行软件开发的工程人员。本文档适用于从事炬芯 ATS282X 系列 IC 的软件开发的初学者及资深工程师。

阅读本文档，将让读者能够更加了解 US282A 系统，并初步具备 US282A 方案的软件开发能力。对于资深工程师，通过阅读本文档，也将能够对系统的一些模块有更深入的认识。

1.2 术语和缩写词

术语	解释
US282A	ATS282X 系列 IC 的方案名称，方案包括硬件平台、软件平台、文档及各种工具。
SDK	软件开发套件，包括源代码、数据、文档和开发工具。
PSP	平台软件包，包括操作系统、LIBC 库、文件系统、存储设备驱动、音频设备驱动、解码编码算法库、中间件、TTS 驱动、USB 驱动组等，CASE 通过 API 方式调用 PSP 功能。
CASE	方案软件包，包括应用程序、COMMON 模块、方案相关的驱动程序、以及各种方案相关的数据。

1.3 参考文档

- [1] 《US282A 音效调试方法和流程.doc》
- [2] 《US282A 通话效果调试方法和流程.doc》
- [3] 《转接板使用说明.doc》

2 阅读指引

本文档与其他很多软件开发教程一样，会先介绍软件开发工具集，我们会按照软件开发和构建顺序依次介绍各种工具。

接着，我们简单介绍了一下 SDK 软件包的目录结构，帮助大家快速检索所要的代码和数据。

然后，我们详细讲解了 PSP 和 CASE 的基本组成和服务，让大家了解 PSP 和 CASE 如何为应用功能提供必须的运行环境。

然后，我们详细讲解了 COMMON 模块，它是软件系统和产品达成概念一致性，行为一致性的关键保证。COMMON 模块为应用封装并实现了软件平台的大部分基础服务，包括 APPLIB、用户交互服务、应用切换、时间闹钟服务、供电充电和电量检测、蓝牙管理器、低功耗模式、等。

接着，我们详细讲解了蓝牙架构，US282A 作为一个以蓝牙技术为核心的多媒体解决方案，需要一个稳定的、扩展性好、高性能的蓝牙架构，该章节会向大家说明我们是如何在稳定性、扩展性和高性能上取得平衡的。

接着，我们详细讲解了前台应用、引擎应用、驱动程序、独立模块 AL 等 CASE 层的软件对象的定义、工程组成、空间资源、运行时模型和原理、以及开发技术，简单介绍一下音频播放模型，再详细讲解了多线程开发等 CASE 开发技术要点。通过这些章节的介绍，大家将具备全面的 CASE 软件开发能力。

接着，我们详细讲解了各个应用功能的设计和实现，以便大家快速将二次开发的需求映射到具体的模块，然后在充分理解的前台下，安全高效的实现个性化功能。

文档附录部分是软件系统 PSP 部分的 API，可以给大家参考查阅。但是，考虑到文档更新可能不会很及时，而 PSP 的头文件的更新会更加及时到位，我们还是建议大家以头文件的 API 说明为准。

3 开发环境介绍

3.1 搭建开发环境

US282A 编译工具链为：GNU 开发工具集，而我们开发所用的操作系统是 Windows 系列操作系统，所以必须安装 cygwin 平台，关于 GNU 开发工具集与 cygwin 相关知识请在网上搜索查阅。

3.1.1 Cygwin 的安装

因为我们只需要使用 GNU 编译工具集，所以只需要安装 tidy 版本就行了。注：full 版本安装很慢，并且很容易安装出错，不建议安装。

如果你之前没有安装过 cygwin，请按照第 1 节全新安装。

如果你曾经安装过 cygwin，但是安装在系统盘，或者不确定是否仍然可用，请按照第 2 节方法完全卸载，再按照第 1 节全新安装。

3.1.1.1 全新安装 cygwin

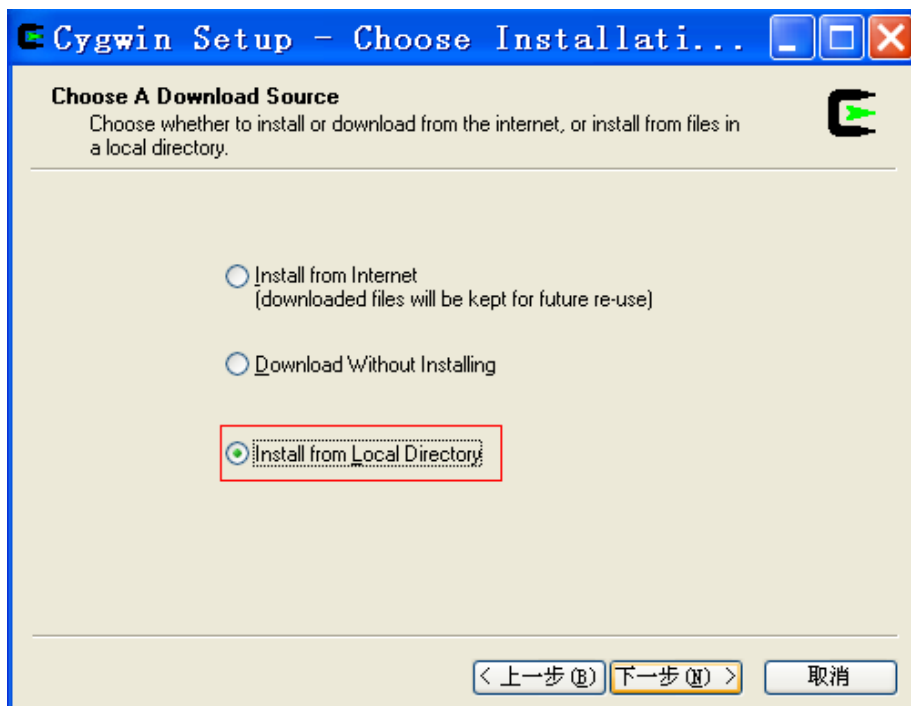
前提：

- 1) cygwin 不可以安装到系统盘符下，因为容易被其他软件尤其是查杀软件误操作。
- 2) 安装 tidy 版本的 cygwin，需要至少 600M 的空间，请安装前确认可用分区。下面的安装步骤是安装到 D:\，也可以是其它分区（除了系统盘）。
- 3) Cygwin 需要安装到根目录下，不可安装到类似 D:\Program Files\cygwin 的路径。
- 4) 如果你之前曾经安装过 cygwin，安装在系统盘，或者不确定是否仍然可用，请先卸载，卸载方法参照第 2 节。

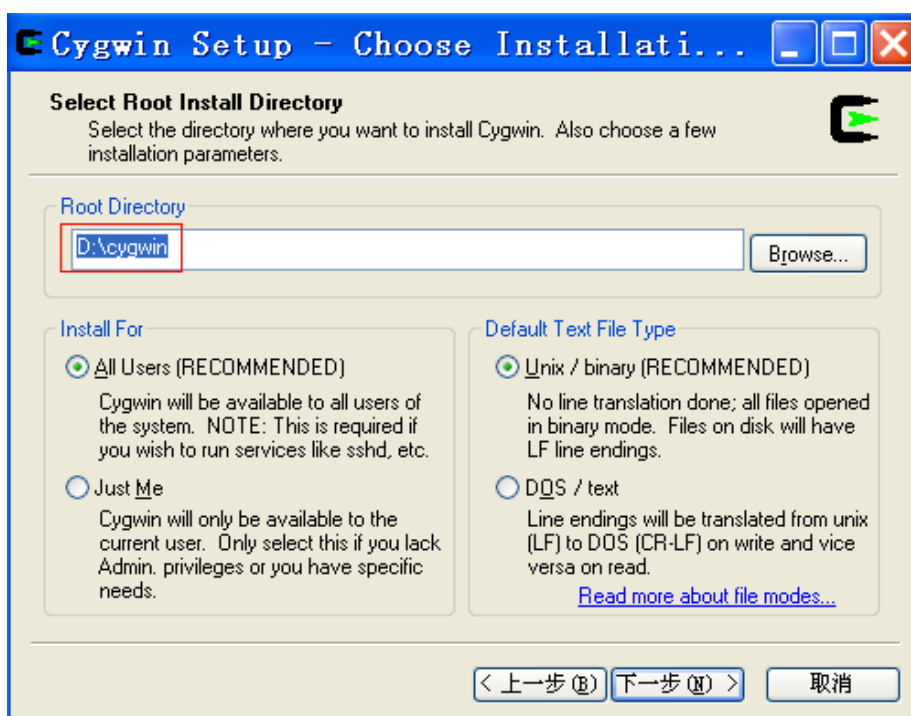
步骤：

- 1) 将 tidy 版本 cygwin 拷贝到本地目录 D:\MIPS_TOOLS\cygwin_tidy，此路径就是下面步骤 5) 的路径。
- 2) 双击 setup.exe，按下一步。

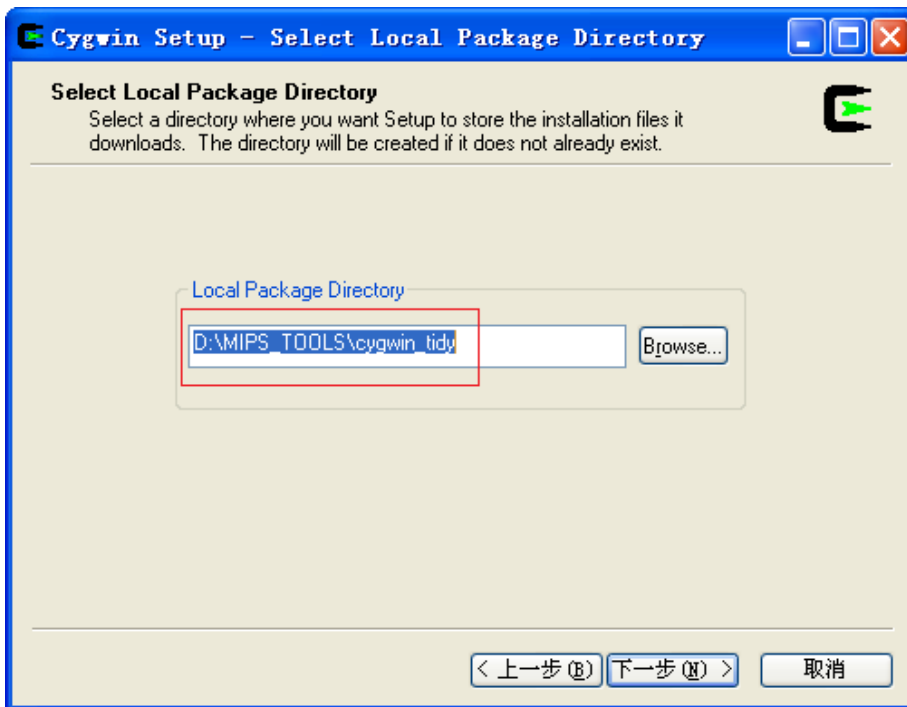
- 3) 选择 “Install from Local Directory”，按下一步。



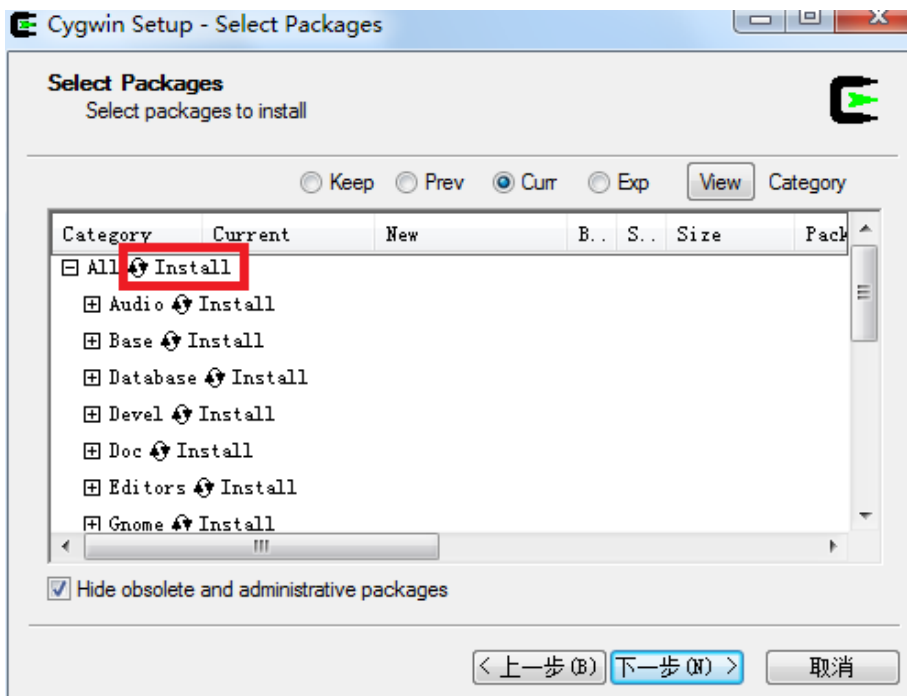
- 4) 选择非系统盘的根目录作为安装路径 D:\Cygwin，按下一步。



- 5) 选择源安装程序的路径，按下一步。



- 6) 点击 All 旁边的红色框位置，Default 会变为 Install（如果 PC 反应慢，可能等几秒钟才会变，再点会继续 Reinstall——>Uninstall——>Default——>**Install** 循环，我们安装时让其处于 Install 状态），按下一步继续就可以了。



- 7) 后面等待安装结束就可以了（需要几分钟才装完）。如果在安装过程中提示类似“内存错误”，很可能是你的 PC 内存不足，可以点击“取消”，然后“重新启动”PC（“注销”不能完全释放内存），

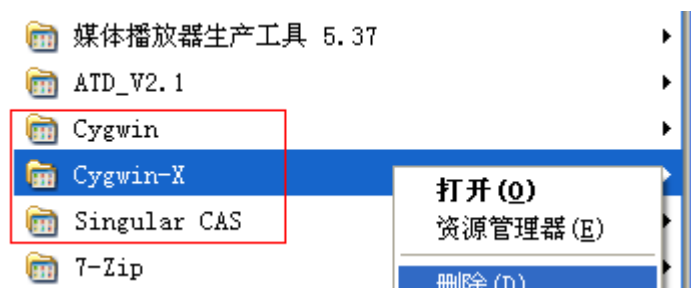
再重新安装。

- 8) 设置环境变量。在“环境变量”窗口下的“系统变量”子窗口中，在“path”值增加“;D:\Cygwin\bin”（路径间用分号，最后的结尾不需要；注意：系统环境变量值为“;D:\Cygwin”，可以在 cygwin 中 make，但是无法在 cmd 命令行中 make，而造成静态代码分析工具 QAC 等需要调用 make 工具的其他工具运行失败）。
- 9) 检查文件 D:\cygwin\cygwin.bat 中的路径是否为“chdir D:\cygwin\bin”，如果是则跳过，否则修改为此路径“chdir D:\cygwin\bin”。

3.1.1.2 卸载 cygwin

假如你之前将 cygwin 安装在 C:\Cygwin，按照下面步骤删除（如果某些步骤中所描述的项目已经不存在，可以略过）：

- 1) 按照第 1 节的 1) 至 4) 步操作，在第 1 节第 5) 步的界面，红色框处于 **Uninstall** 时，按下一步进行卸载，直到完成。
- 2) 删除 C:\Cygwin，以及其所有的子目录。
- 3) 删除系统环境变量 path 中 C:\Cygwin 开始的路径。
- 4) 删除开始——>所有程序中下图中的 3 项（直接右键删除）。



- 5) 删除注册表中的项目（开始——>运行，输入 regedit）
 "HKEY_CURRENT_USER_Software_Cygnus Solutions"
 "HKEY_LOCAL_MACHINE_Software_Cygnus Solutions"

3.1.2 SDE 工具链的安装

Cygwin 自带的 GUN 编译工具链是面向 PC 平台的，而我们需要的是 GNU 编译工具链是面向 MIPS 芯片平台，必须安装 MIPS 公司的 SDE 工具链。SDE 工具链的安装文件为：PN0016-06.61-2B-MIPSSW-MSDE-v6.06.01.tgz。

安装步骤如下（下面使用命令行方式操作，用户也可以直接图形化操作）：

- 1) 拷贝 PN0016-06.61-2B-MIPSSW-MSDE-v6.06.01.tgz 到目录 D:\cygwin\tmp\ 下。
- 2) `cd /usr/local` 进入 /usr/local，用户的应用程序一般放该目录下。
- 3) `mkdir sde60601` 创建目录 sde60601
- 4) `cd sde60601` 进入 sde60601
- 5) 解压 “`gzip -dc /tmp/PN0016-06.61-2B-MIPSSW-MSDE-v6.06.01.tgz | tar xf -`”。
- 6) `sh ./bin/sdesetup.sh` 执行 setup 脚本，设置一些环境变量。

Du you want to add a MIPSSim MDI library?

输入 n

Du you want to create an FS2 probe configuration?

输入 n

set environment globally in /etc/profile.d?

输入 y

- 7) 关掉 Cygwin 再重新打开，sde 工具链就可以使用了。

3.1.3 多媒体固件修改工具的安装

多媒体固件修改工具允许通过图形化工具对方案的某些配置直接进行修改，而避免修改代码再重新编译打包等一系列繁杂缓慢过程，这样的一种高效配置调整方案规格的方式。比如方案是否要有按键音，方案是否要有 FM 收音机应用等。

双击工具安装包 Media Firmware Modify Tool 的 setup.exe 安装，按照默认安装完成即可，无特殊配置。

3.1.4 多媒体产品量产工具的安装

双击工具安装包 Media product tool 的 setup.exe 安装，按照默认安装完成即可，无特殊配置。

3.1.5 量产卡固件烧写工具的安装

双击工具安装包 Card product tool 的 setup.exe 安装，按照默认安装完成即可，无特殊配置。

3.1.6 音效调试工具的安装

音效调试工具，即 ASET 调试工具，是用来图形化、实时调试音效的工具；对于有 USB 接口封装的 IC ATS2825，通过 USB 直接与待调样机连接通信；而对于没有 USB 接口封装的 IC ATS2823，则要借助 USB 转 UART 的转接板与待调样机连接。

双击工具安装包 Audio Sound Effect Tuning Tool 的 setup.exe 安装，按照默认安装完成即可，无特殊配置。

NOTE: 安装完成之后，需要跟我们获取到该工具使用权限方能使用。获取权限的方法请参考工具安装好后的目录下 ReadMe.doc 文档。

音效调试方法和流程请参见 [《US282A 音效调试方法和流程.doc》](#)。

3.1.7 通话效果调试工具的安装

通话效果调试工具，即 ASQT 调试工具，是用来图形化、实时调试通话效果的工具；对于有 USB 接口封装的 IC ATS2825，通过 USB 直接与待调样机连接通信；而对于没有 USB 接口封装的 IC ATS2823，则要借助 USB 转 UART 的转接板与待调样机连接。

双击工具安装包 Audio and Speech Quality Tuning Tool 的 setup.exe 安装，按照默认安装完成即可，无特殊配置。

NOTE: 安装完成之后，需要跟我们获取到该工具使用权限方能使用。获取权限的方法请参考工具安装好后的目录下 ReadMe.doc 文档。

通话效果调试方法和流程请参见 [《US282A 通话效果调试方法和流程.doc》](#)。

3.2 软件构建工具链的使用

在我们发布的 SDK 基础上进行二次开发，编写好代码之后，整个软件构建工具链的使用流程如下：

3.2.1 编译与链接

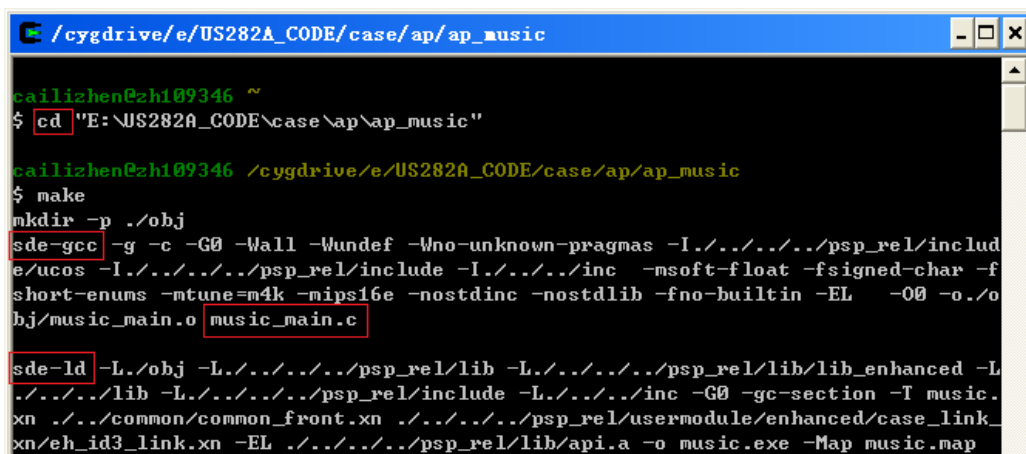
如果二次开发修改了应用程序或驱动程序，COMMON 代码，以及头文件，那么就需要 make 一下所依赖的工程，关于依赖关系，我们会在第 8 章 **SDK 软件包的目录结构** 中说明。

我们有以下几种 **make 方式**：

- 1) **Make clean & make**：清除所有 Obj 文件和结果文件，保证接下来 make 时能全部重新生成，避免 Obj 文件和结果文件更新不到位而出错。如果更新了一些公共头文件那么就需要 make clean & make。
- 2) **Make clean_target & make**：仅清除结果文件，这对于仅更新了当前工程直接依赖（在 makefile 文件中显示体现出依赖关系）的 Obj 文件之外的 Obj 文件时有用，可以快速 make 出结果文件。比如更新了 COMMON 目录下的 C 文件，那么 CASE\lib 目录下相对应的 Obj 文件就会更新，而各个 AP 的 Target 并没有直接依赖该 Obj 文件，故需要先将各个 AP 的 Target 清除，之后 make 时各个 AP 才会更新 Target。
- 3) **Make**：在仅更新了当前工程直接依赖的 Obj 文件时使用，可以快速 make 出结果文件。

说明：Obj 文件隐式依赖于相同名字的源码文件，只要该源码文件更新了，那么 make 时就会更新该 Obj 文件。

假如我们更新了 ap_music 下面的 music_main.c 文件，那么只需要 CD 到 ap_music 目录下，然后执行 make 命令即可更新生成 music.ap 这个最终的结果文件，该文件会自动拷贝到 case\fwpkg\ap 目录下。



```
/cygdrive/e/US282A_CODE/case/ap/ap_music
cailizhen@zh109346 ~
$ cd "E:\US282A_CODE\case\ap\ap_music"

cailizhen@zh109346 /cygdrive/e/US282A_CODE/case/ap/ap_music
$ make
mkdir -p ./obj
sde-gcc -g -c -G0 -Wall -Wundef -Wno-unknown-pragmas -I./../../../../psp_rel/include
e/ucos -I./../../../../psp_rel/include -I./../../../../inc -msoft-float -fsigned-char -f
short-enums -mtune=m4k -mips16e -nostdinc -nostdlib -fno-builtin -EL -O0 -o./o
bj/music_main.o music_main.c

sde-ld -L./obj -L./../../../../psp_rel/lib -L./../../../../psp_rel/lib/lib_enhanced -L
../../../../lib -L../../../../psp_rel/include -L../../../../inc -G0 -gc-section -T music.
xn ./../common/common_front.xn ./../../../../psp_rel/usermodule/enhanced/case_link_
xn/eh_id3_link.xn -EL ./../../../../psp_rel/lib/api.a -o music.exe -Map music.map
```

假如我们增加了一个源码文件 music_test.c，那么必须在链接脚本文件 music.xn 中增加该源码文件对应 Obj 文件的链接项，为其指定在 music.ap 文件中的存放位置，以及运行时加载在那片内存空间上。

```
249     . = ((. + AP_BANK_SPACE) & ~(AP_BANK_SPACE - 1)) + OFFSET;  
250     BANK_CONTROL_1_19 :  
251     {  
252         music_rcp_key_deal.o(.xdata .text .rodata)  
253         . = BANK_CONTROL_SIZE;  
254     }  
255  
256     . = ((. + AP_BANK_SPACE) & ~(AP_BANK_SPACE - 1)) + OFFSET;  
257     BANK_CONTROL_1_20 :  
258     {  
259         music_test.o(.xdata .text .rodata)  
260         . = BANK_CONTROL_SIZE;  
261     }
```

然后再执行 make 命令构建出 music.ap 结果文件。

3.2.2 打包固件

如果仅仅是更新某些结果文件，比如上面更新了 music.ap，则只需要执行 case\fwpkg 目录下的打包固件批处理 buildfw_XXX.bat 或者 buildfw_all.bat 即可。前者只会生成我们想要的单个固件，后者会生成所有固件，耗时较长。

注：同一个 SDK 软件包，根据配置项和打包项的区别，我们可以生成多个固件，每个固件对应于 case\fwpkg\config_txt 目录下的一个配置项文件 XXX.txt 和一个打包脚本 fwimage_XXX.cfg，以及一个由 Gen_buildfw.exe 小工具生成的批处理 buildfw_XXX.bat，其中 XXX 是固件名称。

如果打包项目需要更改，比如需要将不需要的功能的*.ap 文件删除掉，或者增删一些资源文件，那么需要先修改打包脚本 fwimage_XXX.cfg，然后再打包。

注意：

当我们更新了 case\fwpkg 目录下的打包批处理 buildfw_all.bat 时，我们需要执行 Gen_buildfw.exe 小工具更新所有单个固件打包批处理 buildfw_XXX.bat，然后才能使用 buildfw_XXX.bat 打包固件。

3.2.3 升级固件 (ATS2825)

生成固件后，通过下面方式将固件升级到小机：

- 1) 小机以 ADFU 模式（按住按键插入 USB 线开机）或者 U 盘模式连接 PC。
- 2) 打开量产工具，选择要升级的固件，选择相应的下载配置选项，点击下载。
- 3) 等待 10 ~ 20 秒钟下载完成，进度条显示“Successful 100%”表示下载正确，否则下载失败，可

尝试重新下载。



3.2.4 升级固件 (ATS2823)

生成固件后，通过下面方式将固件升级到小机：

版权所有 侵权必究

版本： 1.0

第 32 页

如果是第一次升级，就要使用卡启动的方式，这就需要通过卡物理接口来连接通信。

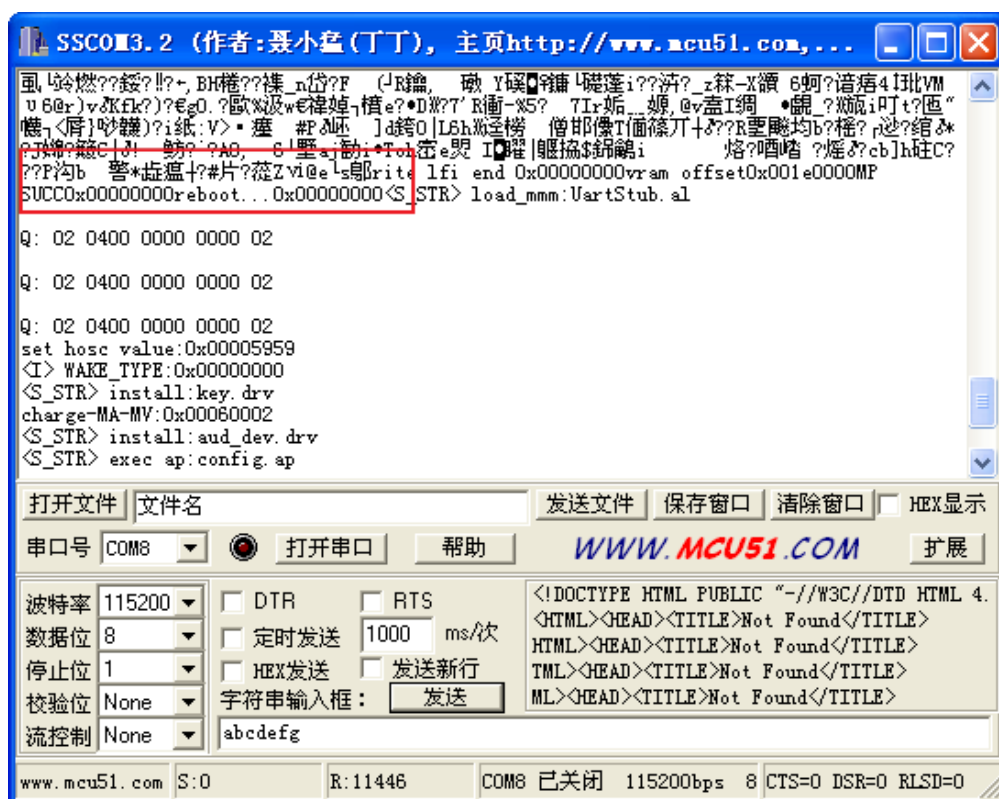
首先要先将固件 *.fw 烧写到 TF 卡中，步骤如下：

- 1) 插入 TF 卡读卡器到 PC 机。注意该卡最好是专用的，以免意外损伤格式化了。
- 2) 打开量产卡固件烧写工具，选择要升级的固件，点击开始烧录卡。
- 3) 等待 10 ~ 20 秒钟下载完成，提示“卡烧录成功！”。这样该 TF 卡就可以用来做卡启动并将固件量产到小机 Nor Flash 中。



接着使用 TF 卡中的固件卡启动，并将固件量产到小机 Nor Flash 中，步骤如下：

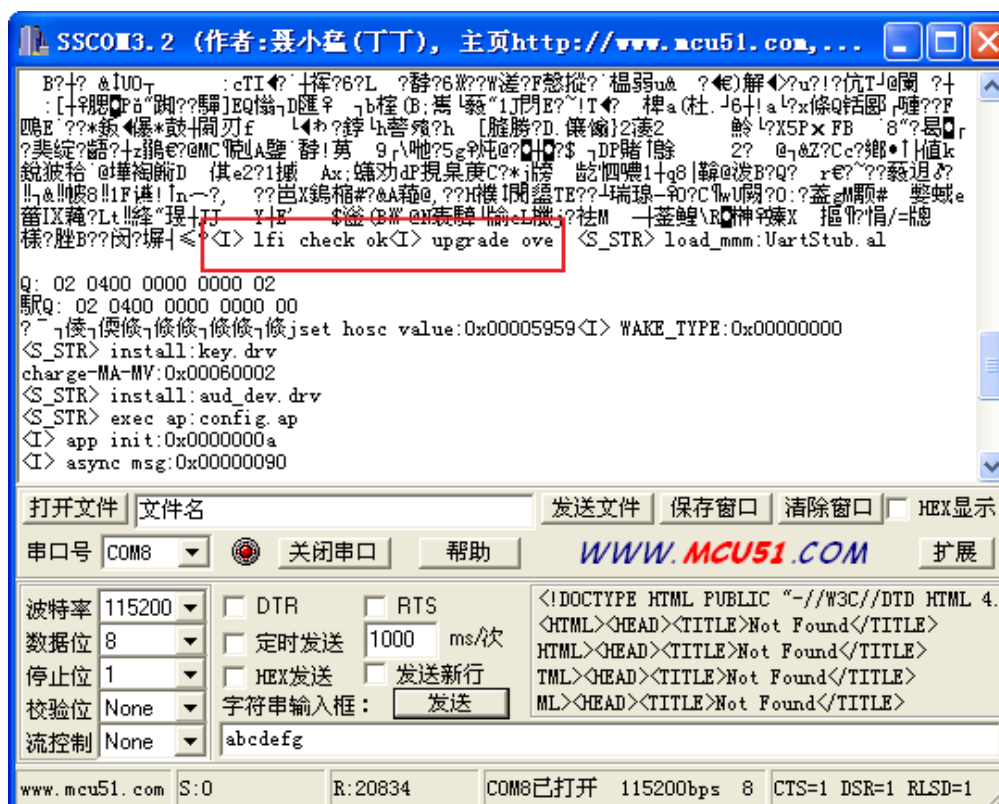
- 1) 确保小机已经与 TF 卡接通，如果小机 PCB 板没有卡外围电路，需要借助转接板，小机与转接板要建立 **1 线卡接口** 连接。
- 2) 确保待量产小机上电后 Nor Flash 不能启动，即空白 Nor Flash 或者短接 Nor Flash 的数据线等；然后小机就会卡启动，卡启动后就将固件拷贝到 Nor Flash 中，从而实现量产。



如果小机已经有支持卡升级的固件，可以借助转接板进行升级，步骤如下：

- 1) 在 U 盘模式下，拷贝 **UPGRADE.HEX** 固件到 TF 卡中。
- 2) 将 TF 卡插到转接板上，然后用 A 口 USB 线给转接板供电。
- 3) 然后将待升级样机与转接板连接通过 **UART 接口/1 线卡接口** 连接好，然后拨动转接板给待升级小机供电的开关到 ON 状态，这样就可以升级了。
- 4) 小机卡升级的几个关键点是：
 - a) 通过 UART 通信握手确认是在升级模式，即只要转接板能回复 **00** 就表示握手成功。
 - b) 然后小机会检测是否有卡插着。
 - c) 如果有卡插着，就会装载卡驱动和文件系统驱动，然后判断是否有 **UPGRADE.HEX** 文件。
 - d) 如果有就跳到 upgrade.ap 开始升级。





关于转接板的介绍，请参见 [《转接板使用说明.doc》](#)。

3.2.5 修改固件

要修改固件配置项的值，比如修改经典蓝牙的设备名称，有 2 种方式：

- 1) 直接修改 case\fwpkg\config_txt 目录下 配置项文件 XXX.txt 的配置项默认值，然后打包固件。

BTSTACK_DEVICE_NAME{240} = "us282a_btbody_demo";

- 2) 打包固件之后，使用固件 Modify 工具进行修改，但这要求该配置项已添加到 case\fwpkg\ 目录下的 config.spc 脚本中，且开放 Modify 权限，即

如果 config.spc 第一行为 debug=0;，那么要求 DISPLAY=1; 才会在 Modify 工具显示该配置项。

如果第一行为 debug=1;，那么表示开放所有配置项的 Modify 权限，则不做这个要求。

```
1171 key=BTSTACK_DEVICE_NAME;
1172 type=STR;
1173 operate=EDIT;
1174 len=29;
1175 RANGE=;
1176 tabname=btstack;
1177 DISP_EN=bluetooth device name:max 29 bytes;
1178 DISP_CH=蓝牙设备名称:最大29字符;
1179 DISPLAY=1;
```

如果配置项文件没有想要的配置项，那么必须自己添加；如果想要通过 Modify 工具修改默认值，那么还需要在 config.spc 中添加对应的配置项。

在配置项文件 XXX.txt 添加配置项的方法：

- 1) 依样画葫芦，选择任意一个 XXX.txt 中找一个相同类型的配置项，拷贝一份，然后修改名称，取值范围，默认值等。

配置项有 4 种类型：

A. 字符串类型：可编辑字符串，需要指定字符串最长字节数。

比如：**BTSTACK_DEVICE_NAME**{270} = "us282a_btbody_demo";

B. 线性数值型：下拉菜单，指定起始值，终值，和步进，以及默认值；通过 Modify 工具只可改默认值。

比如：**SETTING_ONOFF_LOWPOWER_TIMER** = 1[0~10,1];

C. 离散数值型：下拉菜单，列举整个取值空间，以及默认值；通过 Modify 工具只可改默认值。

比如：`SETTING_BAT_CHARGE_CURRENT = 6[0,1,2,3,4,5,6,7]`;

D. 多参数数值型：可编辑多参数整数，有效值为 16 位整型数，默认值可能是无意义的；通过 Modify 工具可修改默认值及所有参数值。

比如：`SETTING_INPUT_KEY_ARRAY = 0[3,2,7,8,5,6,29]`;

E. 直接整型数：可编辑整数，有效值为 32 位整型数。用于取值空间太大无法通过下拉菜单选择的配置项。

比如：`BTSTACK_CLASS_OF_DEVICE = 0x240404`;

- 2) 将该修改同步到所有 XXX.txt 文件中，默认值可以不一样。
- 3) 运行批处理 Gen_Config_ID.bat，生成该配置项文件对应的头文件 case\inc\config_id.h，源码使用生成的 ID 宏来访问该配置项。
- 4) 因为更新了 config_id.h 这种公共头文件，CASE 需要 make clean & make。

在 config.spc 中添加对应的配置项的方法：

依样画葫芦，在 config.spc 中找一个相同类型的配置项，拷贝一份，然后修改 tabname，RANGE，RANGE_DISP_XXX，DISP_XXX 等。

说明：

US282A 的 Config.spc 增加了两种特性：

- 1) 离散型参数别名：在 modify 工具上的显示，可以对参数值取一个别名，比如 0 别名为关闭，1 别名为开启；用别名显示在 modify 工具上，这样用户在 modify 工具上进行选择时会更直观，不易出错。
- 2) 多级菜单：各个配置项之间是有一定层级关系的，现在 Config.spc 中已能支持多级菜单关系的表示了。

另外，US282A 支持在 Welcome 等引导程序中读取配置项的默认值，这需要在 fwimage.cfg 文件中将配置项列出来，表示在量产或升级固件的时候将这些配置项的默认值读取出来，放到引导程序中的一个只读数据区中。在引导启动时，会将这些配置项默认值读取到内存中某个数组，就可以直接访问了。

配置项列表编辑格式如下：

```
//需要解析的CONFIG.BIN文件的ID号,最多2+3(3个数位的十进制数)*8个字符,最多8个ID,该配置项用于普通应用程序
//03表示有三个配置项,080是第一个配置项,对应config id为80,也就是固定ID为80的配置项
//110为第二个配置项,121为第三个配置项
//以下两个配置值允许增加配置项,但不允许删除默认的配置项,不可更改配置项的顺序,配置项ID必须固定!!!
INF_PARSE_CONFIG_INDEX_FOR_APP = "03080110121";

//需要解析的CONFIG.BIN文件的ID号,最多2+3(3个数位的十进制数)*8个字符,最多8个ID,该配置项用于量产程序
//01表示有一个配置项,122表示配置项ID为122,也就是控制量产结束是否重启的配置项
INF_PARSE_CONFIG_INDEX_FOR_PRD = "01122";
```

3.2.6 修改资源文件

在我们 SDK 中有以下几类资源文件：

- 1) TTS 资源文件，这是录制生成的一段一段的语音文件，原文件格式为 MP3 或者 WAV，打包为 tts_rc.lib 文件；一个方案可以支持多种语言的 TTS。
- 2) 当支持电话本人播报时，需要 TTS 语音元素包 package.dat 、索引表 info.pos 、字符转换表 FTBL_GB.\$\$\$ 等文件。
- 3) 按键音、默认来电铃声等 PCM 语音文件，采样率为 8KHz。
- 4) 内置闹铃文件，任何标案支持的歌曲文件格式都可以，当然因为这些文件是打包到固件的，而固件大小很有限，所以内置铃声文件应采用压缩率较高的文件格式，比如 MP3 或者 WMA。

3.2.6.1 TTS 资源文件

如果采用 MP3 格式，那么要求 TTS 资源文件是单声道的 MP3 文件，其采样率和帧长度必须一致，一般比特率也是一致的，并且帧长度必须小于或等于 512 字节。

如果采用 WAV 格式，那么同样要求 TTS 资源文件是单声道的 WAV 文件，并且比特率必须是固定的。

当要更换或者增删资源文件时，先准备好符合要求的资源文件，然后：

- 1) 拷贝到 case\tools\TTS_maker 目录下。
- 2) 如果是增删资源文件，那么需要修改 tts_flist.txt ，并同步更新 TTS 资源 ID 号头文件 case\inc\common_tts_id.h 。
- 3) 执行 tts_maker.exe 生成 tts_rc.lib 文件，并手动拷贝到 case\fwpkg 目录下。
- 4) 如果更新了 common_tts_id.h ，CASE 需要 make clean & make。

注意：如果方案的 TTS 文件数太多，打印“tts_flist.txt 文件太多！”，那么就需要修改 tts_flist.txt 文件的注释头，修改或者添加 INDEX = 1 这样的注释，并将“数字”改大一点。

```
//编写说明：
//1) 脚本文件 tts_flist.txt 必须是ANSI编码，每个词条包含2中语言语音文件，即[中文语音文件;英语语音文件]。
//2) NULL表示该词条没有此种语言语音文件，[END]表示结尾，这两者不能删除。
//3) 支持英文分号;和中文分号；为语音文件分隔符。支持英文空格，制表符和中文空格为空白符号。
//4) 索引表长度：INDEX = 1, 单位为KB，减掉 8B 的头，剩下的除以 8B，结果就是支持的最多文件项，包括NULL项。
蓝牙播歌.mp3;          NULL
```


3.2.6.2 PCM 语音文件

PCM 语音文件是无格式的语音文件，但是为了能够准确播放 PCM 语音，我们将文件的实际长度覆盖填写到文件的头 4 个字节，在播放时，会将这 4 个字节清 0。

我们现在的按键音是 PCM 语音文件，是可以任意定制的，实际上，现在除了可以在任何时候播放按键音外，还可以播放一小段任意语音，比如警告声，蓝牙连接提示音等。

所有的 PCM 文件都必须放在 `case\fwpkg\pcm_file` 目录下。如果我们更新了 PCM 文件，那么必须执行 `pcm_maker.exe` 工具，将头 4 个字节覆盖填写为文件实际长度。如果是增删 PCM 文件，还要同步更新 `pcm_flist.txt` 文件，以及打包脚本 `fwimage_XXX.cfg` 的打包项。

3.2.6.3 内置闹铃文件

如果增删了内置闹铃文件，则需要先同步更新 `case\inc\common_time_alarm.h` 的宏定义 `BUILDIN_RING_MAX` 的值，然后再修改 `case\ap\common\common_func` 目录下源文件 `common_share_data.c` 的 `g_sd_ring_file` 数组定义，还要同步更新打包脚本 `fwimage_XXX.cfg` 的打包项；如果仅仅是更新了内置闹铃文件的声音，那么只需要重新打包固件即可。

3.3 软件调试手段

3.3.1 串口打印

在方案开发时，如果 UART TX **A23** 可用并已引线出来，那么可以使用串口打印来实时调试。

要开启串口打印调试就需要使能 `SETTING_UART_PRINT_ENABLE` 配置项，要关闭也只需要将该配置项关掉即可。

US282A 支持打印接口很丰富，具体参考 `psp_rel\include\debug_interface.h` 文件中的宏函数定义，使用时要注意以下几点：

- 1) 打印接口是分模块的，CASE 应用程序、驱动程序、系统、蓝牙协议栈、蓝牙控制器驱动、等都是使用不同的打印接口，打印不同的前缀字符串，以便识别打印信息的来源。

- 2) 打印是分级别的，即根据打印信息的重要程度分为错误，警告，信息，调试这 4 个等级，不同级别的前缀字符串是不一样的，以便识别打印信息的级别。这 4 个等级的打印接口是分别独立可开关的，SDK 发布包将调试类打印关掉。
- 3) 以 HEX 方式打印字节流的接口只有一个：PRINT_DATA(buf,len)，没有区分等级。
- 4) 我们支持在中断服务例程中打印，提供了一个独立的接口 DEBUG_PRINT_IRQ(str,data,mode)，打印时先将要打印的内容缓存在一个缓冲区中，然后由前台或引擎应用来代理打印出来。
- 5) 打印接口有 3 种打印模式，以打印错误为例：
 - PRINT_ERR(str)：带前缀打印字符串，str 必须是字符串常量。
 - PRINT_ERR_STR(str)：不带前缀打印字符串，str 可以是字符串常量，也可以是字符串指针。
 - PRINT_ERR_INT(str,data)：带前缀打印字符串 + 整型值，str 必须是字符串常量。
- 6) 对于一些打印量非常大的打印需求，比如将蓝牙控制器 UART RX/TX 的数据包全部打印出来，那么我们必须使用 DMA 打印才能够满足打印速度的要求。

```
void libc_dma_print(uint8 *str_buf, dma_print_info_t *data_info, uint8 mode);
```

str_buf : 打印字符串内容 data_info : 打印配置 mode : 打印模式

- 1) DMA_PRINT_VAL : 打印字符串+数值，数值存放在 data_info.data.value 中，此时 len 无意义。
- 2) DMA_PRINT_BUF1 : 打印 HEX，字节流数据存放在 data_info.data.buf 中，每个字节后用空格分隔，每 16 个字节一行。
- 3) DMA_PRINT_BUF2 : 打印 HEX，字节流数据存放在 data_info.data.buf 中，连续打印不使用空格分隔，每 16 个字节一行。

这里以蓝牙控制器 UART RX/TX 打印为例给出示例代码：

```
void print_rxtx(uint8 *buf, uint32 len, uint8 *prefix)
{
    uint32 irq_save = sys_local_irq_save();
    dma_print_info_t data_info;
    data_info.dma_channel = 2; //dma2
    data_info.len = len;      data_info.data.buf = buf;
    libc_dma_print(prefix, &data_info, DMA_PRINT_BUF1);
    sys_local_irq_restore(irq_save);
}
```

4 SDK 软件包介绍

US282A 的 SDK 软件包，除了 3.1 节 构建开发环境 中介绍的工具安装包外，包括了构建完整固件所需的所有源代码、配置文件、库文件和目标文件、数据、工具，其中源代码我们只开放了二次开发所需的 CASE 相关的源代码和 enhanced 模块的源代码，其他都是以库文件和目标文件的形式发布。

名称 ▲	大小	类型
case		文件夹
psp_rel		文件夹

4.1 Psp_rel 目录介绍

Psp_rel 目录是 PSP 发布包的意思，PSP 涉及大量与 CASE 无关的 IC 平台细节，所以我们以库文件和目标文件的方式发布，包含 AFL.bin、API 库文件、进程运行时库文件 ctor.o、API 头文件、PSP 配置文件、多个构建工具等。另外，Psp_rel 目录下还包含 enhanced 的源代码及其库文件。

名称 ▲	大小	类型
bin		文件夹
cfg		文件夹
include		文件夹
lib		文件夹
tools		文件夹
usermodule		文件夹

4.1.1 Bin 目录

名称 ▲	大小	类型
AFL.bin	3,983 KB	FTE Binary Expo...

Bin 目录存放 PSP 所有目标文件和资源文件打包出来的 AFL.bin 文件，我们可能会根据不同产品形态的功能规格要求，打包发布多个 AFL.bin。如果发布了多个 AFL.bin，那么会在当前目录附加文档说明各个版本的 AFL.bin 的功能规格。

AFL.bin 作为固件打包最重要的文件之一，必须谨慎选择。

4.1.2 Cfg 目录

名称	大小	类型
rules.mk	3 KB	Makefile

Cfg 目录存放 PSP 各模块 make 基本规则文件，包括 enhanced 在内。

4.1.3 Include 目录

名称	大小	类型
task_info.xn	2 KB	XN 文件
link_base.xn	11 KB	XN 文件
vm_fwsp_def.h	3 KB	C/C++ Header
vfs_interface.h	23 KB	C/C++ Header
vfs.h	5 KB	C/C++ Header
uhost.h	6 KB	C/C++ Header

Include 目录存放 PSP 对外开放 API 头文件及其他基础公共头文件，还有 2 个空间分配和模块 ID 号分配的公共 XN 文件。

注意： Include 目录下的头文件和 xn 文件更新后，必须 **make clean & make** 一下 enhanced 模块和整个 CASE，以确保 CASE 跟 PSP 是匹配的。

注意： 方案发布之后，对该目录下的头文件和 xn 文件进行修改必须十分谨慎，务必确保不会影响 PSP 各模块的工程编译链接结果，否则会出错。所以如果需要调整这些文件，最好在 IC 原厂 FAE 协助下进行。

4.1.4 Lib 目录

名称	大小	类型
lib_enhanced		文件夹
api.a	36 KB	A 文件
audio_device_op_entry.o	2 KB	O 文件
base_op_entry.o	2 KB	O 文件
bt_op_entry.o	2 KB	O 文件

Lib 目录存放着 PSP 各模块的 API 接口，API 接口以 *.o 方式给出，并且打包为 api.a 库文件。Enhanced 模块编译出来的目标文件 *.o 存放在子目录 lib_enhanced 下。

4.1.5 Tools 目录

名称 ▲	大小	类型
maker_py		文件夹
ap_builder.exe	188 KB	应用程序
bank_addr_info.bmp	476 KB	BMP 文件
bank_addr_info.exe	104 KB	应用程序
bin2hex.exe	694 KB	应用程序
drv_builder.exe	184 KB	应用程序
dsp_lib_analyse.exe	36 KB	应用程序
dsp_lib_builder.exe	172 KB	应用程序
mmm_codec_builder.exe	180 KB	应用程序

Tools 目录包含各种自主开发的小工具，有固件打包工具、AP 打包工具、驱动打包工具、中间件打包工具、解码库打包工具等。

4.1.6 Usermodule 目录

即 enhanced 源码目录，该模块会被 MUSIC 本地播歌应用，所以当 enhanced 目录下的源代码文件更新后，或者 case_link_xn 目录下的链接脚本 XN 更新后，需要 make MUSIC 本地播歌应用的前台应用和引擎应用：

名称 ▲	大小	类型
case_link_xn		文件夹
fsel_simple		文件夹
id3		文件夹
lrc		文件夹
make_build		文件夹

其中：

对象	说明
case_link_xn	enhanced 是直接链接到 AP 的，需要提供单独的 xn 脚本。
fsel_simple	文件选择器源代码。
Id3	ID3 解析器源代码。
Lrc	歌词解析器源代码。
make_build	编译生成 enhanced 的目标文件的 makefile 脚本。

4.2 Case 目录介绍

Case 目录是方案相关的所有源代码、配置文件、库文件和目标文件、数据、工具，多个 Case 可以对应同一个 psp_rel 发布包。

名称	大小	类型
ap		文件夹
ap_test		文件夹
cfg		文件夹
drv		文件夹
fwpkg		文件夹
inc		文件夹
lib		文件夹
tools		文件夹

4.2.1 Ap 目录

名称	大小	类型
ap_alarm		文件夹
ap_btcall		文件夹
ap_btplay		文件夹
ap_cardreader		文件夹
ap_config		文件夹
ap_linein		文件夹
ap_manager		文件夹
ap_music		文件夹
ap_radio		文件夹
ap_upgrade		文件夹
ap_usound		文件夹
btcall_engine		文件夹
btplay_engine		文件夹
common		文件夹
fm_engine		文件夹
linein_engine		文件夹
music_engine		文件夹
music_scan		文件夹
usound_engine		文件夹

Ap 目录是 CASE 面向具体功能业务的应用程序的集合，可以分为 2 类：

- 1) 公共应用程序和模块：CASE 必需应用模块，是 CASE 的基础组成部分。
- 2) 具体功能应用程序：根据 CASE 的规格，可选择支持其中任意多个应用程序，是具体功能的实现主体。

类型	对象	说明
公共应用程序和模块	Ap_manager	CASE 最早加载的应用程序，它是一个特殊 AP，负责其他应用程序的加载和卸载；该 AP 初始化时会装载一些基本模块，比如按键驱动，NOR UD 驱动，AUDIO 驱动
	Ap_config	CASE 开关机应用程序，它是一个前台应用（什么是前台应用，请见 11 章说明），负责处理开关机选项和流程
	COMMON	COMMON 是 CASE 的其他所有前台应用，引擎应用，以及 BT STACK 应用的基础组成部分，为其提供一个概念和行为一致的应用环境。在下一节将继续展示 COMMON 目录结构。
具体功能应用程序	Ap_alarm	闹钟响铃的前台应用，实现了闹钟响铃功能的用户交互，还实现了闹铃目录扫描功能。
	Ap_btcall	蓝牙免提的前台应用，实现了蓝牙免提功能的用户交互。
	Btcall_engine	蓝牙免提的引擎应用，实现了蓝牙免提的播放器后台，负责接收前台应用的命令控制，调用解码中间件和解码库来打电话。
	Ap_btplay	蓝牙推歌的前台应用，实现了蓝牙推歌功能的用户交互。
	Btplay_engine	蓝牙推歌的引擎应用，实现了蓝牙推歌的播放器后台，负责接收前台应用的命令控制，调用解码中间件和解码库来推送音乐。
	Ap_music	本地播放的前台应用，实现了插卡或插 U 盘的本地播歌功能的用户交互。
	Music_scan	插卡或插 U 盘的本地磁盘歌曲扫描，建立播放列表，以实现切歌和点歌功能。
	Music_engine	本地播歌的引擎应用，也是闹钟响铃的引擎应用，实现了本地播歌和闹钟响铃的播放器后台，负责接收前台应用的命令控制，调用解码中间件和解码库来播放音乐。
	Ap_linein	音频输入功能的前台应用，实现了音频输入播放功能的用户交互。
	Linein_engine	音频输入功能的引擎应用，实现了音频输入功能的播放器后台，负责接收前台应用的命令控制，调用解码中间件和解码库来播放音频。
	Ap_radio	FM 收音机的前台应用，实现了 FM 收音机的用户交互。
	Fm_engine	FM 收音机的引擎应用，实现了 FM 收音机的播放器后台，负责接收前台应用的命令控制，调用 FM 驱动实现搜台和收听电台，调用解码中间件和解码库来播放 FM 电台。
	Ap_usound	USB 音箱的前台应用，实现了 USB 音箱的用户交互。

	Usound_engine	USB 音箱的引擎应用，实现了 USB 音箱的播放器后台，负责接收前台应用的命令控制，调用解码中间件和解码库来播放 PC 音乐。
	Ap_cardreader	读卡器应用，实现了读卡器功能的用户交互。
	Ap_upgrade	自动升级固件应用，支持卡升级、U 盘升级等。

4.2.2 Common 目录

名称	大小	类型
applib		文件夹
aset_common		文件夹
bt_common		文件夹
common_display		文件夹
common_event		文件夹
common_func		文件夹
common_rcp		文件夹
common_sound		文件夹
common_view		文件夹
data		文件夹
time_alarm		文件夹
common_engine.xn	3 KB	XN 文件
common_front.xn	12 KB	XN 文件
common共享数据段链接修...	1 KB	文本文档
Makefile	2 KB	文件

COMMON 是 CASE 的其他所有前台应用，引擎应用，以及 BT STACK 应用的基础组成部分，为其提供一个概念和行为一致的应用环境。

注意：如果仅仅修改了 COMMON 目录下的源文件，那么需要且仅需要 **make clean_target & make** 一下整个 CASE，以确保 CASE 下所有程序工程都是同步的。

对象	说明
Applib	实现应用程序管理、消息机制、软定时器、逻辑按键处理、配置项管理等功能
Aset_common	在线音效调试模块
Bt_common	蓝牙管理器，负责与 BT STACK 交互
Common_display	显示模块，实现分立 LED 灯显示功能
Common_event	事件处理模块，包括按键事件和系统事件
Common_func	数字音效调节、音量调节等交互接口，应用切换接口，系统监控

	(省电关机、电量和充电监控、设备热拔插)接口,低功耗模式、应用等待接口、共享数据管理接口、获取初始环境变量
Common_rcp	RCP 协议实现
Common_sound	按键音播放、TTS 播放、声音调节等实现
Common_view	前台应用的视图运行机制,支持嵌套视图显示
data	COMMON 全局变量,分为所有应用程序共享全局变量、所有前台应用共享全局变量、前台应用内全局变量
Time_alarm	时间闹钟服务模块,提供时间显示,日历显示和设置,闹钟设置功能
Common_engine.xn	引擎应用 COMMON 部分链接脚本
Common_front.xn	前台应用 COMMON 部分链接脚本
makefile	COMMON 所有模块的 make 总入口

4.2.3 Cfg 目录

名称	大小	类型
strip_lib		文件夹
common_path	1 KB	文件
common_sub_path	1 KB	文件
Makefile	3 KB	文件
rules.mk	3 KB	Makefile

Cfg 目录是整个 CASE 的 make 配置文件,可以通过这里的 makefile 来 make 整个 CASE。另外,strip_lib 里面的 makefile 用于将 case\lib 和 psp_rel\lib\lib_enhanced 目录下的 OBJ 文件的 debug 段信息去掉,以保证这些 OBJ 文件不会因为各个工程师的工作目录不一致而不同。

这里的 makefile 提供了一下几种 make 方式,参见 6.2.1 节 编译与链接。

4.2.4 Drv 目录

名称	大小	类型
ccd		文件夹
fm_common		文件夹
fm_qn8035		文件夹
Key_boombox		文件夹
key_common		文件夹
LED_7SEG		文件夹
rom_TWI_lib		文件夹
welcome		文件夹

Drv 目录存放着 CASE 相关的多个驱动程序：

对象	说明
Key_common	按键驱动中一些与按键无关的功能，包括调频、设备热拔插检测、电量和充电监控等
Key_boombox	按键驱动的主体功能，实现按键扫描功能
Rom_TWI_lib	TWI 驱动，已固化在 IC ROM 中
Ccd	TI TAS5717 I2S PA 驱动程序，基于 TWI 通信接口；只有当音频输出方式使用 I2S 才需要该驱动程序
LED_7SEG	7 段段码屏控制器驱动程序；只有当使用 LED 段码屏作为显示设备时才需要该驱动程序
Welcome	Welcome 是 CASE 相关的最早运行的程序模块，它几乎是在系统一上电就马上运行，提供低电检测、进入升级模式、上电提示等功能
Fm_common	FM 驱动的逻辑接口层，以及 FM 模组硬件环境初始化，基于 TWI 通信接口
Fm_qn8035	QN8035 FM 模组的硬件驱动程序，实现上述逻辑接口层，基于 TWI 通信接口

4.2.5 Fwpkg 目录

名称	大小	类型
ap		文件夹
config_txt		文件夹
drv		文件夹
PATCH		文件夹
pcm_file		文件夹
phonebook		文件夹
alarm1.mp3	7 KB	cloudmusic.mp3
alarm1.wma	13 KB	cloudmusic.wma
buildfw_all.bat	1 KB	MS-DOS 批处理文件
config.spc	49 KB	PKCS #7 证书
E_ATS2823.fw	3,703 KB	FW 文件
E_ATS2823B.fw	3,703 KB	FW 文件
E_ATS2825.fw	3,703 KB	FW 文件
E_TESTCHIP.fw	3,703 KB	FW 文件
Gen_buildfw.exe	8 KB	应用程序
tts_rc.lib	73 KB	Object File Lib...
US282A_UDA.fw	9,288 KB	FW 文件

Fwpkg 目录是 CASE 固件打包工作目录，它集合了打包成为固件所需的 CASE 层软件对象、数据文件、配置文件、以及打包批处理等。

对象	说明
Ap	Case\ap 目录下的程序工程构建出来的软件对象 *.ap 和 *.al
Config_txt 目录	多个 CASE 的固件配置项文件和固件打包脚本等
Drv	Case\drv 目录下的程序工程构建出来的软件对象 *.drv 和 welcome.bin
PATCH	蓝牙控制器的补丁文件
Pcm_file	CASE 相关的 PCM 音频文件，以及修改文件头的批处理及其脚本
Phonebook	蓝牙免提电话本播报资源文件
ACT_GEN_BTATTR.CFG	自动化测试脚本
Alarm1.mp3, alarm1.wma	内置铃声资源文件
Buildfw_all.bat	固件打包批处理，将 config_txt 下所有配置都对应生成一个固件
Config.spc	配置项 MODIFY 脚本
E_ATS2823.fw,	固件加密相关的“固件”，分别对应不同 CHIP ID 的 IC

E_ATS2823B.fw, E_ATS2825.fw, E_TESTCHIP.fw	
Gen_buildfw.exe	生成对应于每个配置的固件打包批处理，这样可以指定打包任意一个固件
Tts_rc.lib	打包了的 TTS 资源文件
US282A_UDA.fw	转接板固件，用于 ATS2823X IC 辅助升级和调试

4.2.6 Config_txt 目录

名称	大小	类型
fwimage_US282A_BTBOX_DVB_ATS2823.cfg	5 KB	Microsoft Office...
fwimage_US282A_BTBOX_DVB_ATS2825.cfg	5 KB	Microsoft Office...
Gen_Config_ID.bat	1 KB	MS-DOS 批处理文件
Template.ini	18 KB	配置设置
US282A_BTBOX.aset	3 KB	ASET 文件
US282A_BTBOX_DVB_ATS2823.txt	25 KB	文本文档
US282A_BTBOX_DVB_ATS2825.txt	25 KB	文本文档

Config_txt 目录是固件打包的配置项文件和打包脚本文件，它支持多 CASE 开发，每个 CASE 配套使用一个 *.txt 和 fwimage_*.cfg，可以用来实现不同 CASE 使用不同的配置项和打包需求。

对象	说明
*.txt config.txt	多个 CASE 的固件配置项文件，在打包时先拷贝到 fwpkg 目录下并改名为 config.txt，再生成 config.bin
fwimage_*.cfg	多个 CASE 的固件打包脚本，在打包时先拷贝到 fwpkg 目录下并改名为 fwimage.cfg，再打包
Gen_Config_ID.bat	根据找到的第一个固件配置文件生成配置 ID 头文件的批处理
US282A_BTBOX.aset	ASET 工具音效调试参数脚本，记录上一次调试时的参数
Template.ini	ASET 工具导出音效配置项文件的模板

config.txt 文件解析：

- ```
1 //-*- coding: UTF8 -*- 用来指示以什么编码打包字符串内容，从[ANSI, UTF8,
```
- 指定 config.bin 中的字符串内容使用什么编码，默认使用 UTF8，这样可以支持中文等文字。

```
2 //系统配置--应用选项
3 SETTING_APP_SUPPORT_CARDPLAY = 1[0,1]; //是否支持卡播放功能, 0表示不支持, 1表示支持
4 SETTING_APP_SUPPORT_UHOSTPLAY = 1[0,1]; //是否支持U盘播放功能, 0表示不支持, 1表示支持
5 SETTING_APP_SUPPORT_LINEIN = 1[0,1]; //是否支持linein播放功能, 0表示不支持, 1表示支持
```

```
92 //系统设置--数字音效选项
93 DAE_BYPASS_ENABLE{130} = 0[0,1]; //是否禁用DAE数字音效
```

- 一般是分模块写配置项的，除了第一个模块系统配置项之外，后面的每个模块，都会使用花括号{指定 ID}的方式指定该模块的起始配置项 ID，这样前一个模块添加配置项后就不会影响到当前的模块的配置项 ID，这样就不需要重新编译引用该模块配置项的工程。

```
3 SETTING_APP_SUPPORT_CARDPLAY = 1[0,1]; //是否支持卡播放功能, 0表示不支持, 1表示支持
4 SETTING_APP_SUPPORT_UHOSTPLAY = 1[0,1]; //是否支持U盘播放功能, 0表示不支持, 1表示支持
```

- 配置项用法 1：开关配置项，只有开和关两种状态可选。

```
12 SETTING_APP_SWITCH_SEQUENCE = 0[0,1,2,3,4,5]; //应用切换顺序
```

- 配置项用法 2：离线数值型配置项，每个值可改，并且默认值也可以修改。

```
59 SETTING_BAT_CHARGE_CURRENT = 6[0,1,2,3,4,5,6,7]; //电池充电电流值
```

- 配置项用法 3：离散数值型配置项，只有默认值可以修改。

```
17 SETTING_ONOFF_LOWPOWER_TIMER = 1[0~10,1]; //进入低功耗时间
```

- 配置项用法 4：线性数值型配置项，只有默认值可改。

```
236 BTSTACK_DEVICE_NAME{270} = "us282a_btbox"; //经典蓝牙设备名称, 最大55字节
```

- 配置项用法 5：可编辑字符串参数。

```
240 BTSTACK_CLASS_OF_DEVICE = 0x240404; //高字节表示major service
```

- 配置项用法 6：可编辑 32 位整型数。

fwimage.cfg 文件解析：

```
47 //需要解析的CONFIG.BIN文件的ID号,最多2+3(3个数位的十进制数)*8个字符, 最多8个ID,该配置项用于普通应用程序
48 //03表示有三个配置项, 080是第一个配置项, 对应config id为80, 也就是固定ID为80的配置项
49 //110为第二个配置项, 121为第三个配置项
50 //以下两个配置值允许增加配置项, 但不允许删除默认的配置项, 不可更改配置项的顺序, 配置项ID必须固定!!!
51 INF_PARSE_CONFIG_INDEX_FOR_APP = "03080110121";
52
53 //需要解析的CONFIG.BIN文件的ID号,最多2+3(3个数位的十进制数)*8个字符, 最多8个ID,该配置项用于量产程序
54 //01表示有一个配置项, 122表示配置项ID为122, 也就是控制量产结束是否重启的配置项
55 INF_PARSE_CONFIG_INDEX_FOR_PRD = "01122";
```

- US282A 允许 Welcome 和引导程序等使用配置项的默认值，这只需要在打包配置文件中对以上的 2 个变量按照要求的格式填写即可。在 Welcome 中访问这里配置的配置项默认值的方法请直接参见 Welcome 模块中的源码。

```
78 //Specify the flash size that was used, sector unit
79 SPI_STG_CAP=0x1000;
```

2. 指定用于存放固件的 SPI Nor Flash 的容量大小，单位为 512 字节。

```
87 SETPATH=".\";
88 FW_SPEC="config.spc";
89 FW_SPEC="config.txt";
90 //FWIM="alarm1.wma";
91 FWIM="alarm1.mp3";
92 FWIM="tts_rc.lib";
93 FWIM="spk_comp.dat";
94 EFW="E_ATS2825.fw";
```

3. 这基本上涵盖了固件打包配置文件的主要语法点：

- 1) SETPATH: 切换当前目录
- 2) FW\_SPEC: 固件会打包进来，但是最终量产时不会写到 SPI Nor Flash 中，以节省固件空间
- 3) // : 用来临时屏蔽掉不需要打包进来的文件，以节省固件空间
- 4) FWIM : 固件打包进来，且最终量产时也会写到 SPI Nor Flash
- 5) EFW : 一个特殊的文件，用来做 EFUSE 加解密

## 4.2.7 Inc 目录

| 名称                          | 大小    | 类型           |
|-----------------------------|-------|--------------|
| h app_msg.h                 | 18 KB | C/C++ Header |
| h app_timer.h               | 5 KB  | C/C++ Header |
| h app_view.h                | 5 KB  | C/C++ Header |
| h applib.h                  | 16 KB | C/C++ Header |
| h bt_controller_interface.h | 5 KB  | C/C++ Header |
| h btcall_common.h           | 1 KB  | C/C++ Header |
| h btplay_common.h           | 1 KB  | C/C++ Header |
| h btstack_common.h          | 8 KB  | C/C++ Header |
| h case_include.h            | 2 KB  | C/C++ Header |
| h case_independent.h        | 7 KB  | C/C++ Header |

Inc 目录包含 CASE 相关的所有公共头文件，有些是 COMMON 需要的公共头文件，有些是 CASE 相关的驱动程序开放给应用程序的 API 头文件，有些是 CASE 里多个程序工程共享的公共头文件，比如 btcall\_common.h 和 btplay\_common.h 分别是 BTCALL 和 BTPLAY 功能的前台应用和引擎应用的公共头文件。

**注意：**Inc 目录下的头文件更新后，必须 **make clean & make** 一下整个 CASE，以确保 CASE 下所有程序工程都是同步的。

## 4.2.8 Lib 目录

| 名称                         | 大小    | 类型   |
|----------------------------|-------|------|
| app_engine_config.o        | 6 KB  | 0 文件 |
| app_manager_bank.o         | 7 KB  | 0 文件 |
| app_sleep.o                | 9 KB  | 0 文件 |
| app_timer_bank.o           | 11 KB | 0 文件 |
| app_timer_bank_for_engi... | 9 KB  | 0 文件 |
| app_timer_bank_single.o    | 7 KB  | 0 文件 |
| app_timer_rcode.o          | 6 KB  | 0 文件 |

Lib 目录是 COMMON 模块编译出来的 OBJ 目标文件集合，之所以将 COMMON 模块编译出来的 OBJ 目标文件单独存放到一个目录，是为了只编译一次 COMMON，然后其他所有引用 COMMON 的应用程序都可以直接链接，从而加快整个 CASE 的构建速度。

**注意：**如果仅仅修改了 COMMON 目录下的源文件，那么 Lib 目录下的 OBJ 文件就会更新，那么需要且仅需要 **make clean\_target & make** 一下整个 CASE，以确保 CASE 下所有程序工程都是同步的。

## 4.2.9 Tools 目录

| 名称         | 大小 | 类型  |
|------------|----|-----|
| Gen_config |    | 文件夹 |
| PCM_maker  |    | 文件夹 |
| TTS_maker  |    | 文件夹 |

TTS\_maker 展开如下：

| 名称            | 大小    | 类型                |
|---------------|-------|-------------------|
| tts_flist.txt | 2 KB  | 文本文档              |
| tts_maker.exe | 17 KB | 应用程序              |
| tts_rc.lib    | 73 KB | Object File Lib.. |
| 开机.mp3        | 2 KB  | MPEG Layer 3 音频   |
| 蓝牙播歌.mp3      | 3 KB  | MPEG Layer 3 音频   |
| 蓝牙断开.mp3      | 3 KB  | MPEG Layer 3 音频   |
| 蓝牙连接成功.mp3    | 3 KB  | MPEG Layer 3 音频   |

Tools 目录下的工具有 3 个：

- 1) Gen\_config：生成固件配置文件的 bin 格式文件，以及配置 ID 头文件。
- 2) PCM\_maker：修改 PCM 文件头 4 个字节为实际长度的工具，详细见 [3.2.6.2 节 PCM 语音文件](#) 的介绍。
- 3) TTS\_maker：TTS 资源文件打包，详细见 [3.2.6.1 节 TTS 资源文件](#) 的介绍。



## 5.2 引导程序

引导程序是在小机上电后最早运行的程序，引导小机装载存储在外存储器 NOR Flash 或 SD 卡等中的操作系统；或者引导小机进入 ADFU 量产/升级流程。Welcome 模块也属于引导程序中的一部分。

## 5.3 操作系统

操作系统是软件运行的基础设施：

- 1) 使用 POSIX 接口形式对 uC/OS II 进行封装，提供多进程/多线程调度机制，提供多种进程间或线程间的通信方式，包括信号量、互斥锁、条件变量、消息队列等。
- 2) 提供多种软件对象管理和运行机制，包括驱动程序、应用程序、中间件等各种独立程序实体、算法库等。
- 3) 提供各种服务，包括时间闹钟定时、调频、异常和中断管理、BANK 切换、API 接口调用、VRAM、VFS、SD 文件管理、动态内存管理、共享查询管理、共享内存管理、设备检测、打印调试、系统监控等。
- 4) 提供多个 LIBC 基本功能接口，比如 memcpy，memset，memcmp 等。

操作系统实际上也是以驱动程序的形式封装的，只是有点特殊。它是引导程序加载进来的，而其他驱动程序都是通过操作系统的驱动装载接口装载进来的。

下面介绍几个重要的服务或机制。

### 5.3.1 BANK 机制

US282A 同以前的方案一样，都是内存紧缺型系统，其物理内存即 SRAM 大小都只有几十到一二百 KB，很明显是无法把整个驱动和应用加载到内存并常驻的，所以我们只好扩展物理内存空间，使用地址空间足够大的虚拟内存空间，并使用 BANK 机制对驱动和应用实现透明处理。

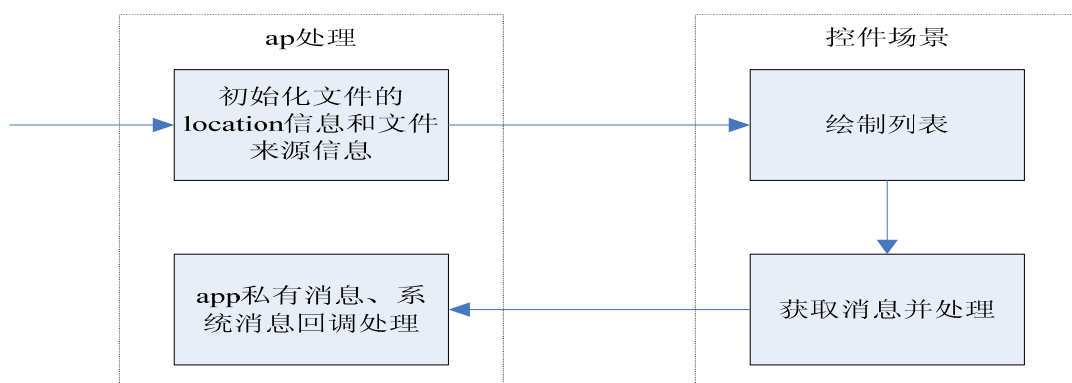
所谓 BANK 机制，就是把一块物理内存空间扩展为多块虚拟内存空间，称为 BANK 页面，对这些 BANK 页面的访问通过一定的机制映射到同一块物理内存空间上；当然，由于多个 BANK 页面共享同一块物理内存空间，所以同一时刻该物理内存空间最多只能存放其中的一个 BANK 页面，所以当程序访问到某个尚未存在物理内存空间的 BANK 页面时，就需要系统将该 BANK 页面切换到该物理内存空间。BANK 机制最重要的就是如何实现透明的 BANK 切换。在 US282A 中，BANK 机制是一种完全的硬件机制，对 MIPS 平



台的 TLB miss 机制进行扩展，以支持 DSP BANK 机制。

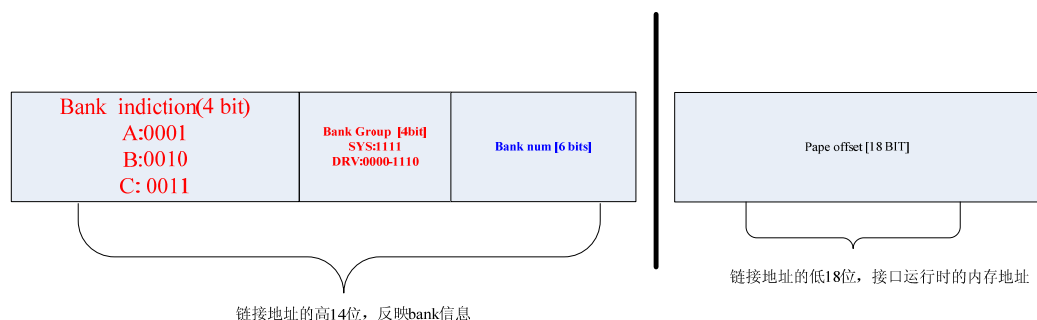
不同的程序实体的 BANK 机制设计稍微有点差异。由于方案开发一般不用修改算法库，我们在这里就不介绍 DSP 的 BANK 机制设计。方案开发涉及的程序实体主要是应用程序和驱动程序，以及一些其他独立程序实体，其 BANK 机制设计分为 2 类：驱动程序 BANK 机制和非驱动程序 BANK 机制。

### 非驱动程序 BANK 机制



- ❖ VA(virtual addr)是 BANK 的链接虚拟运行地址，包括页号 Page Num(14bit)和页偏移 Page Offset(18bit)。
- ❖ AP bank Group(8bit)表示 BANK 分组号。比如
  - AP\_BANK\_FRONT\_CONTROL\_1 = 0x40;
  - AP\_BANK\_FRONT\_UI\_1 = 0x48;
- ❖ Bank num(6bit)表示组内的 BANK 号，因此一组 BANK 最多有 64 个 BANK。
- ❖ Page offset(18bit)表示物理地址，最大支持 256KB 物理内存空间。

### 驱动程序 BANK 机制



- ❖ bank indiction (4bits)表示 kernel bank 的分类，即 Bank A/Bank B 和 Bank C，三者只有 bank 空间大小的不同。
- ❖ Bank Group(4bits)表示 bank 分组，共有 16 组，其中 1111 默认是分给内核的服务组。从空间意义上讲，bank indiction 应该算是 bank 分组的一部分。
- ❖ Bank num(6bit)表示组内的 BANK 号，因此一组 BANK 最多有 64 个 BANK。
- ❖ Page offset(18bit)表示物理地址，最大支持 256KB 物理内存空间。

BANK 使用注意要点：

- ❖ BANK 切换时，原先存放在该 BANK 组对应的物理内存空间内的代码和数据都将被覆盖，而不作任何备份和保留。所以要注意类似这样的使用情形：BANK 1 的 func 1 调用同一 BANK 组的 BANK 2 的 func2，传递 BANK 1 的 const data 的指针给 func 2，这样切换到 func 2 时，由于 BANK 1 的 const data 被覆盖，导致 func2 以该 const data 的指针访问到脏数据而出现不可预期的错误。
- ❖ 另外还需要注意一点，xn 中 bank 段名必须以 BANK 为前缀，具体见应用程序和驱动程序的\*.xn 文件说明。

## 5.3.2 API 机制

US282A 驱动程序 API 机制和 9X 以前的方案有较大的区别。因为 UA282A 是基于 32 位 MIPS M4k 内核的，系统有足够的地址位来表达并实现硬件 bank 机制。所以，US282A 的驱动程序 API，不再需要考虑 BANK 机制。

US282A 的 API 接口形式有 3 种，下面分别介绍。

### 驱动程序 API 和 VFS API

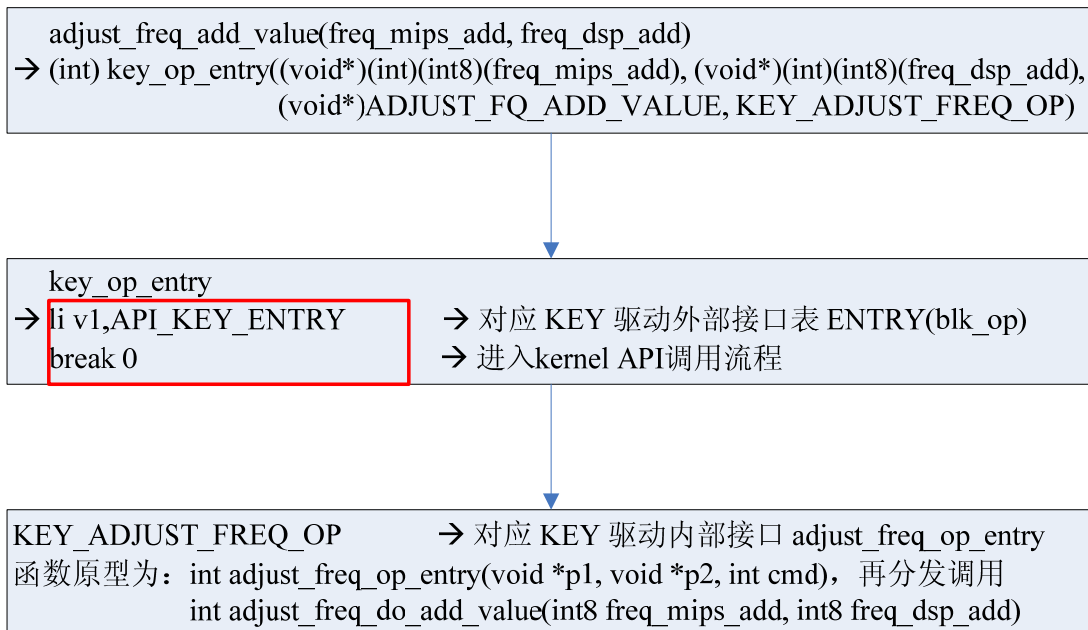
驱动程序 API 由 4 部分组成：

- 由操作系统管理的驱动统一入口：由一系列的驱动统一入口这种系统调用接口组成，统一入口 op\_entry 是 4 字节的小函数，其作用就是传入驱动对应的 op\_entry ID，并利用 break 指令陷入 API 调用流程中。
- 驱动接口表：驱动提供的一个外部接口数组，与 op\_entry ID 相对应。该接口表在链接脚本 \*.xn 中用 ENTRY(xxx\_op\_entry) 的形式告诉操作系统。
- 内部接口声明和定义：驱动内部实现，接口参数严格要求为 3 个（VFS 驱动为 4 个），实际需要参数不足 3 个（VFS 驱动为 4 个）的填充 void \* null 参数。

- 外部接口命令号及宏定义：头文件中的定义，应用和其它驱动通过这里定义的接口宏定义调用 API 接口；宏定义就是 `op_entry` 的调用实例，传入命令号和接口参数来调用指定的 API 接口，与外部接口一一对应。

驱动在安装时会把驱动接口表注册到操作系统的 API 管理器，之后调用 API 时，通过传入的 `op_entry ID` 和外部接口命令号，即可找到对应的真实接口。

API 接口的调用流程如下：（以 KEY 驱动的 API `adjust_freq_add_value()` 为例）



### 中间件和其他独立程序实体 API

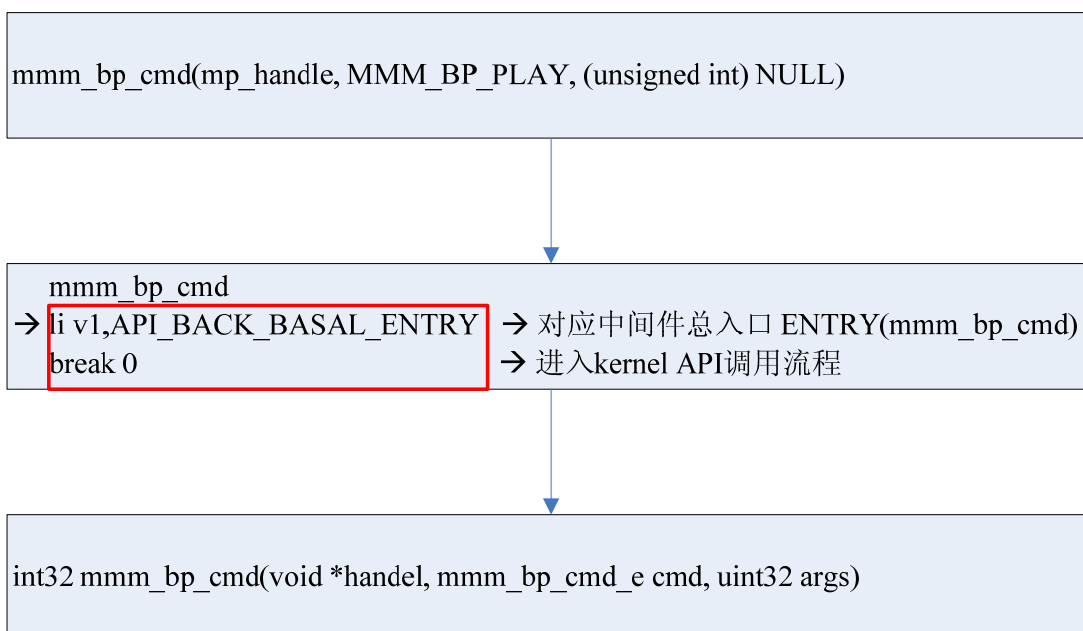
中间件 API 比驱动程序 API 和 VFS API 简单很多，它没有 API 接口表，对具体 API 接口的区分，由中间件的 API 总入口自己决定，操作系统只是负责调用该总入口。

中间件 API 由以下几个部分组成：

- 由操作系统管理的驱动统一入口：由一系列的驱动统一入口这种系统调用接口组成，统一入口 `mmm_cmd` 是 4 字节的小函数，其作用就是传入驱动对应的 `mmm_cmd ID`，并利用 `break` 指令陷入 API 调用流程中。
- 中间件外部 API 总入口：中间件提供的一个总入口函数，接口形式一般为 `int mmm_xxx_cmd(void *hd, mmm_xxx_cmd_e cmd, unsigned int param)`。该函数在链接脚本 `*.xn` 中用 `ENTRY(mmm_xxx_cmd)` 的形式告诉操作系统。
- 命令号定义：头文件中的命令 ID 枚举类型定义，中间件总入口根据该 ID 号执行具体的命令。

中间件在加载时会把总入口注册到操作系统 API 管理器，之后调用 API 时，通过传入的命令号，即可找到对应的真实程序段。

API 接口的调用流程如下：（以蓝牙推歌中间件的 MMM\_BP\_PLAY 命令为例）



### 5.3.3 中断机制

US282A 共支持 32 个中断源，具体如下表所示：

```

enum
{
 IRQ_BT = 0,
 IRQ_NFC,
 IRQ_2HZ, /*WD & 2HZ*/
 IRQ_TIMER1,
 IRQ_TIMER0,
 IRQ_RTC,
 IRQ_UART0,
 IRQ_SIRQ0,
 IRQ_TOUCHKEY,
 IRQ_SPI,
 IRQ_USB,
 IRQ_I2C,
 IRQ_UART1,
 IRQ_SIRQ1,
 IRQ_DAC_I2S_TX,
 IRQ_ADC_I2S_RX,
 IRQ_MPU,
 IRQ_SD_MMC,
 IRQ_DMA0, IRQ_DMA1, IRQ_DMA2, IRQ_DMA3, IRQ_DMA4, IRQ_DMA5,
 IRQ_PCM_RX,
 IRQ_PCM_TX,
 IRQ_SPI1,
 IRQ_OUT_USER0, IRQ_OUT_USER1, IRQ_OUT_USER2, IRQ_OUT_USER3, IRQ_OUT_USER4
};

```

申请中断接口：

```
int sys_request_irq(uint32 type, void *handle);
```

该接口会将相应的中断服务例程挂载到相应的中断源上。

注销中断接口：

```
void sys_free_irq(uint32 type);
```

该接口会将相应的中断服务例程从相应的中断源上卸载掉。

当中断发生时，中断 Pending 寄存器会设置相应的子中断发生标志位。系统采用轮询方式调用某个子中断的中断服务程序。中断服务程序需要自己把相应的中断 Pending 位清掉，否则会重复进入该中断导致死机。

中断服务例程不能放在 BANK 段中，当然也不能调用 BANK 函数；中断服务例程允许调用 API 接口。

对硬件中断使用最多的，一般都是硬件定时器，所以 US282A 开发了硬件定时器申请和注销接口：

申请硬件定时器接口：

```
int8 sys_set_irq_timer1(void* time_handle, uint32 ms_count);
```

我们提供最多 11 个硬件定时器，注意系统已经使用了一些，方案可用的还有 6 个左右。硬件定时器的定时精度为 1ms。

注销硬件定时器接口：

```
int sys_del_irq_timer1(int8 timer_id);
```

### 5.3.4 VFS 机制

虚拟文件系统的最本质特性有几点：

- 各种具体的文件系统对应用层透明，即应用层无需关心具体的文件系统，其接口是统一的。
- 各种具体的存储器驱动对文件系统也是透明的，即文件系统无需关心具体的存储器驱动，其与存储器驱动的交互交由虚拟文件系统的块设备操作层 (block layer) 处理。

在 VFS 机制下，顶层程序实体对文件系统的访问和操作步骤如下：

1) 安装存储器驱动程序

```
sys_drv_install(drv_type, (uint32)drv_mode, drv_name);
```

## 2) 挂载文件系统

```
vfs_mount = sys_mount_fs(drv_type, disk_type, 0);
```

## 3) 这样就可以访问文件系统了，进行目录和文件的操作：

```
vfs_cd(vfs_mount, CD_ROOT, 0);
```

```
vfs_dir(vfs_mount, DIR_HEAD, s_rec_folder_long, EXT_NAME_ALL_DIR);
```

```
lrc_handle = vfs_file_open(vfs_mount, NULL, R_NORMAL_SEEK);
```

```
vfs_file_read(vfs_mount, p_music_lyric_buf, lyric_block, lrc_file_handle);
```

## 4) 卸载文件系统

```
sys_unmount_fs(vfs_mount);
```

## 5) 卸载存储器驱动程序

```
sys_drv_uninstall(drv_type);
```

## 5.4 算法库

算法库是在 DSP 中运行的程序实体，包括各种格式音频的解码库、各种音频编码库、音效库、回声消除算法库等。

方案开发一般不会对算法库进行修改，所以我们也不在这里介绍算法库开发的相关内容。

## 5.5 中间件

中间件是在算法库和应用程序中间，对算法库进行封装的程序，它负责向应用程序呈现一种易于理解使用的功能模型，引擎应用通过中间件命令调用对解码及音效处理/编码/通话效果进行操控；包括各种通道的音频解码中间件、音频编码中间件、蓝牙免提中间件等。

US282A 包括以下几个中间件：

| 应用             | 中间件        | 命令式 API                                                  |
|----------------|------------|----------------------------------------------------------|
| 蓝牙推歌           | mmm_bp.al  | int mmm_bp_cmd<br>(void *, mmm_bp_cmd_e, unsigned int)   |
| 蓝牙免提           | mmm_hfp.al | int mmm_hfp_cmd<br>(void *, mmm_hfp_cmd_t, unsigned int) |
| 插卡播歌<br>插 U 播歌 | mmm_mp.al  | int mmm_mp_cmd<br>(void *, mmm_mp_cmd_t, unsigned int)   |

|                          |           |                                                        |
|--------------------------|-----------|--------------------------------------------------------|
| 音频输入<br>FM 收音机<br>USB 音箱 | mmm_pp.al | int mmm_pp_cmd<br>(void *, mmm_pp_cmd_e, unsigned int) |
| 录音                       | mmm_mr.al | int mmm_mr_cmd<br>(void *, mmm_mr_cmd_t, unsigned int) |

中间件调用流程如下，以蓝牙推歌为例，在蓝牙推歌引擎应用中：

1) 加载中间件

```
sys_load_mmm("mmm_bp.al", TRUE); //第二个参数表示引擎应用对应的中间件
```

2) OPEN，初始化中间件工作环境

```
mmm_bp_cmd(&mp_handle, MMM_BP_OPEN, (unsigned int) NULL);
```

3) 配置解码环境和参数，比如打开音频输出通道

```
mmm_bp_cmd(mp_handle, MMM_BP_AOUT_OPEN, (unsigned int) NULL);
```

等等

4) PLAY，加载蓝牙推歌解码库及音效库，并启动 DSP CORE

```
mmm_bp_cmd(mp_handle, MMM_BP_PLAY, (unsigned int) NULL);
```

5) STOP，停止 DSP CORE，并卸载蓝牙推歌解码库及音效库

```
mmm_bp_cmd(mp_handle, MMM_BP_STOP, (unsigned int) NULL);
```

6) 停止解码环境，比如关闭音频输出通道

```
mmm_bp_cmd(mp_handle, MMM_BP_AOUT_CLOSE, (unsigned int) NULL);
```

等等

7) CLOSE，销毁中间件工作环境

```
mmm_bp_cmd(mp_handle, MMM_BP_CLOSE, (unsigned int) NULL);
```

8) 卸载中间件

```
sys_free_mmm(TRUE); //第二个参数表示引擎应用对应的中间件
```

注意：同时只能有一个引擎应用对应的中间件存在。

## 5.6 驱动程序

驱动程序是一种向其他程序提供服务的独立程序实体，包括文件系统、卡驱动、UHOST 驱动、Nor Flash 驱动、USB 驱动簇、TTS 驱动、音频设备驱动、以及 CASE 各种小驱动，包括蓝牙控制器驱动、KEY 驱动

等。

## 5.7 应用程序

应用程序是产品直接看到的和体验到的功能的程序实体，包括应用管理器、开关机应用、读卡器应用、蓝牙协议栈、各种应用的前台应用和引擎应用等。

应用程序可以分为 4 类：

- 1) 应用管理器：管理其它应用的应用程序。
- 2) 蓝牙协议栈：对蓝牙协议栈的封装。
- 3) 前台应用：负责用户交互的应用程序。
- 4) 引擎应用：负责实际功能控制的应用程序。

典型的应用场景下，会有应用管理器、蓝牙协议栈、一个前台应用和一个引擎应用。

## 5.8 其他独立程序实体

有些功能模块在整个软件上保持一定的独立性，我们希望对其进行模块化，单独开发和维护，那么可以封装为前台应用对应的“中间件”，即除了上述几种程序实体之外的，称为其他独立程序实体；包括本地歌曲磁盘扫描、USB/UART STUB 模块等。

这种模块与上述的中间件基本上是一致的，只是空间不一样和使用方式不同。在系统加载时，调用形式为：`sys_load_mmm(stub_name, FALSE)`；第二个参数表示前台应用对应的“中间件”。当然，卸载对应为：`sys_free_mmm(FALSE)`；。

注意：同时只能有一个前台应用对应的“中间件”存在。

## 5.9 API 接口

不同程序实体之间，需要通过各种 API 接口进行远程调用。API 接口是以 \*.o 或 \*.a 静态库的方式提供出来，供调用方直接链接到程序实体中。



## 6 COMMON 详解

### 6.1 概述

COMMON 模块为应用封装并实现了软件平台的大部分基础服务，包括 APPLIB、用户交互服务 VIEW、应用切换、时间闹钟服务、电池充电管理、蓝牙管理器、低功耗模式、音量调节、音效调节等。

COMMON 虽然属于应用程序中的一部分链接进去，但是 COMMON 在逻辑上它是独立的。有些 COMMON 模块它拥有独立的全局的数据空间，能够在应用切换中保持不变，所以可以看成是一直运行的独立功能模块，比如蓝牙管理器、COMMON 类软定时器、分立 LED 模块等。

### 6.2 APPLIB

APPLIB 应用基本功能接口库，包括应用程序管理、消息通信管理、软定时器管理、配置项解析。这些接口与应用的基本架构有很大的关系。

每个前台应用和引擎应用都必须链接 APPLIB，共享全局数据。

#### 6.2.1 应用程序管理

##### 6.2.1.1 应用管理

US282A 是一个抢占式多任务调度系统，可以同时存在 4 个应用，即进程管理器 ap\_manager，蓝牙协议栈，前台应用和后台应用。应用管理以及应用间的消息通信等这些需求，都要求系统必须管理所有应用。

所以，AppLib 为每个应用分配一个 app\_info\_t 结构体对象，共 3 个结构体对象。该结构体描述如下表所示：

| 结构体成员 | 说明                        |
|-------|---------------------------|
| used  | 结构体使用标志，1 表示已被使用，0 表示未被使用 |

|          |                             |
|----------|-----------------------------|
| app_id   | 进程 ID 号                     |
| app_type | 应用类型，即进程管理器、蓝牙协议栈、前台应用或后台应用 |
| mq_id    | 进程私有消息队列 ID                 |

在接口设计上，我们设计了以下两个接口：

- **bool applib\_init(uint8 app\_id, app\_type\_e type):** 应用程序的注册和初始化。
- **bool applib\_quit(void):** 应用程序注销。

以上 2 个接口除了创建或销毁应用外，还会进行调频，因为现在调频是以线程为基本单元的，创建或销毁应用必然要伴随着调频。

另外，为了方便使用，我们为每个应用程序开设一个全局变量 `g_this_app_info` 指向自身的 `app_info_t` 结构体对象，该变量在 `applib_init` 中初始化。

### 6.2.1.2 引擎管理

现在我们增加了好几个引擎，对引擎的处理显得有点复杂，每个应用都必须去处理引擎冲突，需要知道有多少个引擎，引擎类型与 `app id` 之间的对应关系。

所以，US282A 将引擎相关信息集中到 APPLIB 中，在 `app_engine_config.c` 中增加一张表，来表示所有的引擎类型与 `app id` 之间的对应关系，并增加了一个虚拟 APP ID —— `APP_ID_THEENGINE`，表示当前的引擎应用。应用要杀掉当前引擎，只需要填写这个 ID 好，`ap_manager` 自己可以通过获取引擎类型和查表，就知道对应的真实 APP ID。

用法如下：

```
engine_type = get_engine_type();
if ((engine_type != ENGINE_NULL) && (engine_type != ENGINE_LINEIN))
{
 linein_close_engine(APP_ID_THEENGINE);
}
```

所以，我们提供了以下 2 个小接口：

- **engine\_type\_e get\_engine\_type\_by\_appid(uint8 app\_id):** 判断一个 `app id` 是否引擎。

- 当然，uint8 `get_engine_appid_by_type(engine_type_e engine_type)`: 返回引擎类型对应的 APP ID。

```
typedef struct
{
 uint8 engine_app_id;
 uint8 engine_type;
} app2engine_type_t;

const app2engine_type_t applib_app2eg_type[] =
{
 { APP_ID_MENGINE, ENGINE_MUSIC },
 { APP_ID_FMENGINE, ENGINE_RADIO },
 { APP_ID_LINEIN_EG, ENGINE_LINEIN },
 { APP_ID_BTPLAYEG, ENGINE_BTPLAY },
 { APP_ID_BTCALLEG, ENGINE_BTCALL },
 { APP_ID_UENGINE, ENGINE_USOUND },
};
```

### 6.2.1.3 运行时库 Ctor.o

Ctor.o 运行时库是一个特殊的库文件，AP 加载后会先运行 Ctor.o 中的 `__start` 函数（在 AP 的链接脚本 `xn` 文件中需要把 `ENTRY` 指定为 `__start`），该函数为该 AP 创建一个主线程，然后跳转到 `main` 函数开始运行 AP 的代码。

## 6.2.2 消息通信管理

### 6.2.2.1 概述

在多任务调度系统中，应用间的通信是非常重要的。US282A 选择了消息通信作为应用间的通信方式的主要方式。另外，US282A 还使用了按键消息和系统消息实现按键输入和系统事件捕捉。

由于按键消息、系统消息和应用间消息的内容不一致，所以，我们设计了以下几种类型的消息队列：

- 按键消息队列：用来存储机械按键、红外遥控器、触摸按键等产生的消息，系统只有一个按键消息队列。按键消息队列只有前台应用才会访问到，是用户与前台应用交互的通道。

```
/*!
 * \brief
 * key_phy_msg_t 物理按键消息
 */
typedef struct
{
 uint8 key_type :3;
 uint8 key_status :2;
 uint8 key_ir_code :2;
 uint8 key_reserve :1;
 uint8 key_id; /*0开始, 0x80表示抬起, 0xff表示无按键*/
 uint16 key_timestamp;
} key_phy_msg_t;
```

- 系统消息队列：用来存储系统消息，这种消息非常简单，只需要带有消息类型即可，系统也只有一个系统消息队列。

```
/*!
 * \brief
 * sys_msg_t 系统消息结构体
 */
typedef struct
{
 /*! 应用消息类型 */
 msg_apps_type_e type;
} sys_msg_t;
```

- 应用间消息队列：也称为应用私有消息队列，用来存储其他应用发送过来的应用间消息，这种消息除了带有消息类型外，还需要带输入参数和输出参数。每个应用都需要一个私有消息队列。

```
/*!
 * \brief
 * private_msg_t （应用间）私有消息结构体
 */
typedef struct
{
 /*! 私有消息消息内容 */
 msg_apps_t msg;
 /*! 同步信号量 */
 os_event_t *sem;
 /*! 同步消息回执指针 */
 msg_reply_t *reply;
} private_msg_t;
```

```
/*!
 * \brief
 * msg_apps_t 应用消息结构体
 */
typedef struct
{
 /*! 应用消息类型，参见 msg_apps_type_e 定义 */
 uint32 type;
 /*! 应用消息内容 */
 union
 {
 /*! 消息内容真实数据 */
 uint8 data[4];
 /*! 消息内容缓冲区指针，指向消息发送方的地址空间 */
 void *addr;
 } content;
} msg_apps_t;

/*!
 * \brief
 * msg_reply_t 同步消息回执结构体
 */
typedef struct
{
 /*! 同步消息回执枚举类型，参见 msg_reply_type_e 定义 */
 uint8 type;
 uint8 reserve[3];
 /*! 回执内容缓冲区指针，指向消息发送方的地址空间 */
 void *content;
} msg_reply_t;
```

- 蓝牙协议栈应用与其他应用在通信数据上有些差异，为了支持蓝牙多设备，我们为每条消息预留了一个蓝牙地址成员。蓝牙管理器向蓝牙协议栈发送的消息结构如下：

```
#define BD_ADDR_LEN 6

/*!
 * \brief
 * btstack_msg_t BT MANAGER控制BT STACK的消息结构体
 */
typedef struct
{
 /*! 私有消息内容 */
 msg_apps_t msg;
 /*! 同步信号量 */
 os_event_t *sem;
 /*! 同步消息回执指针 */
 msg_reply_t *reply;
 /*! 保留字节 */
 uint8 reserve[2];
 /*! 命令对象的设备地址；并非所有命令都需要设备地址 */
 uint8 bd_addr[BD_ADDR_LEN];
} btstack_msg_t;
```

蓝牙协议栈向蓝牙管理器发送的消息结构如下，我们称之为蓝牙协议栈事件：

```
typedef struct
{
 /*! 事件内容 */
 msg_apps_t msg;
 /*! 保留字节 */
 uint8 reserve[2];
 /*! 事件发起的设备地址；并非所有事件都有设备地址 */
 uint8 bd_addr[BD_ADDR_LEN];
} btstack_event_t;
```

以上消息类型定义为：

```
typedef enum {
 MQ_ID_MNG = 0, /*进程管理应用消息队列*/
 MQ_ID_DESK, /*前台应用消息队列*/
 MQ_ID_EGN, /*引擎应用消息队列*/
 MQ_ID_BT, /*蓝牙协议栈消息队列*/
 MQ_ID_RES, /*蓝牙协议栈事件队列*/
 MQ_ID_SYS, /*系统消息队列*/
 MQ_ID_GUI, /*按键消息队列*/
 MQ_ID_MAX
} mq_id_e;
```

### 消息通信模块初始化 `bool applib_message_init(void)`

应用启动并且注册和初始化后，必须调用接口 `applib_message_init()` 进行消息通信模块初始化，该接口

会把当前应用的私有消息队列清空。

该接口分为前台应用版本和非前台应用版本，后者给引擎应用和蓝牙协议栈调用。蓝牙协议栈相关的消息队列和事件队列在蓝牙管理器初始化时主动清空。

**检索消息** `int sys_mq_traverse(uint8 queue_id, void* msg, uint32 msg_index)`

当我们仅仅需要查看某个消息队列是否有消息，以及看看有什么消息，而不从消息队列取走消息，那么就可以调用 `sys_mq_traverse()`，比如在判断是否有双击按键发生时，我们需要查看是否已经明确已发生双击按键或者不可能发生双击按键，那么就可以先检索物理按键消息队列；只在确定之后才真正从物理按键消息队列中取走消息并转换为逻辑消息。

**发送消息** `int sys_mq_send(uint8 queue_id, void *msg)`

**接收消息** `int sys_mq_receive(uint8 queue_id, void *msg)`

## 6.2.2.2 按键消息

按键驱动发送出来的按键消息，是物理按键消息，只包含何时按下或按住某个物理按键，或按键抬起，它并不是应用可以使用的逻辑按键消息，所以需要在应用层进行逻辑分析。

由于物理按键消息区分了机械按键、红外遥控器、触摸按键，所以应用层有能力分别独立的处理这 3 种按键，即使同时按下也能够区分开。

物理按键消息发送接口如下：

```
sys_mq_send(MQ_ID_GUI, (void *) &key_msg); // key_phy_msg_t key_msg;
```

应用层并没有直接处理物理按键消息，而是先转换为如下结构的逻辑按键消息：

```
/*!
 * \brief
 * input_gui_msg_t 输入型消息，又称 gui 消息结构体
 */
typedef struct
{
 /*! 消息类型*/
 input_msg_type_e type;
 /*! 消息数据*/
 union
 {
 /*! 按键事件*/
 key_event_t kmsg;
 } data;
} input_gui_msg_t;

/*!
 * \brief
 * key_event_t 按键事件结构体
 */
typedef struct
{
 /*! 按键（逻辑）值*/
 key_value_e val;
 /*! 按键选项: bit0 表示是否需要按键音，用于临时屏蔽按键音 */
 uint8 option;
 /*! 按键类型*/
 key_type_e type;
 /*! 后续按键处理选项*/
 key_deal_e deal;
} key_event_t;
```

获取逻辑按键消息的规则要点如下：

- 1) 如果当前场景检索到的物理按键消息对应的逻辑按键有双击按键事件，那么就会进入双击按键判断状态中，会暂时 HOLD 住逻辑按键消息的转发，直到确定已经发生双击按键事件或不可能发生双击按键事件；才转发逻辑按键消息。所以，带双击按键事件的按键响应会慢一些。
- 2) 逻辑按键各种动作是在该模块中确定的。
- 3) 支持逻辑按键消息暂缓处理机制，即将获取到的逻辑按键消息保存到 `g_buffer_gui_msg` 中：  
`libc_memcpy(&g_buffer_gui_msg, p_gui_msg, sizeof(input_gui_msg_t));`  
然后下次调用 `get_gui_msg()` 时直接将 `g_buffer_gui_msg` 返回。

逻辑按键消息接收接口，由该接口完成物理按键消息转换为逻辑按键消息：

```
bool get_gui_msg(input_gui_msg_t *input_msg);
```



### 6.2.2.3 系统消息

系统消息源有 3 个：

- 1) 操作系统检测到一些事件，比如闹钟起闹、USB/UHOST 拔插等事件，就通过发送系统消息来告诉应用层。
- 2) 按键驱动也会主动周期检测一些事件，比如卡热拔插、AUX 热拔插、耳机热拔插等事件，也是通过发送系统消息来告诉应用层。
- 3) 应用层主动监测某些系统相关的事件，比如低电、充电满、蓝牙连接或断开、蓝牙免提来电等事件，通过发送系统消息，**异步处理**，让其他需要知道该事件的地方有机会知道并处理该事件。

发送系统消息使用以下接口：

```
sys_mq_send(MQ_ID_SYS, &msg); //uint16 msg;
```

接收系统消息使用以下接口：

```
bool get_sys_msg(sys_msg_t *sys_msg)
```

系统消息是在前台应用接收应用私有消息时接收，并以应用私有消息广播到前台应用的私有消息队列，然后由前台应用接收和处理。这样系统消息队列好像是多余的，但因为系统消息一般是在操作系统和驱动发出的，它们并没有应用程序的相关信息，不能给应用程序发送应用私有消息，所以就只能先发送到系统消息队列中。

### 6.2.2.4 应用私有消息

对于应用私有消息，我们提供了 3 种发送方式：

- 同步发送应用私有消息：这种方式用于实现同步控制，消息发送方发送同步消息后，会等待直到消息接收方接收并应答后再继续，保证了发送方和接收方的流程先后关系。目前同步发送用于前台应用发送同步消息给后台应用，或者前台应用发送同步消息创建/杀死后台应用。同步发送通过接口 `bool send_sync_msg(uint8 app_id, msg_apps_t *msg, msg_reply_t *reply, uint32 timeout)` 发送消息。
- 异步发送应用私有消息：这种方式用于消息发送方通知接收方一事件，至于这件事情最终是否通知到接收方则无从知道。异步发送通过接口 `bool send_async_msg(uint8 app_id, msg_apps_t *msg)` 发送消息。

- 广播发送应用私有消息：这种方式用于在不知道具体接收方或者期望所有应用都收到消息的情况下发送消息，广播发送也属于异步发送。广播发送通过接口 `bool broadcast_msg(msg_apps_t *msg)` 发送消息，前台应用转发系统消息则通过接口 `bool broadcast_msg_sys(msg_apps_t *msg)` 发送消息。

前面我们提到，应用私有消息是带有输入参数和输出参数的，下面针对这一点具体说明：US282A 是内存紧缺型的方案，内存资源十分有限，对于大空间参数，我们没有直接把消息参数拷贝到消息队列，而是把消息参数放在发送方的缓冲区，只提供该缓冲区指针给消息接收方。但这样做有一个问题，就是如果接收方还没有处理完消息，而发送方又把消息参数缓冲区给破坏掉了，那么接收方处理消息时就会出现错误。所以这种方式是有限制条件的，只有通过同步发送的方式，才能有效的规避这种情况。

同样的原理也适用于输出参数，只有同步发送方式，才能保证从消息发送回来后，能安全使用输出参数。当然，输出参数的缓冲区也是开在发送方的内存空间上的。

所以，针对输入参数和输出参数，我们规定：

- 系统消息只有异步发送方式，无法带有消息参数和输出参数。
- 私有消息异步发送方式，只能带有 4 个字节的参数；不能带输出参数。
- 私有消息同步发送方式，可以带有 4 个字节的参数，也可以使用大空间参数方式，这两种方式由消息双方决定；可以带输出参数。

应用私有消息接收接口：

```
bool get_app_msg(private_msg_t *private_msg)
```

收到同步消息后，必须在响应消息的最后进行应答，消息发送方才能接着运行下去。同步消息的应答通过接口 `libc_sem_post(sem)`；实现，这样才能避免应答同步消息时发生 BANK 切换。

### 6.2.2.5 蓝牙协议栈消息

蓝牙协议栈消息的发送使用专用接口：

```
int send_sync_msg_btmanager(uint8 *bd_addr, msg_apps_t *msg, msg_reply_t *reply, uint32 timeout);
int send_async_msg_btmanager(uint8 *bd_addr, msg_apps_t *msg);
```

该接口无须指定 APP ID，取而代之的是蓝牙设备地址。

蓝牙协议栈消息的接收也使用专用接口：

```
bool get_app_msg_btstack(btstack_msg_t *btstack_msg);
```

### 6.2.2.6 蓝牙协议栈事件

蓝牙协议栈事件的发送使用专用接口，并且只需要支持异步发送：

```
int send_async_event(uint8 app_id, btstack_event_t *ev);
```

其中，app\_id 必须是 APP\_ID\_BT\_MANAGER，这是一个虚拟的 APP ID，表示蓝牙管理器。

蓝牙协议栈事件的接收直接使用系统接口，这样可以减少常驻代码量：

```
sys_mq_receive((uint32) (MQ_ID_RES), (void *) &btstack_ev);
```

## 6.2.3 软定时器管理

我们在 US282A 的应用程序上设计并实现了软定时器，以方便用户在开发应用程序时方便地开发周期执行业务功能。

软定时器工作原理：应用级定时器不需要硬件 timer 中断的支持，只要求能获取到系统的绝对时间。我们软件设置一个超出当前绝对时间一个周期的时间点 Tout，然后循环检测（在 `get_app_msg` 中调用检测接口 `handle_timers`，检测周期与 `sys_os_time_dly` 睡眠时间相比拟）最新绝对时间是不是超过了时间点 Tout，如果超过就说明已经至少经过了一个周期的时间，那么这时就执行该周期性功能，并且把 Tout 延后一个周期时间。这样便可以实现周期性执行某个功能了。

软定时器能够设置的定时精度为 1ms，但是由于它是挂载在应用的主循环来检测定时的，可能会有 10 ~ 20 ms 的时间偏差。所以，软定时器不能用于非常精准的定时，只能用于粗略的定时场景，比如 LED 灯周期闪烁等。

软定时器实际上就是一个特定条件执行的函数调用，当然可以把定时器 `handle` 实现为 `bank` 函数或者调用其他 `bank` 函数。当然，`handle` 函数也是无法实现函数参数传递的，只能通过全局变量传递操作数据和返回操作结果。

软定时器由于是完全软件实现的定时器，可以设计实现丰富的功能接口，包括创建定时器，修改定时器周期，停止计时，重启计时，以及删除定时器等。

US282A 还实现了一种“闪烁”软定时器，即一个软定时器设置了 2 个交替更新的定时器，可以用来实现分立 LED 等的闪烁效果。

再者，US282A 提供了几个 COMMON 软定时器，其数据是放在 COMMON 的全局数据区的，而不是放在前台应用的全局数据区；所以这类软定时器不会因为切换应用而失效。

- 创建周期性定时器：

```
int8 set_app_timer(uint32 attrb_tagh, uint16 timeout, timer_proc func_proc);
```

其中 attrb\_tagh 是复用的参数，低 8bit 表示属性，高 8bit 表示 TAG 号。

- 创建单发定时器：

```
int8 set_single_shot_app_timer(uint32 attrb_tagh, uint16 timeout, timer_proc func_proc);
```

- 创建闪烁定时器：

```
int8 set_twinkle_app_timer(uint32 attrb_tagh, twinkle_app_timer_param_t *twinkle_param);
```

- 修改定时器周期：bool **modify\_app\_timer**(int8 timer\_id, uint16 timeout);
- 停止计时：bool **stop\_app\_timer**(int8 timer\_id);
- 重启计时：bool **restart\_app\_timer**(int8 timer\_id);
- 删除定时器：bool **kill\_app\_timer**(int8 timer\_id);

另外，为了减少 BANK 切换，减少 BANK 代码，我们特地为蓝牙管理器增加了一个单发软定时器创建接口，以满足蓝牙管理器频繁使用单发软定时器的需求：

- 蓝牙管理器专用的创建单发软定时器：

```
int8 set_single_shot_app_timer_btmanager(uint32 attrb_tagh, uint16 timeout, timer_proc func_proc);
```

另外，当系统进入 S2 低功耗模式，软定时器不再执行，计时也应理解为作废，所以从 S2 低功耗模式退出时，需要重启所有软定时器，可以使用以下接口：

- 重启所有软定时器：

```
void standby_restart_all_app_timer(app_timer_t *timers, uint8 count);
```

#### 软定时器模块初始化：

应用启动并且注册和初始化后，必须调用接口 `bool init_app_timers(app_timer_t *timers, uint8 count)` 进行应用级定时器模块初始化。

COMMON 类软定时器数组的初始化，只需要在开机时初始化一次，后面无须也不能再初始化。

软定时器模块无须销毁，应用退出时，应用内的软定时器数组（存放在应用的全局数据区中）自然就没有了。

软定时器使用说明：

- 定时器类型：分为单发定时器和周期发送定时器，前者创建之后定时器触发执行一次就自动删除了，使用接口 `set_single_shot_app_timer` 创建；后者创建之后周期触发执行，使用接口 `set_app_timer` 创建，使用结束后必须调用接口 `kill_app_timer` 删除。
- 定时器属性：分为 `APP_TIMER_ATTRB_UI` 类软定时器和 `APP_TIMER_ATTRB_CONTROL` 类软定时器，前者只会在背光等亮着时检测并触发执行；后者则没有这样的限制。
- 要让软定时器不受切换应用的影响，需要使用 COMMON 类软定时器的属性，即 `APP_TIMER_ATTRB_COM_UI` 和 `APP_TIMER_ATTRB_COM_CONTROL`，这两种属性仅仅是表示该软定时器是 COMMON 类的，在内部会转换为 `APP_TIMER_ATTRB_UI` 和 `APP_TIMER_ATTRB_CONTROL`。
- 定时器 TAG：因为定时器在数据空间上是属于整个应用的，所以在所有场景中都能够检测到所有定时器并触发执行。但是我们不希望在 A 场景创建的定时器，到了 B 场景还能够触发执行，所以我们为每个定时器都打上一个场景 TAG，用来限制定时器的检测和触发执行。
  - 1) 系统定时器使用 `APP_TIMER_TAG_SYS`，0xff，所有场景都会检测和触发执行。
  - 2) 其他场景 TAG 号使用 VIEW 的 ID 号。

## 6.2.4 配置项解释

case 配置项使用很简单，系统启动时在 `ap_manager` 打开 `config.bin` 文件，得到文件句柄 `config_fp`，之后其他所有应用程序可以直接 `config_fp` 读取解释配置项。

配置项解释提供的接口有 2 个：

1. 读取 case 配置项：

`bool com_get_config_struct(uint16 config_id, uint8 *buf, uint16 buf_len)`

其中 `buf` 实际对应于以下 4 中结构体中的一种：uint32

```

/*!
 * \brief
 * config_string_t 字符串配置项数据结构
 */
typedef struct
{
 /*! 字符串内容，可变长数组，内容与txt输入一致，以'\0'结束 */
 uint8 value[1];
} config_string_t;

/*!
 * \brief
 * config_linear_t 线性数值配置项数据结构
 */
typedef struct
{
 /*! 默认数值 */
 uint16 default_value;
 /*! 取值区间的最小值 */
 uint16 min;
 /*! 取值区间的最大值 */
 uint16 max;
 /*! 步长 */
 uint16 step;
} config_linear_t;

/*!
 * \brief
 * config_nonlinear_t 非线性数值配置项数据结构
 */
typedef struct
{
 /*! 默认数值 */
 uint16 default_value;
 /*! 有效值个数 */
 uint16 total;
 /*! 有效值数组，可变长数组 */
 uint16 value[1];
} config_nonlinear_t;

```

2. 读取 case 配置项默认值，该接口只对数值参数有效：

uint16 com\_get\_config\_default(uint16 config\_id)

## 6.3 APP\_RESULT\_E

对于前台应用和 COMMON 来说，我们广泛使用 app\_result\_e 这个枚举类型来传递模块间或应用间的处理结果。原则上我们要求谨慎返回 app\_result\_e，并且必须谨慎处理该返回值。

app\_result\_e 以下几个值需要特别说明：

1)

```

/*!
 * \brief
 * app_result_e 应用层接口返回结果枚举类型
 */
typedef enum
{
 /*! 没有任何需要通知调用者的情况返回，调用者不需要做任何特殊处理 */
 RESULT_NULL = 0x00,
 RESULT_IGNORE = 0x01,
 /*! 没有任何事件，用于 com_view_loop 返回，表示没有按键消息，系统消息，BT事件等 */
 RESULT_NONE_EVENT = 0x02,

 /*! 无关紧要的结果，一般不会引起场景和应用切换 */
 RESULT_COMMON_RESERVE = 0x20,

```

## 6.4 VIEW 机制

### 6.4.1 概述

以往的方案的应用架构有 2 个问题：

- 1) 场景嵌套和功能嵌套比较多，对栈的压力非常大。
- 2) UI 和控制并没有分开，混杂在一起，UI 开发难度较大。

因此，US282A 打算采用新的应用架构来解决这两个问题：

- 1) 采用扁平化的程序运行架构，减轻栈的压力。
- 2) 采用 MVC 应用架构，将 UI 和控制分开，便于 UI 开发。

采用视图管理器来统一管理场景嵌套，广泛使用“事件机制”进行用户交互，这样编写前台应用的架构将非常简单，开发方式类似于搭积木。

### 6.4.2 基本元素

US282A 的 MVC 应用架构以用户交互场景为基本元素，叫做视图，我们把视图分为 3 类：

- 主视图：实现应用主体功能的场景，有按键输入和系统消息输入。
- 子视图：实现一些独立的小功能的场景，有按键输入，使用主视图的系统消息输入，可以定时退出。
- 消息视图：仅仅进行消息提示，可以定时退出，用户交互还是由上述子视图或主视图（如果没有子视图的话）执行。

每个视图都要实现一个视图刷新回调函数，函数形式为：

```
typedef void (*view_update_cb)(view_update_e mode);
```

分别在以下 4 种情形中执行各自代码分支：

- 1) 创建：创建视图时调用，可以申请一些与视图有关的资源，然后刷一次 UI。
- 2) 销毁：退出视图时调用，将创建时申请的资源释放掉。
- 3) 重绘：当进入上一层视图后返回，需要重绘一次 UI。
- 4) 更新：当发生与当前视图有关的状态或数据变化时，需要更新一次 UI。



支持主视图->子视图->消息视图，这个是最多嵌套视图。创建一个视图时，会先将视图栈中多个视图销毁掉，比如创建子视图，会先将消息视图和子视图销毁掉，然后再添加视图到视图栈中。当应用主动销毁一个视图时，会重绘下一层的视图 UI。

### 视图 ID

每个视图都需要分配一个视图 ID，该 ID 作为软定时器的 TAG，这样就可以做到，在某个主视图创建的 UI 类软定时器，进入到子视图时不会运行，开发者不用自己去将主视图创建的 UI 类软定时器暂停。

### 视图超时退出

对于子视图和消息视图而言，它们都可以理解为嵌套在主视图上的临时视图，所以如果一段时间没操作，那么就要返回主视图，所以我们添加了视图超时机制，当然也可以取消掉超时机制，即将 overtime 赋值为 0。

## 6.4.2.1 主视图

一个主视图，必须实现以下部分：

- 1) 创建主视图的接口。
- 2) 按键事件映射表（必须常驻内存），以及各个按键的处理函数。
- 3) 系统事件映射表（可以没有，如果有必须常驻内存），以及各个系统消息的处理函数。
- 4) 视图 UI 回调函数，以及 UI 显示函数。
- 5) 视图的创建和销毁需要做的事情，封装为一个函数，放到视图 UI 回调函数中的创建和销毁分支。
- 6) 调用 `com_view_loop`，并妥当处理它的返回值。
- 7) 主循环中要自己检测状态，如果状态发生变化，请调用  
`com_view_update(APP_VIEW_ID_MAIN)`；接口来更新 UI。

创建主视图例子：



```
void btplay_create_main_view(void)
{
 create_view_param_t create_view_param;

 create_view_param.type = VIEW_TYPE_MAIN;
 create_view_param.unknown_key_deal = UNKNOWN_KEY_IGNORE;
 create_view_param.overtime = 0;
 create_view_param.ke_maplist = btplay_ke_maplist;
 create_view_param.se_maplist = btplay_se_maplist;
 create_view_param.view_cb = btplay_main_view;
 com_view_add(APP_VIEW_ID_MAIN, &create_view_param);
}
```

注意：

- 1) 应用的按键事件映射表需要放在 .text 或 .data 段中。
- 2) 应用的系统消息映射表需要放在 .text 或 .data 段中。

### 6.4.2.2 子视图

一个子视图，必须实现以下部分：

- 1) 创建子视图的接口。
- 2) 按键事件映射表（必须常驻内存），以及各个按键的处理函数。
- 3) 视图 UI 回调函数，以及 UI 显示函数。
- 4) 视图的创建和销毁需要做的事情，封装为一个函数，放到视图 UI 回调函数中的创建和销毁分支。

US282A 对以下功能模块使用了子视图：

- 1) 时间设置
- 2) 日历设置
- 3) 闹钟设置

创建子视图例子：

```
void tm_alarm_create_clock_view(void)
{
 create_view_param_t create_view_param;

 create_view_param.type = VIEW_TYPE_SUB;
 create_view_param.unknown_key_deal = UNKNOWN_KEY_IGNORE;
 create_view_param.overtime = 8000;
 create_view_param.ke_maplist = clock_ke_maplist;
 create_view_param.se_maplist = NULL;
 create_view_param.view_cb = tm_alarm_clock_view;
 com_view_add(COM_VIEW_ID_DISP_CLOCK, &create_view_param);
}
```

注意:

1) 应用的按键事件映射表需要放在 .text 或 .data 段中。

### 6.4.2.3 消息视图

消息视图很简单，需要实现以下部分：

- 1) 创建消息视图的接口。
- 2) 视图 UI 回调函数，以及 UI 显示函数。
- 3) 视图的创建和销毁需要做的事情，封装为一个函数，放到视图 UI 回调函数中的创建和销毁分支。

创建消息视图例子：

```
void com_create_volume_view(void)
{
 create_view_param_t create_view_param;

 create_view_param.type = VIEW_TYPE_MSG;
 create_view_param.unknown_key_deal = UNKNOWN_KEY_IGNORE;
 create_view_param.overtime = 3000;
 create_view_param.ke_maplist = NULL;
 create_view_param.se_maplist = NULL;
 create_view_param.view_cb = com_set_volume_view;
 com_view_add(COM_VIEW_ID_SET_VOLUME, &create_view_param);
}
```

如果方案不需要显示，就不要创建消息视图。

US282A 对以下功能模块使用了消息视图：

- 1) 音量设置
- 2) 音效设置

## 6.4.3 VIEW LOOP

VIEW LOOP 是前台应用用户交互的总入口，负责处理按键消息、系统时间、软定时器 HANDLE、蓝牙事件包括 RCP 事件、等等。

### 6.4.3.1 按键消息处理

US282A 广泛使用了事件机制，我们把按键消息也理解为事件，直接在按键事件映射表中映射为回调函数，这样就可以由系统自己调用，用户无需关心其中细节，用户要做的就是编写好模块性很好的回调函数，然后填到按键事件映射表中。

我们支持一个视图有 2 个按键事件映射表的机制，即包括视图私有的按键事件映射表和 COMMON 按键事件映射表。按键事件会优先在视图私有的按键事件映射表进行映射，如果映射到了就无须在 COMMON 按键事件映射表中映射；如果映射不到，就会在 COMMON 按键事件映射表中映射。

另外，我们增加多一个灵活的机制，即在视图私有按键事件映射表中映射并处理之后，如果还希望在 COMMON 按键事件映射表中再映射处理，可以返回 RESULT\_KEY\_EVENT\_REDEAL\_BY\_COMMON。

我们还扩展了按键事件结构体，将按键消息过滤功能也直接标志在按键事件映射关系中。

在调用按键事件映射处理时，往往会发生 BANK 切换，而该接口的参数 key\_map\_list 是 const data，在应用程序中是属于 .rodata 段，一般打包到 BANK 段中，如果调用该接口的代码（更准确的说是 key\_map\_list 实例）存放在 UI BANK 段，那么 key\_map\_list 实例就会在切换到该接口所在的 UI BANK 段而被覆盖掉而出错。所以，调用该接口的代码（更准确的说是 key\_map\_list 实例）要么放在 CONTROL BANK 段中，要么把 key\_map\_list 实例放到同该接口相同的 UI BANK 段中，这就是 common 的很多控件的 key\_map\_list 放在 common\_msgmap\_data.c 中的原因。

### 6.4.3.2 系统消息处理

对于前台应用来说，它收到的所有消息，都是系统消息。US282A 修改了系统消息映射机制，直接在系统消息映射表中映射为回调函数，这样就可以由系统自己调用，用户无需关心其中细节，用户要做的就是编写好模块性很好的回调函数，然后填到系统消息映射表中。

对于主视图，我们支持一个视图有 2 个系统消息映射表的机制，即包括视图私有的系统消息映射表和 COMMON 系统消息映射表。系统消息会优先在视图私有的系统消息映射表进行映射，如果映射到了就无须在 COMMON 系统消息映射表中映射；如果映射不到，就会在 COMMON 系统消息映射表中映射。

另外，我们增加多一个灵活的机制，即在视图私有系统消息映射表中映射并处理之后，如果还希望在 COMMON 系统消息映射表中再映射处理，可以返回 `RESULT_SYS_MSG_REDEAL_BY_COMMON`。

我们还扩展了系统消息映射表结构体，增加了一些额外的选项。

### 6.4.3.3 蓝牙事件处理

我们将 RCP 事件也放到蓝牙事件中，作为蓝牙事件中的一种。

蓝牙事件处理，即蓝牙管理器会循环获取蓝牙协议栈的状态并判断是否发生了某些状态变化，如果是则进行响应处理。

另外，有一些蓝牙事件或者状态变化，我们是通过蓝牙协议栈事件发送给蓝牙管理器的，会在这里接收并分发处理。

## 6.5 应用切换

应用切换指的是前台应用的切换，切换过程中可能会伴随引擎应用和蓝牙协议栈的打开或关闭。

应用切换可以由以下几种方式发起：

- 1) 开机：开机后会根据样机外设条件以及关机前状态、唤醒触发条件等决定跳转到哪个应用。
- 2) 按键：用户按模式切换键或功能快捷进入按键即可切换到指定应用。
- 3) 外设热拔插：用户插入或拔出外设会导致应用切换，比如卡、UHOST、AUX 等。
- 4) APK/APP 切换应用：用户可以通过 APK/APP 切换应用，这种方式最灵活。
- 5) 其他系统事件：比如低电关机、空闲一段时间进入低功耗模式、闹钟时间到及其他定时事件等。

US282A 应用切换的对象是功能，而非应用程序；一个应用程序可以有多个功能，功能可以理解为工

作模式，比如本地播歌应用程序，我们分为卡播歌，U 盘播歌，录音回放，特殊目录播放等功能/工作模式，从一个功能切换到另一个功能，要先退出当前应用程序，再以新的功能/工作模式重新进入该应用程序。

在 `case_type.h` 中定义了方案的 APP ID 和 功能 ID。

应用切换模块中有一张应用切换表，记录了某个 `app_result_e` 要对应切换到哪个应用程序的哪个功能。

切换应用的接口如下：

```
void com_ap_switch_deal(app_result_e app_result)
```

有时候在切换应用之前，我们需要先知道切换到哪个应用，以便做出差异化处理，那么可以调用这个接口：`uint8 com_ap_switch_ask_next_func(app_result_e app_result)`，返回下一个应用的功能 ID。

有了这样的接口，我们可以实现这样的功能：

当用户按下模式切换键时，我们先获取下一个应用的功能 ID，如果与当前的功能 ID 一样，那么说明没有下一个应用可切换，就可以避免退出后又重新进入的较差体验。

应用切换的规则由在 `common_ap_switch.c` 的 `ap_switch_info` 应用切换表决定：

1. 每个切换条目都可以指定进入应用 APP ID、参数、是否杀掉当前引擎应用、进入应用的 RESULT 匹配条件、FUNC ID 等信息。

2. 该表分为 2 部分：

```
13 const ap_switch_info_t ap_switch_info[] =
14 {
15 { APP_ID_BTPLAY, 0, 1, RESULT_BLUETEETH_SOUND, APP_FUNC_BTPLAY },
16 { APP_ID_MUSIC, PARAM_MUSIC_SDCARD, 1, RESULT_MUSIC_CPLAY, APP_FUNC_PLAYCARD },
17 { APP_ID_MUSIC, PARAM_MUSIC_UHOST, 1, RESULT_MUSIC_UPLAY, APP_FUNC_PLAYUHOST },
18 { APP_ID_LINEIN, 0, 1, RESULT_ENTER_LINEIN, APP_FUNC_PLAYLINEIN },
19 { APP_ID_RADIO, 0, 1, RESULT_RADIO, APP_FUNC_RADIO },
20 { APP_ID_USOUND, 0, 1, RESULT_USB_SOUND, APP_FUNC_USOUND },
21
22 /*!以下为快捷键进入的功能选项,它们不在功能循环内,需要返回到上一次的状态*/
23 { APP_ID_CONFIG, 1, 1, RESULT_POWER_OFF, APP_FUNC_CONFIG },
24 { APP_ID_CONFIG, 2, 1, RESULT_LOW_POWER, APP_FUNC_CONFIG },
25 { APP_ID_CONFIG, 3, 1, RESULT_SYSTEM_ENTER_S3, APP_FUNC_CONFIG },
26 { APP_ID_CONFIG, 4, 1, RESULT_IDLE_MODE, APP_FUNC_CONFIG },
27 }
```

红色框部分是可以通过模式切换键循环切换到的前台应用 FUNC ID，其总条目数由 `case\inc\common_func.h` 中的 `MAX_FUNCION_CYCLE` 宏值决定，修改这部分需要同步修改该宏值。

/\*! 循环切换应用数目 \*/

```
#define MAX_FUNCION_CYCLE 6
```

另外，增加循环切换的 FUNC ID，需要修改配置项 `SETTING_APP_SWITCH_SEQUENCE`，添加所添

加的前台应用的功能 FUNC ID 到配置项列表中。

蓝色框部分是指定 RESULT\_XXX 切换进入的前台应用 FUNC ID 列表。

当然，红色框部分的各个条目也适用于指定 RESULT\_XXX 切换进入的规则。

## 6.6 快捷键响应

快捷键映射表是 CASE 默认处理的按键消息集合，它的优先级比应用私有的按键消息映射表低。也就是说，如果用户想把某个快捷键作为其他用途，可以把该快捷键添加到应用私有的按键消息映射表中。

另外，前面说过一种灵活的处理方式，就是应用先在私有的按键消息映射表中处理该消息，然后返回 RESULT\_KEY\_EVENT\_REDEAL\_BY\_COMMON 结果，就可以再次按照默认方式处理多一次。

US282A 方案的快捷键列表如下：

- 1) 音量调节，静音和解除静音，音效调节
- 2) 模式切换，以及各种进入指定应用的快捷键
- 3) 关机快捷键
- 4) 蓝牙相关快捷键：断开连接、开关可见性、回拨电话等

## 6.7 按键消息预处理

用户发生按键动作时，一般都伴随着一些特殊处理，比如退出屏保并恢复显示、按键音等等。我们可以通过在前台应用的按键消息接收接口中挂载一个勾函数进行预处理，这样可以简化按键消息接收接口，优化方案架构。

按键消息预处理规格如下：

- 1) 过滤按键消息：使用场景是这样的，有时候用户按下某个按键，在该按键没有放开时就切换到另一个场景或者另一个 AP 了，这样会在新的场景或 AP 中收到该按键动作后续 LONG、HOLD、SHORT\_UP 消息，但用户又不希望在新场景或 AP 中收到这些消息，希望能够透明过滤掉这些按键消息。所以，我们在预处理函数中维护当前按键消息值，当用户希望过滤当前按键后续消息时只需要调用接口 void com\_filter\_key\_hold(void)，之后预处理函数就可以过滤当前按键消息后续消息了，直到收到 SHORT\_UP 消息。另外，我们也提供一种只过滤掉最后的 SHORT\_UP 消息的机制，所调用的接口变成 void com\_filter\_key\_up(void) 了。
- 2) 解除静音，恢复声音。

- 3) 按键音响应, 只对 KEY\_DOWN 消息响应。
- 4) 将省电关机、屏保等计时清 0, 并恢复屏幕显示。
- 5) 调高频率, 以更高效率处理按键消息。

## 6.8 系统消息响应

系统中大部分的公共事件我们都需要提供一种默认方式处理, 这样应用如果不关心该系统消息, 则可以不用处理, 如果应用关心, 则要将其添加到私有的系统消息映射表中。

另外, 前面说过一种灵活的处理方式, 就是应用先在私有的系统消息映射表中处理该消息, 然后返回 RESULT\_SYS\_MSG\_REDEAL\_BY\_COMMON 结果, 就可以再次按照默认方式处理多一次。

US282A 方案的公共事件列举如下:

- 1) 外部设备热拔插事件: UHOST 热拔插、卡热拔插、AUX 热拔插、USB 热拔插、充电适配线热拔插、耳机热拔插等。
- 2) 蓝牙事件: 蓝牙免提来电、蓝牙连接或断开、蓝牙硬件错误、进入 BTT 测试等。
- 3) 其他系统事件: 闹钟起闹、低电、充电满、进入空闲模式、省电关机、进入低功耗、强制应用退出等。
- 4) APK/APP 切换应用事件。

## 6.9 系统消息预处理

前面我们说到应用可以在私有的系统消息映射表中优先接收并处理掉系统消息, 这就意味着有一些系统消息可能不会执行 COMMON 的默认处理, 而如果在默认处理中放置一些必要的处理, 比如系统状态更新, 可能会导致这些必要的处理被忽略跳过, 从而导致系统某些方面失控了。当然我们也可以在应用的系统消息回调函数中进行这些必要的处理, 只是这样会让每个应用都重复并费心维护系统的运行, 并且可能稍微不小心就忘记了。

为此, 我们参考了按键消息预处理的机制, 为系统消息也增加了预处理, 也就是说, 只要收到系统消息就必定会调用预处理函数, 如果我们把所有必要的处理都放在预处理中做, 那么也就不会导致上面的问题了。

系统消息预处理规格如下:



- 1) 允许在预处理时悄悄修改系统消息，比如将 MSG\_USB\_STICK 修改为 MSG\_ADAPTOR\_IN。
- 2) 维护系统外部设备状态，包括卡拔插、UHOST 拔插、USB 拔插、AUX 拔插等；维护其他各个系统状态。
- 3) 对于一些重要的系统消息，将省电关机、屏保等计时清 0，并恢复屏幕显示。
- 4) 调高频率，以更高效率处理系统消息。

## 6.10 分立 LED 显示

对于音箱和耳机这样的产品形态来说，一般都不会有很丰富的显示功能模块，更多的是通过声音和分立 LED 来实现提示反馈功能。

分立 LED 虽然很容易操作，但是为了让方案各个应用都能一致方便的实现丰富的分立 LED 显示效果，比如闪烁功能、呼吸灯功能等，我们也就在 COMMON 增加了一套丰富的分立 LED 显示接口；这样做还有一个原因是，我们在 COMMON 为分立 LED 分配了全局的数据空间，可以让分立 LED 独立于应用，即使切换应用也不会影响分立 LED 的显示行为。

分立 LED 在物理上支持 2 种点亮方式：PWM 和 GPIOOUT。

PWM 可以用来实现缓慢点亮和缓慢熄灭效果，可以用来实现呼吸灯效果。GPIOOUT 只能是简单的点亮和熄灭。这两种方式都在 KEY 驱动中实现，COMMON 层接口要根据分立 LED 的物理连接接口选择点亮方式。

分立 LED 显示包括以下接口：

- 5) 初始化分立 LED: `void discrete_led_init(void);`

该接口只需要在 ap\_config 应用开机时初始化一次，就像驱动只需安装一次。

- 6) 设置分立 LED: `void discrete_led_set(led_id_e id, led_mode_e mode, led_duty_t *duty_param)`

设置分立 LED 点亮，或熄灭，或启动闪烁，闪烁功能是基于闪烁软定时器实现的。

设置闪烁方法如下，表示亮 500ms，灭 500ms，一直闪烁：

```
pw_link_duty.cycle_count = -1;
pw_link_duty.duty_on_time = 500;
pw_link_duty.duty_off_time = 500;
discrete_led_set(LED_ID_POWER, LED_TWINKLE, &pw_link_duty);
```

方案对分立 LED 的使用状况差异较大，需求修改以下几个文件：



- 1) `common_discrete_led.h` : 定义方案有哪几个分立 LED 等, 物理接口如何分配。
- 2) `common_discrete_led.c` : 对应填好 `discrete_led_cfg` 配置表, 以便初始化时使用。
- 3) 然后就可以使用 `discrete_led_set()` 接口进行显示了。

## 6.11 按键音播放

US282A 的按键音模块是一个独立的音频输出通道, 它最终与另一个音频输出通道在 D/A 时进行 MIX。按键音不再是播放一段单频率方波, 而是能播放任意采样率的自然语音, 格式为 PCM, 但是为了兼容低比特率的本地歌曲播放, 我们将按键音的采样率设置为 8KHz。所以我们现在可以更多使用按键音作为声音反馈手段, 比如提示操作失效, 提升音量已调节到最大, 提示蓝牙连接和断开等。

US282A 将按键音播放实现为前台应用的一个子线程, 优先级也定为仅仅比解码线程低。这样它是独立运行的, 系统不会因按键音播放而阻塞其他用户交互的响应速度。

按键音播放规格如下:

- 1) 支持阻塞模式和非阻塞模式, 对于非阻塞模式, 可以在任何地方等待按键音播放结束。
- 2) 对于非阻塞方式, 可以最多缓冲 4 个按键音, 即按键音播放模块提供了一个深度为 4 的 FIFO。
- 3) 按键音播放线程不需要处理任何事件, 只需要把按键音播放完就行了。

按键音播放相关接口如下:

- 1) 按键音模块开关: `void keytone_set_on_off(bool on_off)`

方案可以做到动态开关按键音, 在不需要的场景下可以关闭。

- 2) 设置按键音采样率: `void keytone_set_dac_sample(uint8 dac_sample)`

该接口只能在初始化时调用, 后面不能再修改。

- 3) 分配按键音输出通道: `void keytone_set_dac_chan(dac_chenel_e dac_chan)`

在方案上, 我们需要让按键音与 TTS 或解码分别占用一个独立的音频输出通道, 所以需要做好冲突管理, 我们需要在 TTS 和解码尚未开始之前, 就分配好音频输出通道, 而 TTS 和解码必须分配另一个通道。

- 4) 播放按键音: `bool keytone_play(uint8 kt_id, uint8 mode)`

支持阻塞模式和非阻塞模式, 非阻塞模式仅仅是将按键音放到 FIFO, 然后调用一下 `keytone_play_deal()` 就返回了; 如果是阻塞方式, 必须等待非阻塞模式的按键音播放完之后才能接着播放, 并播放完之后才返回。

- 5) 查询并启动按键音播放: `void keytone_play_deal(void)`

如果有非阻塞按键音在 FIFO, 就会播放起来。

- 6) 等待非阻塞模式按键音播放完: `void keytone_play_deal_wait(void)`

- 7) “按键音”播放: `void com_start_key_tone(uint8 mode)`

之前说的按键音, 指的是使用按键音通道播放的任意声音, 这里的按键音, 指的是对按键动作做出反馈的“按键音”。

方案对按键音的使用状况差异也是比较大, 需要考虑以下几点:

- 1) `common_keytone.h` 中定义按键音 ID, 代码中使用该 ID 来播放。
- 2) `common_keytone_file.c` 中将以上按键音文件名字按顺序列到 `g_keytone_filelist` 数组中。
- 3) 然后, 就可以使用 `keytone_play()` 接口进行播放了。

另外, 我们可以通过配置项配置按键音播放的某些选项, 比如开关, 是否叠加到播放声音上, 最大按键音声音。

## 6.12 TTS 播放

TTS 与播放是互斥的, 播放 TTS 之前必须停止播放, 卸载解码线程, 播放完之后再恢复播放。

与按键音类似, TTS 在 US282A 也是实现为前台应用的一个子线程, 优先级同解码线程一致。这样它是独立运行的, 系统不会因 TTS 播放而阻塞其他用户交互的响应速度。

TTS 播放规格如下:

- 1) 支持多种传参方式:

`TTS_MODE_ONLYONE`: 仅 1 个词条, 一般都是这种方式

`TTS_MODE_SECLIST`: 段序列, 以 `0xff` 为结束标志, 比如 FM 频点和 music 曲目号

`TTS_MODE_STRING`: 字符串, 比如电话本人名播报

- 2) 支持阻塞模式和非阻塞模式播放, 对于非阻塞模式, 我们就有机会在启动了 TTS 之后做一些与 TTS 没有冲突的事情, 然后再等待 TTS 播放结束, 这样可以使一些事情更好并行着跑起来。但是 TTS 的非阻塞模式“应该”在非阻塞模式 TTS 播放之后“显式”调用 `com_tts_state_play_wait()` 等待 TTS 播放结束, 以进行一些播放环境和资源的销毁; 否则容易出现资源冲突而死机。
- 3) TTS 也支持 FIFO 机制, 但是它与阻塞非阻塞选项是独立的, FIFO 深度为 2。TTS 的 FIFO 机制与

按键音播放在处理上有些差异，TTS 使用 FIFO 模式播放，并不会立即启动播放，而是在 `com_view_loop()` 中调用 `com_tts_state_play_deal()` 时启动。一般使用 FIFO 模式播放，也还是阻塞模式播放。

- 4) TTS 播放时可以像主场景一样，响应各种事件，包括按键消息，系统事件，蓝牙事件，软定时器等，只是遇到一些必须终止 TTS 的事件才将 TTS 终止退出，然后回到主场景继续处理事件；并且 TTS 对所有这些事件的处理规格非常灵活，有很多选项，具体参见头文件 `common_tts.h` 中的说明。

TTS 播放相关接口如下：

- 1) 播放 TTS: `tts_play_ret_e com_tts_state_play(uint16 tts_mode, void *tts_info)`

支持阻塞模式和非阻塞模式，如果是非阻塞模式，那么仅启动播放后就退出了，后面应该调用 `com_tts_state_play_wait()` 等待 TTS 播放结束，以进行一些播放环境和资源的销毁；否则容易出现资源冲突而死机。

支持 FIFO 模式，仅仅将 TTS 播放任务放到 FIFO 中，然后就退出了；这种模式对需要在 COMMON 中的某个接口中播放 TTS 比较有利，可以减少嵌套层次，减轻栈的压力。

- 2) 查询并播放 FIFO 中的 TTS: `tts_play_ret_e com_tts_state_play_deal(void)`

如果用 FIFO 模式播放 TTS，那么需要调用 `com_tts_state_play_deal()` 来启动播放；如果是非阻塞模式 TTS 播放，那么该接口还要进行查询状态，当查询到 TTS 播放完，就会进行一些播放环境和资源的销毁。

- 3) 等待非阻塞 TTS 播放结束: `tts_play_ret_e com_tts_state_play_wait(void)`

- 4) TTS 播放功能控制: `void com_tts_state_play_control(tts_control_opr_e opr)`

支持动态开关 TTS，在一些没有资源来播放 TTS 的场景，应该关闭 TTS 播放。

支持任意地方主动终止正在播放的 TTS，主要用在一些需要快速切换进与 TTS 播放有冲突的场景。

方案对 TTS 使用状况差异也比较大，需要考虑以下几点：

- 1) 添加或删除或修改 TTS 资源文件，如果变动了 TTS 词条的顺序编号，需要同步更新 `common_tts_id.h` 的 TTS ID 表，并最好编译 CASE，使用 `make clean & make`。
- 2) 以下几个地方与 TTS 播放有关联，在方案开发时要根据业务需求谨慎选择：

- 系统事件响应是否需要终止正在播放的 TTS，可以通过系统事件映射表来指定：

```
{ { MSG_UH_IN, SYSMMSG_STOP_TTS_YES }, com_sys_deal_uhost_in },
{ { MSG_UH_OUT, 0 }, com_sys_deal_uhost_out },
```

如果某个系统事件，会发生应用跳转，或者其他很复杂的处理流程，就需要添加该标志。

- RCP 命令是否需要终止正在播放的 TTS，是通过 `bool com_rcp_check_for_tts(rmt_ctrl_pkg_t`

\*p\_rcp) 函数来指定的，特别是对于 RCP\_CMD\_SETTING 命令，需要列举所有不需要终止 TTS 的命令 ID:

```
if ((cmd_id == SETTING_ID_SYSTEM_SETTING)
 || (cmd_id == SETTING_ID_SYSTEM_STATUS)
 || (cmd_id == SETTING_ID_DEVICE_NAME)
 || (cmd_id == SETTING_ID_MUSIC_SETTING))
{
 return FALSE;
}
```

方案中有几个使用 TTS 的典型场景:

- 1) 在刚切换进应用时, 播报“进入 XXX”TTS 时, 我们使用非阻塞模式先播放起来, 然后回到应用主场景, 创建或者杀掉后台蓝牙, 再等待 TTS 播放结束。这样做可以让用户感觉到切换应用速度快, 而又不影响后台蓝牙的启动速度。这也要求应用切换模块不要去创建或者杀掉后台蓝牙。
- 2) 一旦进入阻塞模式的 TTS 播放, 或者非阻塞模式的 TTS 播放之后已经调用了等待播放结束的接口, 将会直到 TTS 播放完之后才能出来执行主场景的业务, 所以, 应用显示相关的操作, 应该在 TTS 播放之前进行, 否则会明显觉得显示延迟了。

## 6.13 远程控制协议 RCP

### 6.13.1 概述

US282A 的样机, 除了通过按键和 LED 数码管等“本地”的用户交互方式进行交互外, 还可以通过“远程”的用户交互方式进行交互, 主要有以下两种手段:

- 1) 手机使用 APK/APP, 通过蓝牙数传通道充当样机的键盘和显示器, 极大丰富了用户交互方式。
- 2) 不带蓝牙功能的传统主机需要添加蓝牙功能, 通过串行接口 UART、TWI 等控制充当从设备的样机。

我们为以上两种手段设计并实现了一种远程控制协议 RCP (Remote Control Protocol), 并且深度整合到我们的方案中。

US282A 的 RCP 功能规格如下:

- 1) 支持 SPP 和 BLE 这两种蓝牙数传协议。
- 2) 支持 UART、TWI、SPI 等串行通信接口。
- 3) 支持双向数据交互, 但交互都是手机/主机发起的, 样机不会主动发起数据交互。

- 4) 只要样机正常工作起来并且与手机/主机建立起连接，手机/主机就能够完全控制样机，完成任何用户交互任务。
- 5) **支持组包功能**：即命令包长度超过物理数据包大小时会拆分为多个物理数据包发送过来，就需要组包为一个完整命令包，才能分发给应用层进行业务处理。
- 6) **支持拆包功能**：即手机/主机将多个同时发送过来的小命令包合并为一个物理数据包发送过来，就需要将这个物理数据包拆分为多个小命令包逐个分发给应用层进行业务处理，才不会丢失命令包。
- 7) 支持多种可选的命令包完整性保证方式，比如校验和、CRC8 等；当完整性校验失败，可以请求手机/主机重发。
- 8) 使用事件机制对 RCP 进行封装和响应，这样应用层运用非常简单，可读性良好。

## 6.13.2 命令包格式

RCP 命令包格式如下：

|                   | bit0                                 | bit7 | bit8   | bit15         | bit16                                             |            | bit23          | bit24          | bit31                            |
|-------------------|--------------------------------------|------|--------|---------------|---------------------------------------------------|------------|----------------|----------------|----------------------------------|
| byte 0 - byte 3   | MAGIC                                |      |        |               | FLAGS                                             |            |                |                |                                  |
|                   | 0x01                                 | 0xfe |        | CHAN<br>[0-1] | SEG<br>[2-3]                                      | ACK<br>[4] | Reserve<br>[5] | CHECK<br>[6-7] | SEQ<br>[0-3]<br>Reserve<br>[4-7] |
| byte 4 - byte 7   | CMD TYPE                             |      | CMD ID |               | CMD LENGTH<br>( = 16Bytes + Other Params N Bytes) |            |                |                |                                  |
| byte 8 - byte 11  | PARAM 1<br>(Must)                    |      |        |               |                                                   |            |                |                |                                  |
| byte 12 - byte 15 | PARAM 2<br>(Must)                    |      |        |               |                                                   |            |                |                |                                  |
|                   | OTHER PARAMs (0 ~ N)Byte<br>(Option) |      |        |               |                                                   |            |                |                |                                  |

说明：

- 1) MAGIC：第一个字节 0x01 即 SOH，第二个字节对其按位取反。
- 2) CHAN：通道模式，FLAGS 第一个字节的 bit[0-1]，0h00 表示 SPP 通道，0h01 表示 TWI 或 UART 通道，其他未定义。
- 3) SEG：分段标志，FLAGS 第一个字节的 bit[2-3]，0h00 表示不用分段，0h01 表示第一段，0h10 表示后续段但非最后一段，0h11 表示最后一段。分段机制需要 ACK 和 SEQ 配合实现。
- 4) ACK：ACK 标志，FLAGS 第一个字节的 bit[4]，0h0 表示不需要 ACK，0h1 表示需要 ACK。可以使用该标志置位来确保命令被分发处理，如果收不到 ACK，则对方可以重发命令。
- 5) CHECK：校验方式，FLAGS 第一个字节的 bit[6-7]，0h00 表示没有校验，0h01 表示校验和，其他保留。如果使用校验和，必须在每个命令/分段后添加 4 个字节的校验和，它是可变参数数据的所

有字节相加总和，如果没有可变参数数据，那么就没有这 4 个字节的校验和。如果添加了校验和，那么命令包的长度也要相应加 4 个字节。

- 6) SEQ: 分段传输 SEQ 序号，FLAGS 第二个字节的 bit[0-3]，从 0 到 15 循环更新，第一个段为 0。该标志用于分段重传机制。
- 7) FLAGS 其他 BIT 保留。
- 8) CMD TYPE: 包括以下几类命令：

| 命令类型      | 作用                  | 备注                                                                 |
|-----------|---------------------|--------------------------------------------------------------------|
| 应用切换类 ‘P’ | 用于应用切换              | 这里的应用不同于 AP 这种概念的应用，准确说应该是功能切换。比如插卡播歌和 U 盘播歌是同一个 MUSIC AP 中的两个不同功能 |
| 控制类 ‘C’   | 用于简单动作控制            | 这两类命令，都是通过转发按键消息来发成目的，区别在于控制类定义使用了很多特殊的虚拟按键                        |
| 模拟按键类 ‘K’ | 用于模拟按键控制            |                                                                    |
| 查询类 ‘Q’   | 用于获取状态信息            | 获取状态信息是异步的，上位机发送查询命令，下位机收到命令分发处理时异步应答                              |
| 应答类 ‘A’   | 用于应答查询类的状态信息请求      |                                                                    |
| ACK 类 ‘R’ | 用于简单应答以确<br>保命令处理与否 | 用于非查询类命令的应答                                                        |
| 设置类 ‘S’   | 用于复杂动作控制<br>或设置     | 除非带 ACK，否则不需要应答                                                    |

- 9) CMD ID: 命令号，各种命令类型都有不同的命令号空间，详细见下面命令总表。
- 10) CMD LENGTH: 命令长度，统计范围为整条命令，即 16 字节的固定命令头部再加上不定长参数。
- 11) PARAM 1: 命令第一个参数，各个命令都有不同的意义，详细见下面命令总表。
- 12) PARAM 2: 命令第二个参数，各个命令都有不同的意义，详细见下面命令总表。
- 13) OTHER PARAMs: 命令不定长参数，可带 0~N 字节数据，各个命令都有不同的意义。

### 6.13.3 RCP 事件映射机制

RCP 的事件映射机制设计和实现如下：

- 1) RCP 同样广泛使用事件机制进行映射处理：应用层实现 RCP 相关功能时，只需要填好一张映射表，并实现回调函数，然后注册到 RCP 事件映射分发模块即可。下面以 AP MUSIC 应用为例：



```
/* COMMON set/get/control类命令回调函数注册表 */
const rcp_cmd_cb_t music_rcp_cmd_cb_tbl[] =
{
 { RCP_CMD_SETTING, SETTING_ID_MUSIC_SETTING, (void *) music_set_music_setting },
 { RCP_CMD_SETTING, SETTING_ID_SONG_SEQ, (void *) music_set_song_seq },
 { RCP_CMD_SETTING, SETTING_ID_MUSIC_PLIST, (void *) music_set_music_plist },
 { RCP_CMD_SETTING, SETTING_ID_MUSIC_DISK, (void *) music_set_music_disk },

 { RCP_CMD_QUERY, QUERY_ID_GLOBAL_RCP_INFO, (void *) music_get_global_rcp_info },
 { RCP_CMD_QUERY, QUERY_ID_MUSIC_INFO, (void *) music_get_music_info },
 { RCP_CMD_QUERY, QUERY_ID_MUSIC_LYRIC, (void *) music_get_music_lyric },
 { RCP_CMD_QUERY, QUERY_ID_MUSIC_PLIST, (void *) music_get_music_plist },

 { RCP_CMD_CONTROL, CONTROL_KEY_PLAY, (void *) music_apk_key_play },
 { RCP_CMD_CONTROL, CONTROL_KEY_PAUSE, (void *) music_apk_key_pause },
 { RCP_CMD_CONTROL, CONTROL_KEY_PREV, (void *) music_apk_key_prev },
 { RCP_CMD_CONTROL, CONTROL_KEY_NEXT, (void *) music_apk_key_next },

 /* 结束项 */
 { 0, 0, NULL },
};
```

```
com_rcp_set_callback(music_rcp_cmd_cb_tbl, music_get_global_rcp_info);
```

- 2) 对于应用切换类命令，不需要映射为回调函数，而是转换为系统事件，由系统事件映射来执行应用切换。

```
/* 应用切换类命令，映射为应用切换消息 */
const msg_apps_type_e rcp_cmd_apswitch_event[] =
{
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 EVENT_ENTER_BTPLAY, /* 10 蓝牙推歌应用 */
 EVENT_ENTER_MUSIC, /* 11 music应用 */
 EVENT_ENTER_LINEIN, /* 12 linein应用 */
 EVENT_ENTER_RADIO, /* 13 fm应用 */
 EVENT_ENTER_ALARM_RCP, /* 14 闹钟应用 */
 EVENT_ENTER_MUSIC_CARD, /* 15 插卡播歌 */
 EVENT_ENTER_MUSIC_UDISK, /* 16 U盘播歌 */
 EVENT_RECORD_CPLAY_RCP, /* 17 卡录音回放 */
 EVENT_RECORD_UPLAY_RCP, /* 18 U盘录音回放 */
 EVENT_ENTER_CRECORD_RCP, /* 19 卡录音 */
 EVENT_ENTER_URECORD_RCP, /* 20 U盘录音 */
 EVENT_ENTER_USOUND, /* 21 USB音箱 */
};
```

- 3) 对于一些公共的 RCP 功能，COMMON 有默认的映射表；类似于按键消息和系统事件的处理规则，AP 可以在私有的 RCP 映射表中优先映射处理它所关心的公共命令。其中获取全局状态这个公共命令，一般应用都需要自己实现，添加一些应用私有的全局状态。

```

/* COMMON set/get/control类命令回调函数注册表 */
const rcp_cmd_cb_t com_rcp_cmd_cb_tbl[] =
{
 { RCP_CMD_SETTING, SETTING_ID_SYSTEM_SETTING, (void *) com_set_system_setting }, //系统设置
 { RCP_CMD_SETTING, SETTING_ID_SYSTEM_STATUS, (void *) com_set_system_status }, //APK状态
 { RCP_CMD_SETTING, SETTING_ID_SYSTEM_DIALOG, (void *) com_set_system_dialog }, //对话框提示
 { RCP_CMD_SETTING, SETTING_ID_DEVICE_NAME, (void *) com_set_device_name }, //设置设备名称
 { RCP_CMD_SETTING, SETTING_ID_ALARM_ADD, (void *) time_alarm_set_alarm_add }, //添加或更新闹钟
 { RCP_CMD_SETTING, SETTING_ID_ALARM_DELETE, (void *) time_alarm_set_alarm_delete }, //删除闹钟
 { RCP_CMD_SETTING, SETTING_ID_ALARM_PARAM, (void *) time_alarm_set_alarm_param }, //设置闹钟参数

 { RCP_CMD_QUERY, QUERY_ID_GLOBAL_RCP_INFO, (void *) com_get_global_rcp_info }, //获取全局状态
 { RCP_CMD_QUERY, QUERY_ID_SUPPORT_FEATURE, (void *) com_get_support_feature }, //获取支持特性列表
 { RCP_CMD_QUERY, QUERY_ID_BT_DEVICE_INFO, (void *) com_get_bt_device_info }, //获取设备信息，包括名称等
 { RCP_CMD_QUERY, QUERY_ID_ALARM_LIST, (void *) time_alarm_get_alarm_list }, //获取闹钟列表
 { RCP_CMD_QUERY, QUERY_ID_RING_LIST, (void *) time_alarm_get_ring_list }, //获取内置闹铃列表
 { RCP_CMD_QUERY, QUERY_ID_RING_FOLDER, (void *) time_alarm_get_ring_folder }, //获取本地磁盘闹铃目录列表
 { RCP_CMD_QUERY, QUERY_ID_ALARM_PARAM, (void *) time_alarm_get_alarm_param }, //获取闹钟参数

 { RCP_CMD_CONTROL, CONTROL_KEY_MUTE, (void *) com_ctrl_switch_mute }, //静音开关

 /* 结束项 */
 { 0, 0, NULL },
};

```

## 6.13.4 外部接口设计

RCP 相关的外部接口包括以下几个：

- 1) 应用 RCP 处理环境初始化：void **com\_rcp\_init**(void);

该接口在切换应用时调用，会将 RCP 命令包分析状态机 reset，这样会导致正在分析的长命令包丢失。这是可以解决的问题，需要优化。

- 2) 设置应用私有 RCP 映射表：void **com\_rcp\_set\_callback**(const rcp\_cmd\_cb\_t \* app\_rcp\_cmd\_cb\_tbl, rcp\_get\_func app\_get\_global\_info\_cb);

该接口在应用初始化时设置映射表，在应用退出时清除设置；初始化设置时必须在 **com\_rcp\_init()** 之后调用。

- 3) RCP 命令分发执行入口：app\_result\_e **com\_rcp\_dispatch**(rmt\_ctrl\_pkg\_t \*p\_rcp);

分发规则是这样的：如果是合法命令包，调用 **com\_rcp\_cmd\_dispatch()** 先在应用私有的 RCP 映射表中映射，如果映射不到才在 COMMON 的 RCP 映射表中映射，还映射不到就当作未定义命令包直接丢弃；如果是非法命令包，则调用 **com\_rcp\_dispatch\_invalid()** 进行处理，这样就允许方案自定义命令格式。

对于 SPP 和 BLE 这种通信方式，我们在 BT STACK 应用中将 RCP 以蓝牙事件的方式告诉应用层，然后在 **com\_btmanager\_loop()** 调用蓝牙事件分发 **com\_btmanager\_event\_dispatch()** 中调用该入口。对于 UART 等主从机通信方式，我们需要增加另一套分发接口，入口调用方式也会更改。

- 4) TTS 播放中检查 RCP 包是否需要终止 TTS 播放：bool **com\_rcp\_check\_for\_tts**(rmt\_ctrl\_pkg\_t \*p\_rcp);  
该接口仅在 TTS 播放场景下调用。



## 6.13.5 自定义协议

如果客户不想用我们的 RCP 协议，希望自定义协议，我们建议这样做：

1. 优先考虑复用我们的 COMMON RCP 处理机制，即组包、拆包、事件映射机制，这要求所定义的命令包结构必须与我们的 RCP 命令包结构类似，比如有命令长度，可以在不关心命令类型和 ID 时进行组包、拆包处理，必须有很容易描述的命令类型和 ID，以便进行事件分发映射。

客户必须修改一系列 RCP 相关的接口。

2. 如果客户希望彻底更换协议，或者是某种第三方协议没办法做到上述那样，那么就在 `void com_rcp_dispatch_invalid(uint8 *rcp_data, uint16 rcp_len)`；接口中自行实现协议包的解析和分发处理。如果要回复命令包，自己组好包后，调用 `int com_btmanager_sppble_send(void *data, uint16 data_len)`；进行发送即可。

## 6.13.6 分段机制实现说明（该部分尚未完整实现）

APK 要下载大批量数据（其实只要有效数据大小超过堆空间允许的最大有效数据长度，就称之为大批量数据，比如标案只有 384 字节的堆空间，那么只要有效数据大小超过 200 字节就认为是大批量数据了。）到蓝牙音箱，APK 的通信层（由 APK 自行实现的，不是蓝牙底层通信）会自动将数据分段传输，每一个段都自动添加 SEQ，并且自动添加 ACK 标志，蓝牙音箱收到数据之后会发送 ACK 应答，并且应答包需要将 FLAGS 的 SEQ 标志也照样回复给 APK/APP，这样确保每一段都已经正确接收。

APK 通信层这种机制需要考虑丢包重传，只要蓝牙音箱超过 1000 MS 没有 ACK，那么 APK 就会重传，并且如果连续 3 次没有 ACK，就返回失败给 APK UI 层。

另外，为了加强大批量数据传输的安全性，我们还增加了校验机制。即分段数据包需要采用校验和机制，对所传输的数据进行校验和计算，然后在小机端进行验证，如果验证不通过，那么就说明有（物理）数据包丢失，就会返回 NACK 给 APK/APP。

APK 已经提供了这样的命令模式，只要指定该命令需要 ACK，那么不管该命令有效数据是否超过最大有效数据长度，APK 通信层都保证数据完整发送给蓝牙音箱。APK UI 层可以一次性将所有数据交给 APK 通信层去发送，也可以每次只传输一部分，这样便于在 APK UI 层更新数据传输进度。

## 6.13.7 用户交互说明

在 APK/APP 的用户交互方面上，有以下几点要说明：

- 1) APK/APP 有操作时，可以通过音箱播放 KEYTONE 来响应，表示收到 APK/APP 的命令了。

- 2) APK/APP 因为有很大的显示器，通过显示足以向用户提示各种复杂的状态，所以 APK/APP 有操作引起音箱状态发生变化，除了切换应用等重大变化之外，其他细微变化比如 FM 收音机切换频点等是不用 TTS 来提示的。

## 6.13.8 注意事项

方案对 RCP 的实现需要注意以下几点：

- 1) 应用层要将全局状态查询命令响应函数放在 Rcode 或者主循环运行时不会切换 BANK 的 Control BANK 中，保证正常运行时不会发生 BANK 切换，这样有利于 EMI 和 ESD。
- 2) RCP 事件映射表必须放在 Rcode 或者 data 段中，否则会出错。
- 3) APK/APP 发送给蓝牙音箱的命令及其参数大小要求是 4 字节对齐，这样可以保证不会出现数据非对齐访问异常。这一点实际上已经在 APK/APP 的通信层做了保证，即如果不是 4 字节对齐的，通信层会补齐到 4 字节对齐。
- 4) RCP 逻辑命令包是有大小限制的，它受限于我们系统为 RCP 长命令缓冲分配的缓冲区大小，即 common\_rcp.h 中 RCP\_PKG\_MAX\_PAYLOAD 定义的最大有效负载，命令头 16 个字节不计算在内。所以，如果有某些场景确实需要下传超过 RCP\_PKG\_MAX\_PAYLOAD 的数据，那么请使用分段机制，或者自己在应用层拆分为多个小数据包。

## 6.14 时间日历闹钟

US282A 在 COMMON 中实现了时间显示、日历显示及设置、闹钟显示及管理公共功能模块，这几个模块都是封装为子视图，是非常独立的小模块。

### 6.14.1 时间显示

时间显示规格如下：

- 1) 显示小时和分钟，也可以显示秒；显示：号以示时间在走。
- 2) 每 0.5 秒钟更新一次。
- 3) 如果用户不操作，8 秒钟超时返回主视图。

## 6.14.2 日历显示及设置

日历显示及设置规格如下：

- 1) 显示及设置年份、月份、日期、小时和分钟。
- 2) 当前正在显示和设置的内容会每 0.5 秒闪烁一次。
- 3) 如果用户不操作，8 秒钟超时会取消本次设置并返回主视图。

## 6.14.3 闹钟显示及管理

日历显示及管理规格如下：

- 1) 显示及设置小时和分钟，以及 ON/OFF。
- 2) 当前正在显示和设置的内容会每 0.5 秒闪烁一次。
- 3) 如果用户不操作，8 秒钟超时会取消本次设置并返回主视图。
- 4) 除非增加其他设置项，否则会默认一些闹钟属性为：循环方式为每天，闹铃为第一个内置闹铃。
- 5) 闹铃支持内置闹铃、特殊目录闹铃、卡任意歌曲、U 盘任意歌曲等。

另外，为了更灵活定制闹钟的行为，我们还增加了以下几个参数，可以通过配置项修改默认值：

- 1) 闹钟贪睡时间：即用户选择贪睡后，再过多长时间闹钟再次起闹；默认值为 9 分钟，设置为 0 表示不支持贪睡。（9 分钟贪睡时间跟 iPhone 手机的贪睡时间一致，网上说从 1956 年一直沿用的贪睡时间，甚至从生物钟上来解释这个值。）
- 2) 闹钟贪睡次数：即用户最多可以连续贪睡的次数，次数达到后就将闹钟关闭；默认是 3 次，设置 0 表示无限次。
- 3) 闹钟超时时间：即闹钟起闹后多长时间不操作就自动关闭，默认为 5 分钟，设置为 0 表示一直起闹。

设置好闹钟，只要有一个闹钟开启，那么就会设置闹钟时间到系统时间闹钟硬件模块中，等待闹钟定时到起闹；闹钟时间刚开始是整分钟的，但是选择贪睡时就会精确到秒的设置闹钟时间。

闹钟定时到时，系统会发送系统消息 MSG\_RTCALARM 给应用层，应用层收到该消息后，就根据闹钟记录属性判断是否真的需要在本次起闹，如果是就跳转到 AP ALARM 起闹响应，然后闹钟贪睡或者关闭，就更新闹钟记录并更新下次闹钟时间。如果不需要在本次起闹，那么直接更新闹钟记录并更新下次闹钟时间即可。

我们要求闹钟起闹结束后返回原来状态，如果是关机时闹钟起闹，那么闹钟起闹结束后需要关机。

## 6.15 供电方式与运行模式

一个蓝牙音箱可能同时支持多种供电方式，比如电池供电、USB 线供电、USB 充电适配线供电、专用充电适配线供电等。但是并不一定所有供电方式都能让音箱正常工作起来，比如如果外部功放的工作电压是 7.4V，那么直接使用 USB 线和 USB 充电适配线供电就不能驱动外部功放工作，也就不能发声，但是却可以进行一些不需要发声的事情，比如 U 盘模式。

另外，一个蓝牙音箱可能会设计一个 ON/OFF 拨动式电源开关，并且将该开关理解为音箱正常工作与否的开关，如果将其拨到 OFF 状态，那么不管供电上是否能够让音箱正常工作，都不要进入正常工作的模式。

所以，对于类似上述情况的蓝牙音箱，一种合理的设计方案如下：

- 1) 系统定义了 3 种工作模式：U 盘模式、空闲模式、正常工作模式。
  - U 盘模式：在断电状态下插入 USB 线开机，则会进入 U 盘模式，外部功放关闭。该模式下只能进行 U 盘操作，如果可以充电，会开启充电。
  - 空闲模式：在 OFF 状态下，且不在 U 盘模式，供电方式不限，外部功放关闭，如果可以充电，会开启充电。
  - 正常工作模式：在 ON 状态下，外部功放打开。
- 2) 在 U 盘模式下，将开关拨到 ON 状态，会退出 U 盘模式，跳到正常工作模式。
- 3) 在正常工作模式下，将开关拨到 OFF 状态，如果还没有断电，跳到空闲模式，即回到 AP CONFIG 应用的空闲场景，检测开关是否拨到 ON 状态等。
- 4) 在空闲模式下，如果检测到开关拨到 ON 状态，那么就会退出空闲模式，跳到正常工作模式。

## 6.16 电池供电充电电量测量

上一节说过，供电方式有这几种：电池供电、USB 线供电、USB 充电适配线供电、专用充电适配线供电。而对于电池供电，主要考虑外部功放工作电压以及样机最大输出功率/电流，可能会用多个电池串联增大电压或并联增大电流等组合方式。

ATS282X 集成了内部充电电路，充电电压为 4.2V 左右，充电电流最大为 600mA。如果电池或电池组合的容量太大，选择内部充电电路进行充电效率太低，所以可能会添加外部充电芯片提供更大充电电流。

另外，使用外部充电芯片的方案，需要增加充电检测和充电满检测口，用来识别充电和充电满。

而对于电量检测, AT282X 提供了两种接口, BATADC 和 TEMPADC, 前者测量给 IC 供电的 BAT PIN, 如果电池电压是 7.4V, 那么一般是通过 LDO/DCDC 降压给 IC 供电的, 测量到的电压并不能推算出真实的电池电压; 而 TEMPADC 是一个普通 PIN, 我们可以将电池正极经过外围分压电路分压到符合 TEMPADC 量化范围, 然后接到 TEMPADC, 可以准确的测量 7.4V 电池电压。US282A 按键驱动中已支持这两种电量检测方式了, 对应用层来说是透明的。

BATADC 采样率默认为 250Hz, 启动 BATDAC 后要 8ms 后才能读到稳定的电压值; BATADC 的量化范围为 2.8V ~ 4.4V, 电压值与采样值的对应关系已经反映在 bat\_adc\_def.h 中。

TEMPADC 采样率默认为 250Hz, 启动 TEMPADC 后要 8ms 后才能读到稳定的电压值; TEMPADC 的量化范围为 0.7V ~ 2.2V, 电压值与采样值的对应关系已经反映在 temp\_adc\_def.h 中。

电池电量测量的结果是: 0 ~ 10, 0 表示低电, 电量少于 10%, 1 表示电量多于 10%, 小于 20%, 10 表示充满电, 在应用程会转换为 9, 至于具体电压对应哪个结果, 由 key\_bat\_charge\_comm.c 的 battery\_grade\_vel 表定义。

另外, 我们上报给 APK/APP 的电池电量值的范围是: 0 ~ 5, 0 表示低电, 1 表示 0 格电, 5 表示 4 格点, 它与电池电量测量值的对应关系是:

8 和 9 → 5

6 和 7 → 4

4 和 5 → 3

2 和 3 → 2

1 → 1

0 → 0

最后, iPhone 手机与音箱连接时, 还会上报音箱的电池电量给手机, 上报的电量值也是 0 ~ 9。

说到电池供电, 有一点需要注意: 当输出一个大电流时, 输出电压会明显下降。也就是说, 在电池电压已经比较低的情况下, 输出功率很高, 可能会把样机拉死了。

比如, 对于一个普通锂电池, 输出电流与输出电压的关系如下:

1. 电压为 3.89V 锂电池负载能力:

2. 电压为 4.25V 锂电池负载能力:

| 电压 (V) | 电流 (A) |  | 电压 (V) | 电流 (A) |
|--------|--------|--|--------|--------|
| 3.89   | 0.00   |  | 4.25   | 0.00   |

|      |      |  |      |      |
|------|------|--|------|------|
| 3.72 | 1.00 |  | 4.09 | 1.00 |
| 3.55 | 2.00 |  | 3.93 | 2.00 |
| 3.37 | 3.00 |  | 3.76 | 3.00 |
| 3.18 | 4.00 |  | 3.60 | 4.00 |
| 3.00 | 5.00 |  | 3.40 | 5.00 |

结论:

1. 当电池电量为 3.89V 时, 电池电量下降 5%时, 最大电流输出 1A 左右; 电池电量下降 10%时, 最大电流输出 2A 左右。
2. 当电池电量为 4.25V 时, 电池电量下降 5%时, 最大电流输出 1.3A 左右; 电池电量下降 10%时, 最大电流输出 2.4A 左右。

所以, 当由电池供电时, 如果检测到电池电压低于某个水平, 我们会限制输出功率, 即把音量调低, 以保证输出电流不会太大, 不会将输出电压拉到音箱不能正常工作而死机。

```
void com_update_volume_limit(int8 vol_limit)
```

我们在 COMMON 会周期地进行电量检测, 检测规则如下:

- 1) 检测周期为 500ms, 挂载在 sys\_counter 定时器中执行。
- 2) 电量检测结果, 如果由电池供电, 那么结果只能降低, 不能增加。
- 3) 检测到电量低:
- 4) 检测到充电满:

## 6.17 低功耗模式

当样机由电池供电时且处于空闲状态并维持了一段时间后, 我们可以也应该进入低功耗模式, 有效避免电池电量的浪费。

首先要明确什么是空闲状态, 这个定义具体方案可以不同, US282A 默认是这样的:

- 处于非播放状态下, DSP 已经不再工作了。
- 前台处于空闲状态, 且已经 7 秒钟没有按键或系统事件发生, 前台线程的频率调为很低。
- 蓝牙管理器处于 IDLE 状态。

说明: 前台用户交互状态

```
/*! 前台用户交互状态 */
```

```
typedef enum
```

```
{
 UI_STATUS_IDLE = 0, //空闲状态持续7秒钟后就调低频率
 UI_STATUS_IDLE_LOWFREQ = 1, //空闲且已调低频率状态
 UI_STATUS_BUSY = 2, //表示正在进行初始化, 切歌等短暂需要加速的事情
} ui_status_e;
```

以上状态的变化伴随着调节频率:

UI\_STATUS\_IDLE 调整为 20MHz, 该状态的 IDLE 是相对于 BUSY 而言的, 实际上并非没事干, 只是说只需要进行一些普通的用户交互处理, 我们将频率提升到 20MHz, 正是为了能快速响应短暂的用户交互。

UI\_STATUS\_IDLE\_LOWFREQ 调整为 2MHz, 该状态才是真正的前台没事可干, 频率非常低。

UI\_STATUS\_BUSY 调整为 85MHz, 以最快的频率运行, 加速初始化、切歌等让用户会感觉慢的地方。

另外, 如果我们还可以区分蓝牙等待连接, 这样空闲状态就分为两类:

- 空闲等待状态 SYS\_STATUS\_IDLE
- 空闲不等待状态 SYS\_STATUS\_WAIT\_LINK

说明: 系统工作状态

```
/*! 系统工作状态: 根据各种状态总结出来, 只在 sys_counter 中使用 */
```

```
typedef enum
```

```
{
 SYS_STATUS_IDLE = 0,
 SYS_STATUS_WAIT_LINK = 1,
 SYS_STATUS_WORKING = 2,
} sys_status_e;
```

进入空闲状态后, 开始计时, 计时到 1 分钟等时间段后, 就触发进入低功耗模式。

低功耗模式有 3 种, 区别在于功耗和唤醒手段:

- S2: 可以跑低频和高频, 以及低频和高频切换, 功耗较高, 但支持各种唤醒手段, 包括 DC5V 检测、机械按键、红外遥控器、触摸按键、各种方式的设备热拔插、蓝牙唤醒、闹钟唤醒等。
- S3BT: MCU 几乎断电, 功耗非常低, 支持的唤醒手段也受限了, 只支持 DC5V 检测、机械按键、蓝牙唤醒、闹钟唤醒等。
- S4: MCU 和 BT 控制器都断电了, 功耗极低, 但是不支持蓝牙唤醒, 只能支持 DC5V 检测、机械按键、闹钟唤醒等。

所以, 低功耗模式的选择, 完全是折中考虑功耗和唤醒手段。



另外，进入 S2 低功耗模式后，我们也会一直检测是否电池电量低，检测到低电会关机；也会检测是否有专用充电适配线插入，如果有也会退出低功耗模式。

进入 S2 低功耗模式，所有软定时器都会停下来，我们理解为失效了；然后在退出 S2 低功耗模式时，将所有软定时器重启。

另外，进入 S3BT 低功耗后，如果一直没唤醒，电池电压降低到 3.0V（可以设置为 3.3V），IC 会自动掉电关进，进入 S4 模式。

## 6.18 蓝牙管理器

请参见下一章 [蓝牙框架详解](#)。

## 6.19 PA 与外部功放

US282A 将外部功放的使用也封装为一套接口，将功放类型差异屏蔽掉，包括以下接口：

- 1) 上电使能外部功放：void SPEAKER\_CONTROL\_ENABLE(void);
- 2) 断电外部功放：void SPEAKER\_CONTROL\_DISABLE(void);
- 3) 打开外部功放：void SPEAKER\_ON(void);
- 4) 关闭外部功放：void SPEAKER\_OFF(void);
- 5) 外部功放是否打开：bool IS\_SPEAKER\_ON(void);
- 6) 模拟外部功放 D/AB 类切换：void SPEAKER\_D\_AB\_SWITCH(uint8 type);

当要降低外部功放的电磁干扰时，切换为 AB 类功放，并使用 LDO 降压供电；否则就用 D 类功放，并使用 DCDC 降压供电，这样整体供电效率更好。

另外，带耳机的音箱方案，拔插耳机会伴随着输出通道的切换，由以下接口实现：

void SPEAKER\_OUT\_SWITCH(uint8 hp\_in); 即当耳机插入时，关闭外部功放并断电关闭外部功放；当耳机拔出时，上电使能外部功能并打开外部功能。

如果是外部功放是模拟的，那么需要使能 IC 内部 PA，其上电初始化为了保证没有杂音，需要做 antipop 流程处理，时间需要 1 秒略多一点。为了这个过程体验较好，我们需要在 antipop 过程中并行做其他开机流程，这个环节实现如下：

1. 在 ap\_manager 中创建一个子线程来调用 enable\_pa() 上电初始化，通过 antipop 来消除开机 pa 声；



enable\_pa() 完成之后，会置一个全局标志表示已初始化好。ap\_manager 主线程接着运行，创建 ap\_config 应用继续开机。

2. ap\_config 在开机流程中，如果需要播放声音，就会死等 enable\_pa() 初始化完成，然后播放声音。

关机时，在 disable\_pa() 时也要进行 antipop，时间也要 1 秒多，这是在 ap\_config 应用做的，没有创建子线程，直接在主线程中串行运行，所以关机会感觉慢一点。如果用户操作关机然后马上开机，会明显感觉响应缓慢。

当然，如果外部功放是数字功放，那么就没有这么麻烦，开机时间会缩短一些。

## 6.20 音量调节

US282A 对音量调节进行了简化，集中到 COMMON 统一调节，也就是说只有前台应用才能够调节音量，避免了多个模块调音量而导致混乱。

音量调节的规格如下：

- 1) 当 AUX 和 FM 使用 AA 通路方式播放，只能通过 PA 来调节音量。这种方式几乎不用。
- 2) 当 AUX 和 FM 使用 ADDA 通路方式播放，要通过 DAC 来调节音量，此时将 PA 固定设置为 0db，DAC 调节范围应该是小于等于 0db，否则可能会破音失真。
- 3) 其他纯数字通道，比如蓝牙推歌、蓝牙免提、本地播歌、USB 音箱、TTS、KEYTONE 等，都是通过 DAC 来调节音量。
- 4) 支持 DAC 和 I2S 两种输出/外部功放。IC 支持 SPDIF 输出，但标案暂不支持。
- 5) 支持多音量模式，我们现在有 2 种音量模式，主音量模式和打电话音量模式。各音量模式调节音量是独立的，独立维护一个音量值。
- 6) 支持音量最大级数可配。该规格点尚未完全做好。
- 7) 支持开启静音/解除静音，静音是通过将音量临时改为 0 来实现的。开机默认为解除静音，切换应用也会解除静音，甚至任意按键，APK/APP 控制等，都可以解除静音，具体可以查看源代码。
- 8) 支持 AVRCP/HFP 音量同步。
- 9) TTS 会临时调节音量，即如果音量太小，那么 TTS 播放会听不到，所以就临时将音量提升到刚好能听到的音量，TTS 播放完成后再恢复回去。
- 10) KEYTONE 不会临时调节音量。试想一种情况，如果 KEYTONE 可以临时调节音量，原本在播歌，音量为 0，这时候发生 KEYTONE，临时将音量提高，那么播歌就会突然能听到声音了，体验不好。

不过，我们还有一些很复杂的手段可以让这种问题不会出现，但是太复杂了，我们觉得没必要引进来，这样的规格是可以接受的。

- 11) KEYSTONE 调节音量还有一种方式，就是直接调整 KEYSTONE 信号，将信号进行衰减，就等同于单方面的将按键音音量调小。
- 12) 开机音量有 2 种处理方式，一种是将音量恢复为默认音量，一种是对音量进行限制，如果小于设置的最小值，那么就改为最小值，如果大于设置的最大值，那么就改为最大值。这是可以通过配置项来实现的。

音量调节涉及音量映射表，US282A COMMON 有 3 种音量表，在 common\_phy\_volume\_table.c 中定义：

- 1) AA 通道 PA+DAC 音量表：以 PA 音量为粗调，DA 音量为微调。

```
//IC 内部 PA DB值，最高为0db，单位为0.1dB
const int16 g_ana_pa_db[ANA_PA_VOLUME_MAX] =
{
 -600, -600, -579, -552, -530, -512, -494, -476, -457, -437, -416, -394, -373, -353,
 -327, -312, -297, -281, -265, -250, -235, -221, -205, -190, -174, -159, -143,
 -128, -118, -109, -99, -89, -79, -69, -59, -49, -39, -29, -20, -10, 0,
};

//音量值 -> pa vol + da vol, step by 0.375db * N, or 1db, 1.6db, 2db, 3db
const vol_hard_t g_hard_volume_table[VOLUME_TABLE_MAX+1][VOLUME_VALUE_MAX+1] =
{
 //0db
 {
 /* 0db, -0.375db, -0.75db, -1.375db, -2db -2.75db, -3.375db, -4db */
 {40, 0xbf}, {40, 0xbe}, {40, 0xbd}, {39, 0xbe}, {38, 0xbf}, {38, 0xbd}, {37, 0xbe}, {36, 0xbf},
 /* -4.75db, -5.375db, -6db, -6.75db, -7.375db -8db, -8.75db, -9.375db */
 {36, 0xbd}, {35, 0xbe}, {34, 0xbf}, {34, 0xbd}, {33, 0xbe}, {32, 0xbf}, {32, 0xbd}, {31, 0xbe},
 /* -10db, -11db, -12db, -13db, -14.3db -15.8db, -17.4db, -19db */
 {30, 0xbf}, {29, 0xbf}, {28, 0xbf}, {27, 0xbf}, {26, 0xbf}, {25, 0xbf}, {24, 0xbf}, {23, 0xbf},
 /* -21.25db, -23.5db, -25.75db, -28.1db, -31.2db -35.3db, -41.6db, -60db */
 {22, 0xbd}, {20, 0xbf}, {19, 0xbd}, {17, 0xbf}, {15, 0xbf}, {13, 0xbf}, {10, 0xbf}, {0, 0xbf},
 },
};
```

说明：{40, 0xbf} 表示 PA 音量寄存器设置为 40，即 0db，DA 音量寄存器设置为 0xbf，即 0db。注意 DA 音量寄存器一般不能设置为大于 0db，否则对于一个 0db 的数字信号，D/A 时会有破音。

- 2) ADDA 通道 PA+DAC 音量表：PA 音量固定为 0db，只调节 DA 音量。

```
const vol_hard_t g_hard_volume_table[VOLUME_VALUE_MAX+1] =
{
 /* 0db, -0.375db, -0.75db, -1.5db, -2.25db -3db, -3.75db, -4.5db */
 {40, 0xbf}, {40, 0xbe}, {40, 0xbd}, {40, 0xbb}, {40, 0xb9}, {40, 0xb7}, {40, 0xb5}, {40, 0xb3},
 /* -4.875db, -5.625db, -6.375db, -7.125db, -7.5db -8.25db, -9db, -9.375db */
 {40, 0xb2}, {40, 0xb0}, {40, 0xae}, {40, 0xac}, {40, 0xab}, {40, 0xa9}, {40, 0xa7}, {40, 0xa6},
 /* -10.125db, -11.25db, -12db, -13.125db, -14.25db -15.75db, -17.625db, -19.125db */
 {40, 0xa4}, {40, 0xa1}, {40, 0x9f}, {40, 0x9c}, {40, 0x99}, {40, 0x95}, {40, 0x90}, {40, 0x8c},
 /* -21.375db, -23.625db, -25.875db, -28.125db, -31.5db -35.625db, -41.625db, -60db */
 {40, 0x86}, {40, 0x80}, {40, 0x7a}, {40, 0x74}, {40, 0x6b}, {40, 0x60}, {40, 0x50}, {40, 0x1f},
};
```

说明：{40, 0xbf} 表示 PA 音量寄存器设置为 40，即 0db，DA 音量寄存器设置为 0xbf，即 0db。注意 DA 音量寄存器一般不能设置为大于 0db，否则对于一个 0db 的数字信号，D/A 时会有破音。

- 3) I2S 音量表，不同 I2S PA 的表示方法都不一样：

```
//for volume convert table
//0xc0 for 0db, step 0.125db, 0x3ff for mute
const uint16 g_hard_volume_table_i2s_pa[VOLUME_VALUE_MAX+1] =
{
 0x00aa, 0x00b0, 0x00b6, 0x00bc, 0x00c2, 0x00c8, 0x00ce, 0x00d4,
 0x00da, 0x00e0, 0x00e6, 0x00ec, 0x00f2, 0x00f8, 0x0100, 0x0108,
 0x0110, 0x0118, 0x0124, 0x0130, 0x013c, 0x0148, 0x0154, 0x0160,
 0x0170, 0x0180, 0x0198, 0x01b0, 0x01d0, 0x01f0, 0x0220, 0x03ff,
};
```

音量调节模块向上层提供了以下调节方式:

1) 音量加 1: void **com\_volume\_add**(bool tts\_flag)

音量加 1, 并启动消息视图显示当前音量值, 如果音量加到最大音量, 通过 KEYTONE 或者 TTS 提示。该接口用于音箱音量加, 如果当前在蓝牙应用中, 会同步到手机。

2) 音量减 1: void **com\_volume\_sub**(bool tts\_flag)

音量减 1, 并启动消息视图显示当前音量值, 如果音量减到最小音量, 通过 KEYTONE 或者 TTS 提示。该接口用于音箱音量减, 如果当前在蓝牙应用中, 会同步到手机。

3) 音量设置绝对值: void **com\_volume\_set**(uint8 set\_vol, uint8 set\_mode)

音量设置绝对值, 并启动消息视图显示当前音量, 如果音量设置为 0 或者最大音量, 通过 KEYTONE 或者 TTS 提示。该接口用于 USB 音箱 HID 音量同步到音箱、蓝牙推歌和蓝牙免提音量同步到音箱。

音量调节模块还有以下几个外部接口:

1) 获取当前音量值: uint8 **com\_get\_sound\_volume**(void)

之所以增加这个接口, 我们是不希望各 AP 直接访问全局变量来获取音量值, 这样数据封装性好。

2) 重置当前音量: bool **com\_reset\_sound\_volume**(uint8 set\_mode)

一般是在切换应用、切换音量模式、切换音效模式、解除静音等之后调用。

3) 设置音量: bool **com\_set\_sound\_volume**(uint8 set\_vol, uint8 set\_mode)

应用层所有音量设置, 最终都会调用该接口; 并且只有解除静音情况下才能够设置进去。

4) 切换音量模式: void **com\_switch\_volume\_mode**(volume\_mode\_e mode)

切换音量模式仅仅将音量环境切换过来, 并没有真正设置进去, 所以之后必须调用一次 **com\_reset\_sound\_volume()** 之类的接口。

5) 设置音量相对增益: void **com\_set\_volume\_gain**(uint8 gain)

该增益是对各级音量都进行衰减的, 单位为-1db。

6) 开启静音/解除静音: bool **com\_set\_mute**(bool mute)

7) 开启静音/解除静音: bool **com\_set\_mute**(bool mute)

8) 切换静音状态: bool **com\_switch\_mute**(void)

音量调节底层实现接口：

```
void com_set_phy_volume(uint8 set_vol)
```

该接口一般由 `com_set_sound_volume()` 调用，`com_set_sound_volume()` 根据条件决定是否调用该接口，防止出现混乱。但是某些场景，我们希望无论如何都要设置音量，那么可以直接调用，只是由用户自己保证不会出现问题。

另外，US282A 引进了 MDRC 音效算法，需要动态调节修正音量，导致音量调节的规则复杂很多。所以修改该接口的实现必须十分小心。

## 6.21 数字音效调节

US282A 拥有强大的 DSP CORE，计算能力很强大，所以我们很重视数字音效，搭建了数字音效支持平台，实现了数字音效动态调节。

数字音效调节有以下几种情况：

- 1) 启动播放时，需要初始化音效库的参数，此时必须能够获取到所有音效参数。
- 2) 各种通道切换时，都需要设置数字音效工作模式；切换到非播歌通道时，需要将数字音效设置为 BYPASS。
- 3) 可以通过 APK/APP 的音效参数菜单设置音效。
- 4) 调音音量时，需要动态调整音效参数。
- 5) ASET 调试时，可以实时调节所有音效参数。

为了满足第一个情况，我们特地增加了一个全局共享内存，前台设置音效参数，仅仅是将保存在前台应用的音效参数拷贝到该共享内存；然后在中间件加载音效库后，会用共享内存中的音效参数初始设置音效库。

每次调节数字音效，如果引擎已经加载，那么会发送 `MSG_SET_DAE_CFG_SYNC` 同步消息让引擎间接调用中间件的设置音效参数的命令；另外，引擎会设置音效库运行所需的频率。

在调节数字音效之前，需要初始化音效环境，流程如下：

- 1) 固件升级后第一次开机，需要从 `config.bin` 加载音效参数，然后保存到 VRAM 中。

```
void com_load_dae_config(comval_t *setting_comval);
```

- 2) 每次开机时，初始化标志 `g_comval.dae_initied_flag = FALSE`；，表示尚未初始化完。

- 3) 每次开机时，都从 config.bin 恢复某些数字音效配置项，防止数字音效参数处于一种中间状态被重启后参数不对。

```
void com_reset_dae_config(comval_t *setting_comval);
```

- 4) 每次开机时，创建数字音效共享内存，并初始化一次；再将初始化标志置为 TRUE。

```
void com_init_dae_config(comval_t *setting_comval);
```

以上初始化流程执行过后，就可以正常调节数字音效了。

数字音效调节相关的接口包括：

- 1) APK/APP 设置音效：bool **com\_set\_dae\_config**(daemode\_param\_t \*daemode\_param);
- 2) 数字音效 BYPASS 开关：bool **com\_set\_dae\_onoff**(bool onoff);
- 3) 数字音效使能，disable 掉解码时就会加载一个空音效库，彻底将数字音效占用的空间释放出来：  
bool **com\_set\_dae\_enable**(bool enable);
- 4) 数字音效通道设置：bool **com\_set\_dae\_chan**(bool aux\_flag, bool variable\_mode);

以前 MP3 的方案，有那种传统的 EQ 音效，比如 Rock, Jazz, DBB 等，那么如果 US282A 也想做多种 EQ 音效，然后通过一个 EQ 键让用户切换，应该怎么实现呢？

3. 除了这三个参数 [DAE\_BYPASS\_ENABLE 、 DAE\_GRAPHICS\_EQ\_ENABLE 、 DAE\_GRAPHICS\_EQ\_TYPE]外，其他参数都应该每种音效都有一整套。比如有 3 种音效，那么配置项文件可以这样写：

```
DAE_BYPASS_ENABLE{400}
DAE_GRAPHICS_EQ_ENABLE
DAE_GRAPHICS_EQ_TYPE
```

```
DAE_PRECUT_RATIO{410}
DAE_EQUIVALENT_GAIN
...
DAE_LIMITER_ENABLE
DAE_LIMITER_SETTING
```

```
DAE2_PRECUT_RATIO{510}
DAE2_EQUIVALENT_GAIN
...
DAE2_LIMITER_ENABLE
DAE2_LIMITER_SETTING
```

```
DAE3_PRECUT_RATIO{610}
DAE3_EQUIVALENT_GAIN
...
DAE3_LIMITER_ENABLE
DAE3_LIMITER_SETTING
```

然后代码中访问 3 个公共参数之外的配置项需要改为， $DAE\_PRECUT\_RATIO + g\_eq\_type * 100$ ，其中  $g\_eq\_type$  取值 0 ~ 2 表示第一种到第三种音效。

4. 需要更新 case\ap\common\common\_func\common\_seteq.c 中的 EQ 音效切换接口。

## 6.22 获取能量及频谱等

获取能量及频谱指的是获取帧解码后的平均采样值、最大采样值、多个频段或多个频点的最大采样值。

这需要实时从解码库获取，前台应用和引擎应用都可以按照以下方式获取到以上值：

1) 由引擎在加载解码库，即发送 **PLAY** 命令之后，再调用

```
mmm_bp_cmd(mp_handle, MMM_BP_GET_ENERGY_INFO,
 (unsigned int) &(g_app_info_state_all.p_energy_value)); //获取共享变量的地址
g_app_info_state_all.energy_available_flag = TRUE;
```

2) 在前台应用和引擎应用直接读取能量值：

```
if (g_app_info_state_all.energy_available_flag == TRUE)
{
 //使用 *(g_app_info_state_all.p_energy_value) 获取能量最大值
 //使用 *(g_app_info_state_all.p_energy_value - 1) 获取能量平均值
```

```
}
```

- 3) 当解码库退出时, 需要 `g_app_info_state_all.energy_available_flag = FALSE;`

## 6.23 系统监控功能

系统有些功能是要周期监控的, 在条件满足后就触发一些事情, 比如定时关机、省电关机、低功耗模式、屏保关掉背光、电池电量检测、APK/APP 进入后台、外设热拔插检测等。

为此, 我们在 COMMON 中创建了一个周期为 500ms 的软定时器, 和一个周期为 200ms 的软定时器来执行以上监控任务。实现在 `case\ap\common\common_func\sys_counter.c` 文件中。

在 `void sys_timer_init(void)` 接口中, 创建以下两个软定时器, 该接口在前台应用初始化时调用:

```
sys_counter_timer_id = set_app_timer(APP_TIMER_ATTRB_CONTROL | (APP_TIMER_TAG_SYS << 8),
 500, sys_counter_handle);
```

```
peripheral_detect_timer_id = set_app_timer(APP_TIMER_ATTRB_CONTROL |
 (APP_TIMER_TAG_SYS << 8), PER_DETECT_PERIOD, peripheral_detect_handle);
```

在 `void sys_timer_exit(void)` 接口中将这两个软定时器销毁掉, 该接口在前台应用退出时调用。

## 6.24 其他功能

### 6.24.1 应用睡眠

US282A 的应用睡眠功能相对比较简单, 在睡眠时, 只能通过按键消息中断, 并且中断后该按键消息会继续返回应用场景中响应。应用睡眠的接口如下:

```
app_result_e com_app_sleep(uint32 sleep_timer);
```

## 6.25 COMMON 工程

COMMON 作为应用程序的一部分, 以静态链接库的方式链接到应用程序中, COMMON “工程”使用时, 要点包括:

- 1) COMMON 编译时独立的, 以 `case\ap\common` 目录下的 Makefile 为 Make 脚本, 遍历所有子目录下的 Makefile 并编译, 每个子目录的源文件生成某个 \*.a 归档文件, 并将所有 \*.o 目标文件拷贝



到 case\lib 目录下。

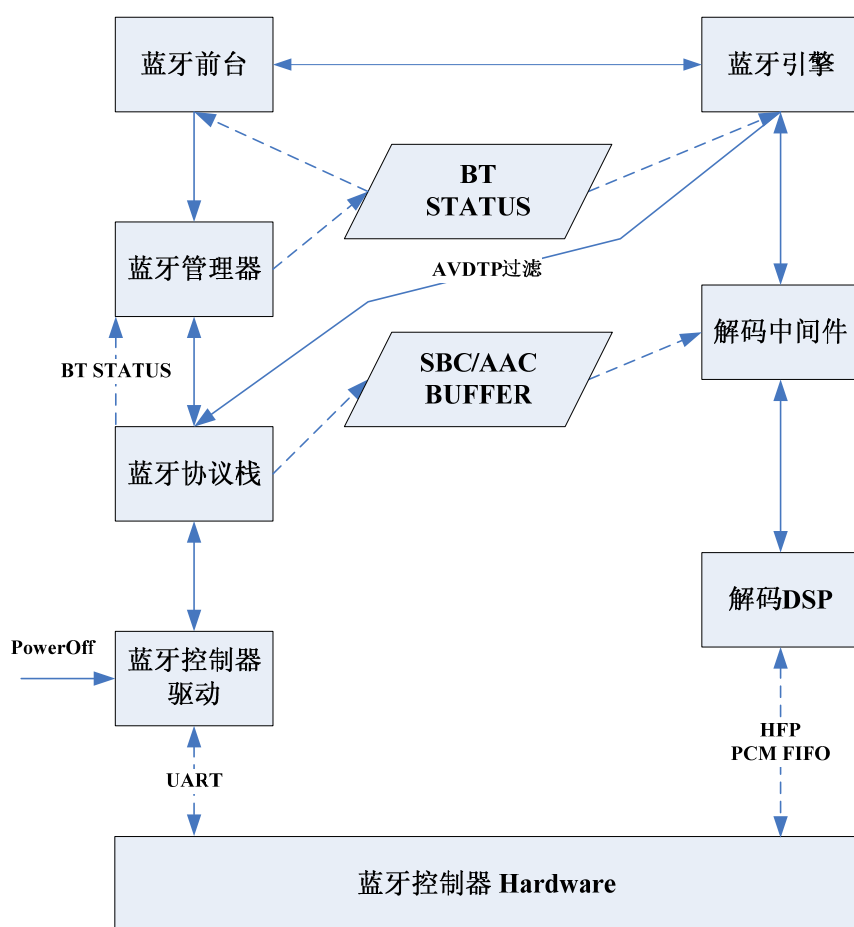
- 2) **COMMON** 提供两个公共的 \*.xn 链接脚本给前台应用和引擎应用链接，所以前台应用和引擎应用工程就不用再过多考虑 **COMMON** 的目标文件链接问题。不过，限制于常驻代码空间和数据空间，我们还是会有一些常驻代码段和数据段放到应用自己的 \*.xn 链接脚本中，特别是前台应用，还有一些特殊要求。这个后面在应用程序工程介绍时再详细说明。
- 3) **Manager AP** 和 **BT STACK AP** 由于应用空间分配比较特殊，并且也只是少量使用了 **COMMON** 的接口和数据，我们就直接将相关的段链接到应用自己的 \*.xn 链接脚本。
- 4) 各个应用程序工程必须确保公共的数据段的链接地址是一致的，否则就会出错。
- 5) 如果更新了 **COMMON** 的代码或数据，编译更新了对应的 \*.o 目标文件，那么需要将所有应用程序的 Target 全部删除，然后重新 Make 以确保各应用程序的 Target 能够重新生成。



## 7 蓝牙框架详解

### 7.1 概述

US282A 对蓝牙框架进行了重构，蓝牙框架大致如下：



US282A 的蓝牙框架在模块化做了很多优化，各模块简单说明一下：

- 1) 蓝牙前台：包括蓝牙推歌前台和蓝牙免提前台，是一个普通前台应用。
- 2) 蓝牙引擎：包括蓝牙推歌引擎和蓝牙免提引擎，是一个普通引擎应用。
- 3) 解码中间件：包括蓝牙推歌中间件和蓝牙免提中间件。

- 4) 解码 DSP：包括 SBC 解码 DSP、AAC 解码 DSP 和回声消除 DSP。
- 5) 蓝牙管理器：蓝牙协议栈与蓝牙应用层之间的抽象层，对应用层屏蔽掉蓝牙协议栈的技术细节，向应用层提供一套友好的蓝牙控制和管理接口，包括连接、回连、AVRCP 控制、HFP 控制、SPP/BLE 数传等；我们将将个性化策略性定制内容放到蓝牙管理器，从而保持蓝牙协议栈的独立性。
- 6) 蓝牙协议栈：与蓝牙管理器对接，实现各种蓝牙 Profile 和 Protocol。
- 7) 蓝牙控制器驱动：驱动蓝牙控制器，通过 UART 接口与蓝牙控制器通信等。

以上模块中，蓝牙协议栈、蓝牙控制器驱动、解码中间件和解码 DSP 都是不开放源码的，也就是说，这些模块的功能都是比较标准的，明确定义的，独立性都很好；而蓝牙前台、蓝牙引擎和蓝牙管理器则是方案个性化功能集中的模块。

可以这么说，蓝牙推歌前台和蓝牙推歌引擎就是一个普通的播歌通道的前台和引擎，与其他播歌通道的区别主要在于数据通道，蓝牙推歌的数据通道是一种流媒体，实时性要求较大，它们没有什么其他的“蓝牙”特性；“蓝牙”特性相关的个性化功能基本都在蓝牙管理器中。

因此，蓝牙控制器驱动和蓝牙协议栈我们就不在这里细说，蓝牙前台、蓝牙引擎、中间件和解码 DSP 也不会在这里细说，我们主要讲蓝牙管理器，以及与蓝牙管理器相关的所有一切。

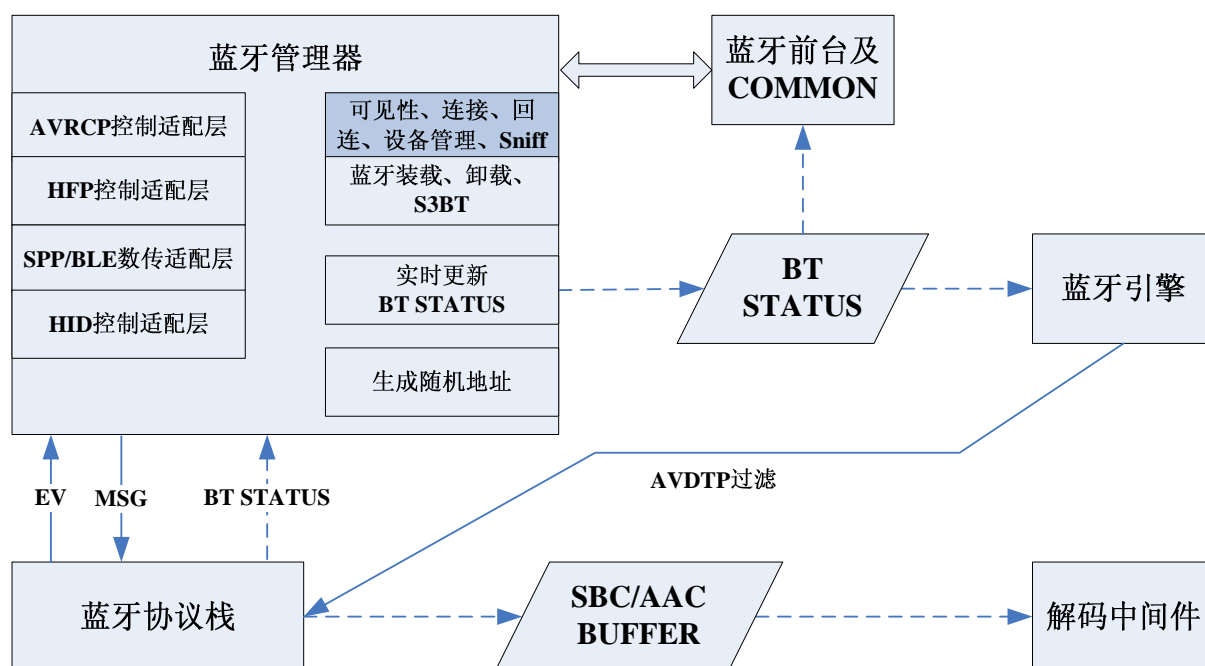
蓝牙管理器与蓝牙协议栈的关系，可以这样理解：蓝牙协议栈不要有 CASE 相关的数据，而蓝牙管理器则可以有蓝牙协议栈相关的数据，即宁愿多开放一些蓝牙协议栈的细节，也不要将一些 CASE 相关的数据封装起来使得客户修改不到。

## 7.2 蓝牙管理器

蓝牙管理器是蓝牙协议栈与蓝牙应用层之间的抽象层，对应用层屏蔽掉蓝牙协议栈的技术细节，向应用层提供一套友好的蓝牙控制和管理接口，包括实时更新 BT Status、可见性、连接、回连、设备管理、Sniff 管理、AVRCP 控制、HFP 控制、SPP/BLE 数传等；我们将将个性化策略性定制内容放到蓝牙管理器，从而保持蓝牙协议栈的独立性。

### 7.2.1 功能框图

蓝牙管理器实际上是一个功能复杂的大模块，这一节我们将其展开，蓝牙管理器内部功能框图如下：



从上图可以看出，蓝牙管理器包括多个子功能模块，下面逐个介绍。

## 7.2.2 可见性、连接、回连、设备管理、Sniff

这是蓝牙管理器最核心的子功能模块，几乎所有的蓝牙相关的行为都是在这个模块中定义的：

- 1) 可见性：可见性分为经典蓝牙可见性和 BLE 蓝牙可见性，默认都是可见的。  
经典蓝牙可见性 `void com_set_discoverable(bool mode)`，规则是：如果当前已连接 ACL 链路的设备数目少于支持的经典蓝牙设备数，那么打开可见性，否则就会关掉可见性。  
BLE 蓝牙可见性 `void com_set_ble_discoverable(bool mode)`
- 2) 连接：包括被动连接和主动回连，需要向用户反馈连接状态，比如蓝牙指示灯、连接事件提示音，并且提供了断开连接的功能，比如长按 **MODE** 键断开连接。
- 3) 回连：开机或切换到蓝牙应用或异常断开后，音箱需要主动回连手机，回连规格如下：
  - 音箱从配对列表中选择一个设备发起连接。
  - 先连接 **HFP** 服务，再连接 **A2DP** 服务，这样连接兼容性会好，能确保大多数手机能连接起来。
  - 支持双手机、对箱等多设备回连。
- 4) 设备管理：包括已连接设备管理和已配对设备管理，规格如下：
  - 支持已连接“多”设备管理
  - 支持最多 8 个已配对设备管理：可以指定回连优先级，可以删除已配对设备。

- 5) **Sniff**: 这是蓝牙控制器的一种工作状态, 这时蓝牙控制器处于一种低功耗, 并且与已连接的设备的通信也会减少。

蓝牙管理器会一直判断是否可以进入 **Sniff**, 确认可以进入 **Sniff** 后发送命令让蓝牙控制器进入 **Sniff**, 然后蓝牙控制器进入低功耗模式。

### 7.2.3 蓝牙装载、卸载、S3BT

这部分是初始化和退出功能。

- 1) 蓝牙装载: 先装载蓝牙管理器, 初始化蓝牙管理器工作环境, 然后装载蓝牙协议栈, 由蓝牙协议栈去装载蓝牙控制器驱动。

```
void com_btmanager_init(bool standby_exit)
```

- 2) 蓝牙卸载: 先卸载蓝牙协议栈, 由蓝牙协议栈卸载蓝牙控制器驱动, 再销毁蓝牙管理器工作环境。

```
void com_btmanager_exit(bool standby_enter, bool need_power_off)
```

- 3) **S3BT**: 当系统处于空闲状态并持续一段时间, 就可以进入 **S3BT** 低功耗模式, 在这种模式下可以通过蓝牙唤醒。进入 **S3BT** 需要把蓝牙控制器、蓝牙协议栈、蓝牙管理等现场环境保存到 **VRAM**, 退出 **S3BT** 时恢复现场, 这样蓝牙功能就可以继续。

### 7.2.4 实时更新 BT STATUS

蓝牙管理器需要实时获取蓝牙协议栈的状态, 以此来了解蓝牙协议栈的状态变化并采取行动。US282A 采用共享查询的方式来实时更新状态, 对于 **BT STATUS** 来说, 即蓝牙协议栈使用 **APP\_ID\_BTSTACK** 创建了一个共享查询, 然后在蓝牙管理器就可以使用下面方式实时更新 **BT STATUS**:

```
if (sys_share_query_read(APP_ID_BTSTACK, &g_bt_stack_cur_info) == -1)
{
 PRINT_ERR("btstack share query not exist!!");
 while (1)
 {
 ; //nothing
 }
}
```

然后前台应用和引擎应用想获取到 **BT STACK**, 我们就通过共享内存方式开放出去, 即蓝牙管理器使用 **SHARE\_MEM\_ID\_BTSTACK\_STATUS** 创建一个共享内存, 蓝牙管理器卸载时要销毁掉该共享内存:

```
if (sys_shm_creat(SHARE_MEM_ID_BTSTACK_STATUS, &g_bt_stack_cur_info, sizeof(bt_stack_info_t)) == -1)
{
 PRINT_ERR("btstack status shm create fail!!");
 while (1)
 {
 ; //nothing for QAC
 }
}

sys_shm_destroy(SHARE_MEM_ID_BTSTACK_STATUS);
```

然后前台应用和引擎应用在访问之前先挂载该共享内存，然后就可以使用指针直接访问了：

```
g_p_bt_stack_cur_info = (bt_stack_info_t *) sys_shm_mount(SHARE_MEM_ID_BTSTACK_STATUS);
if (g_p_bt_stack_cur_info == NULL)
{
 PRINT_ERR("btstack status shm not exist!!");
 while (1)
 {
 ; //nothing for QAC
 }
}
```

## 7.2.5 生成随机地址

我们采用了一种真随机的算法来生成随机地址，即用 ADC 采样一段时间的噪声，然后对噪声进行运算得出一个 24bit 的随机数，作为蓝牙地址低 24bit。

如果需要自己指定蓝牙地址低 24bit，可以自己修改 `bt_common_manager.c` 中创建蓝牙协议栈时的初始化参数 `bt_stack_arg.random_lap`，赋值为自己想要的低 24bit 地址即可。

## 7.2.6 AVRCP 控制适配层

支持播放/暂停/下一曲/上一曲/快进退/音量同步等 AVRCP 命令，其中音量同步是双向的。

## 7.2.7 HFP 控制适配层

支持音量同步/电话控制/电池电量上报等功能，其中音量同步是双向的。

## 7.2.8 SPP/BLE 数传适配层

蓝牙协议栈收到 SPP/BLE 数传数据包，将数据存放到 FIFO BUFFER 中，封装为事件发送给蓝牙协议栈；蓝牙协议栈收到事件后，转发给 RCP 层进行命令包分发处理，命令包分发处理过程通过蓝牙管理器从

FIFO BUFFER 中访问/读取数据。

## 7.2.9 HID 控制适配层

### 7.2.10 蓝牙管理器 LOOP

蓝牙管理器装载起来后，创建了一个回连软定时器，500ms 执行一次回连处理，其他的功能执行，都得依赖于蓝牙管理器的入口函数的执行。该入口函数在应用主循环 `app_result_e com_view_loop(void)` 函数中调用，以及在 TTS 播放主循环 `tts_play_ret_e com_tts_state_play_loop(uint16 tts_mode)` 中调用。

```
app_result_e com_btmanager_loop(bool tts_flag);
```

该接口包括以下功能：

- 1) 实时更新 BT STATUS。
- 2) 蓝牙管理器工作状态切换管理，伴随着调频。
- 3) 蓝牙连接状态变化响应，进行合适的提示等。
- 4) 蓝牙可见性处理。
- 5) 蓝牙 A2DP/AVRCP 相关处理。
- 6) 蓝牙 HFP 相关处理，包括电话状态变化响应。
- 7) 电池电量上报。
- 8) 蓝牙 Sniff 处理。
- 9) 蓝牙协议栈事件响应。

### 7.2.11 蓝牙管理器工作状态

蓝牙协议栈是很独立的一个模块，它自己并不会去调频，而是由蓝牙管理器代理。也就是说，蓝牙工作状态是由蓝牙管理器定义的，包括以下 4 个状态：

```
typedef enum
{
 BTMANAGER_STATUS_IDLE = 0,
 BTMANAGER_STATUS_WORKING = 1,
 BTMANAGER_STATUS_LINKING = 2,
 BTMANAGER_STATUS_POWERON = 3,
} btmanager_status_e;
```

- 1) 上电状态：蓝牙管理器装载，会有很多代码数据要加载，很多初始化代码要执行，我们将调频为最高 85MHz，目的就是想要装载速度快。
- 2) 连接状态：蓝牙管理器主动发起回连，或者 ACL 已连接，表示进入连接流程，会有大量的蓝牙事件命令要处理，所以也要调频为 60MHz，确保能及时响应蓝牙事件命令，保证连接稳定性。
- 3) 空闲状态：当蓝牙控制器进入低功耗模式，此时蓝牙协议栈都不需要做什么事情，就进入空闲模式，我们会将调频为 1MHz。
- 4) 工作状态：空闲状态收到蓝牙控制器的数据就会切换为工作状态；上电状态和连接状态等待 30 秒钟就会切换为工作状态。

## 7.3 BT STACK 解析

蓝牙协议栈的状态分为 2 大类：

总状态机：从设备连接的角度出发，每种服务都只关心是否连接。它包含 2 个状态，一个是设备连接状态，一个是连接了哪种服务。

功能状态机：从应用功能的角度出发，暂时只关心 A2DP 和 HFP 服务，每种服务都有很多功能直接相关的状态，并且除了状态外，还可能包含一些属性。

说明：A2DP 还包括 AVDTP 数据流的编码格式这样的属性。

说明：HFP 还包括 SCO 链路是否已连接这样的属性。

## 7.4 AVDTP 数据管道

蓝牙协议栈创建了 AVDTP 数据管道，这是一种 FIFO BUFFER，用来向蓝牙推歌中间件传递 SBC/AAC 原始数据；AVDTP 数据管道管理结构体使用共享内存方式共享出来，共享内存 ID 为 SHARE\_MEM\_ID\_BTPLAYPIPE。

蓝牙推歌引擎和蓝牙推歌中间件，都是通过挂载该共享内存来获取对 FIFO BUFFER 的访问指针的。

另外，蓝牙推歌引擎还可以通过发送 MSG\_BTSTACK\_PLAY\_PIPE\_CTRL\_SYNC 消息进行 AVDTP 过滤控制。

## 7.5 SPP/BLE 数据管道

蓝牙协议栈创建了 SPP/BLE 数据管道，用来向蓝牙管理器发送事件；SPP/BLE 数据管道管理结构体也

是使用共享内存方式共享出来，共享内存 ID 为 SHARE\_MEM\_ID\_BTRCPPIPE。

## 7.6 蓝牙应用

按照应用模式来分，可以分为蓝牙推歌和蓝牙打电话，而其中的蓝牙打电话又根据产品形态，可以分为主场景模式和子场景模式：

蓝牙音箱产品形态：蓝牙推歌，以及子场景模式的蓝牙打电话。

蓝牙耳机产品形态：主场景模式的蓝牙打电话。

在我们的方案中，蓝牙推歌和蓝牙打电话时互斥的两个功能，即在打电话时候，是不能进行蓝牙推歌的，会把 A2DP 的 AVDTP 音频数据流过滤掉。

蓝牙引擎原则上只负责蓝牙推歌和蓝牙打电话这 2 件本质的事情，它们通过实时获取蓝牙协议栈的状态，在状态发生变化时作出响应。并且向蓝牙前台提供一个播放器状态机。

蓝牙推歌引擎，在检测到 A2DP 已经连接起来，就会主动将 codec 库加载起来，以保证推歌更加及时。



## 8 CASE 空间分配详解

### 8.1 概述

US282A 的内存空间分配主要分为 3 大部分：

- 1) **DSP**: 用来解码和音效处理等，这些空间都是不可分配的，必须由原厂 FAE 配合进行深度开发才允许分配
- 2) **PSP**: 系统运行公共基础模块，包括操作系统、基础驱动程序、中间件等，这些内存空间都是不可分配的，必须由原厂 FAE 配合进行深度开发才允许分配
- 3) **CASE**: 方案可分配的内存空间，其中后台蓝牙占用的内存空间是不可分配的，必须由原厂 FAE 配合进行深度开发才允许分配

所有的内存分配，都应该在 `psp_rel\include` 目录下的 `link_base.xn` 和 `task_info.xn`（需要同步修改 `task_manager.h` 头文件）中定义好，后者只分配栈空间。

另外，本章还将说明系统堆空间和 VRAM 空间分配。

下图是 US282A 整个 CASE 的内存空间分配图，我们只列出 CASE 相关部分，DSP 和 PSP 的内存空间分配是不体现在该内存空间分配图中的。

|                                                      |                                               |                                                          |  |                                  |  |                                                               |
|------------------------------------------------------|-----------------------------------------------|----------------------------------------------------------|--|----------------------------------|--|---------------------------------------------------------------|
| [0x28000---0x287ff]<br>AP_BACK_ENHANCE2 (2k)         |                                               | [0x27000---0x28fff] 各种播放场景占用 (8k)                        |  | [0x27000---0x28fff] Reserve (8k) |  | [0x26000---0x28fff]<br>RECORD_BUFF (12k)<br>暂时没用，保留给录音BUFFER用 |
| [0x27000---0x27fff]<br>TTS_DRV_RCODE (4k)            |                                               |                                                          |  |                                  |  |                                                               |
| [0x26800---0x26fff] AP_BANK_BACK_CTL (2K)            |                                               |                                                          |  |                                  |  |                                                               |
| [0x26700---0x267ff] RCP Command Buffer (0x100 bytes) |                                               |                                                          |  |                                  |  |                                                               |
| [0x26000---0x266ff] AP_BACK_RCODE (1.75K)            |                                               |                                                          |  |                                  |  |                                                               |
| [0x25800---0x25fff] AP_BANK_FRONT_UI (2K)            |                                               |                                                          |  |                                  |  |                                                               |
| [0x25000---0x257ff]<br>FM_DRV_RCODE(2k)              | [0x25000---0x257ff]<br>AP_FRONT_ENHANCE1 (2k) | [0x25400---0x257ff]AP_BANK_BASAL (1k)                    |  |                                  |  |                                                               |
|                                                      |                                               | [0x25000---0x253ff]AP_BASAL_RCODE &RDATA (1k)            |  |                                  |  |                                                               |
|                                                      |                                               | [0x22400---0x24fff] 后台蓝牙占用 (11k)                         |  |                                  |  |                                                               |
| [0x21c00---0x223ff]<br>AP_BACK_ENHANCE1 (2k)         |                                               | [0x20c00--- 0x223ff] 系统占用 (6k)                           |  |                                  |  |                                                               |
|                                                      |                                               | [0x20400---0x20bfff] AP_BANK_FRONT_CTL (2K)              |  |                                  |  |                                                               |
|                                                      |                                               | [0x20200---0x203ff] ap_bank_manager (0.5K)               |  |                                  |  |                                                               |
|                                                      |                                               | [0x20000--- 0x201ff] 系统占用 (0.5k)                         |  |                                  |  |                                                               |
|                                                      |                                               | [0x1f800---0x1ffff] KEY_DRV_RCODE (2K)                   |  |                                  |  |                                                               |
|                                                      |                                               | [0x1d900---0x1f7ff] 后台蓝牙占用 (7.75k)                       |  |                                  |  |                                                               |
|                                                      |                                               | [0x1b600--- 0x1d8ff] 栈空间 (8.75k)                         |  |                                  |  |                                                               |
|                                                      |                                               | [0x1b400--- 0x1b5ff] 系统占用 (0.5k)                         |  |                                  |  |                                                               |
|                                                      |                                               | [0x1b300---0x1b3ff] RESERVE (0.25k)                      |  |                                  |  |                                                               |
|                                                      |                                               | [0x1b200---0x1b2ff] ap_manager_rcode + rdata (0.25k)     |  |                                  |  |                                                               |
|                                                      |                                               | [0x1ac80---0x1b1ff] 系统占用，系统堆空间 (0x580 bytes)             |  |                                  |  |                                                               |
|                                                      |                                               | [0x1a400--- 0x1ac7f] 系统占用 (0x880 bytes)                  |  |                                  |  |                                                               |
|                                                      |                                               | [0x1a380---0x1a3ff] BT_GLOBAL_DATA (0x80 bytes)          |  |                                  |  |                                                               |
|                                                      |                                               | [0x1a1c0---0x1a37f] COMM_GLOBAL_DATA: (0x1c0 bytes)      |  |                                  |  |                                                               |
|                                                      |                                               | [0x1a180---0x1a1bf] APPLIB GLOBAL DATA (64 bytes)        |  |                                  |  |                                                               |
|                                                      |                                               | [0x19a80---0x1a17f] AP_FRONT_DATA : (1.75K)              |  |                                  |  |                                                               |
| [0x19880---0x19a7f]<br>FM_DRV_DATA(0.5k)             |                                               | [0x19580---0x19a7f] AP_BACK_DATA : (1.25K)               |  |                                  |  |                                                               |
|                                                      |                                               | [0x16c00---0x1957f] 卡驱动/UHOST驱动/FS驱动<br>本地播歌占用 (10.375k) |  |                                  |  |                                                               |
|                                                      |                                               | [0x16400---0x16bff] AP_FRONT_RCODE(2K)                   |  |                                  |  |                                                               |
|                                                      |                                               | [0x14200---0x163ff] 后台蓝牙占用 (8.5k)                        |  |                                  |  |                                                               |
|                                                      |                                               | [0x13fc0--- 0x141ff] 系统占用 (0x240 bytes)                  |  |                                  |  |                                                               |
|                                                      |                                               | [0x135e0---0x13fbf] comm BT manager rcode (0x9e0 bytes)  |  |                                  |  |                                                               |
|                                                      |                                               | [0x10000---0x135df] 后台蓝牙占用 (13k + 0x1e0 bytes)           |  |                                  |  |                                                               |
|                                                      |                                               | [0x0c000--- 0x0ffff] 系统占用 (16k)                          |  |                                  |  |                                                               |
|                                                      |                                               | [0x0bf00---0x0bfff] UI_DRV_DATA : (0.25K)                |  |                                  |  |                                                               |
|                                                      |                                               | [0x0be00---0x0beff] KEY_DRV_DATA: (0.25K)                |  |                                  |  |                                                               |
|                                                      |                                               | [0x0a000---0x0bdf] 后台蓝牙占用 (7.5k)                         |  |                                  |  |                                                               |
|                                                      |                                               | [0x09fc0---0x09fff] ccd drv rcode & rdata ( 64bytes )    |  |                                  |  |                                                               |
|                                                      |                                               | [0x09f00---0x09fbf] 系统占用 (0xc0 bytes)                    |  |                                  |  |                                                               |
|                                                      |                                               | [0x09c00---0x09eff] UI_DRV_RCODE : (0.75K)               |  |                                  |  |                                                               |
|                                                      |                                               | [0x09400---0x09bfff] BANK B 系统占用 (2K)                    |  |                                  |  |                                                               |
|                                                      |                                               | [0x09000---0x093ff] BANK A 系统占用 (1K)                     |  |                                  |  |                                                               |

场景颜色标记

本地播歌  
PLAY

蓝牙推歌  
A2DP

FM RADIO  
PLAY

蓝牙通话  
HFP

TTS播报

本地录音  
RECORD

USB 音箱

USB 读卡器

蓝牙协议栈

栈空间

一般场景

### 场景颜色标记

|              |              |
|--------------|--------------|
| 本地播歌<br>PLAY | 蓝牙推歌<br>A2DP |
|              |              |

|                  |             |
|------------------|-------------|
| FM RADIO<br>PLAY | 蓝牙通话<br>HFP |
|                  |             |

|       |                |
|-------|----------------|
| TTS播报 | 本地录音<br>RECORD |
|       |                |

|        |         |
|--------|---------|
| USB 音箱 | USB 读卡器 |
|        |         |

|       |     |
|-------|-----|
| 蓝牙协议栈 | 栈空间 |
|       |     |

一般场景

|  |
|--|
|  |
|--|

## 8.2 COMMON 内存分配

所有前台应用共享一份前台应用的 COMMON 链接脚本，所有引擎应用也共享一份引擎应用的 COMMON 链接脚本。另外，COMMON 有部分数据是所有应用共享的，比如 APPLIB，这些数据段要求所有应用（包括 AP Manager 和 BT Stack）都链接到同一个地址，否则会出现各种莫名其妙的错误甚至死机。再者，对于前台应用来说，COMMON 模块有部分代码和数据空间是独立于应用的，即使切换应用也不会受影响，这主要是因为前台应用的 COMMON 已经相当于一个系统管理的应用了，它需要独立运行。

### 8.2.1 所有应用共享

所有应用共享的数据段空间为：

```
SRAM_APPLIB_GLOBAL_DATA_ADDR = 0x1a080;
```

```
SRAM_APPLIB_GLOBAL_DATA_SIZE = 0x40;
```

如果要将某个全局变量做成所有应用共享的，那么可以将其定义在 `app_info_state_all_t` 结构体中。

#### 8.2.1.1 AP Manager 应用

AP Manager 应用的链接脚本，`process_manager_linker.xn`：

```
. = APPLIB_GLOBAL_DATA_ADDR;
APPLIB_GLOBAL_DATA :
{
 /*common globe 数据*/
 KEEP(applib_globe_data.o(.applib_globe_data.g_app_info_vector))
 . = 0x10;
 KEEP(applib_globe_data.o(.applib_globe_data.config_fp))
 . = 0x14;
 KEEP(applib_globe_data.o(.applib_globe_data.g_app_info_state_all))
 . = APPLIB_GLOBAL_DATA_SIZE;
}
```

该链接脚本还有一个 COMMON 数据段，`g_app_info_state` 需要在 AP Manager 的初始化过程中访问，该数据段必须使用相同的地址链接到前台应用中。

```
. = APPLIB_DATA_ADDR;
APPLIB_DATA :
{
 KEEP(applib_globe_data_com.o(.g_app_info_state))
}
```

### 8.2.1.2 BT STACK 应用

BT STACK 应用的链接脚本，bluetooth\_stack.xn:

```
. = APPLIB_GLOBAL_DATA_ADDR;
APPLIB_GLOBAL_DATA :
{
 /*common globe 数据*/
 KEEP(applib_globe_data.o(.applib_globe_data.g_app_info_vector))
 . = 0x10;
 KEEP(applib_globe_data.o(.applib_globe_data.config_fp))
 . = 0x14;
 KEEP(applib_globe_data.o(.applib_globe_data.g_app_info_state_all))
 . = APPLIB_GLOBAL_DATA_SIZE;
}
```

### 8.2.1.3 引擎应用

引擎应用的公共数据段由 COMMON 链接脚本 common\_engine.xn 链接:

```
. = APPLIB_GLOBAL_DATA_ADDR;
APPLIB_GLOBAL_DATA :
{
 /*common globe 数据*/
 KEEP(applib_globe_data.o(.applib_globe_data.g_app_info_vector))
 . = 0x10;
 KEEP(applib_globe_data.o(.applib_globe_data.config_fp))
 . = 0x14;
 KEEP(applib_globe_data.o(.applib_globe_data.g_app_info_state_all))
 . = APPLIB_GLOBAL_DATA_SIZE;
}
```

### 8.2.1.4 前台应用

前台应用的公共数据段由 COMMON 链接脚本 common\_front.xn 链接:

```
. = APPLIB_GLOBAL_DATA_ADDR;
APPLIB_GLOBAL_DATA :
{
 /*common globe 数据*/
 KEEP(applib_globe_data.o(.applib_globe_data.g_app_info_vector))
 . = 0x10;
 KEEP(applib_globe_data.o(.applib_globe_data.config_fp))
 . = 0x14;
 KEEP(applib_globe_data.o(.applib_globe_data.g_app_info_state_all))
 . = APPLIB_GLOBAL_DATA_SIZE;
}
```

另外，前台应用还有部分数据段与 AP Manager 是公共的，红色框必须放在该数据段的**开头位置**:

```
. = APPLIB_DATA_ADDR;
APPLIB_DATA :
{
 KEEP(applib_globe_data_com.o(.g_app_info_state))
 KEEP(applib_globe_data_com.o(.gl_com_data))
 KEEP(message_key_deal.o(.gl_key_data))
 KEEP(common_tts.o(.gl_tts_data))
 KEEP(common_keytone_play.o(.gl_kt_data))
 KEEP(common_rcp_dispatch.o(.gl_rcp_data))
 KEEP(common_aset_main_deal.o(.gl_stub_data))
 . = APPLIB_DATA_SIZE;
}
```

## 8.2.2 前台应用 COMMON

如上所述：对于前台应用来说，COMMON 模块有部分代码和数据空间是独立于应用的，即使切换应用也不会受影响，这主要是因为前台应用的 COMMON 已经相当于一个系统管理的应用了，它需要独立运行。

所以 US282A 为前台应用 COMMON 划出了一个独立的常驻代码段，以及若干独立的全局数据段：

常驻代码段，用来作为蓝牙管理等模块的常驻代码段：

SRAM\_COMMON\_BT\_MANAGER\_RCODE = 0x135e0;

SRAM\_COMMON\_BT\_MANAGER\_SIZE = 0x9e0;

```
.text1 RCODE_TEXT1_ADDR :
{
 KEEP(bt_common_manager_deal.o(.rcode))
 KEEP(app_timer_bank_single.o(.text .rodata))
 KEEP(common_rcp_dispatch.o(.rcode))
 KEEP(common_aset_main_deal.o(.rcode))
 KEEP(common_rcp_dispatch_block.o(.rcode))
 KEEP(common_rcp_dispatch_misc.o(.rcode))
 KEEP(common_rcp_func.o(.rcode))
 KEEP(bt_common_sppble.o(.rcode))
 KEEP(message_bank_send_btmanager.o(.text .rodata .xdata))
 KEEP(common_setdae.o(.rcode))
 . = SRAM_COMMON_BT_MANAGER_SIZE;
}
```

全局数据空间，即要求在切换应用也不受影响的数据段和缓冲区：

SRAM\_APPLIB\_DATA\_ADDR = 0x1a1c0;

SRAM\_APPLIB\_DATA\_SIZE = 0x1c0;

SRAM\_AP\_RCP\_COMMAND\_BUFFER\_ADDR = 0x26700;

SRAM\_AP\_RCP\_COMMAND\_BUFFER\_SIZE = 0x100;

```
SRAM_BTMANAGER_DATA_ADDR = 0x1a380;

SRAM_BTMANAGER_DATA_SIZE = 0x80;

. = APPLIB_DATA_ADDR;
APPLIB_DATA :
{
 KEEP(applib_globe_data_com.o(.g_app_info_state))
 KEEP(applib_globe_data_com.o(.gl_com_data))
 KEEP(message_key_deal.o(.gl_key_data))
 KEEP(common_tts.o(.gl_tts_data))
 KEEP(common_keytone_play.o(.gl_kt_data))
 KEEP(common_rcp_dispatch.o(.gl_rcp_data))
 KEEP(common_aset_main_deal.o(.gl_stub_data))
 . = APPLIB_DATA_SIZE;
}

. = RCP_COMMAND_BUFFER_ADDR;
RCP_COMMAND_BUFFER :
{
 common_rcp_dispatch.o(.bss.rcp_default_buffer)
}

. = BSMANAGER_DATA_ADDR;
BSMANAGER_DATA :
{
 KEEP(bt_common_manager.o(.bt_auto_connect))
 KEEP(bt_common_manager.o(.bt_globe_data))
 . = BTMANAGER_DATA_SIZE;
}
```

### 8.3 AP Manager 内存分配

AP Manager 功能比较简单，所以分配的内存空间是比较少的，包括：

**常驻代码段&数据段：**

SRAM\_AP\_MANAGER\_RCODE\_ADDR = 0x1b200;

SRAM\_AP\_MANAGER\_RCODE\_SIZE = 0x100;

**AP Manager BANK 段：**

SRAM\_AP\_BANK\_MANAGER\_ADDR = 0x20200;

SRAM\_AP\_BANK\_MANAGER\_SIZE = 0x200;

BANK 组号是 0x7f，即地址空间为 0x7fxxxxxx。

### 8.4 前台应用内存分配

前台应用主要负责用户交互，功能可能比较复杂，并且它是客户方案修改最多的工程，所以内存空间

也稍微多一点；其中前台应用 **COMMON** 部分内存空间已经在前面说过了，这里说的前台应用内存空间分配不包括 **COMMON** 部分：

**常驻代码段：**

SRAM\_AP\_FRONT\_RCODE\_ADDR = 0x16400;

SRAM\_AP\_FRONT\_RCODE\_SIZE = **0x800**;

**常驻数据段：**

SRAM\_AP\_FRONT\_DATA\_ADDR = 0x19a80;

SRAM\_AP\_FRONT\_DATA\_SIZE = **0x700**;

**前台应用 CONTROL BANK：**

SRAM\_AP\_BANK\_FRONT\_CONTROL\_ADDR = 0x20400;

SRAM\_AP\_BANK\_FRONT\_CONTROL\_SIZE = **0x800**;

BANK 组号是 0x40，即地址空间为 **0x40xxxxxx**。

**前台应用 UI BANK**，主要供 **COMMON** 支配，其中 **BANK\_10** 几乎可以理解为常驻代码段，即在正常播放时几乎不会被切走：

SRAM\_AP\_BANK\_FRONT\_UI\_ADDR = 0x25800;

SRAM\_AP\_BANK\_FRONT\_UI\_SIZE = **0x800**;

BANK 组号是 0x48，即地址空间为 **0x48xxxxxx**。

## 8.5 前台中间件内存分配

前台中间件是指 **CASE** 中用 \*.AL 格式打包，通过命令式 **API** 接口访问的独立程序模块，这种程序的任务也比较简单，所以分配的内存空间也比较小：

**常驻代码段&数据段：**

SRAM\_AP\_FRONT\_BASAL\_RCODE\_ADDR = 0x25000;

SRAM\_AP\_FRONT\_BASAL\_RCODE\_SIZE = **0x400**;

**前台中间件 BANK 段：**

SRAM\_AP\_BANK\_FRONT\_BASAL\_ADDR = 0x25400;

SRAM\_AP\_BANK\_FRONT\_BASAL\_SIZE = **0x400**;

BANK 组号是 0x50，即地址空间为 **0x50xxxxxx**。

## 8.6 引擎应用内存分配

引擎应用是应用的播放业务逻辑模块，功能相对比较独立单一，但是也可能比较复杂，所以我们分配的内存空间也还是比较多的：

常驻代码段：

SRAM\_AP\_BACK\_RCODE\_ADDR = 0x26000;

SRAM\_AP\_BACK\_RCODE\_SIZE = **0x700**;

常驻数据段：

SRAM\_AP\_BACK\_DATA\_ADDR = 0x19580;

SRAM\_AP\_BACK\_DATA\_SIZE = **0x500**;

引擎应用 BANK 段：

SRAM\_AP\_BANK\_BACK\_CONTROL\_ADDR = 0x26800;

SRAM\_AP\_BANK\_BACK\_CONTROL\_SIZE = **0x800**;

BANK 组号是 0x60，即地址空间为 **0x60xxxxxx**。

## 8.7 后台蓝牙内存分配

后台蓝牙占用内存空间很大，从上面的内存空间分配图可以看到，它占用超过 48KB 内存空间；如果不支持后台蓝牙，那么在非蓝牙应用场景下，这部分内存空间可以完全自主分配。

另外，蓝牙推歌应用和蓝牙免提应用，都会占用一个很大的数据段：

SRAM\_BTSTACK\_AVDTP\_DATA\_ADDR = 0x16c00;

SRAM\_BTSTACK\_AVDTP\_DATA\_SIZE = **0x2980**;

SRAM\_BTCALL\_CALLRING\_BUF\_ADDR = 0x16c00;

SRAM\_BTCALL\_CALLRING\_BUF\_SIZE = **0x1000**;

SRAM\_BTCALL\_ASQT\_BUF\_ADDR = 0x16c00;

SRAM\_BTCALL\_ASQT\_BUF\_SIZE = **0x2800**;

## 8.8 按键驱动内存分配

按键驱动是方案很多硬件相关的功能的大杂烩，所以其代码空间分配比较大，以便客户方案扩展：

常驻代码段：



```
SRAM_KEY_MESSAGE_RCODE_ADDR = 0x1f800;
```

```
SRAM_KEY_MESSAGE_RCODE_SIZE = 0x800;
```

常驻数据段:

```
SRAM_KEY_MESSAGE_DATA_ADDR = 0xbe00;
```

```
SRAM_KEY_MESSAGE_DATA_SIZE = 0x100;
```

驱动程序 BANK A 段:

```
SRAM_BANK_A_ADDR = 0x09000;
```

```
SRAM_BANK_A_SIZE = 0x400;
```

BANK 组号是 0x1x, 即地址空间为 0x1xxxxxxx。红色的 x 就是驱动程序 ID 号, 编号如下:

```
DRV_GROUP_STG_BASE = 0;
```

```
DRV_GROUP_STG_CARD = 1;
```

```
DRV_GROUP_STG_UHOST = 2;
```

```
DRV_GROUP_FAT32 = 3;
```

```
DRV_GROUP_EXFAT = 4;
```

```
DRV_GROUP_UD = 5;
```

```
DRV_GROUP_LCD = 6;
```

```
DRV_GROUP_FM = 7;
```

```
DRV_GROUP_KEY = 8;
```

```
DRV_GROUP_CCD = 9;
```

```
DRV_GROUP_AUDIO_DEVICE = 10;
```

```
DRV_GROUP_TTS = 11;
```

```
DRV_GROUP_BT = 12;
```

```
DRV_GROUP_SYS = 15;
```

驱动程序 BANK B 段, 按键驱动 BANK B\_1 存放的代码, 我们可以理解为是常驻代码段, 即在正常播放时几乎不会被切走:

```
SRAM_BANK_B_ADDR = 0x09400;
```

```
SRAM_BANK_B_SIZE = 0x800;
```

BANK 组号是 0x2x, 即地址空间为 0x2xxxxxxx。红色的 x 就是驱动程序 ID 号, 编号同 BANK A。

## 8.9 LED 驱动内存分配

LED 驱动只是小部分方案会用到，并且其功能非常简单，所需内存空间比较小：

**常驻代码段：**

```
SRAM_UI_RCODE_ADDR = 0x9C00;
```

```
SRAM_UI_RCODE_SIZE = 0x300;
```

**常驻数据段：**

```
SRAM_UI_DATA_ADDR = 0xbf00;
```

```
SRAM_UI_DATA_SIZE = 0x100;
```

**驱动程序 BANK A 段：**

```
SRAM_BANK_A_ADDR = 0x09000;
```

```
SRAM_BANK_A_SIZE = 0x400;
```

**驱动程序 BANK B 段：**

```
SRAM_BANK_B_ADDR = 0x09400;
```

```
SRAM_BANK_B_SIZE = 0x800;
```

很多方案都不需要 LED 驱动，所以 LED 驱动的常驻代码段和常驻数据段可以自主分配。

## 8.10 CCD 驱动内存分配

CCD (Case Common Driver) 驱动我们没有分配常驻代码段，并且数据段也只有很小的 64B，所以该驱动程序只能用来实现一些不常调用功能，比如 I2S PA 驱动，只需要初始化、设置音量等不常用接口，而像 LED 段码屏显示就不合适，因为有可能会 2Hz 闪烁等功能接口。

```
SRAM_RAM6_CCD_GLOBAL_DATA_ADDR = 0x9fc0;
```

```
SRAM_RAM6_CCD_GLOBAL_DATA_SIZE = 0x40;
```

## 8.11 栈空间分配

栈空间分配由 psp\_rel\include 下的 task\_info.xn 和 task\_manager.h 共同定义。

当前栈空间分配如分配图所示，总共占用了 8.75KB，这种分配情况适用标案所有功能都开启的情况，如果客户方案将某些功能关掉，那么其对应的栈空间就可以释放出来，可以加大其他任务的栈空间，或者

拿来做其他用途。

对于栈空间，我们在编写代码时要注意以下几点：

- 应用程序在运行时，函数调用对栈空间的消耗比较多，每个函数调用至少消耗 24 字节，所以在某些功能复杂的情景下，应该尽量让功能扁平化，以减少栈空间的使用，避免栈溢出
- 函数内的局部变量会占用栈空间，因此要避免在函数内申请很大的局部变量，而可以使用 `sys_malloc()` 动态申请一片较大的空间来满足这种需求

## 8.12 内存分配复用说明

上述列出的，基本上都是独占的内存空间，还有一些内存空间分配是采用分时复用的。在 CASE 层面上，内存空间分配复用情况具体如下：

**复用 1：**KEYTONE PLAY 子线程和 TTS PLAY 子线程复用了前台应用主线程的 **UI BANK** 段

这种属于多线程复用同一个 **BANK** 段，需要特别注意一点：较低优先级的线程所调用的 **UI BANK** 段，在不锁调度或关中断的情况下，不能使用 **XDATA** 数据段，也不能使用 **Const Data** 传参调用不是该 **UI BANK** 中的函数，否则就有出错的风险。

类似这种空间分配复用关系还有：

- **BANK A、BANK B：**即不同驱动程序都复用 **BANK A** 和 **BANK B**。

**复用 2：**FM 驱动常驻代码段复用前台中间件所有空间

这种属于分场景复用，因为在 **FM** 收音机场景下没有前台中间件，所以不用担心空间冲突。

类似这种空间分配复用关系还有：

- 前台 **Enhanced BANK** 段复用前台中间件所有空间：这是本地播歌场景下的分场景复用，刚进入时会先运行 **music scan** 进行扫描，扫描结束后才可调用 **Enhanced BANK** 段接口。
- **TTS** 驱动常驻代码段复用解码预处理模块空间：**TTS** 和解码是互斥的。

**复用 3：**引擎应用 **Enhanced1 BANK** 段复用中间件 **BANK** 段

这种属于共用复用，必须在满足这个条件的情况下才能使用：分时复用即第二种复用关系，或者复用的几个 **BANK** 段都是很少调用的，对运行效率没有什么影响。类似于复用 1，需要特别注意在多线程环境下的 **XDATA** 和 **Const Data** 传参等的使用。这种复用需要由系统 **BANK** 管理模块支持。

类似这种空间分配复用关系还有：

- 引擎应用 Enhanced2 BANK 段复用解码预处理模块的 BANK 段
- 前台 Enhanced BANK 段复用前台中间件的 BANK 段

## 8.13 内存分配调整说明

内存分配调整分为 2 次层次的调整：

5. 单个应用程序或驱动程序等程序实体内部的内存分配调整，比如将前台应用的 Control BANK 段长度调小一点，将省出来的空间给常驻代码段或常驻数据段用。  
这种调整，只需要调整单个程序实体的链接脚本 \*.xn，修改段起始地址和长度，并重新 make 一下就好了。
6. 在整个系统的层面上，将某些不用的模块独占的内存空间或者保留空间挪给其他模块使用，比如某个方案不需要 LED 段码屏显示，那么可以将 LED 驱动独占的常驻代码段分配给 KEY 驱动作为 .text1 常驻代码段。  
这种调整，一般需要调整 link\_base.xn 文件，并重新 make 整个 CASE。它需要保证调整后所有工程都 make 到位，如果需要 make PSP 的工程，或者蓝牙协议栈应用等没有开发源码的工程，那么就需要 FAE 协助了。

内存分配调整，必须确保所有参与空间调整的段都没有复用关系，否则可能会导致出错。比如按键驱动不能单独调整 BANK A 和 BANK B 的段大小，因为这是所有驱动程序复用的空间，并且系统同时会有多个驱动程序同时运行。

## 8.14 链接脚本 \*.xn 说明

### 8.14.1 输入段与输出段

以前台应用 COMMON 的常驻数据段为例进行说明：

```
.text1 RCODE_TEXT1_ADDR :  输出段
{
 KEEP(bt_common_manager_deal.o(.rcode))  输入段
 KEEP(app_timer_bank_single.o(.text .rodata))  输入段
 KEEP(common_rcp_dispatch.o(.rcode))
 KEEP(common_aset_main_deal.o(.rcode))
 KEEP(common_rcp_dispatch_block.o(.rcode))
 KEEP(common_rcp_dispatch_misc.o(.rcode))
 KEEP(common_rcp_func.o(.rcode))
 KEEP(bt_common_sppble.o(.rcode))
 KEEP(message_bank_send_btmanager.o(.text .rodata .xdata))
 KEEP(common_setdae.o(.rcode))
 . = SRAM_COMMON_BT_MANAGER_SIZE;
}
```

每种程序实体，都有对应的目标映像文件结构以及打包工具，指定了它允许包含哪些输出段，比如应用程序支持以下输出段：.text .text1 .text2 .data .bss BANK\*，其他的输出段将会被忽略掉。

而输入段是编译器编译生成的，根据具体编译器的规则来生成，各种输入段生成规则如下：

| 输入段名    | 说明                                                                                                            |
|---------|---------------------------------------------------------------------------------------------------------------|
| .text   | 代码段，包括代码中的常数（数字和字符串，注意是直接使用才会放到 .text 段中，直接放在函数结束后，即 jrc 指令后面）                                                |
| .rodata | 只读数据段，包括 const 全局变量                                                                                           |
| .data   | 初始化了的全局变量，包括没有带 const 声明的初始化字符串                                                                               |
| .bss    | 未初始化的全局变量（初始化为 0 也属于未初始化），.bss 段在加载时会被清为 0                                                                    |
| .xdata  | Bank 数据段，声明为 _BANK_DATA_ATTR_ 的变量（注：没有在 *.xn 中指定链接的 .xdata 会被自动放在 .data 段后面，也就是说等于占用了常驻数据空间）                  |
| 指定名字输入段 | 使用 __section__（“section”）重命名输入段名，然后在链接脚本 XN 文件中用 filename.o(section) 表示该输入段；这样我们就可以将同一个 *.o 的不同函数放在不同的输入段分别链接 |

## 8.14.2 链接脚本 \*.xn

链接脚本 \*.xn 就是程序实体工程内所有目标文件 \*.o 和 \*.a 进行地址重定位的脚本文件，它使用显式的或隐式的规则对所有输入段链接输出到特定的输出段中，并且根据程序实体映像文件结构体定义的需

要指定一些其他信息，包括程序实体的入口地址等。

一个工程支持由多个 \*.xn 进行链接，它们分别对某些目标文件的某些输入段进行链接，比如前台应用分为应用私有的链接脚本和 COMMON 独立的链接脚本 common\_front.xn，这样每个前台应用都只需要维护自己私有的链接脚本，并且如果修改了 COMMON 模块，也只需要修改 common\_front.xn。

链接脚本文件要点如下，以 AP BTPLAY 前台应用为例：

- 每个链接脚本都要包含系统空间分配脚本 link\_base.xn，该脚本放在 psp\_rel\include 目录下；\*.xn 直接应用 link\_base.xn 中定义的段地址和长度来链接，这样更新空间分配只需要更新 link\_base.xn 即可。

3 INPUT(link\_base.xn)

- 将 link\_base.xn 中定义的段地址和长度标识符，用更恰当和更简短的标识符重命名一下。

```
17 BANK_CONTROL_1_ADDR_BASE = (AP_BANK_FRONT_CONTROL_1 << 24) + SRAM_AP_BANK_FRONT_CONTROL_ADDR;
18 BANK_UI_1_ADDR_BASE = (AP_BANK_FRONT_UI_1 << 24) + SRAM_AP_BANK_FRONT_UI_ADDR;
```

```
26 BANK_CONTROL_SIZE = SRAM_AP_BANK_FRONT_CONTROL_SIZE;
27 BANK_UI_SIZE = SRAM_AP_BANK_FRONT_UI_SIZE;
```

- 指定入口地址，对于应用程序来说，入口函数是运行时库 ctor.o 中的 \_\_start() 函数。

35 ENTRY(\_\_start)

- 常驻代码段、常驻数据段编写规则，对于数据段，可以将 .bss 放在前面，也可以将 .data 放在前面，两者一般是紧接着链接，所以后面的输出段不用指定起始地址。

```
/* 常驻代码段: .text */
.text RCODE TEXT ADDR : ———> 指定输出段名和起始地址
{
 KEEP(sys_op_entry.o(.text)) ———> KEEP 确保固定链接进来
 ...
 KEEP(common_view_manager.o(.rcode))

 btplay_message_loop.o(.text)
 btplay_message_handle.o(.rodata.ke_maplist .rodata.se_maplist)
 . = SRAM_AP_FRONT_RCODE_SIZE; ———> 保留大小，确保段不会越界
}

/* 未初始化的全局数据段，系统会自动清零*/
.bss RDATA_DATA_ADDR :

/* 初始化了的全局数据段*/
.data :
```

- BANK 段编写规则，第一个 BANK 编写方式如下，第一个 BANK 号也可以不是 0，如果是 10，那么起始地址指定表达式为 . = BANK\_CONTROL\_1\_ADDR\_BASE + AP\_BANK\_SPACE \* 10; :

```

. = BANK CONTROL 1 ADDR BASE;
/*OFFSET为ui_bank 1组的实际物理地址*/
OFFSET = . & 0x3ffff;
BANK_CONTROL_1_0 :
{
 ctor.o(.text .rodata)
 btplay_main.o(.text .rodata)
 btplay_message_loop.o(.bank_2)
 . = BANK CONTROL SIZE;
}

```

第一个BANK段逻辑地址  
OFFSET 表示BANK段物理地址  
BANK 输出段名必须以BANK开头  
保留大小，确保段不会越界

后续 BANK 编写方式如下：

```

. = ((. + AP_BANK_SPACE) & ~(AP_BANK_SPACE - 1)) + OFFSET;
BANK_CONTROL_1_1 :
{
 btplay_message_handle.o(.text .rodata)
 . = BANK_CONTROL_SIZE;
}

```

BANK段名区别于上个BANK即可  
逻辑地址指向下个BANK：高14bit加1

- \*.xn 文件中的输入段描述支持通配符，比如 \*(.bss) 表示所有目标文件尚未链接的 .bss 段，rcode\_\*.o(.text) 表示所有前缀为 rcode\_ 的\*.o 目标文件的尚未链接的 .text 段。

## 8.15 Makefile 说明

所有工程和模块的编译链接，都由 Cygwin Make 工具完成，它需要 Makefile 脚本来指定 Make 规则和选项；Makefile 脚本支持嵌套引用。

Makefile 要点如下，以 AP BTPLAY 前台应用为例：

- 引用其他 Makefile，将公共变量名称和基本 Make 规则引用进来。

```

34 #把公共路径加载进来
35 include ../../../../cfg/common_path
36 #把公共定义makefile加载进来
37 include ../../../../cfg/rules.mk

```

- 指定映像文件的名称，包括 \*.exe、\*.map、\*.info、\*.lst、\*.ap 等名称。

```

40 #Name of application 开发人员需修改处(modify-1)
41 IMAGENAME = btplay

```

- 指定源文件目录，以获取源文件列表。

```

43 #所要编译的源文件的存放位置,开发人员需修改处(modify-2)
44 SRC = $(CASE)/ap
45 SRCDIR_16 = $(SRC)/ap_btplay
46 SRCDIR_32 =
47 SRCDIR_16_O2 =

```



- 引用栈空间描述脚本，给后面 AP\_PRIO 等变量赋值。

```
50 include ../../../../psp_rel/include/task_info.xn
```

- 根据上述的源文件目录，填写下述搜索路径。

```
61 #指定依赖过程的文件的搜索路径，把源文件的路径写上即可
62 VPATH = $(SRCDIR_16) $(SRCDIR_32) $(SRCDIR_16_O2) $(OBJ_DIR)
```

- 定义映像文件名称变量。

```
68 #指定结果文件的名称
69 IMAGE_ELF = $(IMAGENAME).exe
70 IMAGE_LST = $(IMAGENAME).lst
71 IMAGE_INFO = $(IMAGENAME).info
72 MAP = $(IMAGENAME).map
73 TARGET_AP = $(IMAGENAME).ap
74 TARGET_EXT = exe lst map ap info
75 TARGET_FILES = $(foreach n, $(TARGET_EXT), $(IMAGENAME).$(n))
```

- 指定编译头文件搜索路径和链接库文件搜索路径。

```
81 #指定程序中引用的头文件的搜索路径，一般源码include时只写上头文件的名称，没有路径信息。这时就：
82 INCLUDE = -I$(PSP_REL)/include/ucos -I$(PSP_REL)/include -I$(CASE)/inc
83 #指定链接时搜索的路径
84 LINCLUDE = -L$(OBJ_DIR) -L$(PSPLIBDIR) -L$(ENHANCEDIR) -L$(CASELIBDIR) -L$(PSP_REL)/include
```

- 指定该工程的链接脚本，支持由多个脚本进行链接。

```
86 #指定自定义链接脚本的名称，开发人员需修改处(modify-4)
87 LD_SCRIPT = ap_btplay.xn ../../common/common_front.xn
```

- 指定链接器链接选项。

```
92 #链接命令行选项
93 LD_OPTS_1 = $(LINCLUDE) -G0 -gc-section -T $(LD_SCRIPT) $(OFORMAT)
94 LD_OPTS_2 = -o $(IMAGE_ELF) -Map $(MAP)
```

- 指定栈空间，只有应用程序才需要指定，会抽取到 \*.ap 的头部，在创建应用程序主线程时传递到操作系统中，以便做栈溢出检测监控。

```
99 AP_PRIO = $(AP_FRONT_LOW_PRIO)
100 AP_STK_POS = $(AP_FRONT_LOW_STK_POS)
101 AP_STK_LEN = $(AP_FRONT_LOW_STK_SIZE)
```

- 定义源文件列表和目标文件列表变量。

```
107 #获得.c后缀源码
108 SRC_C_16 = $(foreach dir, $(SRCDIR_16), $(wildcard $(dir)/*.c))
109 #转换为.o格式文件名称，不带路径信息
110 OBJ_C_16 = $(notdir $(patsubst %.c, %.o, $(SRC_C_16)))
```

- Make 规则，生成 \*.o , \*.exe , \*.ap 等目标文件。



```

139 #make 目标
140 .PHONY : all → make 指定的目标
141
142 #dump出lst文件
143 all : creat_dir $(IMAGE_ELF) → *.exe 是标准的ELF映像文件
144 $(OBJDUMP) -D $(IMAGE_ELF) > $(IMAGE_LST)
145 $(READELF) -a $(IMAGE_ELF) > $(IMAGE_INFO)
146 $(AP_BUILDER) $(IMAGE_ELF) $(TARGET_AP) $(AP_PRIO) $(AP_STK_POS) $(AP_STK_LEN)
147 cp $(TARGET_AP) $(OBJECT_BIN_PATH) → *.ap 是US282A 平台特定的映像文件
148
149 creat_dir:
150 mkdir -p $(OBJ_DIR) → 将*.ap拷贝到fwpkg/ap目录下
151
152 #链接过程
153 $(IMAGE_ELF) : $(OBJ)
154 $(LD) $(LD_OPTS_1) $(LD_OPTS_2)
155
156 #编译过程
157 $(OBJ_C_16) : %.o : %.c
158 $(CC) $(CC_OPTS_O0_16) -mlong-calls -o $(OBJ_DIR)/$@ $<
159 @echo

```

- Make 规则，make clean 和 make clean\_target 。

```

181 #删除结果文件
182 .PHONY : clean → 将所有目标文件都删掉，会重新生成
183 所有目标文件，比较慢
184 clean :
185 rm -f $(LINKDIR)/$(OBJ)
186 rm -f $(LINKDIR)/$(TARGET_FILES)
187 rm -rf $(OBJ_DIR)
188
189 #只删除 target 文件，保留*.o，以避免没修改的源文件也要重编译
190 .PHONY : clean_target → 只删除结果映像文件，只重新生成
191 映像文件，速度快
192 clean_target :
193 rm -f $(LINKDIR)/$(TARGET_FILES)

```

## 8.16 \*.map 和 \*.info 解读

一般 Makefile 中有 Make 规则会从 \*.exe 中提取一些链接信息生成 \*.map 和 \*.info 文件，通过这两个文件我们可以了解实际空间分配结果。

### 8.16.1 \*.map 解读

\*.map 文件包含应用程序或驱动程序等映像文件的内存分配信息，可以用来具体了解各个接口和变量的链接地址及其大小，映像文件的各种输出段的信息，包括链接地址、段大小、是否越界、段内包含的代码和数据等。

## 8.16.2 \*.info 解读

\*.info 文件则仅仅以总结列表形式列出 \*.exe 映像文件中的所有输出段的概要信息，包括链接地址、段大小等信息。

## 8.17 BANK 使用考量

1) 充分考虑 BANK 机制的特点：

- BANK DATA 在生命周期中不能被切换出去。
- 同一个 BANK 组内不同 BANK 的函数互相调用会发生 BANK 切换，这种情况除了来回增加 2 次 BANK 切换外，还有一个问题需要特别注意，就是常量数据传参，包括 `const data` 和函数内的常量数据，不能以指针方式传递参数，因为函数调用后发生 BANK 切换，常量数据也被切走了，也就是指针指向的内容被冲掉了，这样会导致参数使用错误。

2) 满足 BANK 切换目标：

- 1) 对于引擎应用，原则上要做到在后台应用处于正常状态并且没有收到应用私有消息的情况下不会发生 BANK 切换。
- 2) 对于前台应用，原则上要做到在没有按键动作且没有收到应用私有消息的情况下不会发生 BANK 切换。

3) 软定时器 HANDLE 切 BANK 要求：尽量避免将软定时器的 HANDLE 放在消息处理循环所在 BANK 组上不同 BANK 上，或者应该尽量将所有软定时器的 HANDLE 集中存放在同一个 BANK 上。

## 8.18 系统堆空间

操作系统提供了 0x580 字节的系统堆空间，为整个系统共用，目前使用的并不多，应用程序可以临时申请一片较大的内存。

我们现在也提供了标准的动态内存管理接口，接口如下：

**申请堆空间：** `void* sys_malloc(uint32 size);`

如果返回 NULL 则表示找不到这么大一块连续的空间了。

**释放堆空间：** `void sys_free(void* addr);`

使用系统堆空间要注意申请与释放匹配，否则会导致内存泄漏。

## 8.19 VRAM 空间分配

VRAM 是一块从 SPI Nor Flash 中预留的非易失存储器空间，容量较大，但是读写速度很慢。在对速度要求不高的情景下，可以用来存放应用环境变量和临时数据缓冲。

US282A 总共有 64KB 的 VRAM 空间，为 PSP 和 CASE 共用。

US282A VRAM 采用全新的管理方式，其 32 位地址组织方式如下：

**高 16 位的 INDEX（每个 INDEX 的地址空间都是 64KB） + 低 16 位的偏移地址**

每个 INDEX 实际长度可以是 1 Byte ~ 64KB，但是一般建议是 512 字节以内。

US282A 最多支持使用 64 个 INDEX，其中 0x00200000 ~ 0x003f0000 分配给 PSP 使用，定义在 `psp_rel\include\vm_fwsp_def.h` 中，原则上 CASE 不能使用这部分 INDEX，但是如果某个客户方案明确不需要这些 INDEX 使用者模块，那么 CASE 就可以拿来使用，这一点需要在 FAE 支持下进行。

CASE 使用的 INDEX 定义在 `case\inc\vm_def.h` 中，INDEX 范围为 0x00000000 ~ 0x001f0000。

VRAM 读写接口定义如下：

VRAM 读：int sys\_vm\_read(void \*buf, uint32 offset, uint32 len); VRAM 读的 offset 的高 16 位是 0x00000000 ~ 0x003f0000，低 16 位偏移地址可以是该 INDEX 实际长度以内任意偏移，即可以读取任意偏移位置任意长度的数据，当然必须保证不会超出该片 VRAM 的有效边界。

VRAM 写：int sys\_vm\_write(void \*buf, uint32 offset, uint32 len); VRAM 写的 offset 的高 16 位是 0x00000000 ~ 0x003f0000，低 16 位偏移地址必须是 0，长度必须是实际长度，也就是说，每个 INDEX 对应的 VRAM 写必须一次性写完。

所以，如果有个数据长度是 1KB，但是如果内存缓冲区只能得到 512B，需要分为 2 次写，那么必须使用 2 个 INDEX 来写这 1KB 的数据，每个 INDEX 的 VRAM 实际长度为 512 字节。当然，VRAM 读也要分为 2 个 INDEX 用 2 次 VRAM 读完成。

注意：VRAM 不允许在一次上电进行超高频度地擦写，否则可能会导致 VRAM 空间变成坏块，而造成样机永久性损坏，必须更换 SPI Nor Flash 才能修复过来。

## 8.20 FAQ

### 8.20.1 如何增加一个 BANK?

从 BANK 机制上看, 增加一个 BANK, 就是要增加一个除了 BANK 号不一致外其他都一致的逻辑内存代码段, 我们一般是在最后一个 BANK 的后面, 增加一个 BANK 号加 1 的 BANK。

那么在链接 \*.xn 上的编写方法如下:

以 common\_front.xn 为例:

```
. = ((. + AP_BANK_SPACE) & ~(AP_BANK_SPACE - 1))) + OFFSET;
```

```
BANK_UI_1_38 :
```

```
{
 KEEP(common_seteq.o(.text .rodata .xdata))
 KEEP(common_setdae_init.o(.text .rodata .xdata))
 . = BANK_UI_SIZE;
}
```

要增加一个 BANK BANK\_UI\_1\_39, 编写脚本如下:

```
. = ((. + AP_BANK_SPACE) & ~(AP_BANK_SPACE - 1))) + OFFSET;
```

```
BANK_UI_1_39 :
```

```
{
 KEEP(common_test.o(.text .rodata .xdata))
 . = BANK_UI_SIZE;
}
```

蓝色部分, OFFSET 就是低 18 位物理内存地址, AP\_BANK\_SPACE 的值是 0x40000, 即低 18 位全 0, 运算过程是高 14 位加 1, 即 BANK 号加 1, 然后低 18 位先清 0, 然后再加上低 18 位的物理内存地址, 刚好是下一个 BANK 的逻辑起始地址。

红色部分, 后面的 39 比 38 大 1, 这仅仅对用户阅读代码有用, 链接器并不理会它。所以, 即使改为 BANK\_UI\_1\_99, 也没有任何问题。

粉色部分, 是为了让链接器能够主动监测 BANK 段是否溢出了, 即是否 BANK 段大小超出了该 BANK 所分配的物理内存大小。

## 8.20.2 如何将相应的代码放置到对应的 BANK?

源文件编译为目标文件 \*.o，它包含了多个输入段，链接器是以输入段为对象进行内存分配的。

比如 common\_test.o 有以下几个输入段：.text、.bss、.bank2。

其中 .bank2 是通过 \_\_section\_\_(“.bank2”) 对某个或某几个函数或全局变量进行声明才生成的。

```
void __section__(“.bank2”) my_func1(void)
{
}
```

然后，我们可以将 .bank2 放到不同于 .text 的 BANK 段中，比如：

```
. = ((. + AP_BANK_SPACE) & ~(AP_BANK_SPACE - 1))) + OFFSET;
```

```
BANK_UI_1_39 :
```

```
{
```

```
 KEEP(common_test.o(.text .rodata .xdata)) //common_test.o 没有 .rodata 和 .xdata 段,没有关系,
```

这样写没有什么负面影响

```
 . = BANK_UI_SIZE;
```

```
}
```

```
. = ((. + AP_BANK_SPACE) & ~(AP_BANK_SPACE - 1))) + OFFSET;
```

```
BANK_UI_1_40 :
```

```
{
```

```
 KEEP(common_test.o(.bank2))
```

```
 . = BANK_UI_SIZE;
```

```
}
```

## 8.20.3 如何增加常驻代码段?

US282A 的应用程序、驱动程序和前台中间件等都支持最多 3 个常驻代码段，所以如果 .text 段已经放不下了，我们又能找出一片没用的空间作为扩展的常驻代码段，那么可以将新增的输出段命名为 .text1（前台应用的 COMMON 模块已占用）或 .text2，然后将某些常驻代码段放到该输出段里面。

## 8.20.4 如何处理空间越界问题？

链接或 maker 为 \*.ap/\*.drv 时偶尔会发生各种空间越界报错，现列举如下：

1. 如果我们在输出段的末尾添加了 `. = SEG_SIZE;`，那么如果有发生段越界，就会打印如下错误：

```
link_base.xn:371 cannot move location counter backwards (from 00000000400a0c74 to 00000000400a0c00)
```

我们主要看红色的结束地址，它是段起始地址加上段长度得到的地址。这是一个前台应用 Control BANK 的地址，BANK 号为 2，即取高 14 位的低 6 位，也可以用 0a 这个字节直接除以 4 得到，所以断定是 Control BANK 2 这个段代码太多了。

这种情况，就需要将该输出段的实际长度调整小一点，如果是 BANK 段越界了，可以将部分代码挪到另一个 BANK 去。

2. 有时候有些输出段我们忘记在末尾添加 `. = SEG_SIZE;` 了，那么如果在这个工程里面，它恰好覆盖了别的输出段，那么会报错：

```
section BSMANAGER_DATA [000000009fc26780 -> 000000009fc267ff] overlaps section
```

```
RCP_COMMAND_BUFFER [000000009fc26700 -> 000000009fc267d7]
```

BSMANAGER\_DATA 在 \*.xn 中位置较 RCP\_COMMAND\_BUFFER 后一点，前者的地址空间覆盖了后者的地址空间。

这种情况，可以将后者的实际长度调小一点，或者将前者的整个地址空间往后挪一点。

3. 我们在 maker 工具上添加了 BANK 段越界检测，即使在 \*.xn 中没有添加 `. = SEG_SIZE;`，在 maker 时也会将越界了的 BANK 段检测出来，这是因为 maker 工具已经固化了 BANK 段的起始地址和长度信息，它可以根据这些信息对 BANK 段进行地址合法性检查。检测到非法地址会打印警告：

warning bank :

front control bank number[2] is too large

max size: 0x800

根据这样的信息，我们直接查看 \*.xn 的 Control BANK 2 。

## 8.20.5 如何处理栈越界？

US282A 在操作系统中增加了栈越界监控模块，实时的监控是否发生了栈溢出问题，检测到后会打印出来，因为这一点，我们也要求在开发时要把打印调试用起来。

## 9 应用管理器介绍

AP Manager，应用管理器，专门负责创建和回收其他应用程序，以及一些其他协调应用程序的任务。另外，作为操作系统第一个加载的应用程序，它还负责完成一些在 CASE 中尽可能早执行的初始化动作。

### 9.1 AP Manager 的地位与功能

US282A 是一个多任务抢占式调度系统，其内核是 UCOS II，调度基本单元是任务。我们说的 AP/应用程序会被装载为进程/主线程，在进程/主线程中又可以创建子线程，这是 POSIX 风格的多线程架构。而线程在系统中是用 UCOS II 中的任务来实现的。

所以，我们可以把 US282A 看成是 2 个层次的“多任务”调度系统：

1. 应用程序层次：以 ap\_manager 为调度核心，ap\_manager 负责创建和回收其他应用程序。
2. UCOS II 任务层次：由 kernel UCOS II 为调度核心。

设备上电启动，加载操作系统并运行，操作系统加载 ap\_manager 并运行，再由 ap\_manager 创建 ap\_config 开机，之后 ap\_manager 进入应用程序调度循环中。

Ap\_manager 作为应用程序的调度核心，具体的功能如下：

1. 完成一些在 CASE 中尽可能早执行的初始化动作。
2. 创建 ap\_config，执行开机流程。
3. 接收前台应用创建其他前台应用的异步消息，异步创建前台应用。
4. 接收前台应用创建引擎应用的同步消息，同步创建引擎应用。
5. 接收前台应用创建蓝牙协议栈应用的同步消息，同步创建蓝牙协议栈应用。
6. 接收前台应用杀死引擎应用/蓝牙协议栈应用的同步消息，同步杀死引擎应用/蓝牙协议栈应用。

### 9.2 AP Manager 的设计要点

应用的启动：由操作系统创建，操作系统启动并完成初始化后，就会创建 ap\_manager 应用。

应用的退出：AP Manager 应用永远不退出，在有电情况下一直在内存中运行。



Ap\_manager 是一个很特殊的应用，功能简单，为了尽量少链接 COMMON 的代码，Ap\_manager 尽量避免调用 COMMON 的接口，除了引擎管理小接口、配置项读取接口，SPEAKER 相关接口等很独立的接口外。

比如，bool applib\_init(uint8 app\_id, app\_type\_e type) 和 int send\_async\_msg(uint8 app\_id, msg\_apps\_t \*msg) 等接口都是直接在 AP Manager 自己实现，而不是使用 COMMON 的公共接口。

又比如，AP Manager 获取私有消息，是直接调用 sys\_mq\_receive (MQ\_ID\_MNG, &pri\_msg) 获取，而非调用 COMMON 的公共接口。

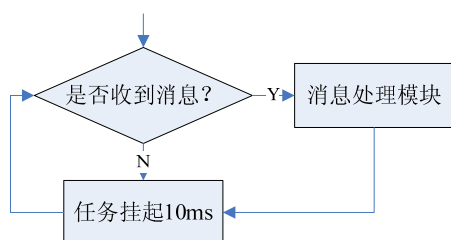
AP Manager 仅仅承担着应用管理的任务，该任务的时效性不高，所以我们把 ap\_manager 的任务优先级定得比较低。

### 9.3 AP Manager 初始化

1. 安装 nor\_ud.drv 驱动，这样在 AP Manager 就可以进行 SPI NOR Flash 读写了。
2. 初始化所有应用的公共数据区以及 g\_app\_info\_state 结构体。
3. 打开 config.bin 文件，并设置 UART 打印使能。
4. 配置 26MHz 晶振匹配电容。
5. 安装按键驱动，并做必要的初始化设置。
6. 创建一个子线程来初始化内部 PA，这样能让内部 PA 初始化与其他初始化流程并行走。该动作会安装 aud\_dev.drv 驱动，并使能外部 PA 和打开外部 PA。
7. AP Manager 自身初始化。

### 9.4 AP Manager 主循环

Ap\_manager 的主循环功能简单，主流程如下：



AP Manager 只接受应用私有消息，然后调用 `manager_msg_callback(&pri_msg)`；进行分发处理。

## 9.5 创建应用程序的流程

对于 AP Manager 来说，创建应用程序有 3 种方式，或者说通过 3 个不同的消息：

- **MSG\_CREAT\_APP** 异步方式创建前台应用，这是因为前台应用同时只能有一个在运行，并且一般是由前台应用发起创建另一个前台应用，所以必须采用异步方式创建，即先告诉 AP Manager 要创建另一个前台应用，然后 AP Manager 等待当前的前台应用退出，之后才创建另一个前台应用，具体流程如下：
  - 1) 前台应用发送 **MSG\_CREAT\_APP** 给 AP Manager。
  - 2) AP Manager 收到消息后，调用 `wait_ap_exit()` → `libc_waitpid()` 等待当前前台应用退出，AP Manager 也进入睡眠状态。
  - 3) 当前前台继续执行，完成应用程序退出流程，这时会唤醒 AP Manager。
  - 4) AP Manager 从 `wait_ap_exit()` 返回，调用 `sys_free_ap(0)`；重置前台应用运行环境，然后调用 `sys_exece_ap(ap_name, 0, (int32) ap_param)`；创建另一个前台应用。
- **MSG\_CREAT\_APP\_SYNC** 同步方式创建引擎应用，一般是由前台应用发起，在此之前前台应用保证已经没有引擎应用存在，所以可以采用同步方式创建，具体流程如下：
  - 1) 前台应用发送 **MSG\_CREAT\_APP\_SYNC** 给 AP Manager。
  - 2) AP Manager 收到消息后，调用 `libc_waitpid(0, 1)`；重置进程运行环境，再调用 `sys_free_ap(1)`；重置引擎应用运行环境，最后调用 `sys_exece_ap(ap_name, 1, (int32) ap_param)`；创建引擎应用。
  - 3) 创建完毕后，调用 `libc_sem_post(pri_msg->sem)`；回应前台应用，前台应用继续运行。
- **MSG\_CREAT\_APP\_EXT\_SYNC** 同步方式创建蓝牙协议栈应用，一般是由前台应用发起，在此之前前台应用保证已经没有蓝牙协议栈应用存在，所以可以采用同步方式创建；之所以用一种新方式来创建蓝牙协议栈，主要是传参的需要，创建前台应用和引擎应用都只能传递 1 个字节的参数，而创建蓝牙协议栈应用需要传递一些额外的参数，必须采用新的传参方式；具体流程如下：
  - 1) 前台应用发送 **MSG\_CREAT\_APP\_EXT\_SYNC** 给 AP Manager。
  - 2) AP Manager 收到消息后，调用 `libc_waitpid(0, 1)`；重置进程运行环境，再调用 `sys_free_ap`

(AP\_BTSTACK\_FILE); 重置蓝牙协议栈应用运行环境，最后调用 `sys_exece_ap(ap_name, AP_BTSTACK_FILE, (int32) ap_param)`; 创建引擎应用。

- 3) 创建完毕后，调用 `libc_sem_post(pri_msg->sem)`; 回应前台应用，前台应用继续运行。

## 9.6 杀死应用程序的流程

前台应用的杀死实际上是伴随着另一个前台应用的创建而主动杀死的，这里就不用说了。

引擎应用和蓝牙协议栈应用的杀死是用同一个消息实现的，`MSG_KILL_APP_SYNC` 同步杀死引擎应用或蓝牙协议栈应用，具体流程如下：

- 1) 前台应用发送 `MSG_KILL_APP_SYNC` 消息给 AP Manager。
- 2) AP Manager 收到消息后，转发 `MSG_APP_QUIT` 强制应用程序退出的异步消息给引擎应用或者蓝牙协议栈应用，并调用 `wait_ap_exit()` → `libc_waitpid()` 等待对方退出，AP Manager 也进入睡眠状态。
- 3) 引擎应用或者蓝牙协议栈应用收到该消息后，执行应用程序退出流程，最终唤醒 AP Manager。
- 4) AP Manager 从 `wait_ap_exit()` 返回，调用 `sys_free_ap(1)`; 或者 `sys_free_ap(AP_BTSTACK_FILE)`; 重置运行环境。
- 5) 杀死完毕后，调用 `libc_sem_post(pri_msg->sem)`; 回应前台应用，前台应用继续运行。

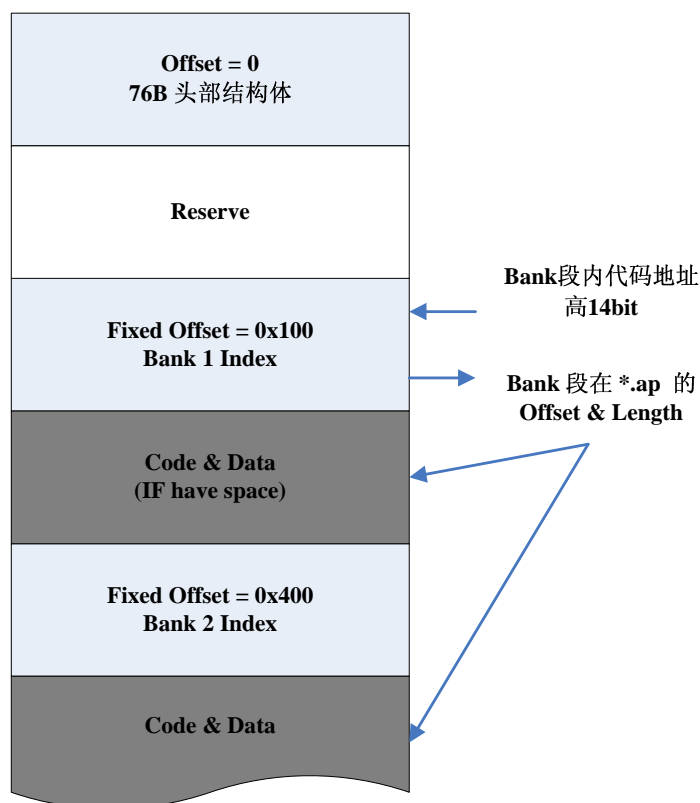
## 10 前台应用详解

前台应用，指负责用户交互的应用程序，另外由于系统内存资源有限，我们把系统管理的功能也放到前台应用中，由前台应用的 COMMON 模块实现。

### 10.1 应用程序 \*.ap 文件结构

应用程序通过 make 的编译链接，生成 \*.exe 结果文件，但是因为 \*.exe 文件包含了很多对运行没用实际作用的调试信息等，这也就浪费了很多空间；并且 \*.exe 这种标准的 ELF 文件结构的解释也比较复杂。所以，我们提供应用程序的 maker 工具，去掉多余的调试信息，仅仅抽取我们需要的代码段和数据段，并且以一种非常简洁的文件结构进行打包，maker 工具把应用程序打包为 \*.ap 文件。

这种应用文件的结构示意图如下：



从上面的示意图可以看到，应用程序文件结构将 BANK 段索引表放在扩展结构中，位置是固定的，有多少个 BANK 段索引表根据应用程序本身需求而定。为了最大限度减少固件大小，我们将 BANK 段索引表的剩余空间，也拿来存放代码段或数据段，只要代码段或数据段的长度小于剩余空间，那么就可以填充到该空间。

**Bank 段的装载：**从 Bank 段内代码地址高 14bit 获取对应的索引表和 Index 号，就可以得到 Bank 段在 \*.ap 的 Offset & Length，然后就可以装载到物理内存中。

应用程序的头部包含了很多信息，当操作系统加载应用程序时，根据头部结构的信息装载代码和数据到内存中，并建立应用程序运行环境，最终再跳转到入口函数去，头部结构定义如下：

```
52 typedef struct
53 {
54 unsigned char file_type;
55 unsigned char ap_type;
56 unsigned char major_version;
57 unsigned char minor_version;
58 unsigned char magic[4];
59 unsigned int text_offset;
60 unsigned int text_length; → .text 输出段在*.ap中的索引
61 unsigned int text_addr;
62 unsigned int text1_offset;
63 unsigned int text1_length; → .text1 输出段在*.ap中的索引
64 unsigned int text1_addr;
65 unsigned int text2_offset;
66 unsigned int text2_length; → .text2 输出段在*.ap中的索引
67 unsigned int text2_addr;
68 unsigned int data_offset;
69 unsigned int data_length; → .data 输出段在*.ap中的索引
70 unsigned int data_addr;
71 unsigned int bss_length; → .bss 输出段的描述
72 unsigned int bss_addr;
73 unsigned int entry; → 应用程序的入口函数 __start()
74 task_attr_t task_attr; → 主线程栈空间描述
75 } ap_head_t;
```

说明：

- .text .text1 .text2 是常驻代码段，在应用程序加载时装载到内存中；.data 是初始化的常驻数据段，在应用程序加载时装载到内存中；.bss 是尚未初始化的常驻数据段，在应用程序加载时清为 0。
- entry 是运行时库 ctor.o 的 \_\_start() 函数，是在链接脚本 \*.xn 文件中用 ENTRY(\_\_start) 声明的。
- task\_attr 是主线程栈空间描述，也是在 Makefile 文件中对 AP\_PRIO 等定义得到的。

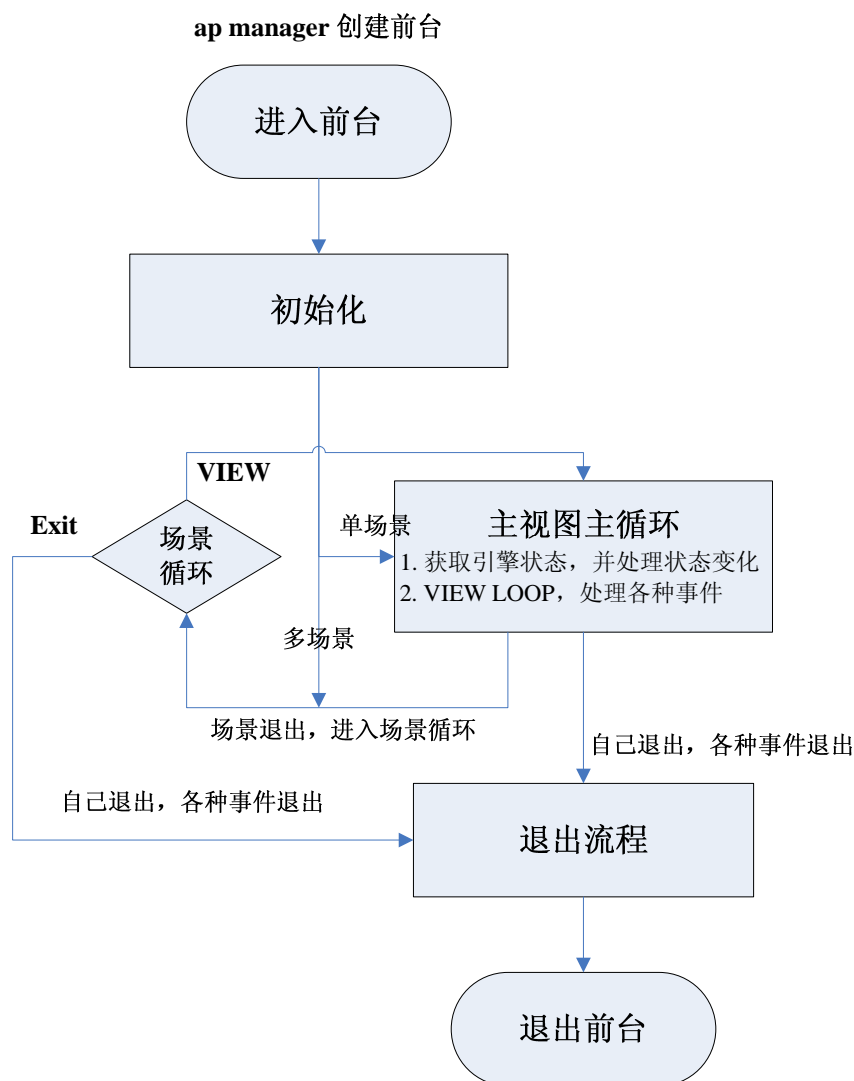
## 10.2 前台应用工程概述

### 10.2.1 工程组成

前台应用工程包括以下若干部分：

- 前台应用的业务逻辑相关源文件和头文件
- 前台应用的链接脚本 \*.xn
- 前台应用的 Makefile 脚本
- COMMON 模块及其链接脚本
- 运行时库 ctor.o，包含 \_\_start() 入口函数
- 各种其他程序实体的 API 接口库，比如操作系统、按键驱动等
- 公共头文件，包括 case\inc 和 psp\_rel\include 目录下的头文件
- 其他库文件、源文件和头文件，及其链接脚本，包括 Enhanced 模块及其链接脚本等
- 其他数据文件和配置文件，包括 config.bin、按键音文件、TTS 资源文件等

## 10.2.2 应用的一般流程



## 10.2.3 初始化流程

前台应用装载之后，跳转到 `ctor.o` 中的 `__start()` 函数运行，在 `ctor.o` 中会创建应用程序的主线程，然后跳转到 `main()` 函数运行，接下来的初始化流程如下，以 AP MUSIC 前台应用为例：

1. `applib_init(APP_ID_MUSIC, APP_TYPE_GUI);` 向 APPLIB 模块注册该前台应用；该接口最先调用。
2. `applib_message_init();` 消息模块初始化。
3. `init_app_timers(music_app_timer_vector, COMMON_TIMER_COUNT + APP_TIMER_COUNT);` 初

始化软定时器。

4. `sys_timer_init()`; 创建 COMMON 模块的系统管理定时器; 该接口必须在 `init_app_timers()` 之后调用。
5. `com_rcp_init()`; 初始化 RCP 解析器。
6. `music_rcp_var_init()`; → `com_rcp_set_callback(music_rcp_cmd_cb_tbl, music_get_global_rcp_info)`; 向 RCP 解析器注册 RCP 事件映射表; 该接口必须在 `com_rcp_init()` 之后调用。
7. `com_view_manager_init()`; 初始化视图管理器。

另外, 可以在上述合适位置读取保存在 VRAM 的环境变量、初始化全局变量等。

每个前台应用都必须定义 `comval_t` 系统环境变量, 并调用 `com_setting_comval_init(&g_comval)`; 接口读取保存在 VRAM 的系统环境变量, 该接口还将 `g_comval` 地址注册到 COMMON 模块中, COMMON 模块就可以使用公共指针 `sys_comval` 来访问系统环境变量了。

完成上述初始化步骤后, 就可以进入视图调度和视图主循环, 执行真正的业务逻辑。

## 10.2.4 退出流程

当前台应用收到应用切换的事件后, 就会走退出流程, 以 AP MUSIC 前台应用为例:

1. `com_ap_switch_deal(retval)`; 调用应用切换接口, 向 AP Manager 发送创建下一个前台应用的消息, 然后 AP Manager 就会等待本前台应用退出。
2. `music_rcp_var_exit()`; → `com_rcp_set_callback(NULL, NULL)`; 向 RCP 解析器注销 RCP 事件映射表。
3. `com_view_manager_exit()`; 销毁视图管理器。
4. `sys_timer_exit()`; 杀死 COMMON 模块的系统管理定时器。
5. `applib_quit()`; 向 APPLIB 模块注销该前台应用。

另外, 可以在上述合适位置将环境变量保存到 VRAM 中, 以供下次重新进入该前台应用时恢复环境。

完成上述退出步骤后, 很快就运行到 `main()` 的结束位置, 从 `main()` 退出, 就会回到 `ctor.o` 中, 销毁主线程, 这样该前台应用就彻底销毁了。



## 10.2.5 主视图/场景调度

在前面 COMMON 模块我们已经介绍了前台应用的用户交互框架——VIEW 机制。前台应用主要就是由一个或多个主视图/场景构成的，每个主视图都有一个主视图主循环。

（这里保留场景的说法，是历史遗留原因，以前的方案都叫场景，并且 US282A 的代码中还有挺多地方保留 SCENE 这样的标识符。）

如果前台应用有多个主视图，那么前台应用就得有一段简单的主视图/场景调度程序，即在初始化流程完成之后，就进入主视图/场景调度程序中，程序结构大致如下，以 AP RADIO 前台应用为例：

```
while (g_radio_scene != FM_SCENE_EXIT)
{
 switch (g_radio_scene)
 {
 case FM_SCENE_PLAYING:
 scene_result = radio_scene_playing();
 if (scene_result == RESULT_AUTO_SEARCH)
 {
 //进入搜台主视图
 g_radio_scene = FM_SCENE_SEARCH;
 }
 else
 {
 g_radio_scene = FM_SCENE_EXIT;
 }
 break;
 case FM_SCENE_SEARCH:
 scene_result = radio_scene_auto_search(DIR_UP);
 if ((scene_result > RESULT_COMMON_RESERVE) && (scene_result != RESULT_STOP_SEARCH))
 {
 //退出 RADIO
 g_radio_scene = FM_SCENE_EXIT;
 }
 else
 {
 //回到播放主视图
 g_radio_scene = FM_SCENE_PLAYING;
 }
 break;
 default:
 //退出场景调度
 g_radio_scene = FM_SCENE_EXIT;
 break;
 }
}
```

进入 radio\_scene\_playing() 和 radio\_scene\_auto\_search() 之后，就创建主视图，然后进入主视图主循环，退出主视图时最好将主视图删除掉，这样会及时释放主视图占用的资源；如果没有主动删除主视图，我们在切换到新的主视图或者销毁视图管理器时也会自动删除所有视图，将资源销毁掉。

## 10.2.6 主视图主循环

创建了主视图，并做完业务逻辑初始化之后，就进入到主视图主循环，在主循环中主要有两件事：

1. 前台应用需要从引擎应用获取当前状态信息，然后主动监测是否发生了状态变化，并响应状态变化，如果需要更新 UI，那么可以调用 `com_view_update()` 对该主视图 UI 进行更新。
2. 调用 `com_view_loop()` 进行用户交互处理，包括按键处理、系统消息处理、蓝牙事件（包括 RCP）处理、软定时器处理等。这些用户交互处理，可能会发送消息给引擎应用，或发生主视图切换/应用切换等。`com_view_loop()` 返回 `app_result_e` 类型结果，需要我们谨慎处理每一种值，确保不会遗漏重要的结果。

程序结构大致如下，以 AP MUSIC 为例：

```
while (1)
{
 _play_check_status(); //从引擎获取当前状态信息

 ret_vals = com_view_loop();
 if (ret_vals > RESULT_COMMON_RESERVE)
 {
 break;
 }

 sys_os_time_dly(1);
}

void _play_check_status(void)
{
 play_get_status(&g_music_info); //获取状态信息

 //正常播放, 没有错误
 if (g_music_info.eg_status.err_status == EG_ERR_NONE)
 {
 if (s_prev_time != (g_music_info.eg_playinfo.cur_time / 1000))
 {
 s_prev_time = g_music_info.eg_playinfo.cur_time / 1000;
 g_play_refresh_flag |= PLAY_REFRESH_TIME;
 com_view_update(APP_VIEW_ID_MAIN);
 }
 }

 if (g_music_info.eg_status.play_status != s_status_old)
 {
 s_status_old = g_music_info.eg_status.play_status;
 g_play_refresh_flag |= PLAY_REFRESH_STATUS;
 com_view_update(APP_VIEW_ID_MAIN);
 }
}
```

## 10.2.7 链接 \*.xn 脚本与 Makefile 脚本

前台应用的链接 \*.xn 脚本和 Makefile 脚本与 [CASE 空间分配详解](#) 一章中对应章节说明是相符合的，这里就不再赘述了。

## 10.3 前台应用开发要点

### 10.3.1 引擎应用的创建和杀死

引擎应用是由前台应用创建的。前台应用必须在初始化阶段创建对应的引擎应用，然后就可以控制引擎应用了。

前台应用必须保证在创建自己的引擎应用之前，将其他引擎应用杀死掉。

另外，有一种情形，我们也会把自己对应的引擎应用杀死，然后重新创建一次：如果引擎应用有多种工作方式，并且一旦进入一种工作方式，要切换到另一种工作方式很麻烦，那么建议干脆先杀死引擎应用，然后以新的工作方式重新创建引擎应用。

前台应用杀死其他引擎应用，可以直接使用 APP\_ID\_THEENGINE 这个虚拟 APP ID 来泛指当前的引擎应用，这样就不用自己去区分当前的引擎应用是哪一个，代码很简洁，也避免了每添加一个引擎应用，所有与引擎类型有关的地方都要修改的麻烦。

前台应用退出时，可以主动将自己的引擎应用杀死，或者在 com\_ap\_switch\_deal() 时通过应用切换表配置为杀死引擎应用的方式杀死，也可以不用杀死，留到下一个前台应用时再杀死。

### 10.3.2 与引擎应用的通信方式

前台应用必须与引擎应用配合起来才能有所作为。前台应用与引擎应用的通信方式有这么几种：

#### 10.3.2.1 共享查询

前台应用需要不断的查询引擎应用的工作状态，我们现在采用共享查询的机制来实现，这种方式不需要发生任务切换，系统消耗比较低，只是会用一点数据空间。

下面以 AP MUSIC 前台应用和 MUSIC ENGINE 引擎应用为例说明用法。

- 引擎应用创建共享查询

```
engine_info_t g_engine_info[2]; //必须是两份，这样才能安全进行共享/更新
engine_info_t *g_engine_info_p; //使用过程中会不断交替指向元素[0]和元素[1]

g_engine_info_p = &g_engine_info[0]; //指针必须指向第一个元素
g_engine_info_p = sys_share_query_create(APP_ID_MENGINE, g_engine_info_p,
 sizeof(engine_info_t)); //共享查询的 ID，可以使用引擎应用的 APP ID
if (g_engine_info_p == NULL)
{
 PRINT_ERR("music share query create fail!\n");
}
```

然后就可以使用了，引擎应用更新状态，前台应用获取状态，引擎应用退出时将其删掉。

- 引擎应用更新状态，必须使用 `g_engine_info_p` 指针来更新状态

```
g_engine_info_p = sys_share_query_update(APP_ID_MENGINE);
```

- 前台应用获取状态，即把状态拷贝到前台应用自己的全局变量

```
if (sys_share_query_read(APP_ID_MENGINE, p_music_info) == -1)
{
 PRINT_ERR("music share query not exist!\n");
}
```

- 引擎应用退出时删除共享查询

```
if (sys_share_query_destroy(APP_ID_MENGINE) == -1)
{
 PRINT_ERR("music share query destroy fail!\n");
}
```

从上面的使用方式可以得知，前台应用和引擎应用必须自己保证使用顺序，必须是先创建，再更新和获取，删除之后就不能再更新和获取了。

### 10.3.2.2 共享内存

共享内存类似共享查询，比共享查询简单，也不需要额外的数据空间，但是它的线程安全性由用户自

已把握，一般通过锁调度、或者关中断等方式来访问，以确保访问安全。

下面以蓝牙管理器共享蓝牙协议栈状态给引擎应用为例说明：

- 蓝牙管理器创建一个共享内存

```
if (sys_shm_creat(SHARE_MEM_ID_BTSTACK_STATUS, &g_bt_stack_cur_info, sizeof(bt_stack_info_t))
== -1) //共享内存 ID 需要在 psp_rel/include/share_memory_id.h 中定义
{
 PRINT_ERR("btstack status shm create fail!!");
}
```

然后蓝牙管理器对该共享内存的更新，是通过获取共享查询状态进行的，该接口本身就是关中断的。

- 蓝牙推歌引擎应用需要访问该共享内存，要先挂载

```
g_p_bt_stack_cur_info = (bt_stack_info_t *) sys_shm_mount(SHARE_MEM_ID_BTSTACK_STATUS);
if (g_p_bt_stack_cur_info == NULL)
{
 PRINT_ERR("btstack status shm not exist!!");
}
```

之后用 `g_p_bt_stack_cur_info` 就可以访问该共享内存了；在引擎应用访问，因为优先级较高，所以不用担心被蓝牙管理器的更新动作打断，这也保证了安全性。

- 蓝牙管理器退出时删除共享查询

```
sys_shm_destroy(SHARE_MEM_ID_BTSTACK_STATUS);
```

从上面的使用方式可以得知，共享查询相关模块必须自己保证使用顺序，必须是先创建，再挂载和访问，删除之后就不能再访问了。

### 10.3.2.3 消息通信

消息通信是一种很常见的进程间/线程间通信方式，COMMON 模块也说了很多，我们这里不再赘述。

## 10.4 FAQ

### 10.4.1 如何增加一个前台应用

我们提供了一个最小的前台应用 Demo——user1 前台应用，可以在此基础上完成增加一个前台应用的

任务，步骤如下：

- 1) 需要在 `case_type.h` 为前台应用分配一个 APP ID，可以使用 `APP_ID_USER1` 和 `APP_ID_USER2` 等保留 ID。
- 2) 需要在 `case_type.h` 为前台应用分配一个 FUNC ID，可以使用 `APP_FUNC_USER1` 等保留 ID。
- 3) 需要在 `case_type.h` 的 `app_result_e` 中添加 `RESULT` 类型用于创建该前台应用。
- 4) 需要在 `common_ap_switch.c` 的 `ap_switch_info` 应用切换表中添加切换到该前台应用的项目。具体添加方法要根据前台应用在方案中的属性决定，请参考 **COMMON** 模块应用切换一节说明。  
如果增加的前台应用需要通过模式切换键切换到，那么需要添加到 `ap_switch_info` 的前面几条应用切换项目中，并修改 `case\inc\common_func.h` 中的 `MAX_FUNCTION_CYCLE` 宏，将其值加 1。  

```
/*! 循环切换应用数目 */

#define MAX_FUNCTION_CYCLE 6
```

  
同时修改配置项 `SETTING_APP_SWITCH_SEQUENCE`，添加所添加的前台应用的功能 FUNC ID 到配置项列表中。
- 5) 创建应用程序需要其名字字符串，我们将其放在 `manager_get_name.c` 的 `app_name_ram` 数组中，以 APP ID 为索引来获取。所以增加一个应用程序，需要将其名字增加到 `app_name_ram` 中。
- 6) 在 `case\cfg\Makefile` 中添加该前台应用工程 make 规则，然后 make 一下整个 CASE，生成 \*.ap 映像文件。
- 7) 在 `fwimage*.cfg` 中将 \*.ap 添加进来，然后打包生成固件。
- 8) 至此，就把一个新的前台应用加到方案中了。

接下来就可以实现该前台应用的业务功能了。

## 10.4.2 如何删除一个前台应用

如果方案确定不需要某个前台应用，那么可以将该前台应用屏蔽或删除掉。

- 1) 如果该前台应用有开关配置项，那么可以简单的配置为关闭即可将该前台应用屏蔽掉。
- 2) 如果不能，那么只能通过修改代码，保证不会创建该前台应用。

而如果要将该前台应用彻底删除到一点痕迹都没有，那么请参考上一节增加一个前台应用的步骤描述，反过来做就行了，基本上就可以将某个前台应用彻底删除掉。

## 10.5 开关机应用

AP config 是一个特殊的前台应用，负责方案开关机功能，是 CASE 必不可少的一部分。

### 10.5.1 AP config 的功能设计

AP config 应用负责开机和关机，以及维护两个稳定场景：充电场景和空闲场景。

对于开机，它与 Welcome 和 AP Manager 一起完成整个开机流程，负责一些复杂的个性化选择功能。可以说，Welcome 实现一些非常简单的开机提示，比如点亮某个分立 LED 指示灯，AP Manager 实现一些基本运行环境的初始化，而其他的开机初始化全部放在 AP config 做，AP config 可以用各种手段进行用户交互。

对于关机，它全权负责，所有细节都在 AP config 应用进行。

如果是插入充电适配线开机，那么可以进入充电场景，不做什么实质性事情。

如果音箱没有什么事情可做，但是又有除了电池之外的其他供电方式供电，那么可以进入空闲场景。比如 7.4V 工作电压（功放工作电压）的音箱，如果电池断开，然后又用 USB 线供电，那么这时候就只能进入到空闲场景，等待用户打开电池开关。

### 10.5.2 AP config 的开机功能

AP config 开机功能，从整个音箱开机初始化需要做的事情看，还有以下事情：

- 1) 还有一些前台应用的共享全局变量尚未初始化。
- 2) LED 驱动尚未装载。装载之后可以显示开机 LOGO。
- 3) COMMON 模块很多子模块尚未初始化，包括声音调节、按键音、音效调节、分立 LED 灯、按键逻辑分析、应用切换等。
- 4) 一些硬件参数配置，比如卡配置、USB/UHOST 检测初始化等。
- 5) 开机选项，可以根据唤醒原因、外设状况、供电状况、以及上次关机状况、以及一些配置项来决定开机后进入到什么应用场景。
- 6) 当然，还有一些开机声音提示等。

开机流程大致如下：

- 1) 如果是升级后第一次开机，在调用 `com_setting_comval_init(&g_comval);` 时会初始化 `comval_t` 系



统环境变量。

- 2) 调用 `config_globe_data_init()` 初始化前台应用的共享全局变量，这些全局变量只需要在开机时初始化一次，之后就不能再初始化了。
- 3) 调用 `config_sys_init_display()` 装载 LED 驱动程序并显示开机 LOGO；调用 `discrete_led_init()` 初始化分立 LED 灯管理器。放在这里做，是因为如果是从 S3BT 唤醒回来，需要尽快恢复蓝牙工作环境，而蓝牙工作是需要调用一些显示接口，所以必须把显示初始化放到很前的位置执行。
- 4) 如果是从 S3BT 唤醒回来，调用 `com_btmanager_init(TRUE)`；恢复蓝牙工作环境。
- 5) 调用 `card_set_drv_power()` 进行卡配置。
- 6) 调用 `config_key_module()` 初始化按键逻辑映射表。
- 7) 调用 `com_reset_dae_config(&g_comval)`；和 `com_init_dae_config(&g_comval)`；初始化音效调节环境。
- 8) 调用 `config_sys_init()` 初始化音量调节、按键音、初始化系统日期和时间（仅升级后第一次开机）。
- 9) 如果是 S3BT 唤醒回来，跳过步骤 10，跳到步骤 11。
- 10) 否则，调用 `config_poweron_option()` 选择要进入到哪个应用场景，选择规则请参考函数实现。注意，结果也可能是关机或者进入空闲场景。
- 11) 调用 `config_power_on_dispatch()` 跳到目标应用场景。

开机流程中，有几个点要特别说明：

### 1) 开机唤醒原因

开机唤醒原因有多种，比如按键开机、闹钟定时到唤醒、插入 USB 线或充电适配线、或者电源硬开关从 OFF 拨到 ON、在 S3BT 通过蓝牙唤醒等方式，开机时需要分别处理。

比如如果是长按 Power 键唤醒，用户一直按着 Power 键，我们要避免触发关机又关机了，需要过滤掉 Power 按键。

### 2) 开机选项问题

开机后去到哪个应用场景由哪些因素决定呢？带 USB 线可能进入到 U 盘模式，带充电适配线可能就去到充电场景，带 TF 卡可能就到卡播歌，等等因素考虑，我们完全由具体方案决定，方案通过修改 `config_poweron_option()` 接口实现达到目的。

### 3) 开机提示音时机问题

如果是使用模拟功放的音箱，开机后并不能马上播放声音，而是要等到 AP Manager 的 PA 初始化子线程执行完毕之后才能播放。所以在开机流程的多个需要出声音的地方，我们都会先调用 `config_wait_pa_ok()` 接口来等待 PA 初始化完成，然后设置一次音量和 DAC 采样率，然后才播放声音。

### 4) 开机创建下一个应用



根据开机选项确定了要去到哪个应用场景，在创建下一个应用之前，如果下一个应用场景需要后台蓝牙，那么会先创建后台蓝牙。这还牵扯着另一个问题，就是开机体验的问题，如果要用户感觉开机速度快，那么就要尽早播放 TTS，要先播放 TTS 后再创建后台蓝牙，但这样又会导致后台蓝牙创建时间推迟了一点，回连蓝牙的速度又变慢了。这个问题得不到两全其美，具体方案请根据自己方案要求修改代码实现（注：如果是 I2S PA 因播放提示音不用等待，这个问题可能会简单很多）。

### 10.5.3 AP config 的关机功能

主要有以下几件事情会触发关机：

- 1) 按键关机：短按 Power 键或者长按 PLAY 键等
- 2) 低电关机
- 3) 省电关机或定时关机，包括 S3BT 进入低功耗模式
- 4) 关机时闹钟定时到，响应结束后又关机

当收到以上任意事件后，当前前台应用会创建关机应用，然后自己退出，之后切换到 AP config 关机流程，具体流程如下：

- 1) 播放关机提示音，需要根据不同的关机事件确定，比如如果是闹钟响应后又关机，就不需要播放提示音。
- 2) 如果是 S3BT，那么备份蓝牙工作环境，否则彻底关掉蓝牙。
- 3) 关掉外部 PA，调用 `disable_pa()` 关掉内部 PA。
- 4) 调用 `en_play_wake_up()` 配置好唤醒方式。
- 5) 将电源模式切换为 S3BT 模式或者 S4 断电模式。

关机流程中，有几个点要特别说明：

#### 1) 关机过程中插入 USB 线或充电适配线

如果在关机过程中插入 USB 线或充电适配线，那么就不需要真正关机了，而是直接进入到充电场景，但是无论如何，既然用户选择了关机，那么我们还是做到像关机一样，将段码屏显示都关掉，只是静悄悄的充电。

### 10.5.4 AP config 的充电场景

充电场景是一个主视图，可以响应按键消息和系统事件，比如按 PLAY 键退出充电场景进入某个应用

场景播歌，或者闹钟定时到跳到闹钟响应中去，又或者插卡就跳到卡播歌等。

如果是开机时进入充电场景，LED 段码屏会显示充电状态，蓝牙也是处于工作状态的，可以连接 APK/APP，远程遥控切换到其他应用场景。

如果是关机时进入充电场景，LED 段码屏会关掉，蓝牙也会关掉，这时候用户看来是关机状态的。

### 10.5.5 AP config 的空闲场景

开机时如果判断到没有什么实质事情可做，那么可以进入空闲场景；或者在工作过程中，检测到已无法满足工作的条件，也会进入空闲场景。

进入空闲场景，不能响应按键消息和系统消息；我们会主动检测是否已能满足工作的条件，如果是则退出空闲场景，进入工作应用场景。

## 11 引擎应用详解

引擎应用，指负责实际功能控制的应用程序，一般来说，引擎应用负责控制中间件和解码库来实现真正的播放功能。

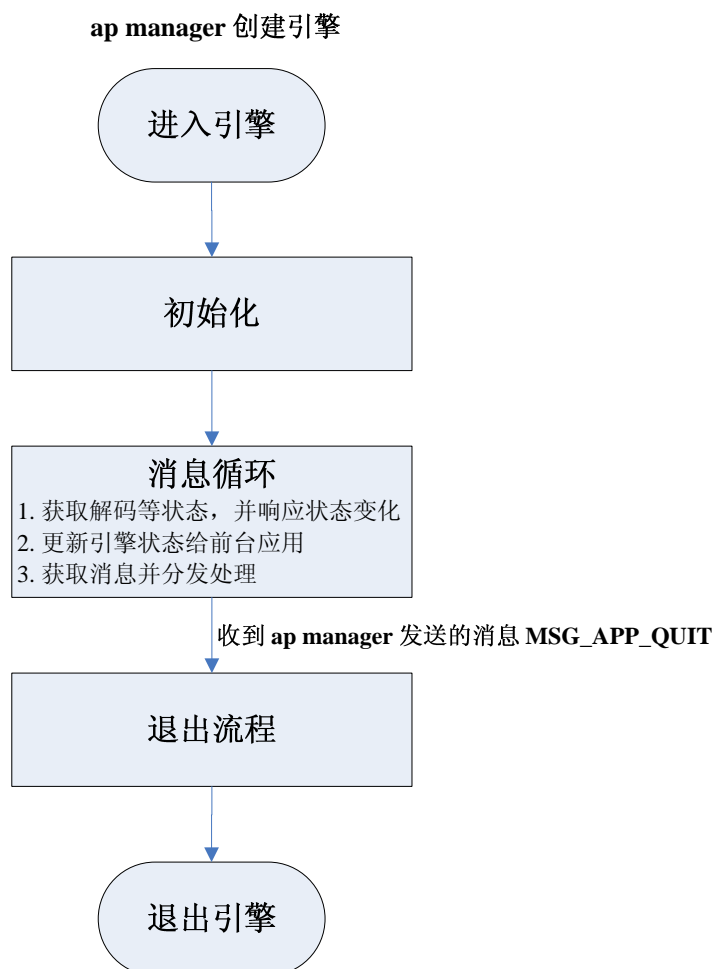
### 11.1 引擎应用工程概述

#### 11.1.1 工程组成

引擎应用工程包括以下若干部分：

- 引擎应用的业务逻辑相关源文件和头文件
- 引擎应用的链接脚本 \*.xn
- 引擎应用的 Makefile 脚本
- COMMON 模块及其链接脚本
- 运行时库 ctor.o，包含 \_\_start() 入口函数
- 各种其他程序实体的 API 接口库，比如操作系统、解码中间件等
- 公共头文件，包括 case\inc 和 psp\_rel\include 目录下的头文件
- 其他库文件、源文件和头文件，及其链接脚本，包括 Enhanced 模块及其链接脚本等
- 其他数据文件和配置文件，包括 config.bin 等

### 11.1.2 应用的一般流程



### 11.1.3 初始化流程

引擎应用比前台应用功能独立很多，也不用处理用户交互，所以它的初始化流程也会简单很多。

下面以 MUSIC ENGINE 为例说明：

- 1) 创建共享查询，以供前台应用查询引擎应用状态。创建方法请参见上一章的介绍。
- 2) `applib_init(APP_ID_MENGINE, APP_TYPE_CONSOLE);` 向 APPLIB 模块注册该引擎应用；该接口最先调用。
- 3) `applib_message_init();` 消息模块初始化。
- 4) `init_app_timers(mengine_app_timer_vector, APP_TIMER_COUNT);` 初始化软定时器。

另外，可以在上述合适位置读取保存在 VRAM 的环境变量、初始化全局变量等。

完成上述初始化步骤后，就可以进入业务主循环，执行真正的业务逻辑。

### 11.1.4 退出流程

当引擎应用收到 AP Manager 发送过来的 MSG\_APP\_QUIT 异步消息，就会无条件的退出应用。引擎应用的退出流程非常简单：

- 1) 调用 `applib_quit()`；向 APPLIB 模块注销该引擎应用。
- 2) 注销共享查询。

另外，可以在合适位置将环境变量保存到 VRAM 中，以供下次重新进入该前台应用时恢复环境。

### 11.1.5 业务主循环

引擎应用的业务主循环也很简单，主要做以下几件事情：

- 1) 更新共享查询，这样前台应用就可以更新引擎应用的状态了。
- 2) 查询中间件的状态，并主动监测是否发生了状态变化，并响应状态变化，特别是要响应一些异常/错误状态，比如播放到尾，帧解码错误等。
- 3) 获取前台应用发送过来的消息 `get_app_msg(&cur_msg_val)`，如果收到消息则分发处理。另外，引擎应用在 `get_app_msg()` 中会执行软定时器 HANDLE。

### 11.1.6 链接 \*.xn 脚本与 Makefile 脚本

引擎应用的链接 \*.xn 脚本和 Makefile 脚本与 [CASE 空间分配详解](#) 一章中对应章节说明是相符合的，这里就不再赘述了。

## 11.2 FAQ

### 11.2.1 如何增加一个引擎应用

我们提供了一个最小的引擎应用 Demo——user1eg 引擎应用，可以在此基础上完成增加一个引擎应用的任务，步骤如下：

- 1) 需要在 `case_type.h` 为应用程序分配一个 APP ID，可以使用 `APP_ID_USER3` 和 `APP_ID_USER4` 等保留 ID。
- 2) 需要在 `applib.h` 的 `engine_type_e` 中添加引擎类型。
- 3) 在 `app_engine_config.c` 的 `applib_app2eg_type` 表中添加 APP ID 与 引擎类型对应的项目。
- 4) 创建应用程序需要其名字字符串，我们将其放在 `manager_get_name.c` 的 `app_name_ram` 数组中，以 APP ID 为索引来获取。所以增加一个应用程序，需要将其名字增加到 `app_name_ram` 中。
- 5) 在对应的前台应用中添加创建该引擎应用的代码。
- 6) 在 `case\cfg\Makefile` 中添加该引擎应用工程 make 规则，然后 `make` 一下整个 CASE，生成 \*.ap 映像文件。
- 7) 在 `fwimage*.cfg` 中将 \*.ap 添加进来，然后打包生成固件。
- 8) 至此，就把一个新的引擎应用加到方案中了。

接下来就可以实现该引擎应用的业务功能了。

### 11.2.2 如何删除一个引擎应用

引擎应用是由对应的前台应用创建的，所以只要其前台应用被删除了，该引擎应用也就不可见了。

而如果要将该引擎应用彻底删除到一点痕迹都没有，那么请参考上一节增加一个引擎应用的步骤描述，反过来做就行了，基本上就可以将某个引擎应用彻底删除掉。

## 12 驱动程序详解

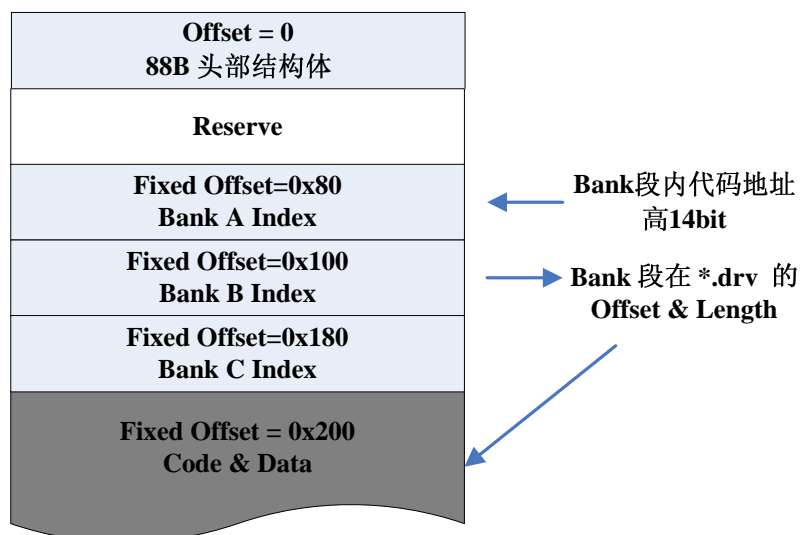
驱动程序是一种向其他程序提供服务的独立程序实体。在进程/线程调度机制上，驱动程序是完全被动的，它自己不会创建一个进程/线程，而是直接被某个进程/线程以 API 机制进行远程调用，所以它的任务优先级与调用方一致。

另外，驱动程序可以注册硬件中断，实现硬件功能服务。如果驱动程序注册了硬件定时器，那么驱动程序也可以周期循环的进行一些独立业务处理，比如按键驱动注册了 20ms 的硬件定时器，用来按键检测和电量检测等。

### 12.1 驱动程序 \*.drv 文件结构

驱动程序通过 make 的编译链接，生成 \*.exe 结果文件，但是因为 \*.exe 文件包含了很多对运行没用实际作用的调试信息等，这也就浪费了很多空间；并且 \*.exe 这种标准的 ELF 文件结构的解释也比较复杂。所以，我们提供驱动程序的 maker 工具，去掉多余的调试信息，仅仅抽取我们需要的代码段和数据段，并且以一种非常简洁的文件结构进行打包，maker 工具把应用程序打包为 \*.drv 文件。

这种驱动程序文件的结构示意图如下：



从上面的示意图可以看到，驱动程序文件结构将 BANK 段索引表放在扩展结构中，位置是固定的。

Bank 段的装载：从 Bank 段内代码地址高 14bit 获取对应的索引表和 Index 号，就可以得到 Bank 段在 \*.drv 的 Offset & Length，然后就可以装载到物理内存中。

驱动程序的头部包含了很多信息，当操作系统加载驱动程序时，根据头部结构的信息装载代码和数据到内存中，并建立驱动程序运行环境，然后调用 drv\_init\_entry() 完成驱动初始化，接着其他模块就可以通过 API 调用驱动外部接口；当操作系统卸载驱动程序时，调用 drv\_exit\_entry() 完成驱动资源释放。头部结构定义如下：

```
23 typedef struct
24 {
25 unsigned char file_type;
26 unsigned char drv_type;
27 unsigned char Major_version;
28 unsigned char minor_version;
29 unsigned char magic[4];
30 unsigned int text_offset;
31 unsigned int text_length; → .text 输出段在 *.drv 中的索引
32 unsigned int text_addr;
33 unsigned int text1_offset;
34 unsigned int text1_length; → .text1 输出段在 *.drv 中的索引
35 unsigned int text1_addr;
36 unsigned int text2_offset;
37 unsigned int text2_length; → .text2 输出段在 *.drv 中的索引
38 unsigned int text2_addr;
39 unsigned int data_offset;
40 unsigned int data_length; → .data 输出段在 *.drv 中的索引
41 unsigned int data_addr;
42 unsigned int bss_length; → .bss 输出段的描述
43 unsigned int bss_addr;
44 unsigned int drv_init_entry; → 驱动程序初始化接口
45 unsigned int drv_exit_entry; → 驱动程序资源销毁接口
46 unsigned int drv_banka_file_offset;
47 unsigned int drv_bankb_file_offset; → BANK代码在 *.drv 中的索引
48 unsigned int drv_bankc_file_offset;
49 unsigned int drv_op_entry; → 驱动程序API表地址
50 } drv_head_t;
```

说明：

- .text .text1 .text2 是常驻代码段，在驱动程序加载时装载到内存中；.data 是初始化的常驻数据段，在驱动程序加载时装载到内存中；.bss 是尚未初始化的常驻数据段，在驱动程序加载时清为 0。
- drv\_init\_entry 是在驱动程序的源代码中，用 module\_init(drv\_init) 来指定的。
- drv\_exit\_entry 是在驱动程序的源代码中，用 module\_exit(drv\_exit) 来指定的。
- drv\_op\_entry 是驱动程序 API 表地址，它是在链接脚本 \*.xn 文件中用 ENTRY(op\_entry) 声明的。



## 12.2 驱动程序工程概述

### 12.2.1 工程组成

驱动程序工程包括以下若干部分：

- 驱动程序的功能服务相关源文件和头文件
- 驱动程序的链接脚本 \*.xn
- 驱动程序的 Makefile 脚本
- 操作系统等底层 PSP 模块的 API 接口库
- 公共头文件，包括 case\inc 和 psp\_rel\include 目录下的头文件
- 其他库文件、源文件和头文件等
- 其他数据文件

### 12.2.2 驱动程序装载

应用程序调用驱动程序 API 接口之前，必须确保已经正确装载了驱动程序。

有些公共驱动程序，比如操作系统、NOR\_UD 驱动、audio\_device 驱动、按键驱动、LED 驱动、ccd 驱动等，我们在开机初始化阶段装载后就不卸载了，可以放心使用；其他一些场景相关的驱动程序，比如文件系统和卡驱动等，只有在进入对应场景后才会装载，这种情况下必须谨慎。

装载驱动程序接口：

```
int drv_install(uint8 drv_type, void* drv_para, char* drv_name);
```

比如：sys\_drv\_install(DRV\_GROUP\_FM, &fm\_arg, "drv\_fm.drv"); 注意 DRV\_GROUP\_FM 必须和驱动程序的链接 \*.xn 中的 BANK\_GROUP\_INDEX = DRV\_GROUP\_FM; 一致。

代码中访问的 DRV\_GROUP\_FM 是定义在 driver\_manager.h 中的 drv\_type\_t 枚举类型值。

我们现在可以通过 drv\_para 向驱动程序初始化接口传递任意参数了，这样就可以根据业务场景需要详细定制驱动程序的工作方式，对整个方案提供丰富定制化是有好处的。

比如按键驱动，装载时传递了充电模式、充电电流、充电电压、红外遥控器类型等，而这些都是通过 config.txt 配置项来配置的。

装载驱动程序时，操作系统装载完毕后，会调用驱动程序的初始化接口，其函数形式为：

```
int drv_init(void* null0, void *null1, void* drv_para);
```

drv\_para 就是 drv\_install() 函数的参数 void\* drv\_para，这样写并不是说参数必须是以“地址”传递，而是表示参数是任意一种 32 位的数据，它可以是一个 uint32/int32 常量或变量，也可以说任意类型的指针常量或变量。

注意：驱动程序的文件结构要求我们要用 module\_init(drv\_init) 来指定 drv\_init 为驱动程序初始化接口。

### 12.2.3 驱动程序卸载

当驱动程序不再使用时，我们应该将其卸载掉，以释放出内存空间，供其他复用模块使用，比如当从卡播歌模式退出时，将文件系统和卡驱动卸载掉，将内存空间释放出来，作为蓝牙推歌模式的数据缓冲区。

卸载驱动程序接口：

```
int sys_drv_uninstall(uint8 drv_type);
```

比如：sys\_drv\_uninstall(DRV\_GROUP\_FM);

卸载驱动程序时，操作系统会调用驱动程序的退出接口，释放该驱动程序占用的各种系统资源，比如硬件中断、动态内存空间、DMA 通道等；其函数形式为：

```
int drv_exit(void* null0, void *null1, void *null2);
```

即该接口不能带任何参数，当然，对于这样的接口形式，驱动程序的退出函数实现为 void drv\_exit(void); 也是不会运行出错的。

注意：驱动程序的文件结构要求我们要用 module\_exit(drv\_exit) 来指定 drv\_exit 为驱动程序退出接口。

### 12.2.4 链接 \*.xn 脚本与 Makefile 脚本

驱动程序的 Makefile 脚本与 [CASE 空间分配详解](#) 一章中对应章节说明稍微有点差异，下面只列出差异点：

1. TARGET\_DRV = \$(IMAGENAME).drv Target 后缀名从 \*.ap 改为 \*.drv 。
2. \$(DRV\_BUILDER) \$(IMAGE\_ELF) \$(TARGET\_DRV) 生成映像文件的 Maker 工具从 ap\_builder.exe 改为 drv\_builder.exe。

驱动程序的链接脚本 \*.xn 与 [CASE 空间分配详解](#) 一章中对应章节说明稍微有点差异，下面只列出差异点：

1. BANK\_GROUP\_INDEX = DRV\_GROUP\_KEY; 驱动程序需要填写 BANK\_GROUP\_INDEX，它是在 link\_base.xn 中定义的。
2. ENTRY(op\_entry) 驱动程序的 ENTRY() 填写的是驱动程序 API 表地址，而非某个函数地址。

## 12.3 驱动程序 API 接口

我们之前在 [软件基本组成介绍](#) 介绍过驱动程序的 API 接口，我们这里就不再说明了。

这里要说的是一个使用技巧。

因为每个 API 接口都要占用 4 个字节的常驻数据空间，所以对于分配的常驻代码空间或常驻数据空间非常有限的驱动程序，扩展一个 API 接口有时就很困难了。

所以，我们可以将多个功能相关的简单 API 接口复用合并为一个 API 接口，用 3 个参数中的某一个来区分是要执行那个功能就行了。

比如按键驱动的调频类接口，我们有 2 个调频接口，并且这两个调频接口都可以将参数限制为 2 个，能够节省出一个参数来区分是哪个调频接口，所以就共用一个 API 接口，详细见 key\_interface.h。

## 12.4 注意事项

1. 驱动程序开发时要特别注意一点：不能没有锁调度和关中断的前提下，在 BANK A 和 BANK B 中使用 BANK DATA，或者使用 const 数据指针传参来调用不是本 BANK 的接口，否则就有可能出现 BANK 被切走导致数据和参数错误的情况。

## 12.5 FAQ

### 12.5.1 如何增加一个驱动程序

因为增加一个新的驱动程序需要额外的常驻数据空间和常驻代码空间，以及一个空闲的驱动程序 op\_entry，所有这些资源都是比较难以获取到的。

所以我们不建议增加一个新的驱动程序，而是建议将所要添加的驱动功能合并到已有的驱动程序中，比如将 UART 通信模块合并到 KEY 驱动中。

如果真的要增加一个新的驱动程序，那么只能是“借壳”增加一个驱动程序，直接修改就行了。

## 12.5.2 如何删除一个驱动程序

如果某个方案不需要使用某个驱动程序，那么只要确保不装载该驱动程序就行了，它所占用的内存空间和资源就会释放出来，供其他模块使用。

我们可以在 `fwimage*.txt` 中将该驱动程序打包项删除，这样还可以释放出所占用的固件空间。

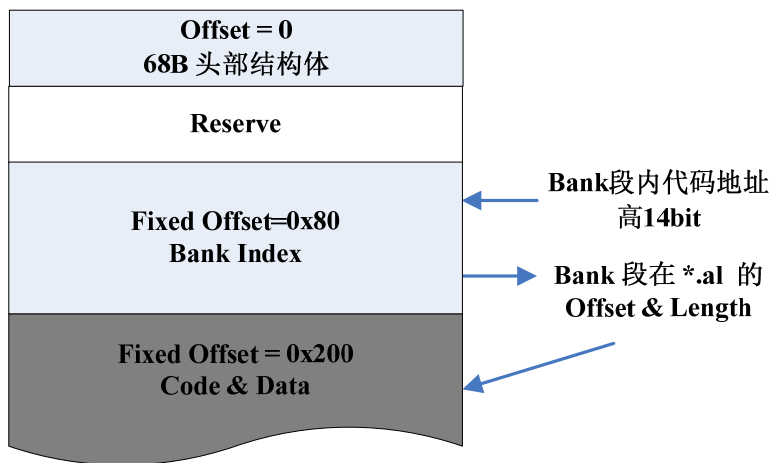
## 13 前台中间件详解

有些功能模块在整个软件上保持一定的独立性，我们希望对其进行模块化，单独开发和维护，那么可以封装为前台应用对应的“中间件”，即除了上述几种程序实体之外的，称为其他独立程序实体；也可以形象一点称之为前台中间件。

### 13.1 前台中间件 \*.al 文件结构

前台中间件通过 make 的编译链接，生成 \*.exe 结果文件，但是因为 \*.exe 文件包含了很多对运行没用实际作用的调试信息等，这也就浪费了很多空间；并且 \*.exe 这种标准的 ELF 文件结构的解释也比较复杂。所以，我们提供中间件的 maker 工具，去掉多余的调试信息，仅仅抽取我们需要的代码段和数据段，并且以一种非常简洁的文件结构进行打包，maker 工具把中间件打包为 \*.al 文件。

这种中间件文件的结构示意图如下：



从上面的示意图可以看到，中间件文件结构将 BANK 段索引表放在扩展结构中，位置是固定的。

**Bank 段的装载：**从 Bank 段内代码地址高 14bit 获取对应的 Index 号，就可以得到 Bank 段在 \*.al 的 Offset & Length，然后就可以装载到物理内存中。

中间件文件的头部包含了很多信息，当操作系统加载中间件时，根据头部结构的信息装载代码和数据到内存中，并建立中间件运行环境，接着其他模块就可以通过统一命令入口控制和访问中间件了。头部结构定义如下：

```
84 typedef struct
85 {
86 unsigned char file_type;
87 unsigned char media_type;
88 unsigned char major_version;
89 unsigned char minor_version;
90 unsigned char magic[4];
91 unsigned int text_offset;
92 unsigned int text_length; → .text输出段在*.ap中的索引
93 unsigned int text_addr;
94 unsigned int text1_offset;
95 unsigned int text1_length; → .text1输出段在*.ap中的索引
96 unsigned int text1_addr;
97 unsigned int text2_offset;
98 unsigned int text2_length; → .text2输出段在*.ap中的索引
99 unsigned int text2_addr;
100 unsigned int data_offset;
101 unsigned int data_length; → .data输出段在*.ap中的索引
102 unsigned int data_addr;
103 unsigned int bss_length; → .bss输出段的描述
104 unsigned int bss_addr;
105 unsigned int module_api_entry; → 中间件入口函数
106 } codec_head_t;
```

说明：

- .text .text1 .text2 是常驻代码段，在中间件加载时装载到内存中；.data 是初始化的常驻数据段，在中间件加载时装载到内存中；.bss 是尚未初始化的常驻数据段，在中间件加载时清为 0。
- module\_api\_entry 是中间件命令入口，它是在链接脚本 \*.xn 文件中用 ENTRY(cmd\_entry) 声明的。

## 13.2 前台中间件工程概述

### 13.2.1 工程组成

前台中间件工程包括以下若干部分：

- 前台中间件的功能服务相关源文件和头文件
- 前台中间件的链接脚本 \*.xn

- 前台中间件的 Makefile 脚本
- 操作系统等底层 PSP 模块的 API 接口库
- 公共头文件，包括 case\inc 和 psp\_rel\include 目录下的头文件
- 其他库文件、源文件和头文件等
- 其他数据文件

### 13.2.2 前台中间件装载

前台中间件装载接口如下：

```
int sys_load_mmm(char *name, uint32 type);
```

其中 type 必须为 FALSE。

前台中间件装载时操作系统没有自动调用初始化函数，而是在装载完成之后，通过调用 OPEN 命令主动进行初始化。

### 13.2.3 前台中间件卸载

前台中间件卸载接口如下：

```
void sys_free_mmm(uint32 type);
```

其中 type 必须为 FALSE。

前台中间件卸载时操作系统没有自动调用退出函数，所以我们在卸载之前，通过调用 CLOSE 命令主动销毁资源，然后再将前台中间件卸载掉。

### 13.2.4 链接 \*.xn 脚本与 Makefile 脚本

前台中间件的 Makefile 脚本与 [CASE 空间分配详解](#) 一章中对应章节说明稍微有点差异，下面只列出差异点：

1. TARGET\_AL = \$(IMAGENAME).al Target 后缀名从 \*.ap 改为 \*.al 。
2. \$(MMM\_CODEC\_BUILDER) \$(IMAGE\_ELF) \$(TARGET\_AL) 生成映像文件的 Maker 工具从 ap\_builder.exe 改为 mmm\_codec\_builder.exe。

前台中间件的链接脚本 \*.xn 与 [CASE 空间分配详解](#) 一章中对应章节说明稍微有点差异，下面只列出差异点：

1. ENTRY(cmd\_entry) 前台中间件的 ENTRY() 填写的是前台中间件的命令入口函数地址。

### 13.3 前台中间件 API 接口

我们之前在 [软件基本组成介绍](#) 介绍过中间件的 API 接口，我们这里就不再说明了。

### 13.4 FAQ



## 14 CASE 驱动程序详解

CASE 驱动程序包括 KEY 驱动、LED 驱动、Welcome 模块等，是方案相关的驱动，即具体方案的 KEY 硬件电路设计、LED 模组可能有所区别，并且即使是相同的硬件，其软件功能规格也可能会有所差异，所以我们将这几个驱动程序放在 CASE 中来设计和实现。

其中的 KEY 驱动、LED 驱动是与用户交互的基础模块，即负责前台应用的 I/O 任务，对构造一个人性化方案起到至关重要的作用。

另外，Welcome 模块也是 CASE 中必不可少的组成部分，它也属于开机初始化流程中的第一个环节，也是 CASE 能够进行二次开发的第一个环节，那些需要在上电后马上执行的功能就放到 Welcome 中。

### 14.1 KEY 驱动程序

#### 14.1.1 KEY 的地位和功能

KEY 驱动程序最直接的功能，就是循环地进行按键检测并发出按键消息；然而，正如我们前面所述的一样，我们一般不会为了某些很小的硬件功能而增加一个独立的驱动程序，而是将这些小硬件功能添加到现有的驱动程序中，而 KEY 驱动程序就是首选。所以我们也为 KEY 驱动程序分配了比较大的内存空间。

作为输入设备的 KEY 驱动程序，我们支持 GPIO 按键、ON\_OFF 按键、线控 LRADC 按键等机械按键、红外遥控器、触摸按键等，并且这 3 类按键是相互独立的，KEY 驱动程序支持同时按下以上多种按键并将其分开解析。

所以，KEY 驱动程序支持以下功能：

1. 支持 GPIO 按键、ON\_OFF 按键、线控 LRADC 按键等机械按键、红外遥控器、触摸按键等按键
2. 充电管理及电池电量检测
3. 设备热拔插检测，包括 UHOST、卡、AUX、耳机等
4. 调频管理
5. 分立 LED 灯显示，支持呼吸灯模式

## 14.1.2 KEY 按键检测原理

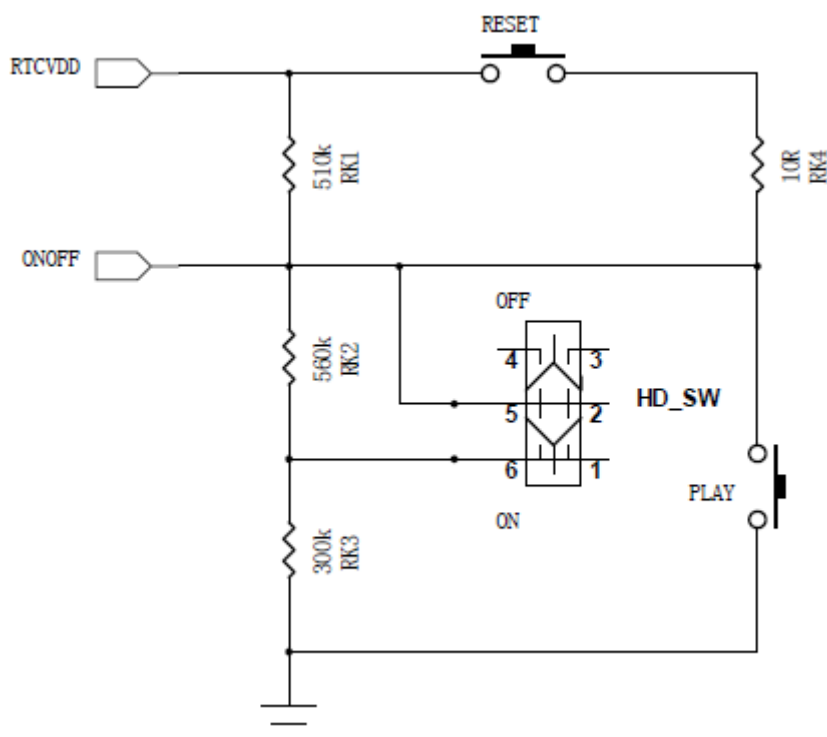
### 14.1.2.1 GPIO 按键的原理

GPIO 按键的设计，通常有两种方式：每个 GPIO 对应一个按键的方式和 GPIO 矩阵的方式，其设计参考如下：

按照上述原理，GPIO 按键可以实现多个按键同时按下的检测。

### 14.1.2.2 ON\_OFF 按键的原理

ON\_OFF 多功能按键由 Analog 电路负责检测按键状态：其原理图如下：



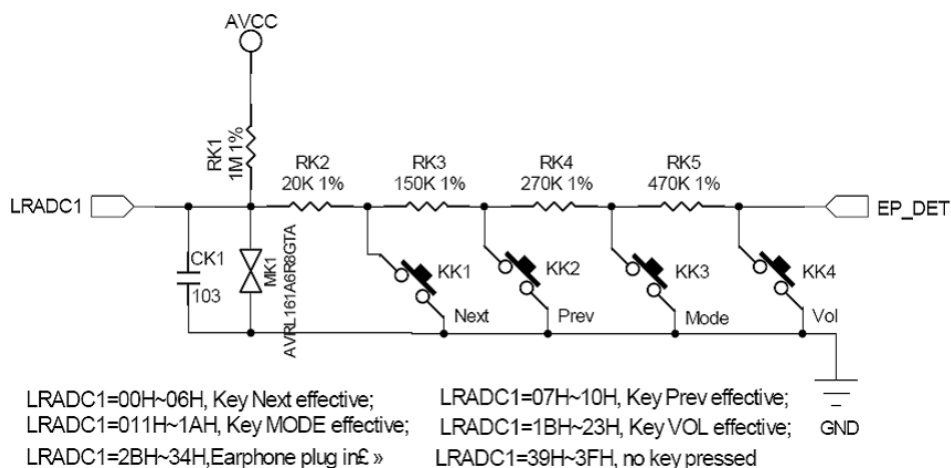
寄存器 ONOFF\_KEY Bit0,1,2 实时指示 ONOFF 电平，Bit3 指示硬开关状态，不受是否按下软开关影响。

即 ONOFF 电平为 0 表示 PLAY 按键按下；为 2/3 RTCVDD 表示硬开关处于 OFF 状态；为 1/3 RTCVDD 表示硬开关处于 ON 状态；为 RTCVDD 表示 RESET 按键按下。

软件中只需读取寄存器 ONOFF\_KEY 的值就可以判断上图中按键的状态。

### 14.1.2.3 LRADC1 按键的原理

LRADC1 按键，其原理是，当按键按下时，不同的按键按下，从 AVCC 到 GND 的电阻值是不一样的，从而导致了从 LRADC1 到地的分压不一样，软件将通过读取 LRADC1 的值，不同的按键产生不同的 ADC 值，去判断具体是哪个按键。不同的电阻值对应的 ADC 值的计算方法如下，比如下图：



直接用电压表测量 LRADC1 的电压。以 NEXT 键为例，测得 LRADC1 处的电压是 0.35v，套用下面公式。AVCC 为 2.8v，得出值为：

$$\frac{V_{lradc1}}{AVCC} \times 2^7$$

根据电阻值计算。以 NEXT 为例，使用电阻分压比乘以 128，得出值为 0.25：

$$\frac{20K}{1M + 20K} \times 2^7$$

上面两个公式中的  $2^7$  表示 LRADC1 为 7 位 ADC。使用不同 ADC 时注意其位数。

按照以上原理，ADC 按键无法实现两个按键同时按下这样的功能的。

### 14.1.2.4 红外遥控器的原理

红外遥控器是由 ATS282X IC 内部集成了红外接收控制器来实现的，支持多种红外协议，支持 36KHz，38KHz，40KHz 载波。

当我们使能了红外接收控制器，只要收到红外遥控器信号，控制器就会将收到的按键编码保存起来，然后置其 Pending 位；如果我们使能了中断，那么就由中断去处理，如果没有就通过硬件定时器循环去检测 Pending 位。

### 14.1.3 KEY 驱动程序工程概述

#### 14.1.3.1 源代码组成

KEY 驱动程序包含一些源代码，分别说明如下：

| 源文件名称                              | 简介             |
|------------------------------------|----------------|
| key_bankb0_init.c                  | 按键驱动初始化        |
| key_bankb1_exit.c                  | 按键驱动退出         |
| key_rcode_i2c_tk.c                 | I2C 触摸按键       |
| key_rcode_irq_ir.c                 | 红外接收           |
| key_rcode_irq_tk.c                 | IC 内置触摸按键      |
| key_rcode_op_entry.c               | 按键驱动 API 接口表定义 |
| key_rcode_scan.c                   | 按键检测实现         |
| key_common\adjust_freq_comm.c      | 调频管理模块         |
| key_common\key_bat_charge_comm.c   | 充电和电池电量检测      |
| key_common\key_discrete_led_op.c   | 分立 LED 灯显示     |
| key_common\key_peripheral_detect.c | 设备热拔插检测        |

#### 14.1.3.2 KEY 驱动初始化与退出

KEY 驱动初始化流程如下：

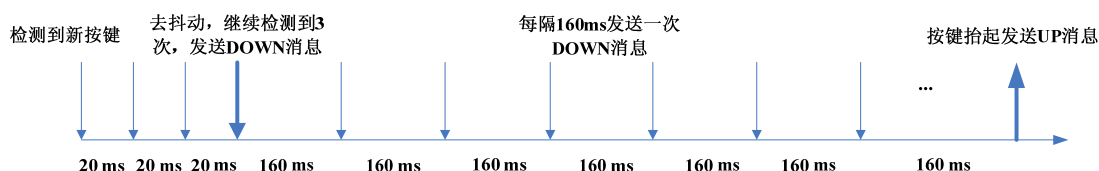
- 使能 LRADC，以捕捉按键动作。
- 启动 20ms 硬件定时器中断，以实现按键周期扫描。
- 初始化充电状态为停止充电。
- 其他模块初始化。

KEY 驱动卸载需要完成上述事情的逆事件。

## 14.1.4 KEY 按键扫描

GPIO 按键和线控 LRADC 按键，都是以 20ms 周期扫描的方式实现按键动作的捕捉和处理的。按键按下时，可能会因按键接触不完全产生抖动，如果一捕捉到电压变化就认定为有按键按下，则可能会错误发送按键消息，所以需要进行去除抖动处理。去除抖动确定按键动作稳定后，即可开始按照按键生命周期发送按键消息了。

按键生命周期如下所示：



按键驱动只需要简单的把上述检测到的按键动作发送出去就行了，交给应用层自己去解释长按、保持、双击等按键动作。

对于触摸按键，其检测方式跟机械按键一样，支持获取物理采样值的方式以及判断某个物理采样值对应哪个物理按键的方式不一样而已。

红外遥控输入开启了中断，按键消息也直接在中断服务例程发送出去。

但是，红外遥控需要在扫描循环中检测什么时候按键抬起了；我们增加了一个计时器，该计时器会在按键按下的中断服务例程清 0，所以只要计时到 7 次以上，就认为按键抬起了。

按键发送的接口为：

```
bool post_key_msg(key_phy_type_e key_type, key_phy_status_e key_status, uint8 key_id);
```

### 14.1.5 调频管理

US282A 调频管理采用按线程调频的方式，即每个线程分配它自己需要的 M4K、DSP、ASRC 这 3 个主要硬件计算模块的工作频率，然后按键驱动会将所有线程需要的各模块的工作频率分别相加，计算出能满足该场景的最小工作频率，然后调用操作系统的 `sys_adjust_clk()` 执行调频。

线程有时会代理其他程序模块设置频率，比如引擎代理 DSP 解码和音效库等设置 DSP 的工作频率。

KEY 驱动提供出来的调频接口有以下两个：

1. 设置绝对频率值： `int adjust_freq_set_level(uint8 prio, uint8 level_mips, uint8 level_dsp);`
2. 整个场景再加上一些频率值： `int adjust_freq_add_value(int8 freq_mips, int8 freq_dsp);`

`level_mips` 和 `level_dsp` 对应实际频率的关系如下：

```
typedef enum
{
 FREQ_NULL = 0, //不耗用 MIPS 或 DSP 频率，即频率值为 0
 FREQ_LEVEL1, //如果是 MIPS->1MM, 如果是 DSP->35M
 FREQ_LEVEL2, //如果是 MIPS->2M, 如果是 DSP->40M
 FREQ_LEVEL3, //如果是 MIPS->4M, 如果是 DSP->55M
 FREQ_LEVEL4, //如果是 MIPS->6M, 如果是 DSP->60M
 FREQ_LEVEL5, //如果是 MIPS->8M, 如果是 DSP->70M
 FREQ_LEVEL6, //如果是 MIPS->10M, 如果是 DSP->80M
 FREQ_LEVEL7, //如果是 MIPS->20M, 如果是 DSP->100M
 FREQ_LEVEL8, //如果是 MIPS->30M, 如果是 DSP->120M
 FREQ_LEVEL9, //如果是 MIPS->40M, 如果是 DSP->140M
 FREQ_LEVEL10, //如果是 MIPS->60M, 如果是 DSP->160M
 FREQ_LEVEL11, //如果是 MIPS->85M, 如果是 DSP->180M
 FREQ_LEVEL_MAX
} adjust_freq_level_e;
```

### 14.1.6 充电和电量检测

系统要不断检测电源，电池状态，电池电量等。在 key 驱动初始化函数中会创建一个定时器每隔 20ms 会检测一下电源，电池状态，电池电量。在各个应用的初始化函数中会每隔 500ms 获取一下 key 驱动得到的这些参数。

电源：项目中支持适配器，锂电池，USB 线 3 种方式供电。

电池状态：电池有 5 种状态，key 驱动中每隔 20ms 会根据供电方式，以及电池电量更新电池状态。电池的 5 种状态如下：

```
typedef enum
{
 BAT_NORMAL = 0, //电池供电且不在充电状态
 BAT_CHARGING, //正在充电
 BAT_FULL, //充满
 BAT_NO_EXIST, //没有电池
 BAT_CHECKING, //正在检测电池中
} battery_status_e;
```

电池电量：读取寄存器 TEMPADC\_DATA 或 BATADC\_DATA(7bit Voltage ADC)的值，就得到电池电压的采样值，根据 ADC 的基准电压，就可以获得实际的电压，为了减小误差，获取 50 个采样值，并将最小的 5 个去掉，然后求取平均值作为电池电压的采样值。本项目中电池电压等级数为 0 ~9 共 10 个等级，0 表示显示低电关机，1 表示低电需提示，9 表示电池满电量。程序中会根据电池电压的采样值将其转换成对应的等级。在应用中会根据获得的电池电量进行相应的 TTS 播报，提醒用户，如会报电量低，请充电等。

检测电源，检测电池，和开关充电 void ker\_battery\_charge\_deal(void)

处理跟电量有关功能，显示充电标志，检测低压(<3.6V 为低压)等 void com\_battery\_charge\_deal(void)

## 14.1.7 分立 LED 灯显示

分立 LED 灯用来反映系统运行状态和用户交互过程，起提示的作用。

分立 LED 灯显示的代码分为 2 部分，一部分在 COMMON 的 common\_discrete\_led.c 中，一部分在 KEY 驱动中。之所以这样划分，有 2 个原因：

1. 放在 KEY 驱动中的部分代码，确实是属于很底层的 IC 硬件模块的操作，比如 PWM 呼吸灯效果实现等，理应放在驱动程序中。

2. 放在 KEY 驱动中的部分代码，它链接到 BANKB 段中，这个段在正常播放时是不会被切走的，等同于常驻代码，这样有利通过 ESD 和 EMI 的认证。

## 14.1.8 设备热拔插检测

设备热拔插支持 AUX、卡、U 盘/UHOST、耳机等外部设备。U 盘/UHOST 比较特殊，我们在 KEY 驱动中只是负责进行 U 盘/UHOST 检测模式切换，然后由操作系统检测 U 盘/UHOST 的热拔插；AUX、卡、耳机都可以有多种检测方式，比如 GPIO、LRADC、SIO、GPIOB0 等，对于卡检测，还有一种命令检测方式可用。

设备热拔插 API 接口是 `int key_peripheral_detect_handle(uint32 detect_mode)`，由 COMMON 的设备热拔插检测定时器 200ms 周期调用。这个周期是可以修改的，然后 KEY 驱动真正运行时，各种设备可以按此周期再计时实施检测。如果方案要求检测得非常及时，那么必须采用检测消耗很小的方式，比如卡不要使用命令检测方式，并且将定时器周期改小，比如改为 50ms，然后再修改各种设备的检测计时。

## 14.1.9 FAQ

### 14.1.9.1 如何修改物理按键设计

修改物理按键设计后，需要检查一下各按键 ADC 值是否发生变化；并更新物理按键 ADC 表。  
同时，如果增加一个新的意义的按键，则要在 `gui_msg.h` 中的增加一个逻辑按键序号。

接下来要修改物理 - 逻辑映射表，即修改 `SETTING_INPUT_KEY_ARRAY` 配置项或 `SETTING_INPUT_TK_KEY_ARRAY` 配置项中的物理按键序号对应的逻辑按键值。

### 14.1.9.2 如何修改按键周期时间参数

我们在按键驱动中定义了以下几个宏参数：

```
#define DOWN_KEY_TIMER 3//按键按下消息时间为 60ms
#define UP_KEY_TIMER 3//按键抬起消息时间为 60ms
#define HOLD_KEY_TIMER 8//按键连续 hold 消息间隔为 160ms
```

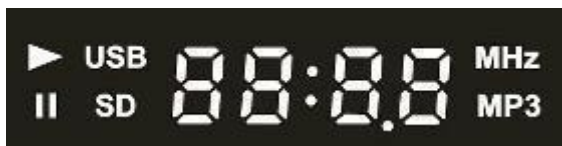
一般可以通过修改这几个宏参数来达到修改按键周期时间参数的目的。比如修改 `HOLD_KEY_TIMER` 就是修改了重复按键发出的时间间隔，但是如果要修改长按按键的时间，就得修改应用层的参数了。



## 14.2 LED 驱动程序

### 14.2.1 LED 的地位和功能

US282A 的目标产品形态，比如音箱和耳机，大部分都是作为功能附件的角色，比如手机声音输出的附件，所以我们在音箱操作的并不会很多，也就降低了对复杂操作进行有效反馈的全功能“显示器”的需求。所以我们标案不支持 LCD 显示屏，而只支持 LED 数码屏这种显示功能很有限的“显示器”模块。



一般 LED 数码管可以进行有限的状态提示和能够数字化的信息显示，比如：

1. 指示当前处于哪种应用模式：比如卡播放、AUX 播放、FM 播放
2. 指示当前正在播放还是暂停
3. 指示当前电池电量
4. 显示当前歌曲曲目号和播放时间、FM 收音频点

甚至，很多音箱连这么简单的 LED 数码屏都不需要，因为我们可以用分立 LED 灯、声音反馈等来有效完成用户交互的反馈。

所以说，我们在标案中保留 LED 数码管，是为了让需要 LED 数码屏的方案可以不用当心资源问题；而很多不需要 LED 数码屏的方案，大可以将它占用的内存空间拿去做其他用途。

### 14.2.2 LED 驱动程序的接口设计

根据上面的需求，我们设计了以下 LED 数码管接口：

- 显示接口：void **led\_display**(uint8 addr, uint8 content, uint8 type);

用来点亮或关闭某个状态灯和数码管。

- 清屏接口：void **led\_clear\_screen**(void);

用来关掉整个数码屏的所有状态灯和数码管。

- 闪屏接口：void **led\_flash\_screen**(bool onoff);

用来让整个数码屏闪烁，以进行一种特殊状态的提示，比如低电，或报警等。

- 休眠接口: `void led_sleep_display(void);`

当用户一段时间不操作, 可以让 LED 数码屏进入休眠状态, 以降低功耗。

- 唤醒接口: `void led_wake_up_display(void);`

当用户有操作时, 从休眠状态唤醒, 继续显示。

### 14.2.3 LED 驱动程序的实现

ATS282X IC 内部集成了 LED 数码管控制器, 其具体规格如下:

- Support 3com / 4com / 5com / 6com SEG\_LCD Driving Timing
- Support 4com or 8com DIG\_LED Driving Timing
- Support 7 / 8 pin matrix\_LED driving timing
- Support LED segment analog constant current configuration
- Support HOSC / LOSC for SEG\_LCD & DIG\_LED clock source

所以, US282A 对 LED 数码屏的操作将非常简单, 不用理会时序细节问题, 不会关心状态灯和数码管的亮度控制细节问题, 非常省事。

LED 数码屏的亮度是有其对应 IO 口的驱动能力决定的, 驱动能力越大, 亮度越高, 与刷屏频率无关。

LED 数码屏分为很多显示单元, 每个状态灯是一个单元, 每个数码管也是一个单元, 我们需要对这些单元进行编址。这是逻辑层面上的编址, 放在 `led_driver.h` 中定义。另外, 每个 7 段数码管都有 7 个小灯管, 也需要根据显示内容进行编码, 显示内容可以是数字、某些字母和某些特殊符号, 实际编码是放在 LED 驱动程序中的, 但是编码序号定义是放在 `led_driver.h` 中的。

而在 LED 驱动程序中, 我们需要对 LED 数码屏进行物理编址, 包括每个 7 段数码管的 7 个灯管。编址细节请参考所选 LED 数码屏的 SPEC, 我们只是提供了 `COM0_SEG[0~31]` 和 `COM1_SEG[0~31]` 的设置算法。

### 14.2.4 LED 驱动程序的使用说明

鉴于 LED 数码管不是每个音箱都需要, 所以我们增加了宏开关和配置项开关来对 LED 数码管进行屏蔽。

如果方案不需要 LED 数码管并且想把 LED 数码管的所有接口调用都去掉, 以节省代码空间, 那么请将 `case_independent.h` 中的 `SUPPORT_LED_DRIVER` 宏开关定义为 0。

如果同一个 SDK, 我们想做到有些方案支持 LED 数码管, 有些方案不支持, 并且希望在不修改代码的

情况下实现，那么可以让 `SUPPORT_LED_DRIVER` 宏开关定义为 1，然后通过修改配置项 `SETTING_DISPLAY_SUPPORT_LED_SEGMENT` 达到目的。

在支持 LED 数码管显示的方案中，如果在用户无操作的情况下，我们可以将 LED 数码管休眠，以降低功耗。无操作多长时间可以通过配置项 `SETTING_DISPLAY_LED_SEGMENT_LIGHT_TIMER` 修改，0 表示不启用不操作休眠功能。

## 14.2.5 FAQ

### 14.2.5.1 如何更换一款 LED 数码屏

LED 数码屏的显示分为 NUM\_X 显示和 ICON 显示。NUM\_X 表示数字和字母，例如 1,2, A,B 等；ICON 表示图标，例如 FM 等。

更换一款新的 LED 数码屏，首先要知道该 LED 数码屏的编码。可以通过 led 的 datasheet 或通过 try 的方式获得。标案中用的 LED 数码管见图 14.2.5.1.1。



图 14.2.5.1.1

假设用下面的长条形数码管图 14.2.5.1.2 替换掉标案中的圆形数码管：



图 14.2.5.1.2

步骤 1：获取两种数码管的编码表：

圆形数码管编码表

|   | 1  | 2     | 3   | 4  | 5    | 6    | 7  |
|---|----|-------|-----|----|------|------|----|
| 1 | /  | 1a    | 1b  | 1e | card | aux  | X  |
| 2 | 1f | /     | 2a  | 2b | 2e   | 2d   | X  |
| 3 | 1g | 2f    | /   | :  | 3b   | full | FM |
| 4 | 1c | 2g    | 3f  | /  | 3c   | 4e   | X  |
| 5 | 1d | 2c    | 3g  | 4a | /    | 4c   | 4g |
| 6 | 3d | empty | 3e  | 4d | 4f   | /    | 4b |
| 7 | x  | x     | usb | x  | x    | 4a   | /  |

长方形数码管编码表

|   | 1  | 2   | 3   | 4  | 5  | 6    | 7  |
|---|----|-----|-----|----|----|------|----|
| 1 | /  | 1a  | 1b  | 1e | SD | play | X  |
| 2 | 1f | /   | 2a  | 2b | 2e | 2d   | X  |
| 3 | 1g | 2f  | /   | :  | 3b | stop | FM |
| 4 | 1c | 2g  | 3f  | /  | 3c | 4e   | X  |
| 5 | 1d | 2c  | 3g  | 3a | /  | 4c   | 4g |
| 6 | 3d | usb | 3e  | 4d | 4f | /    | 4b |
| 7 | X  | X   | aux | X  | .  | 4a   | /  |

圆形数码管编码表中 2b 的含义是，第 2 个数码管的 b 段，要将其点亮的话，7PINled 数码屏的第 2 脚为高电平，第 4 脚要为低电平。

步骤 2：观察上述编码表发现，NUM\_X 的编码方式相同，ICON 编码部分错位。例如圆形数码管的图标 aux 的编码和长方形数码管编码表中的图标 play 对应。在 led\_driver.h 中可知#define LCD\_AUX S13，只需将长方形数码管图标 play 的宏定义为 S13 即可。其它错位图标同理。

按照上述方法既可以完成 LED 数码屏的更换。

注意:为了不修改底层驱动，上面编码不一致只能是两种数码管 NUM\_X 编码错位，ICON 编码错位，不能出现下面的类似的情形，长方形数码管编码表中 play 的编码和圆形数码管编码表 2b 的编码对应的情形。

## 14.3 Welcome 模块

### 14.3.1 Welcome 的地位和功能

Welcome 模块，顾名思义，就是在设备启动时用来显示“欢迎”界面的模块；Welcome 模块是 CASE 中必不可少的组成部分，它属于开机初始化流程中的第一个环节，也是 CASE 能够进行二次开发的第一个环节，那些需要在上电后马上执行的功能就放到 Welcome 中。

从以上定义可以想到，我们需要在 **Welcome** 模块中完成以下任务：

1. 板子硬件初始化
2. 我们想在某个简单的条件下，比如按着某个按键的同时插 USB 线，让设备进入 ADFU 升级模式，所以必须挑一个特别的时机点，就是开机，并且是在 **Welcome** 这个很短暂的阶段；这样也符合人们的自然预期
3. 我们想在当只有电池供电时，电池电量极低的时候不让用户开机，并且尽量少耗电，那么在 **Welcome** 中做低电检测关机也是很合理的
4. 在确定了可以正常开机后，马上给出一些反馈，比如点亮电源 LED 灯，表示设备已经上电进入开机流程；在 **Welcome** 做这个事情，能让用户觉得设备响应很快，用户控制感非常好
5. 其他一些需要在一上电就做的事情，比如设置看门狗、给蓝牙控制器上电以加快蓝牙就绪的时间等

### 14.3.2 Welcome 的设计要点

**Welcome** 是在操作系统加载之前运行的一段代码，它是由上电引导程序调用的，采用绝对地址调用，必须将 **Welcome** 的入口函数链接到约定好的地址。如果引导程序分配给 **Welcome** 的内存地址改变了，必须修改 `welcome.xn` 中的 `WELCOME_START_ADDR` 地址变量。

**Welcome** 最终封装为一个 `*.bin` 文件，它只包含一个已链接了的输出段 `.text`，该输出段的起始地址就是上述入口地址，最大长度是由引导程序分配的，目前定为 **2KB**。

所以，**Welcome** 的 `Makefile` 只需要链接生成一个 `*.exe` 结果文件，然后用 `OBJCOPY` 工具将 `.text` 输出段抽取出来输出到 `*.bin` 文件中。

另外，US282A 支持在 **Welcome** 模块中访问配置项，方法如下：

1. 在打包脚本 `fwimage*.cfg` 中，修改一下关键字：

```
47 //需要解析的CONFIG.BIN文件的ID号,最多2+3(3个数位的十进制数)*8个字符,最多8个ID,该配置项用于普通应用程序
48 //03表示有三个配置项,080是第一个配置项,对应config id为80,也就是固定ID为80的配置项
49 //110为第二个配置项,121为第三个配置项
50 //以下两个配置值允许增加配置项,但不允许删除默认的配置项,不可更改配置项的顺序,配置项ID必须固定!!!
51 INF_PARSE_CONFIG_INDEX_FOR_APP = "03080110121";
```

这样就将最多不超过 8 个配置项的默认值，按照所列顺序排列，每个配置项配置项 2 个字节。

2. 在 **Welcome** 模块中，用以下方式就可以访问到配置项默认值了：

```
__section__(".FIRST") void disp_starting(void (*adfu_launch)(void),
 void (*read_res)(uint8*, uint16, uint16), welcome_argument_t *welcome_arg)
{
 uint16 *cfg_value = (uint16 *) welcome_arg->cfg_buffer;
 uint8 charge_mode = (uint8) cfg_value[0];
}
```

### 14.3.3 Welcome 的启动流程

引导程序调用 Welcome 入口函数，进入 Welcome 启动流程，步骤如下：

1. 板子硬件初始化，以及一些可以在下面步骤 2、3 之前做的小事情，比如打开看门狗
2. 判断是否要进入 ADFU，如果是则跳到 ADFU
3. 判断是否电池低电关机，如果是进入 S4 关机
4. 开机 Welcome 反馈，一般通过显示功能进行反馈，这必须在步骤 2、3 之后做
5. 其他需要在步骤 2、3 之后做的事情，比如蓝牙控制器上电

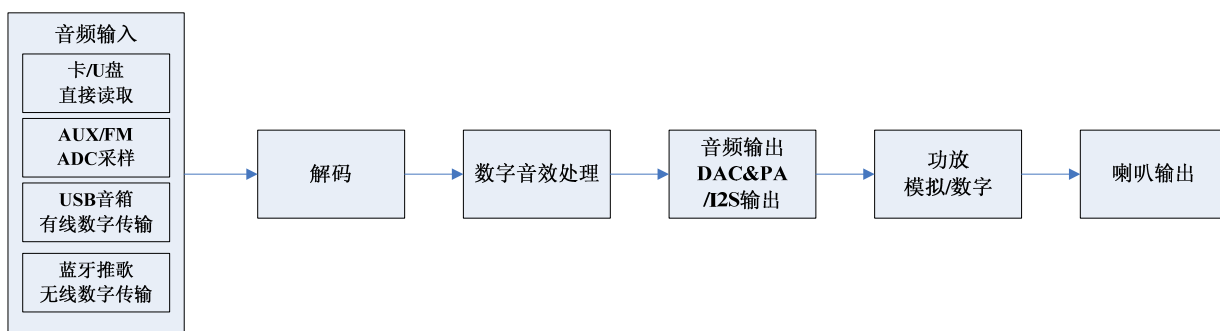
### 14.3.4 注意事项

1. Welcome 模块生成的 welcome.bin 是由大小限制的，比如在 2KB 以内。
2. Welcome 模块运行时操作系统尚未加载，不能调用操作系统 API 接口。



## 15 音频播放模型介绍

US282A 是一个音箱方案，主要功能就是音频播放，所以这里简单介绍一下音频播放模型，以便大家对音频播放有完整的概念，对分析音频播放的问题有较大的帮助。



下面以 AUX 通道为例，说明音频播放模型。

- 1) 外部输入：AUX 的输入是模拟输入，手机/PC 机通过 AUX 输出到音箱，US282A 要求模拟信号输入到 IC AIN 引脚的信号大小最大不能超过 1V，否则就会产生失真。
- 2) AIN 模拟输入处理：模拟输入到 IC，可以进行增益控制，一般为 0dB。如果输入超过 1V，那么 AIN 就会失真。
- 3) ADC 采样：AD 采样也可以进行增益控制，一般为 0dB。如果 ADC 输入超过 1V，那么 ADC 就会失真导致破音。
- 4) 解码：解码一般比较独立，对于 AUX 通道而言，并不需要实际的解码功能，所谓的解码只是主导整个 DSP 运行，包括调用数字音效处理。解码模块的输出是 16bit 的 PCM 音频数据。
- 5) 数字音效处理：输入的是 16bit 的 PCM 音频数据，输出也是 16bit 的 PCM 音频数据，它根据数字音效参数对 PCM 进行处理，将某些频段提升以突出音效，将某些腔体喇叭不适合播放的频段消除掉以防止出现杂音破音。如果数字音效处理的参数配置错误，会导致破音问题。
- 6) 音频通道输出选项：可以将最终输出的数字信号进行后再输出，比如将左右通道混合后输出到左声道和右声道，将左右通道互换输出等。这样就不用在外围电路进行处理，节省成本。
- 7) 音频输出 DAC+PA：这是 IC 内部输出的模拟信号控制，最大音量为 0dB+0dB。如果不小心将 DAC 的音量设置到大于 0dB，那么如果数字信号输出也为 0dB，那么就会破音（即信号+DAC 音量大于 0dB 就会破音）。
- 8) 外部功放：外部功放对 IC 输出的模拟的或数字的信号进行放大，如果放大后输出功率超出额定功

率，就会出现破音失真等音质问题。

## 16 CASE 软件开发要点

上面基本上覆盖了 CASE 软件开发涉及的所有大方面，本章我们将对 CASE 软件开发的要点进行介绍，希望能够对 CASE 软件开发的某些方面进行更好的指导。

### 16.1 多线程设计与开发

#### 16.1.1 多线程架构

##### 16.1.1.1 应用程序、进程、线程、任务及其优先级

**应用程序**被操作系统装载进来就变成了**进程**，即拥有独占的代码和数据空间的主体，引进进程概念，是为了支持**多线程**，以便实现一个应用程序内多个相对独立的功能模块的并行运行；应用程序的主体就被封装为一个主线程。US282A 最多支持 5 个进程，实际上已经用完了。

在一个**应用程序**里面，如果包含一些与**主功能模块**相对**独立的功能模块**，比如本地播歌前台应用，主功能模块就是处理本地播歌的用户交互，但是它还有其他独立功能模块(扫描磁盘建表、按键音播放、TTS 播放)，为了实现这些模块与主功能模块的并行运行，以提升用户体验，减少串行延迟，就将这些独立功能模块实现为**子线程**。US282A 最多支持 8 个线程同时运行，每个线程都要有自己的运行栈空间，所以并不能一味添加线程，它受可用栈空间的限制。

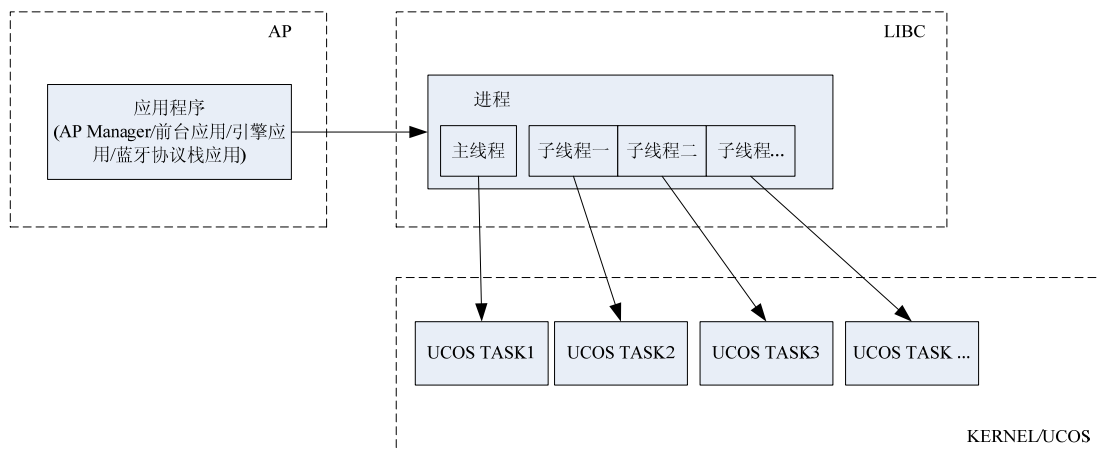
**线程**在 UC/OS II 中被封装为**任务**实体，最终是以任务进行调度的。**任务**是 UC/OS II 的核心主体，是实时抢占式调用的基本单元。任务优先级是任务调度最重要的依据，任务优先级越高，即其值越小，任务就越容易被调度到。US282A 支持 16 个任务，优先级为 0 ~ 15 。

US282A 的优先级分配如下所示，定义在 `psp_rel\include\task_manager.h` 中：



| 任务优先级（值越小优先级越高）            | 任务名称/类型      |
|----------------------------|--------------|
| KEYTONE_PLAY_PRIO 4        | KEYTONE 子线程  |
| AP_BACK_HIGH_PRIO 3        | 解码线程/TTS 子线程 |
| AP_BACK_LOW_PRIO 6         | 引擎应用主线程      |
| AP_BT_HIGH_PRIO 7          | 蓝牙协议栈应用      |
| AP_FRONT_HIGH_PRIO 9       | 前台子线程        |
| AP_FRONT_LOW_PRIO 10       | 前台应用主线程      |
| AP_PROCESS_MANAGER_PRIO 12 | 应用管理器        |
| OS_LOWEST_PRIO 15          | 系统 IDLE 任务   |

应用程序、进程、线程、任务的关系如下图所示：

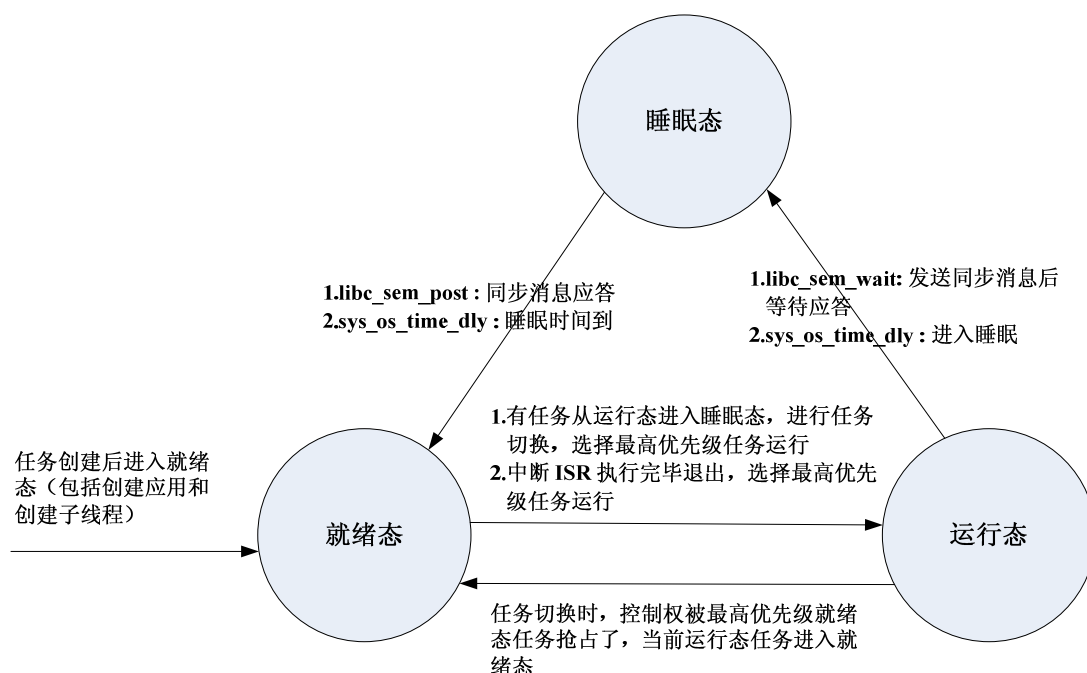


### 16.1.1.2 任务调度机制

UC/OS II 的基本单元是任务，任务的运行状态有 3 种：

- 睡眠态
- 就绪态
- 运行态

任务的这 3 种状态的转换关系（并不是完整的，是在应用层常见的）如下图所示：



说明：

- 为了保证优先级较低的任务，比如 AP manager，能够有机会获得控制权运行，高优先级的任务必须适当的主动进入睡眠，即调用 `sys_os_time_dly` 函数。一般来说，我们要求每个线程主循环都要在循环体内调用 `sys_os_time_dly` 主动睡眠一段时间，睡眠时间长度则依据具体场景而定。
- 操作系统设计了一个 10ms 的 timer tick 硬件定时器中断，所以每隔 10ms 就会检查是否有任务睡眠时间已到，如果有则把任务置为就绪态。timer tick 中断 ISR 执行完毕退出，会进行任务切换。

### 16.1.1.3 创建子线程

如果在某个场景的栈空间允许，我们就可以复用一個沒用的线程，或者创建一个新的线程，创建方法如创建 KEYTONE 子线程：

```
pthread_ext_param_t param;
param.pthread_param.start_rtn = (void *) keytone_play_loop;
param.pthread_param.arg = (void *) (uint32) kt_id;
param.pthread_param.ptos = (void *) KEYTONE_PLAY_STK_POS;
param.stk_size = KEYTONE_PLAY_STK_SIZE;
```

```
if (libc_pthread_create(¶m, KEYTONE_PLAY_PRIO, CREATE_NOT_MAIN_THREAD) < 0)
{
 //创建失败
}
```

创建子线程时，要仔细描述好栈空间和优先级，如果有冲突，就会造成非常严重的错误。

#### 16.1.1.4 销毁子线程

正如我们不会强制杀死应用程序一样，我们并不会强制销毁子线程，而是子线程在执行完成后，在子线程 LOOP 函数的最后调用 `libc_pthread_exit()`；接口主动退出子线程。

需要特别注意的一点是，子线程 LOOP 函数在调用了 `libc_pthread_exit()`；接口后，不会再返回，所以不能在之后编写任何有意义的代码，否则将会出错。

#### 16.1.1.5 多线程注意事项

1. 如果子线程与主线程共享一个 BANK 组，那么就要谨慎了。在没有锁调度或关中断的前提下，在 BANK 中使用 BANK DATA，或者使用 `const` 数据指针传参来调用不是本 BANK 的接口，否则就有可能出现 BANK 被切走导致数据和参数错误的情况。
2. 多线程是一把双刃剑，它简化了多个任务并行运行时的运行控制权问题，但是引进了线程同步和资源冲突的问题，比如上述的 BANK 使用的问题。使用多线程时，必须在设计上充分验证其线程按键性，以及线程调度实时性，否则会带来无尽的麻烦。

### 16.2 进程/线程间通信方式

#### 16.2.1.1 通信方式概述

US282A 支持以下多种进程/线程间通信方式：

1. 信号量
2. 互斥锁
3. 条件变量
4. 消息队列

5. 共享查询
6. 共享内存
7. 跨进程的全局变量

### 16.2.1.2 信号量

### 16.2.1.3 互斥锁

### 16.2.1.4 条件变量

### 16.2.1.5 消息队列

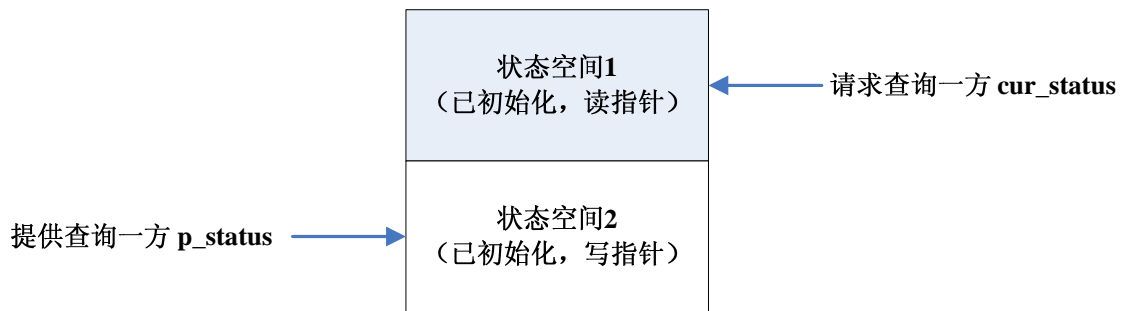
### 16.2.1.6 共享查询

共享查询是一种进程/线程通信交互开销非常小，并且很安全的一种进程/线程间通信方式。它主要应用于状态查询，比如前台应用查询引擎应用的状态。

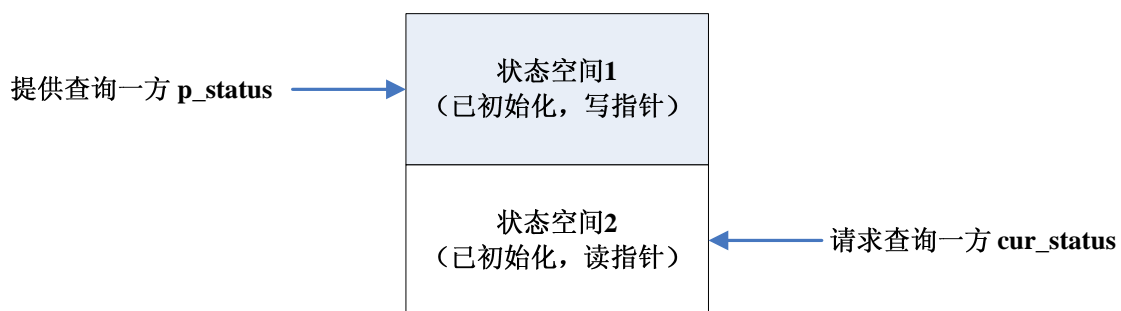
共享查询的实现机制是这样的：

共享查询要求提供查询服务的一方提供所要查询的状态的双倍内存空间，用于乒乓读写；而请求查询服务的一方，需要提供一份用于拷贝所要查询的状态的内存空间。

首先，提供查询的一方对状态空间 1 进行初始化，然后调用 `void* sys_share_query_creat(int8 query_id, uint8 *mem_addr, uint16 size)`；创建共享查询，状态指针指向状态空间 1。



之后，提供查询一方使用 **p\_status** 去访问和更新状态，然后在合适的时机调用 `void* sys_share_query_update(int8 query_id)` 更新状态，更新后的共享查询视图变为：



请求查询的一方，任何时候来请求更新状态，都只是将读指针的状态空间 2 拷贝到 **cur\_status**，不会改变共享查询视图，其接口形式为 `int sys_share_query_read(int8 query_id, uint8 *read_addr);`。

当应用场景退出时，提供查询的一方调用 `int sys_share_query_destroy(int8 query_id);` 销毁共享查询。

以上所有接口执行时都会关中断，所以不用担心发生线程访问安全问题。

从上面分析可以知道，共享查询开销很小，它不会导致请求服务的一方进入睡眠状态，也就是说不会主动产生任务切换，只是需要对占用提供查询一方多 1 份的状态空间。

目前 US282A 提供了 4 个共享查询资源，在大部分应用场景下，我们已经使用了以下 2 个：

1. 前台应用查询引擎应用的状态。
2. 蓝牙管理器查询蓝牙协议栈应用的状态。

也就是说，我们还能够使用 2 个共享查询。

使用共享查询要注意以下几个点：

1. 共享查询结构体中不能使用指针成员，否则会出错。使用指针成员有 2 种情形：
  - a) 指向某块 `buffer` 的指针：共享查询要求提供 2 份完全的状态空间，使用指针成员，由于共享查询更新时使用指针软拷贝，必定不能达到 2 份 `buffer` 更替的效果。
  - b) 指向共享结构体某个数组成员的某个元素的指针：一旦指针在某一时刻指向了某个元素，那么它就被固定了指向状态空间 1 或状态空间 2 的某个元素，不会再改变了，这样就不能达到 2 份 `buffer` 更替的效果，结构体定义举例如下：

```
Struct
{
 Struct dev my_devs[2];
 Struct dev *p_dev;
};
```

2. 共享查询的 ID 不能重复，我们现在使用应用程序的 APP ID，是不会重复的；但是预留了 2 个共享查询注意不要使用 APP ID (0 ~ 31) 的值。

### 16.2.1.7 共享内存

共享内存则更加简单，我们是为了避免多个进程/线程使用“全局变量”进行通信而产生的数据共享耦合而增加的一种进程/线程间通信方式。这是一种降低系统耦合性的有效手段，能够提升系统的健壮性。

共享内存的用法如下：

提供共享内存的一方要先创建一个共享内存，即：`int sys_share_query_creat(int8 shm_id, uint8 *shm_addr, uint16 shm_size);`；然后共享的另一方就可以挂载该共享内存了 `uint8* sys_shm_mount(int8 shm_id);`，它得到了一个指向该共享内存的指针，然后就用该指针来访问该内存；当应用场景退出时，提供共享内存的一方调用 `int sys_shm_destroy(int8 shm_id);` 删除共享内存。

目前 US282A 提供了 8 个共享内存，我们现在已经在 `psp_rel\include\share_memory_id.h` 分配了以下几个了：

```
#define SHARE_MEM_ID_BTPLAYPIPE 0 //蓝牙推歌 pipe 管理结构体，由 bt stack 共享
#define SHARE_MEM_ID_BTSTACK_STATUS 1 //蓝牙协议栈当前状态，由 bt common 共享
#define SHARE_MEM_ID_BTRCPPPIPE 2 //蓝牙 RCP pipe 管理结构体，由 bt stack 共享
#define SHARE_MEM_ID_DAECFG 3 //DAE 配置结构体，由 config 申请空间并共享
#define SHARE_MEM_ID_USOUND 4 //usound 引擎与前台共享 pc 同步的音量和标志
```

```
#define SHARE_MEM_ID_AGC_AEC
```

```
5 //btcall and dap lib share memory
```

使用共享内存要注意以下几点：

1. 在数据访问线程安全性上，共享内存实际上跟“全局变量”是一样的，它的线程安全性由用户自己把握，一般通过互斥锁、或者关中断等方式来访问，以确保访问安全，特别是低优先级的线程。比如，高优先级线程共享内存给了低优先级线程，那么低优先级线程在访问时，如果没有锁调度或者关中断，那么访问到一半，高优先级线程抢占运行，又更新了该共享内存，这样低优先级线程原本需要一次性访问的数据就被破坏了，导致出错。
2. 共享内存的 ID 不能重复，请在 `psp_rel\include\share_memory_id.h` 中统一定义。

### 16.2.1.8 跨进程的全局变量

使用全局变量或者绝对地址来通信，是非常简单，但是有 2 大问题：

1. 工程问题：如果某个进程链接时修改了全局变量的地址，或者修改了绝对地址的值，而另一个进程没有同步修改，那么必定会导致通信错误。甚至，修改了绝对地址宏定义，某个进程重新 `make`，而另一个没有，也都会导致通信错误。
2. 数据访问的线程安全性问题：必须由用户自己把握，需要自己使用锁调度或关中断等方式来保证数据访问的安全性。

所以我们不推荐这种用法，尤其是因为第一个原因，它是系统健壮性的一个麻烦制造者，会在不经意之时引进一些非常诡异的问题。

我们建议使用共享内存和共享查询来代替跨进程的全局变量。

## 16.3 头文件依赖关系

1. `Case_include.h`：只包含公共头文件。
2. `Case_independent.h`：作为 `CASE COMMON` 和 驱动模块的公共头文件。
3. `Psp_includes.h`：`psp` 公共模块头文件。
4. `PSP` 其他外部头文件，`CASE` 各模块按照依赖关系具体包含进去。
5. 每个应用的前台应用和引擎应用共享的头文件，比如 `btplay_common.h` 由蓝牙推歌前台和蓝牙推歌引擎共享，可以用来定义它们共享的宏、枚举类型和结构体等。

## 16.4 多方案开发环境

多方案开发有几种情况：

1. 多个方案使用相同的 SDK，仅仅是功能配置上的差异，或者功能有无的差异。这在同一厂商的同一阶段产品中挺常见的。
2. 基于同一个 PSP 的多个不同 CASE。
3. 由于升级了 PSP，后续的方案必须使用一个全新的 SDK。

case\_independent.h 中定义了方案很多模块的宏开关，它是驱动和应用程序都会包含的头文件，可以用来定义一些同时作用于驱动和应用程序的宏开关；然后每个方案都有一个方案宏开关，如下所示：

```
#define CASE_BOARD_EVB 0
#define CASE_BOARD_DEMO 1
#define CASE_BOARD_ATS2823 2
#define CASE_BOARD_DVB_ATS2825 3
#define CASE_BOARD_DVB_ATS2823 4
#define CASE_BOARD_TYPE CASE_BOARD_DVB_ATS2825
```

然后各个模块开关根据该宏分别定义：

```
#if (CASE_BOARD_TYPE == CASE_BOARD_DEMO)
#define SUPPORT_AUX_DETECT DETECT_BY_GPIOB0 //是否支持 AUX 拔插检测
#elif (CASE_BOARD_TYPE == CASE_BOARD_DVB_ATS2825)
#define SUPPORT_AUX_DETECT DETECT_BY_GPIO //是否支持 AUX 拔插检测
#else
#define SUPPORT_AUX_DETECT DETECT_NO_SUPPORT //是否支持 AUX 拔插检测
#endif
```

config.txt 固件配置项文件，每个方案有一个，其名字可以不要 config 前缀，比如直接是方案名称，US282A\_BTBOX\_DVB\_ATS2825.txt。

fwimage.cfg 固件打包脚本，每个方案有一个，其名字必须是 fwimage\_XXX.cfg，XXX 表示对应的 config.txt 的名称，比如 fwimage\_US282A\_BTBOX\_DVB\_ATS2825.cfg。



## 16.5 EJTAG 调试接口关闭

我们以前使用 GPIO 口，有时候会遇到它与当前配置的 EJTAG 口有冲突，在不知情之下，调试好久，才发现是 EJTAG 口冲突，浪费时间。

那么我们现在这样处理：在 welcome 中增加一个 welcome\_ejtag 接口，在该接口，我们会说明 EJTAG 功能开关情况，以及占用那些 IO 口资源。

这样，就很清楚了。如果某个 CASE 没有预留一整个空的 EJTAG 口，那么 EJTAG 功能就必须关掉。如果有，那么就可以使用。

另外，为了让方案能够在某个可控位置进入调试模式查看现场，我们在 COMMON 中增加了 com\_enable\_jtag\_function 接口，可以在关闭 EJTAG 的情况下调用，使能某组 EJTAG，进入调试模式。

## 16.6 使用打印调试

在 3.3.2 串口打印 一节我们已经说过了，这里再强调一下，并添加一些具体实现的细节。

1. 除了 DMA 打印接口之外，其他打印接口在真正打印时都不会关中断，而是锁调度。
2. 因为 DEBUG\_PRINT\_IRQ(str,data,mode) 打印需要由前台或引擎应用来代理打印出来，所以如果前台或引擎应用调度不到，那么就不会打印出来。这种打印缓冲，最多只能缓冲 8 条，也就是说，如果在调度到前台或引擎应用之前，一口气打印了超过 8 条的，那么就会将前面的一些条目给覆盖掉。另外，每条打印最多只能打印 31 个有效字符，包括前导字符串在内，超过 31 字符的部分将被丢弃，如果是打印数值，那么前导字符串最长只能是 21 个字符，这样能够保证数值会被完整打印出来。

## 16.7 编译工具链

### 16.7.1.1 编译优化选项

GCC 有多种编译优化选项，我们常用的有几种：

O0：关闭所有优化选项，优点在于汇编代码阅读起来很容易，调试很方便。

O1：最基本的优化等级，编译器会在不花费太多编译时间的同时试图生成更快更小的代码。

O2: 推荐的优化等级, 编译器会试图提高代码性能而不会增大体积和大量占用的编译时间。

Os: 用来优化代码尺寸, 这对于磁盘空间极其紧张或者 CPU 缓存较小的机器非常有用。但也可能会产生一些问题, 比如下面将说到的 `__section__` 问题。所以, 除非别无他法, 我们是不建议使用 Os 的, 如果使用了, 必须小心谨慎。

以上优化选项是在 `case\cfg\rules.mk` 定义的, CASE 中的各个工程或目录根据自己的需要进行选择, 比如 COMMON 模块现在选用了 Os 的编译选项, 规则如下:

```
$(OBJ_C_16): %.o: %.c
$(CC) $(CC_OPTS_Os_16) -o $@ $<
```

使用 Os 有时能比 O2 节省 10% 的代码, 还是挺划算的。

但是使用 Os 需要注意一种情况:

如果某个源文件中有 inline 函数, 如果将某个调用了 inline 函数的非 inline 函数使用 `__section__` 输出为一个输出段, 那么该非 inline 函数对 inline 函数的调用地址会出错, 这种出错情况同使用了头文件中 inline 小函数一致。所以, 为了系统的稳定性, 请不要将使用了 inline 函数的非 inline 函数使用 `__section__` 输出到一个单独的输出段中。

### 16.7.1.2 \*.o strip debug

\*.o 文件一般包含了很多调试信息段, 里面有很多路径信息, 这样会导致不同开发者, 同样从 SVN 下载下来的源代码, 如果存放的目录路径不一致, 编译出来的 \*.o 在二进制上是不相等的。

如果我们需要把这些 \*.o 上传到版本管理系统上, 那么就会对版本管理产生一些负面影响。

所以, 我们要求在将 \*.o 上传到 SVN 之前, 先用 `sde-strip --strip-debug $(OBJ)` 命令将调试信息段删除。

这样带来的一个问题是, 直接使用 strip debug 的 \*.o 进行链接的 \*.exe 映像文件, 这些 \*.o 对应的源文件不能在调试模式下看到源代码与现场的对应视图, 很难调试。所以, 当我们在开发调试时, 可以先把 sde-strip 命令暂时去掉, 再重新 make 出 \*.o 和 \*.exe, 进行调试。

### 16.7.1.3 Makefile 的陷阱

1. Makefile 规则中,并没有将所有实际的依赖关系表达出来,比如 \*.exe 对 COMMON 模块的 \*.o 的依赖关系是没有建立起来的。这样我们更新了 COMMON 模块的某个源文件,直接对整个 CASE 工程输入命令 make, \*.exe 并不会自动更新。我们必须自己先将 \*.exe 删除掉,然后再 make,输入命令为 make clean\_target; make 或者 make clean; make 。

2. 我们想通过 case\cfg 目录下的 Makefile 脚本对所有工程进行 make,但是如果 make 到其中的某一项发生错误,那么就会终止那个 make 命令,后面的 make 项目就不会执行。

但是,需要注意这样的陷阱,如果我们输入 make clean\_target; make,那么这实际上是 2 个命令,如果在执行 make clean\_target 时发现某个工程被删除了,它的 make clean\_target 自然会失败,从而导致后面的 make clean\_target 没有执行到;但是 make 命令还是会执行,并且所有工程都能够执行。问题来了,正如 1 点说到,如果没有 make clean\_target 将 \*.exe 删除,那么接下来的 make 我们并不会更新 \*.exe,导致该 \*.exe 没有更新到位,与系统其他工程不匹配,将出现各种很奇怪的问题。

例如,当我们编写出这样的 Makefile 时:

all:

```
make -C ../ap/ap_music
```

```
make -C ../ap/ap_config
```

clean\_target:

```
make -C ../ap/ap_music clean_target
```

**make -C ../ap/ap\_alarm clean\_target ap\_alarm 工程被删除了,该命令会失败,导致 make\_clean\_target 提前终止,下面的 ap\_config 的 make clean\_target 没有被执行。**

```
make -C ../ap/ap_config clean_target
```

因为 Cygwin 一直 make 下来,中间虽然报错了,但是工程师没有看到,结果就导致了这样的悲剧。

case\cfg 目录下的 Makefile 脚本还有另一种疏忽,就是把某个工程的 make 项目注释掉后忘记打开了,这种情况在工程师头晕晕时也可能发现不了,需要引起注意。

所以,我们必须谨慎维护好 case\cfg 目录下的 Makefile 脚本。

## 17 蓝牙推歌功能

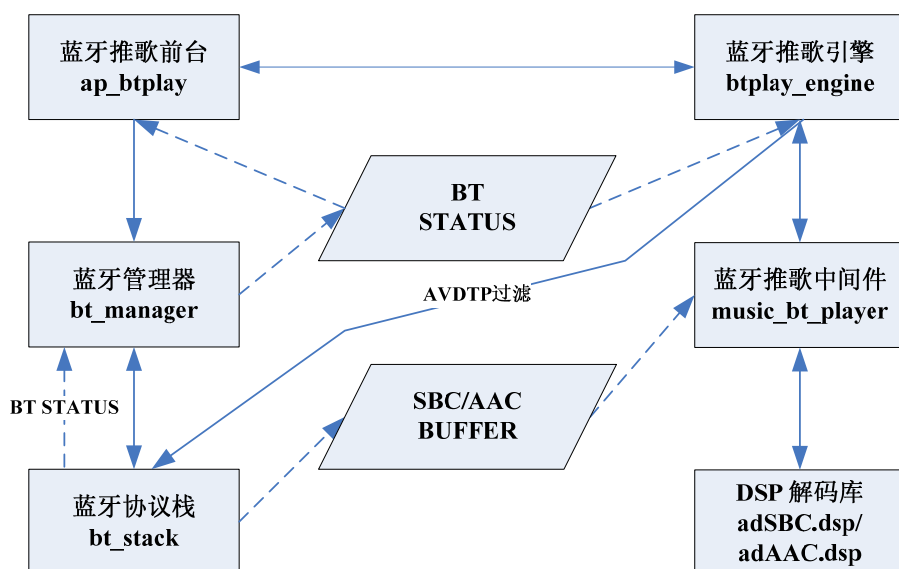
### 17.1 应用规格

蓝牙推歌应用的规格如下：

1. 支持 SBC 和 AAC 编码格式，其中 AAC 编码格式可以通过配置项配置是否支持，默认不支持。
  - a) SBC 规格：比特率 328Kbps@44.1KHz, 345Kbps@48KHz
  - b) AAC 规格：iPhone 手机大约是 160 ~ 192 Kbps@44.1KHz，所以音质并没有明显比 SBC 好
2. 支持 AVRCP 控制，可以进行音量同步、切换上下曲、播放/暂停、快进快退等控制。
3. 支持 AVRCP 语音提示，包括暂停、播放、上一曲、下一曲等。
4. 支持连接状态语音提示，包括等待连接、连接成功、连接断开等。
5. 支持播放延时可配置，但是配置得太短会导致一些蓝牙推歌功能做得不好的手机或平板比较容易断音。
6. 支持连接起来后自动播放，可以通过配置项开关，默认不开启。

### 17.2 应用架构

蓝牙推歌应用主要由 6 个程序实体组成：



## 17.3 蓝牙推歌前台设计

### 17.3.1.1 功能及设计

蓝牙推歌前台的主要功能及其设计如下：

- 负责与用户交互，用户通过按键可以控制蓝牙播歌暂停/播放/切上下曲/快进快退/调节音量等。

**设计：** 蓝牙推歌用户所做的以上操作实际上并非直接对本地播放器进行操作，而是转为对远程设备音乐播放器的控制，即调用 `bt_manager` 的 AVRCP 适配层接口；调节音量会调节本地的音量，并且如果建立了音量同步，会同时发送音量同步命令给远程设备，当然远程设备调节音量也会同步给音箱，这时候并不需要前台干预，`bt_manager` 自己回调音量设置接口。

- 一般情况下，作为蓝牙功能的主场景，提示蓝牙连接状态变化。

**设计：** 关于蓝牙连接状态变化的提示，US282A 有多种解决方案，现在一一说明：

- 在蓝牙推歌前台中，可以选择仅对 A2DP 服务连接状况进行提示，或者选择对 A2DP 或 HFP 服务连接状况进行提示。第一种方式，我们能够做到一旦提示“连接成功”，就立即可以推歌出声音，但是由于回连时是先连接 HFP 服务再连接 A2DP 服务，所以回连时间会慢一点；第二种方式，回连时间会快一些，但是提示“连接成功”后并不能马上推歌出声音，如果在这时候手机马上推歌，会先从手机喇叭出声音几秒钟，然后等 A2DP 连接起来后再从音箱出声音，用户体验不是很好。

- 2) 可以仅由 `bt_manager` 进行提示，目前 `bt_manager` 提示只根据蓝牙物理连接状态进行提示。
- 3) 提示可以用 `KEYTONE`，也可以用 `TTS`，甚至仅仅通过蓝牙指示灯 `LED` 提示。`KEYTONE` 提示不用打断播放，体验更佳。

### 17.3.1.2 状态

蓝牙推歌前台通过获得以下 2 个状态作为其业务逻辑执行的条件：

#### 1. 蓝牙推歌引擎共享查询的播放状态机： `btplay_status_e`

```
typedef enum
{
 BTPLAY_IDLE = 0,
 BTPLAY_STOP = 1,
 BTPLAY_PLAY = 2,
 BTPLAY_PAUSE = 3,
} btplay_status_e;

typedef struct
{
 btplay_status_e status;
 uint8 reserve[3]; //共享查询结构体需要 4 字节对齐
} btplay_info_t;
```

这是由蓝牙推歌引擎维护的状态机，蓝牙推歌引擎根据复杂的规则维护这个状态机，具体后面说明。

#### 2. 蓝牙管理器共享内存的蓝牙状态 `bt_stack_info_t`（在 `btstack_common.h` 中定义）

该状态包括蓝牙物理连接状态、蓝牙 A2DP 服务状态、蓝牙 HFP 服务状态等。

### 17.3.1.3 应用结构

蓝牙推歌前台应用比较简单，它只包含一个主视图——播放主视图，所以不需要主视图调度程序。

应用初始化：

1. 平台要求的初始化 `_app_init()`；。
2. 创建播放主视图 `btplay_create_main_view()`；

3. 杀死其他引擎应用 `kill_conflict_console()`;
4. 解除静音并重新设置音量 这个是每个前台应用都必须做的，确保新进入应用是有声音的。
5. 播报 TTS “蓝牙推歌”
6. 如果之前蓝牙管理器尚未创建，那么创建蓝牙管理器（蓝牙管理器会启动蓝牙）。
7. 判断是否要快速退出应用，如果是则退出应用。
8. 切换按键音通道为通道 1，`keytone_set_dac_chan(DAF1_EN)`；，因为蓝牙推歌播放通道为通道 0。
9. 创建蓝牙推歌引擎应用 `create_console()`；。
10. 进入 `get_message_loop()`，即蓝牙推歌主视图。

应用退出：

1. 杀死蓝牙推歌引擎应用 `kill_console()`；。
2. 切换按键音通道为默认通道 0，`keytone_set_dac_chan(DAF0_EN)`；。
3. 切换应用 `com_ap_switch_deal(result)`；。
4. 平台要求的退出 `_app_exit()`；。

### 17.3.1.4 播放主视图设计

按键消息响应：`btplay_ke_maplist`

包括上一曲、下一曲、暂停/播放、快进及取消快进、快退及取消快退。

系统事件响应：`btplay_se_maplist`

没有什么系统事件需要做特殊处理。

视图初始化：

1. 即应用初始化，这时基本上已初始化好了。
2. 等待获取蓝牙推歌引擎的共享查询成功，表示引擎已经正常工作了。
3. 初始化用户交互状态机。
4. 创建自动播放处理定时器。

视图主循环：

1. 获取蓝牙推歌引擎的共享查询状态 `bt_status_deal()`;
2. 如果判断有状态变化，需要刷新 UI，则调用 `com_view_update(APP_VIEW_ID_MAIN)`;

3. 如果判断有状态变化，需要 TTS 提示，则调用 `btplay_tts_play()`;
4. 自动播放处理 `btplay_autoplay_deal()`;
5. VIEW LOOP 调用 `result = com_view_loop()`;

视图退出：

1. 为了防止快进快退的过程中退出应用后没有发送取消快进快退的命令，这里保证一下。
2. 杀死自动播放处理定时器。

## 17.4 蓝牙推歌引擎设计

### 17.4.1.1 功能及设计

蓝牙推歌引擎的主要功能及其设计如下：

1. 负责装载和卸载蓝牙推歌中间件，我们要在快速播放和尽量降低功耗之间取得平衡。

**设计：**我们采用预加载的方式来装载蓝牙推歌中间件，并在蓝牙进入空闲状态时卸载以降低功耗。

- 1) 预加载条件：

- a) 尚未加载中间件
- b) 没有在 TTS 播放状态中
- c) A2DP 服务已连接
- d) 没有设置 AVDTP 数据过滤，并且已经有 AVDTP 数据传输过来了

- 2) 卸载条件：

- a) 已经加载了中间件
- b) A2DP 服务已断开，或者蓝牙进入空闲状态，或者收到 TTS 播放请求

2. 向蓝牙推歌前台提供一个蓝牙推歌播放状态机，前台需要根据这个状态机了解播放器状态，以正确执行用户的请求。

**设计：**该状态机就是前面说的 `btplay_status_e`，它的维护需要考虑以下几个方面的信息：

- 1) 蓝牙协议栈维护的 A2DP 播放状态，它由以下 3 个方面信息决定：

- a) AVDTP Stream 命令，有 start 命令、suspend 命令，但是手机有可能不会发出该命令就直接推送或停止推送数据。
- b) AVDTP Stream 状态变化同步，有 play 状态、pause 状态、stop 状态，需要支持播放状态同步的手机才会上报这个状态，但是手机有可能不发出状态变化同步事件。



- c) 蓝牙推歌前台调用蓝牙管理器的发送 AVRCP 控制命令，为了配合 UI 和 TTS 等交互需要，蓝牙协议栈会临时将 A2DP 状态修改为预期的状态，比如蓝牙协议栈收到 AVRCP PAUSE 命令，会立即将 A2DP 状态更新为 A2DP\_STATUS\_PAUSE 状态。
- 2) 由于蓝牙协议栈的 A2DP 状态有可能因为概率性的重要事件没有收到而不正确，所以我们必须实时监控 AVDTP Stream 数据流，即蓝牙协议栈会反馈“当前有 Data”和“当前无 Data”这两种状态，它们以 200ms 为统计窗口，如果连续 200ms 没有收到数据包，就认为停止推送数据了。蓝牙推歌引擎会根据这个状态来主动修改播放状态。
- 3) 蓝牙推歌前台发起暂停命令，不同手机/播放器的响应是不一致的，有些能够停下来，有些不能停下来，淡出时间也不一致，淡出结束后可能会持续一段时间发送静音数据，并且不同手机/播放器的时长也不一致。所以，我们需要通过一些手段获取到手机类型信息以区分手机，并且为了解决淡出结束后可能会持续一段时间发送静音数据，我们最好能够检测静音数据。

### 17.4.1.2 状态

#### 1. 蓝牙管理器共享内存的蓝牙状态 `bt_stack_info_t`（在 `btstack_common.h` 中定义）

该状态包括蓝牙物理连接状态、蓝牙 A2DP 服务状态等。

#### 2. AVDTP Data Pipe

这是蓝牙协议栈共享内存的 AVDTP 流数据管道管理结构体，通过该状态可以了解当前有无数据存储到蓝牙推歌输入缓冲区中。

### 17.4.1.3 应用结构

蓝牙推歌引擎应用非常简单，它不用像本地播歌那样进行复杂的本地环境初始化，并且也不用一进入就创建中间件，其结构大致如下：

应用初始化：

1. 平台要求的初始化 `_app_init()`；。
2. 创建 `btplay_status_e` 的共享查询。
3. 挂载音效参数共享内存，以便为音效库设置运行频率。
4. 解除 AVDTP 流数据过滤。

5. 进入消息主循环。

应用退出：

1. 收到 MSG\_APP\_QUIT 消息，退出消息主循环。
2. 设置 AVDTP 流数据过滤。
3. 销毁 btplay\_status\_e 的共享查询。
4. 平台要求的退出 \_app\_exit(); 。

消息主循环：

1. btplay\_status\_e 状态更新。
2. 中间件预加载和卸载处理。
3. 自身状态更新，并以此来更新 btplay\_status\_e 的状态。
4. 应用私有消息接收并处理。
5. 更新引擎播放状态，该状态与低功耗、省电关机有关。

## 17.5 蓝牙推歌中间件

蓝牙推歌是被动播放的播歌应用，播放和停止都是由手机决定；前台应用发起的播放/暂停命令，只是发送到蓝牙协议栈，由蓝牙协议栈转给手机，手机再异步响应，最终结果我们并不能确定。

所以，引擎并不能完全控制播放声音，它让中间件播放起来，只表示允许中间件启动播放，至于什么时候启动并播放出声音，由手机推送流数据决定。

播放启动条件为：

1. 数据已经缓冲了至少 46ms。
2. 再满足以下条件之一：
  - a) 如果数据已经缓冲超过了“播放起始延时”配置项 BTPLAY\_SBC\_PLAY\_DELAY\_START 的值，马上启动播放。
  - b) 如果数据缓冲没这么多，那么经过一段时间等待，也会启动播放。所以短暂的按键音能够播放起来。

还有，引擎让中间件暂停，实际上是我们强制让中间件停止播歌并设置过滤后续 AVDTP 流数据，所以很快就停止下来了。

蓝牙播歌断音问题，是指手机发送过来的 AVDTP 流数据短暂中断了或一段时间阻塞了，导致中间件没有足够的数据提供连续播放而断音；断音时中间件会主动做淡出处理，并进入暂停状态，之后数据恢复过来并达到播放启动条件，就会恢复播放，播放时会做淡入处理。

## 17.6 注意事项

1. 引擎应用接收并处理前台应用的私有消息，很多都是同步消息，务必保证所有同步消息都处理到了并调用 `libc_sem_post(p_sem);` 进行答复了。

## 18 蓝牙免提功能

### 18.1 应用规格

蓝牙打电话：有 6 种状态，即空闲，停止，来电，拨号，音箱通话，手机通话。

- 1) 蓝牙打电话也是一个完整的稳定应用，可以进行连接等用户交互。
- 2) 蓝牙打电话有 2 种模式可以进来，一种是正常模式，进入一种稳态。一种是嵌套打电话模式，打完电话后就返回原应用，也叫后台打电话模式。
- 3) 蓝牙打电话支持 **HFP** 音量同步控制，支持手机\小机音源切换。
- 4) 蓝牙通话时时，避免语音的干扰，屏蔽 **TTS** 播报。
- 5) 进入正常模式，要播报蓝牙免提、等待连接和连接成功，后续如果已经连接成功，就只播报蓝牙免提；如果是嵌套打电话模式，就不会播报任何 **TTS**。

后台打电话功能，是通过蓝牙管理子模块实时查询蓝牙 **HFP** (Hands-free Profile) 服务状态，监控到已进入打电话模式，就强制切换到蓝牙打电话应用下。

### 18.2 应用架构

打电话的主要架构图

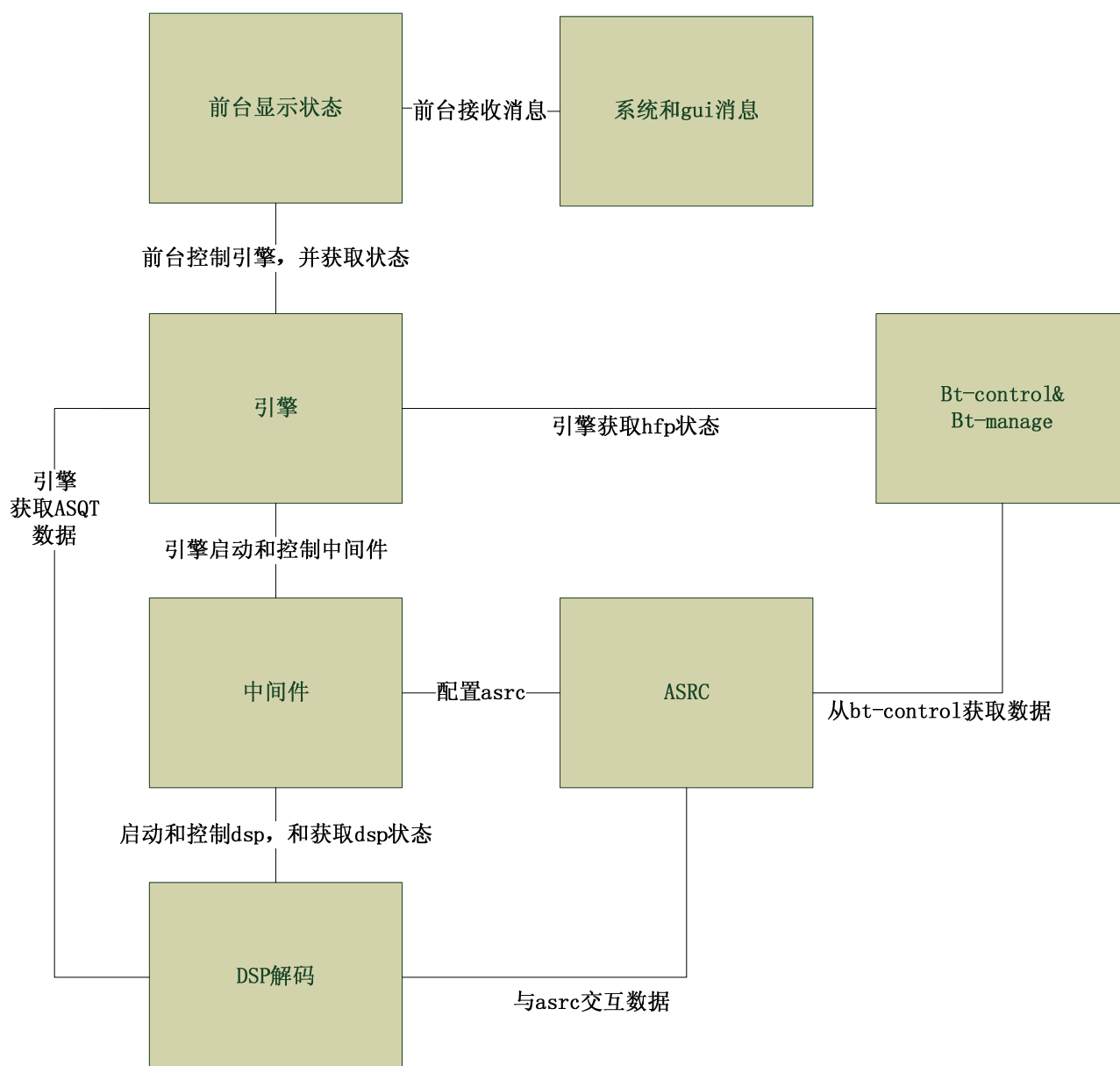


图 18.1 蓝牙通话架构图

### 18.3 前台介绍

蓝牙通话前台的主要功能及其设计如下：

- 1、负责与用户交互，用户通过按键可以选择、接听、拒接、挂断电话、调节音量和切换手机或样机通话等。

**设计：**蓝牙通话，实际是对远程设备（一般指手机，下同）通话的控制，基于 SCO 连接方式的 HFP 通

讯，接收设备的 speaker 语音数据和上传本地 mic 采集并处理后的数据上给设备过程。所以一般情况下，只有来去电话，蓝牙通话场景才会出现。蓝牙通话中，有 6 种状态的变化，call-idle, call-in, call-out, call-hfp, call-stop, call-phone，状态变化如下：

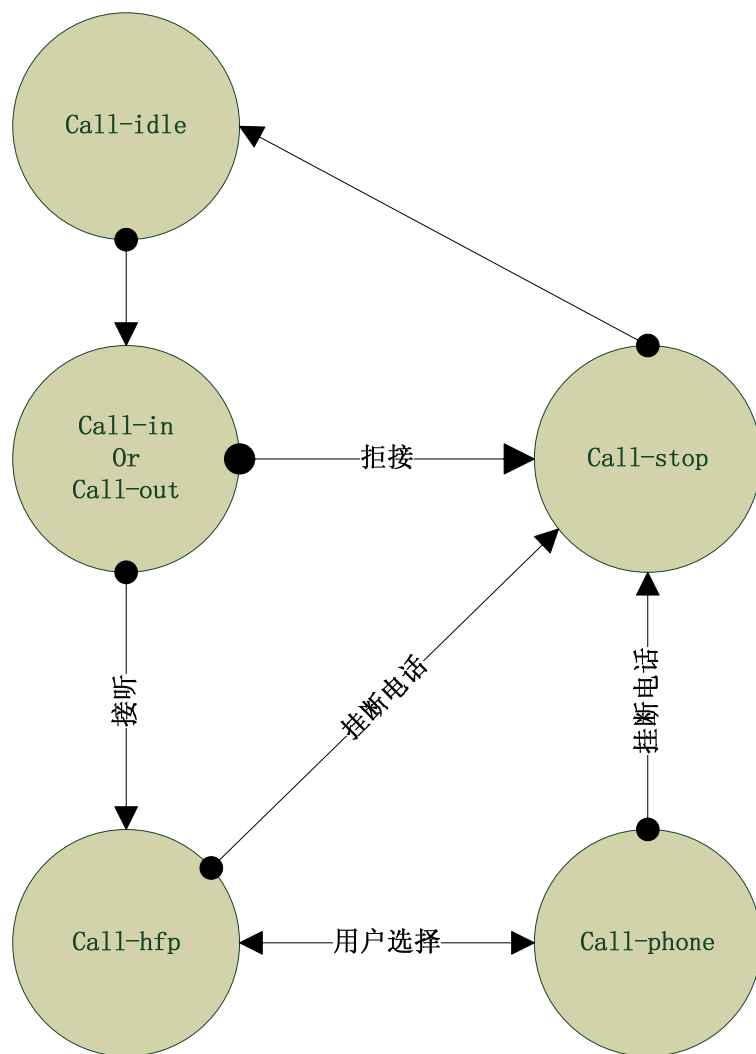


图 18.2 通话场景状态转换图

蓝牙通话 Call-idle 状态比较少出现，从其他应用切换到 btcall 时，此时必须是已经接收到了 hfp 请求，引擎识别的状态已经至少是 call-in 状态。此 6 个状态，前台和引擎是通过共享内存同步的，前台只读取，引擎定时更新。前台通过状态关系控制引擎，如架构图 22.1 所示。

## 18.4 引擎介绍

负责装载和卸载蓝牙通话中间件，并从通过中间件获取 dsp 的解码状态，和获取 bt-control 的状态，实际是获取 hfp 状态，并转换为前台和引擎共用的状态，hfp 状态转换如下，然后同步给前台。

```
HFP_STATUS_LINKED → CALL_STOP;
HFP_STATUS_INCOMING → CALL_CALLIN;
HFP_STATUS_OUTGOING → CALL_CALLOUT;
HFP_STATUS_CALLING → CALL_HFP;
HFP_STATUS_PHONE → CALL_PHONE;
HFP_STATUS_NONE → CALL_IDLE;
```

引擎会根据 HFP 的不同状态，主动控制中间的，引擎还有一个比较重要的功能是响应前台的消息，例如用户切换到手机接听或从手机切勿回音箱接听等。

## 18.5 中间件及 dsp

蓝牙通话的中间件相对简单，大部分工作都交给了 DSP 来完成，中间件主要是加载和启动 DSP，任务已经基本完成。DSP 从 ASRC 模块的 in-buffer 从 PCM 模块获取语音数据，并处理，把处理好的数据，传到 ASRC 的 out-buffer，ASRC 模块再将数据传给 PCM 模块，结构图如下所示：

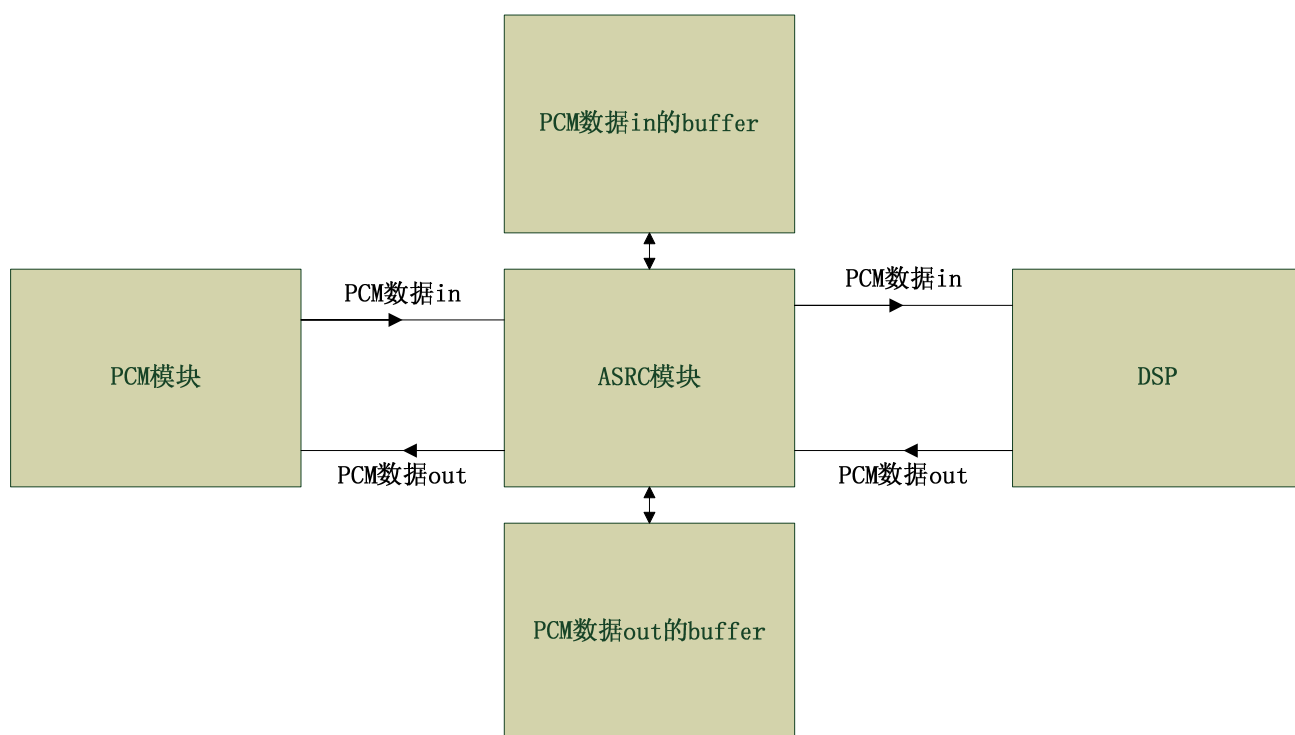


图 18.3 通话语音数据传输图

DSP 的解码主要过程:

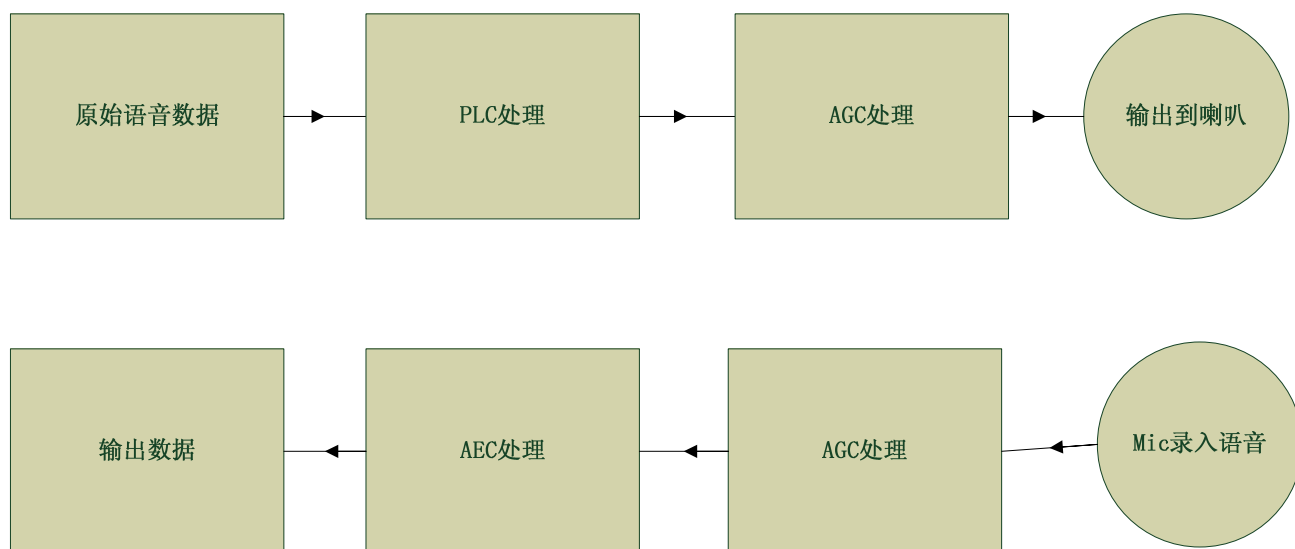


图 22.4 DSP 的解码主要过程



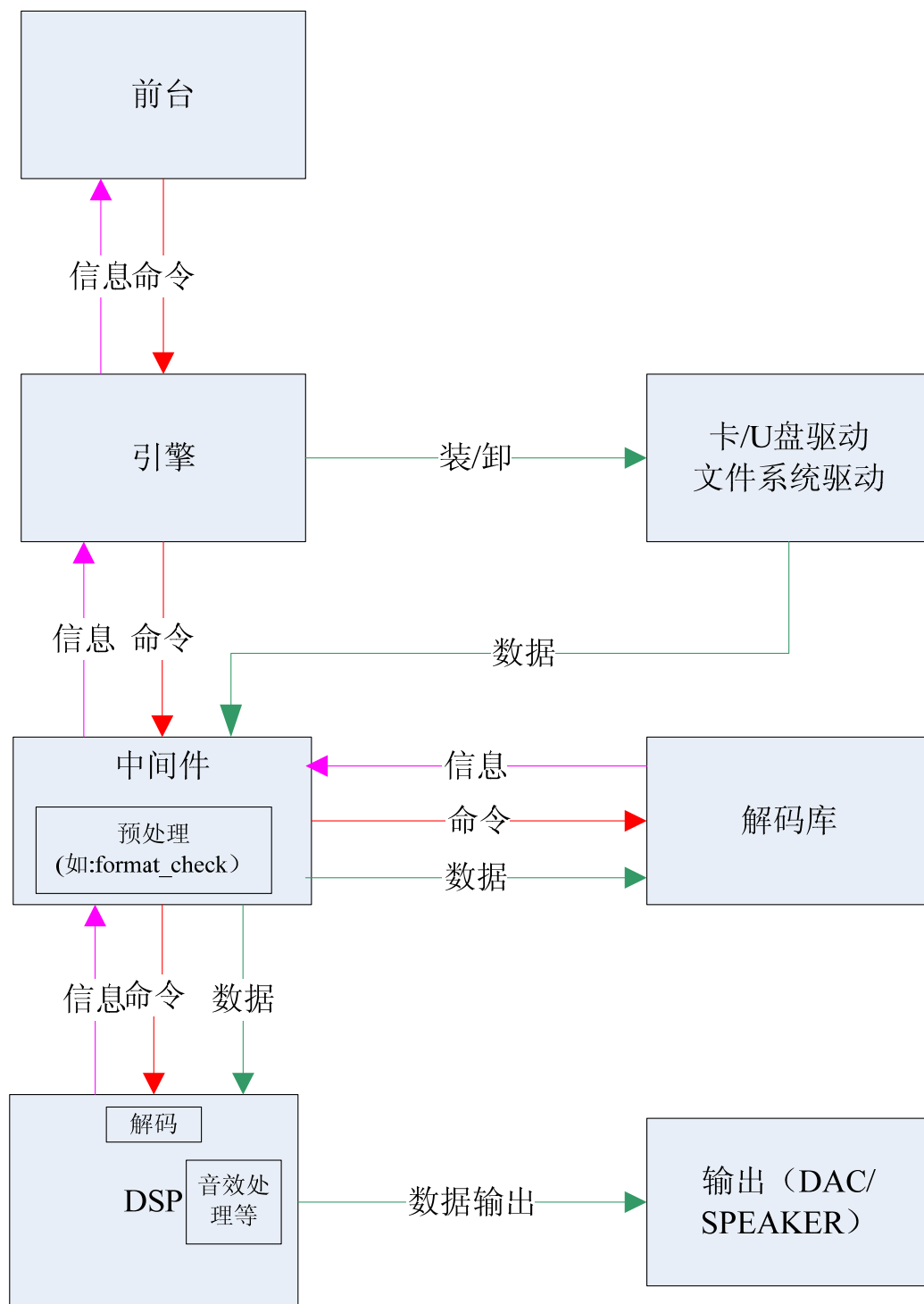
## 19 本地播歌功能

音乐应用的实现，包括了三个子应用：music 前台、music 引擎(也称为“后台”)、music 文件扫描。牵涉到中间件、解码库、dsp、文件选择器、ID3 解析器、LRC 检查(判断 LRC 是否存在)、文件系统、卡/U 驱动、音频设备驱动等。

### 19.1 功能介绍

- 支持音乐播放的基本功能，包括：断点续播、播放暂停、上下首切换、快进/快退、音量加减、淡入淡出等功能。
- 曲目选歌、单曲\列表循环模式、随机播放模式、AB 复读、错误提示、RCP 控制。
- 特定目录播放、特定文件播放(快捷播放刚录文件、闹钟文件)、内置 (SD 区) 文件播放(如闹钟文件)、内置文件无缝播放、显示歌曲信息。
- 功能可通过 config.txt 配置。
- 支持最多 8 级子目录，存放于大于 8 级目录的文件，将会被忽略。
- 单目录下不限制文件个数，音乐文件总数 2048 以内，不包括大于 8 级目录的文件。

## 19.2 总体结构介绍



前台、引擎、DSP 都是独立的线程（DSP，我们把它抽象成一个独立的线程）

前台主要是处理人机交互的功能，例如音量控制、切歌、设置、显示等

引擎主要是为系统承上启下，连接各资源的桥梁作用。

中间件主要是连接引擎和 DSP 的数据\命令交互，同步 DSP 的状态。

DSP 主要是数据解码，音效处理，和输出。

前台作为开启线程，启动引擎，引擎加载相应的外设驱动及文件系统，和加载中间件，中间件启动 DSP。

系统运行过程中，接收到操作命令，例如按下按键 NEXT，前台读取到 UI 的信息，发送同步命令给引擎，引擎获取下一首歌的播放信息，配置给中间件，中间件同步 DSP 播放的状态。

中间件作为连接引擎和 DSP 的桥梁，为引擎提供函数库。比如，引擎会定时查询播放状态，例如现在播放到第几秒，DSP 处于怎样状态。当系统开始播放音频文件时，中间件要首先对文件做预处理，如判断文件是什么类型，是 MP3 还是 WA。预处理后，才加载相应的解码库调。

解码库有几个版本，对不同的音频文件，都有专门的解析处理，为顶层提供统一的接口。

## 19.3 前台应用详解

### 19.3.1 总体介绍

- 作用：处理人机交互、显示播放状态。
- 启动：music 播放的前台代表着 music 这个应用，初始化时，需要一些参数，例如是从卡播放还是从 U 盘播放，是否为恢复模式，是否为播放录音文件模式。所谓恢复模式，即是过滤 TTS 的流程，达到快速地静默地断点继播。然后加载扫描线程和初始化相应的模块。
- 退出：  
应用退出时，需要记录当前的播放状态，以便下次断点继播，和释放相应的资源，通知管理中心切换应用。
- US282A 使用多个视图(view)的方式，实现模块化设计。music 前台分成：播放视图、曲目号视图、信息提示视图、数字选歌视图、循环模式视图、AB 视图。其中，播放视图是主视图。除主视图外，其他视图都没有自己的消息循环，他们都依赖于主视图的消息循环，被主视图的消息循环所调用。

更多细节请参照 14 章“前台应用详解”

### 19.3.2 与其他模块的同步和交互

共享内存通信方式：前台通过引擎获取播放状态的通讯方式是基于共享内存实现的，由引擎生成共享内存数据块，前台定时读取，规则制定是前台只读共享内存，不修改，共享数据的运动，及销毁都由引擎完成。

同步消息通信方式：前台跟引擎（或其他模块）通讯的另一种方式是消息和信号量。同步消息是主要的通信方式，包括按键消息，系统消息，其他模块的消息等

### 19.3.3 视图及功能

视图只在显示的方案才有，对于没有显示的方案，虽然没有视图，但其功能及过程也是大同小异的。

前台的主要功能是人机交互，实现人机交互功能的模块是视图，视图的实现是公共部分，在前面 14.2.4 已经有详细的介绍，更多信息可以返回阅读。这里主要介绍人机交互的功能及 music 特有部分的功能。

#### 19.3.3.1 播放状态视图

播放视图是 music 应用的主视图，显示当前的播放状态，一般情况下，显示的是播放时间，暂停时时间静止不动，歌曲切换时，显示当前播放的曲目号。

#### 19.3.3.2 曲目号视图

显示当前是第几首，当当前的歌曲播放到结束，切到下一首时，由播放视图转为曲目号视图，显示 3 秒，又回到播放视图，或者用户按按键上\下曲时，显示曲目号视图。

#### 19.3.3.3 数字选歌视图

数字选歌视图是小机接收用户输入数字，指定播放此序号的曲目，在主视图中，按下一个数字键后，就进入本视图，此时显示一个数字，直到输入 4 个数字、确定或超时时，播放此曲目，进入播放视图。

#### 19.3.3.4 循环模式视图

用户设置循环模式时，显示的视图，支持单曲循环、列表循环和随机播放

#### 19.3.3.5 AB 播放视图

在主视图中，按下 AB 键后，就进入本视图。进入后，首先显示的设置 A 点的视图，按 NEXT 键用来进行状态切换设置 B 点。

### 19.3.4 RCP 处理

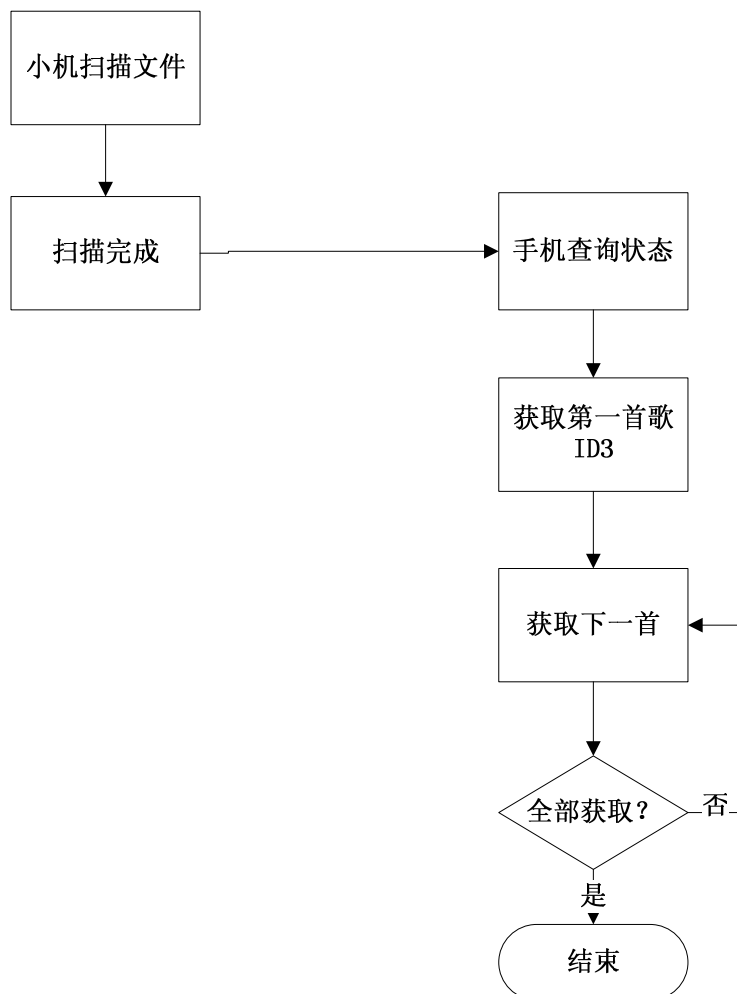
前台启动时，需要注册和初始化 RCP 函数，就是要初始化相应的回调函数，回调函数被 BT-stack 调

用。

RCP 分两种，一种是模拟按键、另一种信息交互。模拟按键就是用户在手机应用端操作，例如切到下一首歌，跟物理按键的功能一样，支持物理按键和模拟按键同时操作。信息交互就是手机中小机之间的信息及状态同步的过程，比如获得文件列表、得到 ID3 信息等。下面是获取 ID3 的过程。

### 19.3.5 获取 ID3

获取 ID3 是手机跟小机的信息交互过程，所以必须遵循一定的流程：



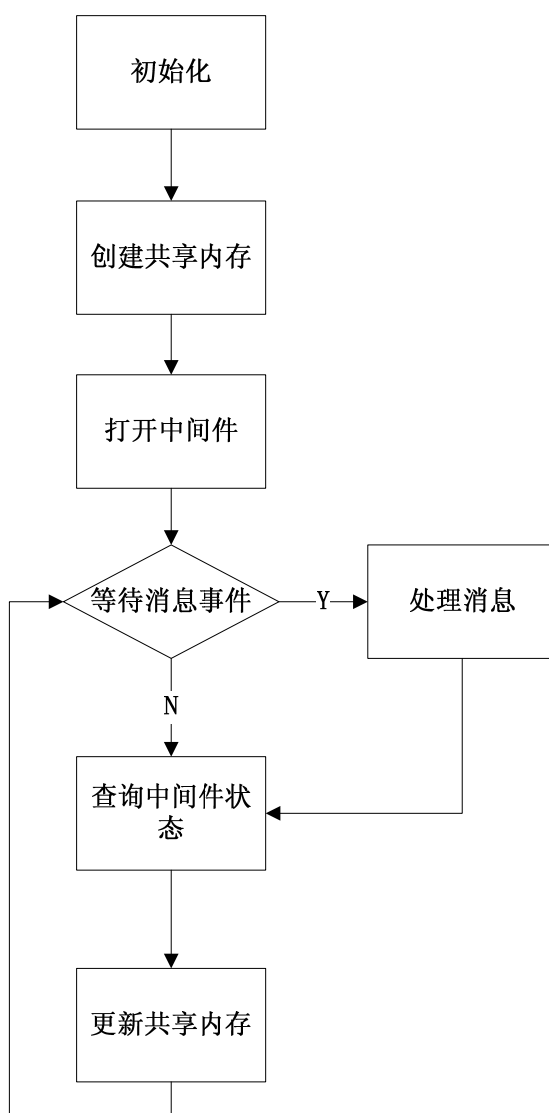
因此，mus\_scan 和 ID3 解析器不能同时存在。

## 19.4 引擎应用详解

- 功能介绍

- 响应前台的同步消息请求。
- 启动中间件，并与中间件实时交互，获取播放状态等。
- 实现自动切歌，包括列表模式、单曲模式和随机模式。
- 启动扫描线程。

### 19.4.1 引擎的总体流程



## 19.4.2 引擎的状态处理

引擎的主要功能是状态处理，向中间件获取播放状态，转为前台可以查询的状态。接收前台的控制消息，并同步给中间件。引擎是一个承上启下的桥梁身份。

引擎跟前台的信息交互有共享内存和消息两种方式，共享内存的方式是主要用于播放状态的查询，其他消息通过同步消息实现。共享内存比消息可以更实时更直接，但只适用于状态的共享，引擎定时向中间件查询当前播放状态，并将获取的状态转为前台可以识别的结构，存放于共享内存中，前台随时查询。另一种交互是同步消息，同步消息主要响应前台的操作请求，例如切歌暂停等。

## 19.5 文件扫描详解

- 功能介绍

文件扫描是为播放时快速响应作准备，事先将所有音频文件的目录存储路径读取并记录，切歌、数字点歌、随机播放等操作时，引擎通过读取记录的信息，从而达到快速响应的效果。

文件扫描是引擎的一个子线程，由引擎启动，扫描结束后，自行退出。在扫描结束前，引擎只能通过文件系统直接获取文件的信息，扫描结束后，引擎在记录获取文件信息。

文件扫描是先序顺序的过程，所谓先序顺序，是遍历树型结构的一个术语，指遍历时先访问当前的数据，再进入到下一层结构，在这里是指，先读取当前文件夹下的全部文件再进入子文件夹，以此递归。

## 19.6 中间件及解码部分

- 功能介绍

中间件负责为 DSP 提供音乐的源数据，并对数据进行预解析，获取 DSP 的解码状态和同步 DSP 的状态，并将状态转为引擎可以识别的结构。中间件是直接跟文件系统打交道的，需要完成一些 `read()` 等操作(比如快进快退)、报错的操作、获取文件信息。中间件完成这些操作，是通过自身的代码以及调用解码库中的函数来实现的。

中间件也是一个线程，它对外提供 API 函数，算是引擎的一部分，但中间件需要独立加载。对外提供的接口 API 只有一个 `mmm_mp_cmd()`，通过标志来区分，例如 `MMM_MP_OPEN` 是要打开中间件。

## 19.6.1 中间件的线程

中间件的线程相对简单，线程主要处理两种事件：一个是等待 DSP 信号量和另一个是快进快退。DSP 信号里包括，始播放结束，解码出错等，而快进退功能，上面说过，中间件为 DSP 解码提供音乐源数据，要快进退则是通过 seek 来实现。

## 19.7 卡拔出后的处理

卡拔出后，可能会触发两个“引信”。

- 一个是从 DSP 开始：
  - DSP 再要数据的时候，解码库报 AD\_RET\_DATAUNDERFLOW;
  - 于是解码库被 AD\_CMD\_CLOSE，中间件把状态改为 MMM\_MP\_ENGINE\_STOPPED;
  - 引擎收到 MMM\_MP\_ENGINE\_STOPPED 状态后，触发正常的切歌流程。但在切歌时，发现卡不在了，于是引擎的状态变成 EG\_ERR\_DISK\_OUT。
  - 通过引擎将返回状态到前台，进入 play\_err\_deal;
  - play\_err\_deal 返回 RESULT\_NEXT\_FUNCTION;
  - 显示 ER03。
- 另一个是从 COMMON 定时器开始：
  - 每隔 PER\_DETECT\_PERIOD 毫秒调用  
peripheral\_detect\_handle->key\_peripheral\_detect\_handle->key\_inner\_peripheral\_detect\_handle->check\_card\_status->发送 MSG\_SYS\_SD\_OUT，即 MSG\_SD\_OUT。
  - 前台进入 \_play\_card\_out->\_play\_disk\_out;
  - 如果前一个“引信”先点着，此时 play\_status 可能已经是 STOP;
  - 于是进入 play\_err\_deal;
  - play\_err\_deal 返回 RESULT\_NEXT\_FUNCTION;
  - 显示 ER03。

一般情况下，是 DSP 先发现问题，显示 ER03，然后定时器又发现问题，再次显示 ER03。但某些低码流的音频文件，读一次可以播很久的，这种情况下，就是定时器先发现问题。

## 19.8 格式不支持文件

有两种情况：

一种是文件一打开就知道它不支持。

- 用户按 NEXT 键或者数字选歌或者 APP 上选择一首;
- 引擎最后会调用到 play->set\_file;
- 中间件 SET\_FILE 时，会进行 format\_check



- 会 set\_file 失败 → play 失败 → mengine\_save\_errno\_no;
  - 前台\_play\_check\_status 时，切换到下一首。
- 另一种是文件播放的过程中出错。
- DSP 要数据->解码器报错;
  - 中间件的状态变成 MMM\_MP\_ENGINE\_ERROR; (顺便说一下：此时中间件线程变成了空转)
  - 引擎 mengine\_status\_deal-> error\_handle-> mengine\_save\_errno\_no
- 前台\_play\_check\_status 时，切换到下一首。

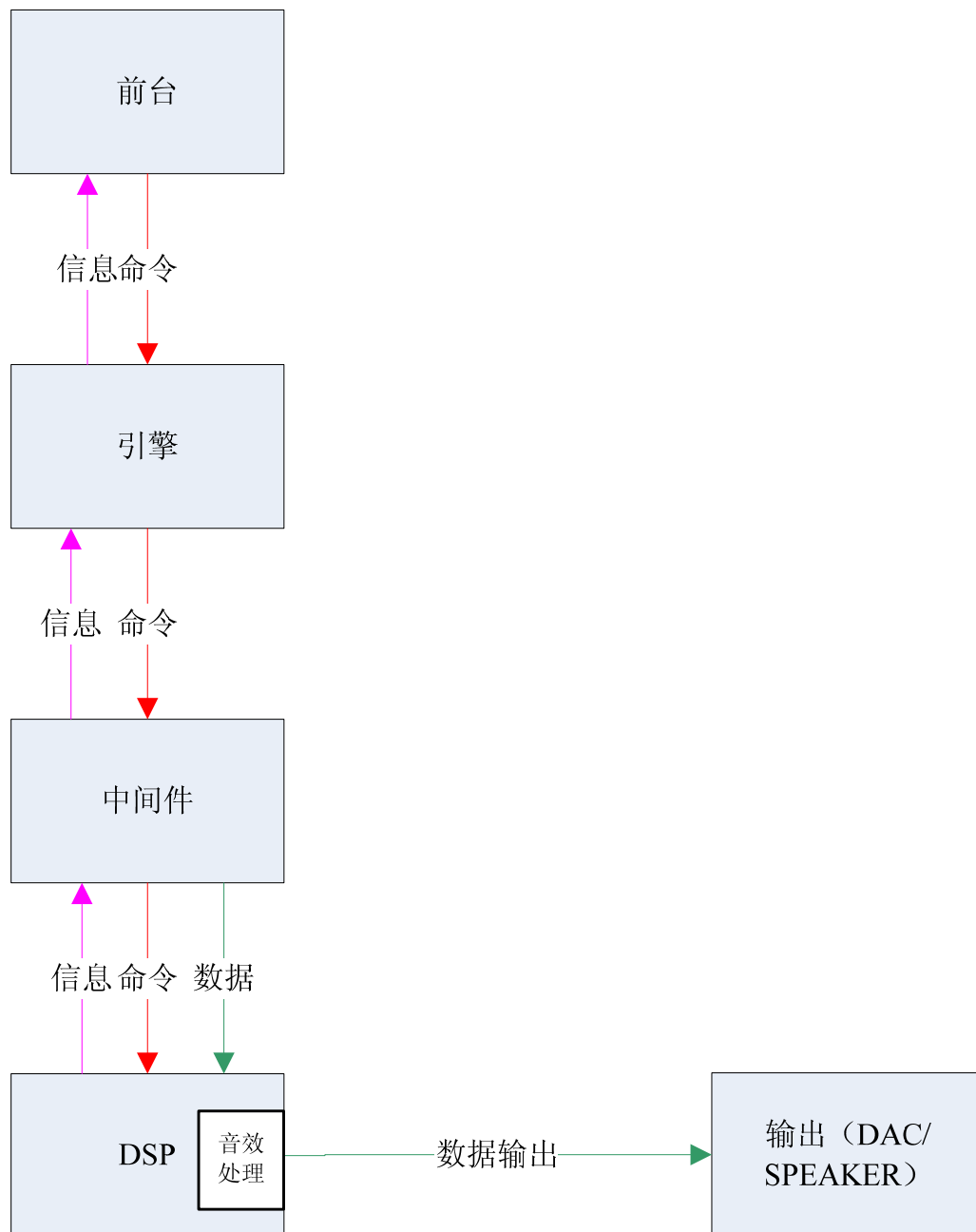
## 20 音频输入功能

音频输入应用的实现，包括了两个子应用：LineIn 前台、LineIn 引擎(也称为“后台”)。牵涉到中间件、dsp、音频设备驱动等。

### 20.1 功能介绍

- 支持 AA 输出、ADDA 输出。
- ADDA 输出时，能进行 EQ 等处理。
- 功能可通过 config.txt 配置。

## 20.2 总体结构介绍



前台、引擎、中间件、DSP 都是独立的线程。

DSP 的作用是把得到的数据进行处理，然后输出。

(ps: DSP 实际上是个单独的核，但在这里，我们可以把它理解成一个独立的线程。)

中间件是与 USB 音响等公用的。LineIn 用到的中间件部分的代码相对简单一些。而且仅在 ADDA 方式时才会用到。

引擎主要的作用是：前台和中间件之间的桥梁。

前台的作用时完成用户交互。接收用户的按键，把它们转换成发往引擎的指令；接收 RCP 命令。

## 20.3 前台应用详解

- 作用：前台应用用来响应用户按键、显示播放信息。
- 入口参数：
  - 一般进入；
  - “从 S3 退出”后进入；
    - ◆ 从 S3 退出时，不用过滤一些消息(一般情况下，是要过滤掉一些消息的)。
    - ◆ 从 S3 退出时，不用播报进入本应用时的 TTS(一般情况下，进入本应用时，会播报“音频输入”)。
    - ◆ 从 S3 退出是，也不会马上就开始播 LineIn 上的内容。(这是通过传递给 LineIn 引擎的一个参数来实现的)
- 退出：

退出时会向 `com_ap_switch_deal` 传递一个值，这个值有以下几种：

`RESULT_NEXT_FUNCTION`：切换到下一个应用；

`RESULT_SYSTEM_ENTER_S3`：进入 S3；

`RESULT_LASTPLAY`：回到原先的应用；

`RESULT_BLUETEETH_SOUND` 等：进入对应的应用。比如通过 APK 切换应用的时候；

`RESULT_POWER_OFF` 等：进入关机或者低功耗等。比如按关机键的时候；

US282A 使用了一个设计理念：分成多个视图(view)，实现模块化设计。

前台只有一个视图，即主视图。

`linein_main.c` 是应用的出入口。

`scene_linein.c` 和 `linein_control.c` 是主视图的处理。这里完成：引擎的创建、UI 的刷新、按键的处理等。

`linein_rcp_deal.c` 是 RCP 指令的处理。

我们重点说一下主视图的按键(比如 PLAY 键)处理。

- 在视图生成的时候，指定了按键映射表 `linein_ke_maplist`；
- `_linein_loop_deal` 是整个应用的中枢。所谓“线程在运行”实际大部分情况下就是在它里面执行。
- `_linein_loop_deal` 里面有个 `com_view_loop`。当用户按下 PLAY 键时，通过 `com_view_loop_key` 会调用到 `key_play_deal`。
- 向引擎发送 `MSG_LINEIN_EG_PLAY_SYNC` 消息。

## 20.4 引擎应用详解

- 作用：
  - 实现与前台的通信，完成对前台命令的响应。
  - 能够完成对系统消息的响应。
- 入口参数：

引擎初始化完成后，是播还是不播。
- 退出：

只有一种情况：收到前台发送的 MSG\_APP\_QUIT 消息后。
- 初始化：

如果是 AA 通道，就很简单；如果是 ADDA 通道，这时就要装载中间件。(但此时中间件的线程还未运行起来。)
- 消息循环：

消息循环是应用的中枢。

以用户按下 PLAY 键为例：当收到前台发送的 MSG\_LINEIN\_EG\_PLAY\_SYNC 消息后。如果是 AA 通道，就只是简单地调用驱动使能 AIN 等；如果是 ADDA 通道，

  - 将会向中间件发送 MMM\_PP\_OPEN，启动中间件的线程。(注意：是此时才启动中间件的线程，不是引擎一进入就启动中间件线程)
  - 发送 MMM\_PP\_AINOUT\_OPEN，设置驱动用的参数。
  - 发送 MMM\_PP\_PLAY，使 DSP 跑起来。

## 20.5 中间件介绍

- 作用：
  - 接收引擎的指令，操作 DSP。
  - 跟 DSP 通讯：比如 DSP 说初始化时，就打开 ADC。
  - 中间件完成这些操作，是通过自身的代码以及调用解码库中的函数来实现的。
- 入口参数：

引擎给中间件下达的命令不同，参数也不一样，也很简单易懂，就不一一列举。
- 退出：

只有一种情况：引擎调用 MMM\_PP\_CLOSE 接口后。

LineIn 的中间件是跟 USB 音箱等公用的。但 LineIn 用的部分相对简单。

- 中间件是怎么被调用的

中间件提供一组 API 函数接口(即 MMM\_PP\_OPEN 等), 同时有自己的线程。当引擎调用 API 函数时, 会导致中间件自身的线程被挂起。

在 ADDA 通道时, 当引擎发送了 OPEN 指令后, 中间件的线程启动。

当引擎发送了 PLAY 指令后, DSP 启动。

此时 DSP 会发来 NEED\_INIT\_PARA, 中间件线程收到后, 才使能 ADC。这是为了避免开始的杂音。

除了这个处理外, 中间件线程就没有什么别的事情要处理了。

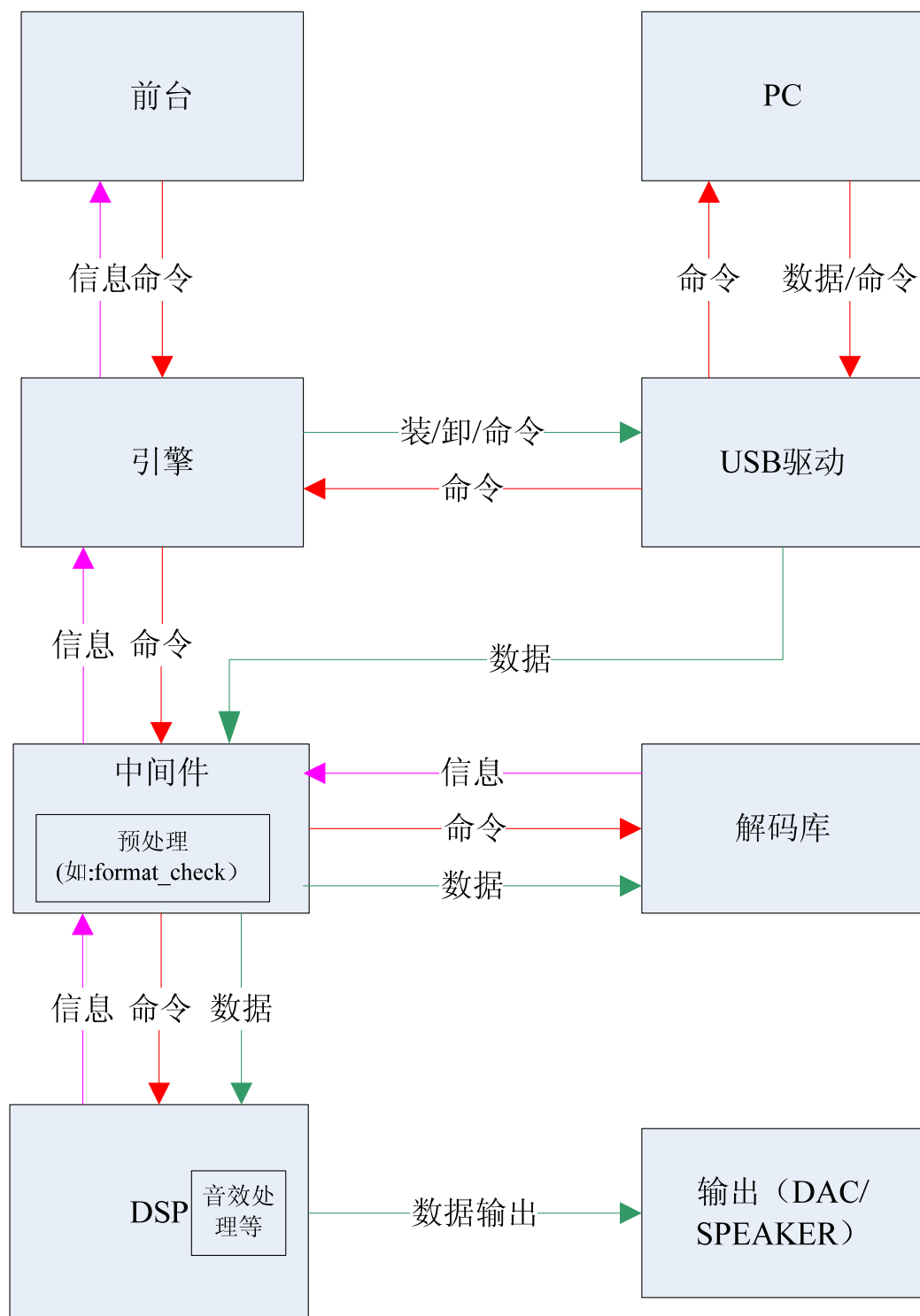
## 21 USB 音箱功能

USB 音箱主要包括两个应用和一个驱动，分别为 usound 前台、usound\_engine 引擎和 usb 驱动。涉及到中间件、dsp 以及一些其它的驱动。

### 21.1 功能介绍

- USB 音箱目前支持两个功能：USB 音箱(uspeaker)和 USB 声卡(usound)。
- USB 音箱：
  - 支持 PC 通过 USB Audio 功能下传音频数据，音箱将其转为声音输出；
  - 支持数字音效；
  - 支持 16bit 位深，44.1K、48K、96K（需要相应的 PA 配合）采样率输出；
  - 支持 HID 控制，可通过音箱按键控制 PC 端歌曲的播放、暂停、切换；
- USB 声卡：
  - 支持 PC 通过 USB Audio 功能下传音频数据，音箱将其转为声音输出；
  - 支持数字音效；
  - 支持 16bit 位深，44.1K、48K 采样率输出；
  - 支持 HID 控制，可通过音箱按键控制 PC 端歌曲的播放、暂停、切换；
  - 支持 MIC 录音，并通过 USB 将录音数据上传至 PC；
- 功能可通过 config.txt 的 USOUND\_TYPE 字段配置。

## 21.2 总体结构介绍





前台、引擎、中间件、DSP 都是独立的线程。

由于 USB 线上的数据是 PC 播放器解码后的数据，DSP 的作用是把得到的数据作音效处理，然后输出。在音效处理的过程中，DSP 会进行能量检测，通过分析 USB 线传过来的数据，判断是否需要更改音箱的播放/暂停状态（主要针对 win7 以上版本的操作系统存在 PC 暂停播放后仍然有数据下发的情况）。DSP 运行过程中，会发送各种命令。

(ps: DSP 实际上是个单独的核，但在这里，我们可以把它理解成一个独立的线程。)

中间件在本应用中的主要功能是：响应引擎下发的命令，完成 DAC 采样率的设置；控制打开、关闭 DSP；控制音箱的播放与暂停；向引擎同步播放的相关信息（如播放状态、当前采样率、能量检测值等）。

引擎主要完成：加载 USB 驱动，通过 USB 驱动将 PC 下发的数据传送给音箱，将用户在音箱上的操作反映给 PC，控制 PC、播放器的播放、暂停、切歌；根据用户的按键，产生按键信息，控制音箱播放、暂停。

前台的作用是：完成用户交互，接收用户的按键，把它们转换成发往引擎的指令，并在显示屏上显示相应的视图。接受 RCP 指令，控制音箱播放/暂停。

## 21.3 前台应用详解

- 作用：前台应用用来响应用户按键、显示播放信息。

- 入口参数：无。

- 退出：

退出时会向 com\_ap\_switch\_deal 传递个值，这个值有以下几种：

RESULT\_NEXT\_FUNCTION:

RESULT\_POWER\_OFF 等：进入关机或者低功耗等。比如按关机键的时候；

US282A 使用了一个设计理念：分成多个视图(view)，实现模块化设计。USB 音箱应用包括播放视图和音量设置视图。

其中，播放视图是主视图，其他是子视图(SUB\_VIEW，能处理按键消息)或者消息视图(MSG\_VIEW，只是显示，不能处理按键消息)。

各主视图是互斥的，即：它们不可能同时存在，所以，在应用里面有个场景调度。

主视图有自己的消息循环，但非主视图没有自己的消息循环，它们都依赖于主视图的消息循环，被主视图的消息循环所调用。

### 21.3.1 与其他模块的同步和交互

前台与引擎之间的通讯是通过消息和信号量。比如前台生成后台时就是通过发送消息给 manager,同时用信号量来等待引擎打开完毕。

```
bool usound_open_engine(uint8 engine_id)
{
```

```
bool bret = FALSE;
msg_apps_t msg;
//msg_reply_t temp_reply;
//无消息内容
msg.content.data[0] = engine_id;
//消息类型(即消息名称)
msg.type = MSG_CREAT_APP_SYNC;
//发送同步消息
if (send_sync_msg(APP_ID_MANAGER, &msg, NULL, 0) == ERR_NO_ERR)
{
 bret = TRUE;
}
return bret;
}
```

这里的“等待引擎打开完毕”是用所谓“发送同步消息”来实现的。

下面是“发送同步消息”的代码：

```
int send_sync_msg(uint8 app_id, msg_apps_t *msg, msg_reply_t *reply, uint32 timeout)
{
...
 if (sys_mq_send((uint32) (target_app_info->mq_id), (void *) &cur_send_pmsg) < 0)
 {
 PRINT_ERR("sync msg ERR_MSGQUEUE_FULL!!");
 retvalue = ERR_MSGQUEUE_FULL;
 }
 else
 {
 //等待同步消息回应
 if (libc_sem_wait(sem, timeout) < 0)
 {
...

```

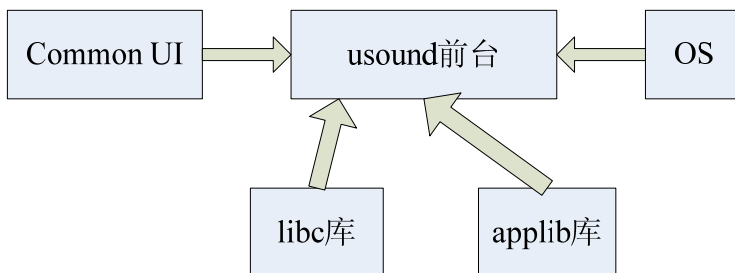
可以看到，在 `sys_mq_send` 把消息发送出去后，这里有一个 `libc_sem_wait`，它的作用就是等到消息被处理完成后才返回。

## 21.3.2 依赖库及其接口说明

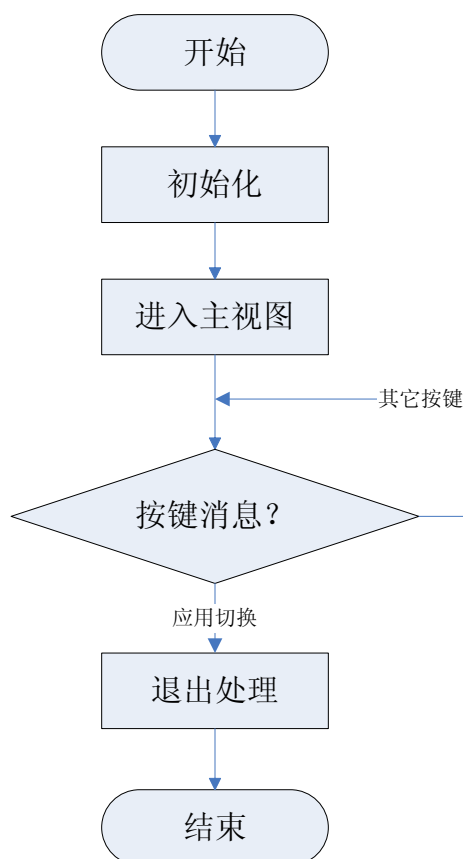
系统和 libc 的接口 `api.a`

应用运行时库 `ctor.o`

`common`



### 21.3.3 应用的业务流程

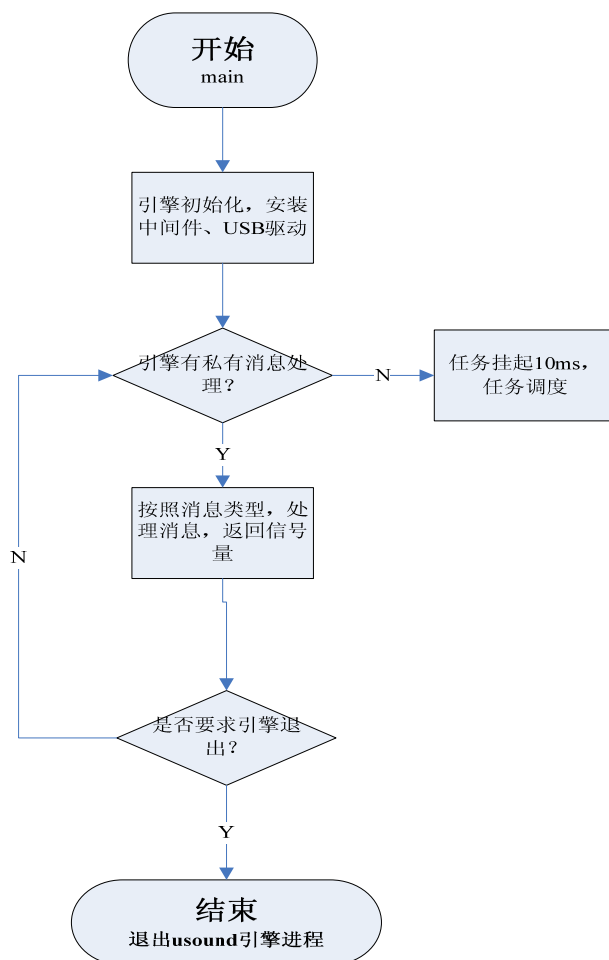


应用刚进入，进入播放视图（主视图）；  
 当用户按下按键，要调节音量，就从播放视图中退出，在这里进入调节音量的视图；  
 当音量调节完成后，又从这里进入播放视图。  
 USB 音箱视图较为简单，在此不做赘述。

## 21.4 引擎应用详解

- 作用：
  - 接收来自前台的消息，并作出处理。
  - 调用 USB 驱动的各个接口，执行音频数据的接收，HID 命令的发送，PC 端命令的接收等一系列通过 USB 进行的数据交互工作。
  - 调用中间件接口，处理音效，控制音箱的播放暂停等。
- 入口参数：无
- 退出：
  - 只有一种情况：收到前台发送的 MSG\_KILL\_APP\_SYNC 消息后。

## 21.4.1 引擎的总体流程



## 21.4.2 引擎的调用

USB 音箱引擎的功能较为简单，只需要加载中间件和 USB 驱动，再根据从前台或者驱动接收到的用户操作的相关消息（按键、RCP、HID 命令），调用 USB 驱动或者控制音箱执行一系列的操作。

引擎根据 config.txt 配置文件中所设置的音箱类型，加载不同的 USB 驱动。USB 声卡应用加载 "usound.drv" 驱动，而 USB 音箱加载 "uspeaker.drv" 驱动，前者支持录音和播放功能，而后者仅支持播放。引擎加载 USB 驱动的同时会作 USB 设备初始化，初始化成功后用户可从 PC 端看到 USB-Audio 这一播放设备。

引擎启动时会加载中间件，设置默认的采样率；会调用中间件接口不断从 DSP 获取当前能量检测的结果(能量检测是通过采样分析 USB 线下发的音频数据流，判断这些数据对应的声音输出是否为小信号，如果

小信号强度保持，则认为 PC 播放器已进行暂停操作)，并根据此结果修改前台、引擎的播放状态。

## 21.5 中间件介绍

- 作用：
  - 接收引擎消息，加载与启动、停止 DSP 及音频解码库、音效库等。
  - 与 DSP 通讯，比如从 DSP 获取能量检测的结果。
  - 向引擎提供 DAC 采样率、ASRC 配置等一系列接口函数。
- 入口参数：

引擎给中间件下达的命令不同，参数也不一样，也很简单易懂，就不一一列举。
- 退出：
  - 只有一种情况：引擎调用 MMM\_PP\_CLOSE 接口后。

虽然 USB 音箱应用从 PC 端下传的音频数据流为 PCM 格式，DSP 不需要对音频文件做解码处理，但是为了保证音效处理流程的一致性，中间件依然会加载音频解码库。

USB 音箱应用与 FM 应用共用同一个中间件，参考 FM 收音机的中间件介绍。

## 21.6 驱动详解

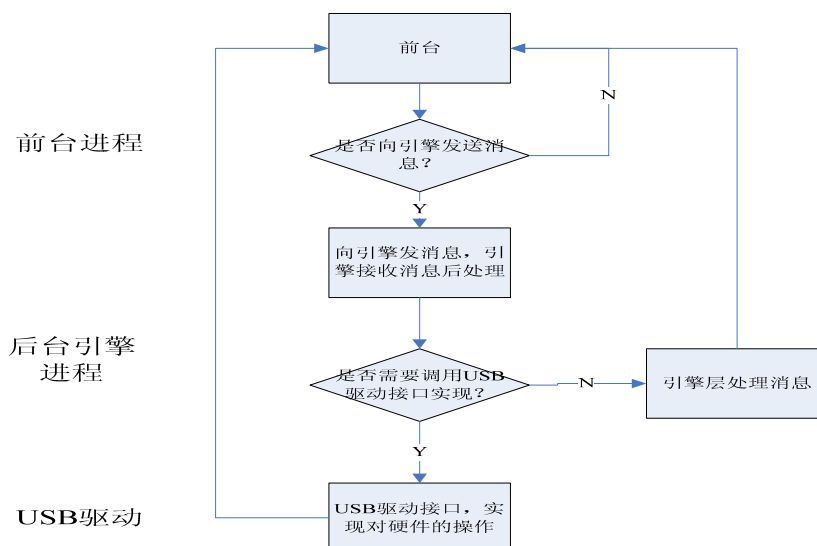
- 作用：
  - 实现音箱与 PC 的数据通信（PCM 数据和 HID 命令），为上层提供一系列 USB 通讯接口，使上层应用不需要考虑 USB 通讯的细节。
  - 将 USB 通信的协议部分模块化，确保其完整性与可靠性，并且易添加，易修改。
  - 通过接口控制 USB 通信，增加了这一功能使用的灵活性。
- 入口参数：
  - 驱动向引擎提供一些接口，功能不一样，参数也不一样，也很简单易懂，就不一一列举，较常用的接口在 23.6.3 中介绍。
- 退出：
  - 只有一种情况：引擎卸载驱动。

驱动是一组函数库，它没有自己的线程，它被引擎的线程所调用，是在引擎的线程空间中跑的，它用的也是引擎的栈空间。

- 接口函数（以 USB 声卡为例）：

```
usound_operations_t usound_ops =
{
 (ud_op_i) usound_inner_get_status, //获取驱动状态
 (ud_op_i) usound_inner_module_start, //USB 模块初始化
 (ud_op_i) usound_inner_module_stop, //USB 模块卸载
 (ud_op_i) usound_inner_set_cfg, //USB 属性设置
 (ud_op_i) usound_inner_get_info, //USB 相关信息更新
 (ud_op_i) usound_inner_set_info, //USB 信息发送（预留）
 (ud_op_i) usound_inner_set_cmd, //USB 命令设置
 (ud_op_i) usound_inner_hid_deal, //发送 HID 命令
};
```

## 21.6.1 总体设计



USB 驱动是 USB 硬件操作层，需要时，调用 `sys_drv_install(...".drv")` 进行装载，引擎会根据配置项中的 `USOUND_TYPE` 段判断应该加载哪个 USB 驱动，将驱动常驻代码以及初始化数据搬入内存，并执行驱动初始化函数。

驱动装载完成后，对驱动的使用完全由上层控制，当前台需要操作 USB 模组时，可通过向 USB 音箱引擎发送消息，来调用 USB 驱动的接口，实现硬件操作。即使 PC 端通过 USB 向音箱发送控制命令，USB 驱动也仅仅是改变自身状态，再等待引擎来同步状态信息。

比如：用户按下播放/暂停按键，前台将按键消息（`TTS_PLAY_SONG`/`TTS_PAUSE_SONG`）发送到引擎，引擎调用函数 `uengine_play_pause`，该函数调用 `ud_set_cmd` 接口是 USB 驱动向 PC 发送控制命令，令

PC 播放器暂停。

PC 端停止播放，以 win7 为例，分两种情况：

播放器点击暂停，USB 线仍然会有小信号的 PCM 数据下发，此时引擎通过中间件从 DSP 获取能量检测的值，判断出当前状态应该为暂停状态，则控制中间件暂停，再调用驱动接口同步 USB 状态。

播放器关闭，则 USB 线上无数据发送，USB 驱动收不到数据，认为此时应该为暂停状态，并将自身状态修改为暂停，待引擎前来同步状态时再将引擎状态修改为暂停，再将这一状态同步到前台与中间件。

## 21.6.2 USB 驱动的应用接口

USB 驱动要实现硬件上的 USB 数据交互，必须要在驱动初始化阶段完成枚举过程，必须要保持与 PC 端的连接，并且响应 USB 的中断并回复各种命令。

初始化阶段：

从配置项获取 USB 相关属性，引擎调用 `ud_set_config` 接口设置驱动的属性；调用 `ud_module_start` 接口，完成 USB 设备的枚举工作，此后可从 PC 机的设备管理器上看到本设备。

连接建立：

引擎调用 `ud_get_status` 接口获取 USB 状态，并同步到前台、中间件。

命令传输、播放：

前台发送按键消息到引擎，引擎调用 `ud_set_cmd` 接口控制驱动给 PC 发控制命令；驱动状态因 PC 命令而改变后，引擎调用 `ud_get_status` 同步状态，再将其同步到前台、中间件上。

常用应用接口函数

```
ud_get_status(usound_status_t *pstatus) //获取当前 usb 状态函数
ud_module_start(isr_cbk *p_isr_cbk) //启动 usb 模块,注册中断等操作
ud_module_stop() //退出 usb 模块,注销中断、释放资源等
ud_set_config(uint32 cfg_index, uint32* pcfg, uint32 cfg_len) //设置 usb 的配置项
ud_get_info(uint32 info_index, uint32* pinfor, uint32* info_len) //获取 usb 信息
ud_set_info uint32 info_index, uint32* pinfor, uint32 info_len) //配置 usb,暂未使用
ud_set_cmd(uint32 cmd, uint32 cmd_param) //USB 驱动的主要控制接口
typedef enum
{
 SET_PLAY_FLAG = 0, //play status set
 SET_HID_OPERS, //cmd use
 SET_SAMPLE_RATE, //sample set
 SET_ADJUST_TIMER, //adjust
 SET_LINE_STATUS, //status
 SET_VOLUME_FLAG, //volume syn flag
 SET_HID_CHANGE, //hid need report
 SET_CARD_INOUT,
 //deal card pull plug
} usound_set_cmd_e; // ud_set_cmd 命令字
```



这些应用接口函数可在 USB 音箱应用中被各模块调用。

`ud_get_status(usound_status_t *pstatus)`

该接口在引擎线程主循环调用，周期性轮询 USB 驱动的状态，并同步。

`ud_module_start(isr_cbk *p_isr_cbk)`

该接口在 USB 驱动初始化阶段调用，开始 USB 设备枚举操作，并将 USB 中断的指针返回给引擎。

`ud_module_stop()`

该接口在 USB 驱动卸载阶段调用，注销中断，释放资源。

`ud_set_config(uint32 cfg_index, uint32* pcfg, uint32 cfg_len)`

该接口在 USB 驱动初始化之前调用，引擎从配置项读取 USB 的相关内容，再通过该接口将其写到驱动的配置段。

`ud_get_info(uint32 info_index, uint32* pinfor, uint32* info_len)`

该接口用于获取 USB 的数据帧长度。

`ud_set_info(uint32 info_index, uint32* pinfor, uint32 info_len)`

该接口为预留接口，未定义。

`ud_set_cmd(uint32 cmd, uint32 cmd_param)`

该接口为引擎对驱动的控制接口，各操作对应命令字见 `usound_set_cmd_e` 的定义。

例如，用户在音箱上按播放/暂停按键，前台发送按键消息到引擎，引擎调用

`ud_set_cmd(SET_HID_OPERS, 0x08)`，发送 HID 命令 08 给 PC。

HID 设备对 PC 发送的数据并不是同步发送的，引擎调用 `ud_set_cmd(SET_HID_OPERS, 0x08)` 后，需要等待 PC 轮询才能将命令发送给 PC。在 USB 音箱应用中将轮询周期设置为 32ms，也就是说，PC 最多要在 32ms 后才会响应 HID 命令。

## 21.6.3 USB 驱动的配置说明

修改配置项文件的字段

`USOUND_TYPE{250} = 0[0,1];` //usb 声卡 or 音箱选择（0，声卡；1，音箱）

可切换引擎加载的 USB 驱动。

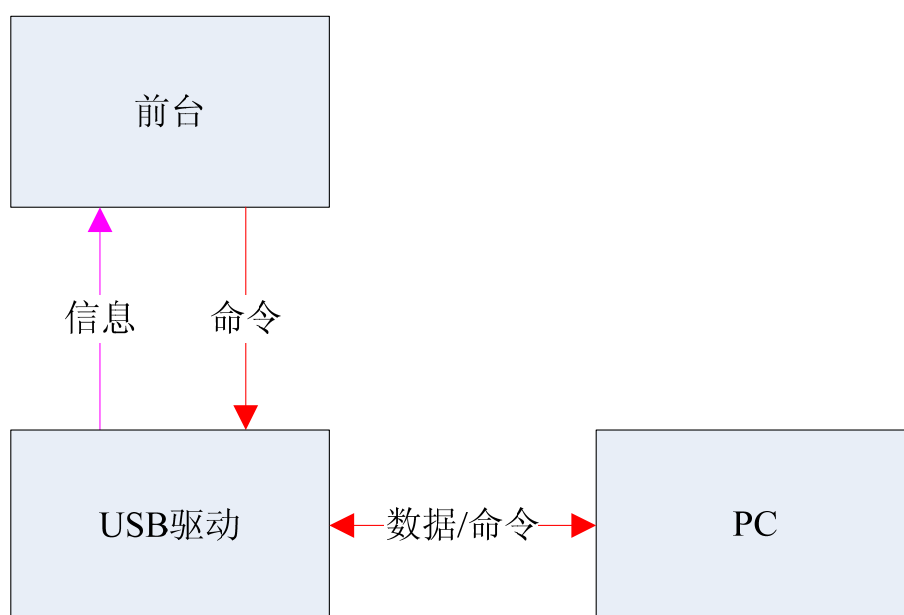
## 22 USB 读卡器功能

将 US282A 与 PC 通过 USB 连接，US282A 会进入读卡器应用，该应用功能较为简单，只有一个前台应用和一个驱动。

### 22.1 功能介绍

- 支持卡的读/写。
- 在此应用下可进行量产操作

### 22.2 总体结构介绍



USB 读卡器功能较为单一，且无需与 DSP 产生交互，所以不需要加载引擎。USB 驱动直接在前台被加载，

## 22.3 前台应用介绍

- 作用：前台应用用来响应用户按键、显示应用信息。
- 入口参数：无。
- 退出：

退出时会向 `com_ap_switch_deal` 传递一个值，这个值有以下两种：

`RESULT_NEXT_FUNCTION`：按任意按键（除关机操作）切换到下一个应用。

`RESULT_POWER_OFF` 等：按下关机键关机。

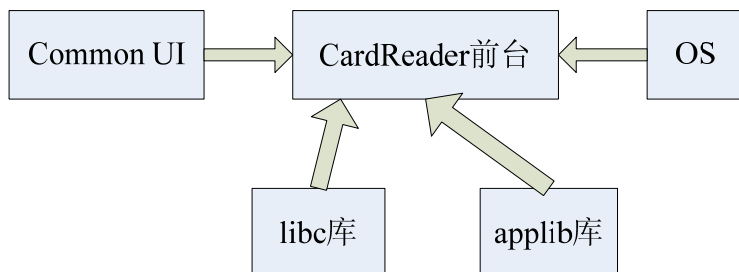
USB 读卡器应用只存在一个视图，默认为显示 `USBH` 来指示当前应用状态。由于在应用场景下，按下任意按键切换到下一应用，所以 USB 读卡器应用执行的过程中不需要 TTS 播报和按键音提示，这两个功能在此应用下无法使用。

### 22.3.1 依赖库及其接口说明

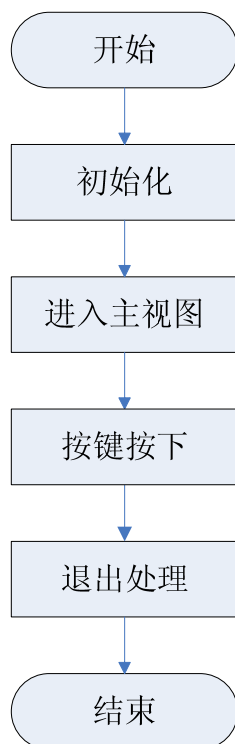
系统和 `libc` 的接口 `api.a`

应用运行时库 `ctor.o`

`common`



## 22.3.2 应用的业务流程



USB 读卡器应用只有一个视图，在此不作赘述。

## 22.4 驱动介绍

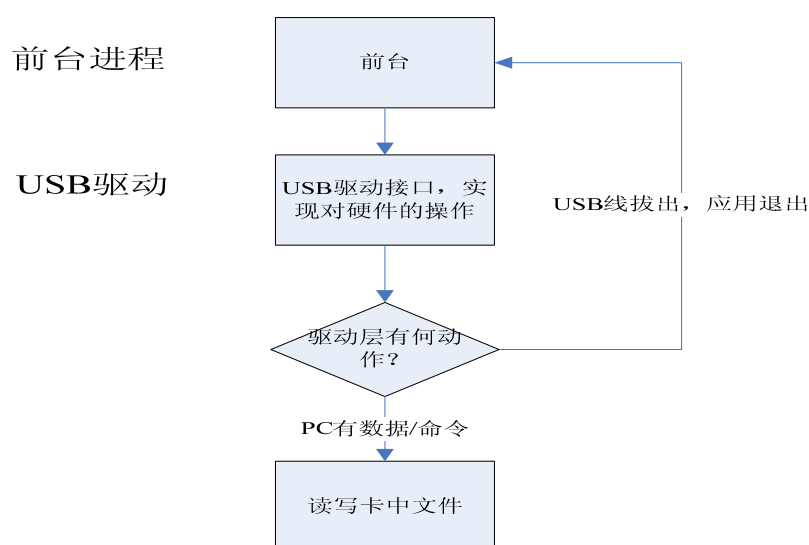
- 作用：
  - 实现 US282A 与 PC 的数据通信（PCM 数据和 HID 命令），为上层提供一系列 USB 通讯接口，使上层应用不需要考虑 USB 通讯的细节。
  - 将 USB 通信的协议部分模块化，确保其完整性与可靠性，并且易添加，易修改。
  - 通过接口控制 USB 通信，增加了这一功能使用的灵活性。
- 入口参数：
  - 驱动向上层应用提供一些接口，功能不一样，参数也不一样，也很简单易懂，就不一一列举。
- 退出：
  - 只有一种情况：上层应用卸载驱动。

USB 读卡器驱动是一组库函数，它没有自己的线程，被前台应用调用，在前台线程空间中运行，用的是前台的栈空间。

● 接口函数：

```
ureader_operations_t ureader_ops =
{
 (ud_op_i) ureader_inner_get_status,
 (ud_op_i) ureader_inner_module_start,
 (ud_op_i) ureader_inner_module_stop,
 (ud_op_i) ureader_inner_set_cfg,
 (ud_op_i) ureader_inner_msc_send,
 (ud_op_i) ureader_inner_msc_receive,
 (ud_op_i) ureader_inner_set_cmd,
 (ud_op_i) ureader_inner_msc_deal,
};
```

## 22.4.1 总体设计



## 22.4.2 USB 驱动的应用接口

### 常用应用接口函数

|                                                                    |                        |
|--------------------------------------------------------------------|------------------------|
| ureader_get_status(usound_status_t *pstatus)                       | //获取当前 usb 状态函数        |
| ureader_module_start(isr_cbk *p_isr_cbk)                           | //启动 usb 模块,注册中断等操作    |
| ureader_module_stop()                                              | //退出 usb 模块,注销中断、释放资源等 |
| ureader_set_config(uint32 cfg_index, uint32* pcfg, uint32 cfg_len) | //设置 usb 的配置项          |

```
ureader_set_cmd(uint32 cmd, uint32 cmd_param) //USB 驱动的主要控制接口
typedef enum
{
 SET_PLAY_FLAG = 0, //play status set
 SET_HID_OPERS, //cmd use
 SET_SAMPLE_RATE, //sample set
 SET_ADJUST_TIMER, //adjust
 SET_LINE_STATUS, //status
 SET_VOLUME_FLAG, //volume syn flag
 SET_HID_CHANGE, //hid need report
 SET_CARD_INOUT, //deal card pull plug
} usound_set_cmd_e; // ureader_set_cmd 命令字
```

这些应用接口函数可在 USB 读卡器应用中被各模块调用。

`ureader_get_status(ureader_status_t *pstatus)`

该接口在前台应用主循环调用，周期性轮询 USB 驱动的状态，并同步。

`ureader_module_start(isr_cbk *p_isr_cbk)`

该接口在 USB 驱动初始化阶段调用，开始 USB 设备枚举操作，并将 USB 中断的指针返回给前台。

`ureader_module_stop()`

该接口在 USB 驱动卸载阶段调用，注销中断，释放资源。

`ureader_set_config(uint32 cfg_index, uint32* pcfg, uint32 cfg_len)`

该接口用于将配置文件中的信息写到驱动模块的配置段。

`ud_set_cmd(uint32 cmd, uint32 cmd_param)`

该接口为前台对驱动的控制接口，各操作对应命令字见 `usound_set_cmd_e` 的定义。

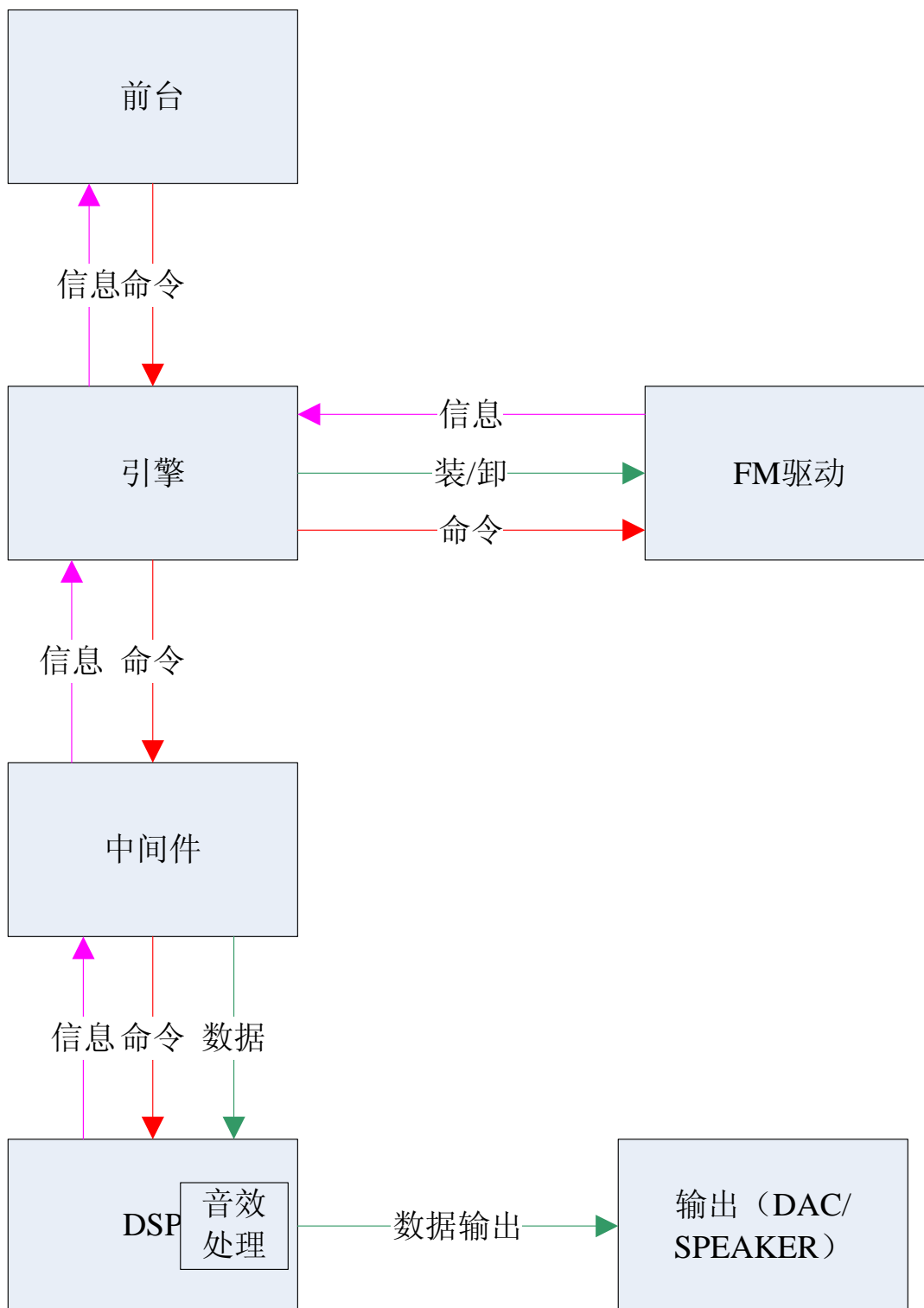
## 23 FM 收音机功能

收音机主要包括了两个应用和一个驱动：FM 前台、FM 引擎(也称为“后台”)、FM 驱动。牵涉到中间件、dsp、音频设备驱动等。

### 23.1 功能介绍

- 支持预设电台的播放和手动调频。
- 支持三种电台频段：普通频段，日本频段和欧洲频段。

## 23.2 总体结构介绍





前台、引擎、中间件、DSP 都是独立的线程。

DSP 的作用是把得到的数据进行处理，然后输出。

(ps: DSP 实际上是个单独的核，但在这里，我们可以把它理解成一个独立的线程。)

中间件是与 USB 音响等公用的。FM 用到的中间件部分的代码相对简单一些。而且仅在 ADDA 方式时才会用到。

引擎主要的作用是：前台和中间件之间的桥梁。

前台的作用时完成用户交互。接收用户的按键，把它们转换成发往引擎的指令；接收 RCP 命令。

## 23.3 前台应用详解

- 作用：前台应用用来响应用户按键、显示播放信息。
- 入口参数：
  - 本应用有两类入口：“从 S3 中退出”、RTC 自动启动后进入。但“RTC 进入”这个入口现在没有做任何处理，保留将来使用。
  - 从 S3 退出时，不用播报进入本应用时的 TTS(一般情况下，进入本应用时，会播报当前播放的频率)。
- 退出：

退出时会向 `com_ap_switch_deal` 传递一个值，这个值有以下几种：

`RESULT_NEXT_FUNCTION`：切换到下一个应用；

`RESULT_SYSTEM_ENTER_S3`：进入 S3；

`RESULT_MAIN_APP`：回到缺省的应用；

`RESULT_BLUETEETH_SOUND` 等：进入对应的应用。比如通过 APK 切换应用的时候；

`RESULT_POWER_OFF` 等：进入关机或者低功耗等。比如按关机键的时候；

US282A 使用了一个设计理念：分成多个视图(view)，实现模块化设计。FM 前台分成：播放视图、硬件搜台视图、软件搜台视图、错误提示视图、数字选台视图、频道号显示视图。

其中，播放视图、硬件搜台视图、软件搜台视图都是主视图，其他是子视图(SUB\_VIEW，能处理按键消息)或者消息视图(MSG\_VIEW，只是显示，不能处理按键消息)。

各主视图是互斥的，即：它们不可能同时存在，所以，在应用里面有个场景调度。

主视图有自己的消息循环，但非主视图没有自己的消息循环，它们都依赖于主视图的消息循环，被主视图的消息循环所调用。

### 23.3.1 与其他模块的同步和交互

前台与引擎之间的通讯是通过消息和信号量。比如前台生成后台时就是通过发送消息给 `manager`，同时用信号

量来等待引擎打开完毕。

```
bool radio_engine_create(uint8 eg_id)
{
 //消息返回
 msg_reply_t msg_reply;
 msg_apps_t msg;
 //创建 radio 后台引擎进程
 msg.type = MSG_CREAT_APP_SYNC;
 msg.content.data[0] = eg_id;
 //发送同步消息
 return (send_sync_msg(APP_ID_MANAGER, &msg, &msg_reply, 0) == ERR_NO_ERR);
}
```

这里的“等待引擎打开完毕”是用所谓“发送同步消息”来实现的。

下面是“发送同步消息”的代码：

```
int send_sync_msg(uint8 app_id, msg_apps_t *msg, msg_reply_t *reply, uint32 timeout)
{
 ...
 if (sys_mq_send((uint32) (target_app_info->mq_id), (void *) &cur_send_pmsg) < 0)
 {
 PRINT_ERR("sync msg ERR_MSGQUEUE_FULL!!");
 retvalue = ERR_MSGQUEUE_FULL;
 }
 else
 {
 //等待同步消息回应
 if (libc_sem_wait(sem, timeout) < 0)
 {
 ...
 }
 }
}
```

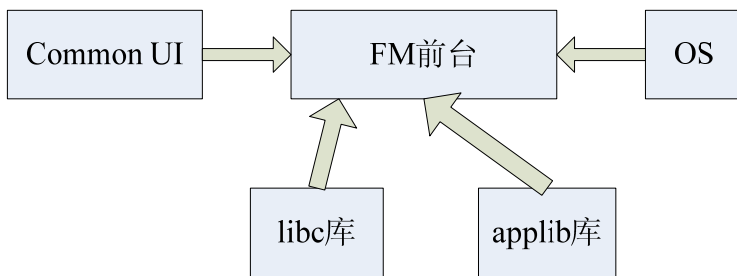
可以看到，在 `sys_mq_send` 把消息发送出去后，这里有一个 `libc_sem_wait`，它的作用就是等到消息被处理完成后才返回。

## 23.3.2 依赖库及其接口说明

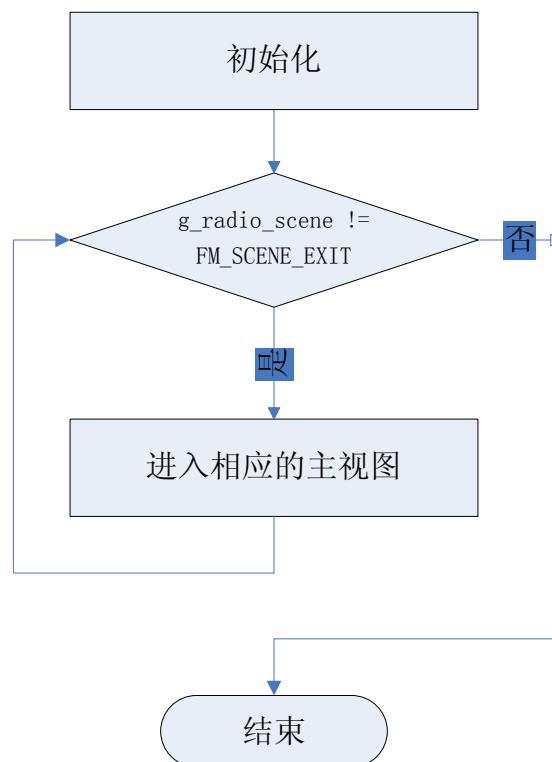
系统和 libc 的接口 `api.a`

应用运行时库 `ctor.o`

`common`



### 23.3.3 应用的业务流程



刚进入，肯定是进入播放视图；

当用户按下按键，要进入搜台时，就从播放视图中退出，在这里进入搜台的视图；

当搜台完成后，又从这里进入播放视图。

### 23.3.4 播放视图

- 初始化：装载引擎，初始化 FM，并得到当前频率。

```
result = radio_engine_create(APP_ID_FMENGINE); // 装载引擎
```

```
...
 result = radio_modul_open((uint8) g_radio_config.band_mode); //初始化 FM
}

if (result == TRUE)
{
 //获取当前频点
 radio_get_freq(); //得到当前频率
}
```

- 生成主视图。
- 显示当前是 CHx 或者，报频率。

显示 CHx 是用消息视图(MSG\_VIEW)的方式显示的，显示 2 秒后，自动关掉。

- 进入消息循环。

以用户按 MUTE 键为例：

用户按下 MUTE 键后，com\_view\_loop 中会调用 radio\_play\_key\_deal\_play\_mute->radio\_set\_mute(SetMUTE)->向引擎发送 MSG\_FMENGINE\_SETMUTE 消息，最终在驱动程序中，对 FM 模块进行 MUTE。

这里说一下 radio\_set\_mute：

它采用计数器的方式。也就是说：如果是这样的调用顺序 Mute-Mute-Release-Release，只有最外层的那个 Release 才能真正解除静音。

如果传统方式，举例来说：如果有两个函数 f1 和 f2，f1 执行的过程中会产生杂音，于是 f1 的开头 mute，f1 的结尾 release；f2 也是如此。比如按了 A 键，我调用 f1；按 B 键时调用 f2。这都没问题，但现在多了一个 C 键，按 C 键时先调用 f1，然后调用 f2，这就会导致：f1 执行的时候没声音出来，f2 执行的时候也没声音出来，但 f1 和 f2 之间有声音出来。而我要求：C 键执行的过程中都没有声音出来，执行完了才有。这个时候，传统的方法就没法解决。而用计数器，就能轻松解决了这个问题。

### 23.3.5 硬件搜台视图

自动搜台和手动搜台，都会生成硬件搜台视图。现以自动搜台为例。

- 在生成视图之前，先进行初始化：radio\_auto\_search\_init(dir);
- 进入消息循环：deal\_hard\_auto\_seek。
- 当搜到一个有信号的电台时，会通过 \_insert\_tab 保存电台，并显示在屏上。
- 当按下按键时，会通过 com\_view\_loop 进入对应的按键处理函数。比如按下 PLAY 键，com\_view\_loop 会调用 search\_key\_deal\_cancel\_search，然后跳出消息循环。
- 当搜台完成后，会返回，此时 scene\_result 为 RESULT\_NULL，于是在 radio\_scene\_dispatch 的循环中，g\_radio\_scene 被再度赋值为 FM\_SCENE\_PLAYING，然后，再次进入播放视图。

### 23.3.6 软件搜台视图

从前台的角度来看，软件搜台跟硬件搜台的差别不大，这里就不再累述。

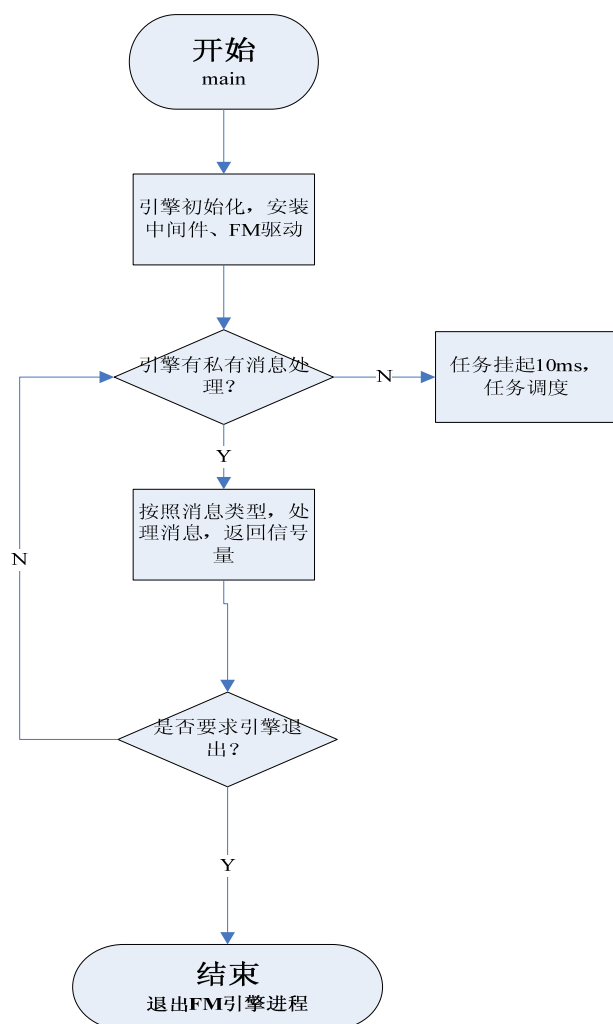
## 23.4 引擎应用详解

- 作用：
  - 接收来自前台的消息，并作出处理。
  - 调用 FM 驱动提供的各个接口，对 FM 进行控制，实现模组的初始化，设置频率，配置搜台，设置频段，退出等等一系列功能。
  - 调用中间件接口，处理音效等。
- 入口参数：

无。
- 退出：

只有一种情况：收到前台发送的 MSG\_FMENGINE\_CLOSE 消息后。

## 23.4.1 引擎的总体流程



## 23.4.2 引擎的调用

加载到内存就可以了。

大体上说，当前台发送来消息后，引擎接收到后，一般会调用驱动接口，实现对 FM 的控制、读取。

这里特别说明一下 ADDA 通道时，对中间件的处理：

在前台调用 `radio_connect_ain->MSG_FM_AIN_OPEN->fm_Ain_open->play` 后，引擎才会向中间件发送 `MMM_PP_OPEN`(启动中间件线程)、`MMM_PP_PLAY`(启动 DSP)。所以，在引擎刚装载进来的时候，中间件的线程是没有跑起来的。这一点要注意。

## 23.5 中间件介绍

- 作用：
  - 接收引擎的指令，操作 DSP。
  - 跟 DSP 通讯：比如 DSP 说初始化时，就打开 ADC。
  - 中间件完成这些操作，是通过自身的代码以及调用解码库中的函数来实现的。
- 入口参数：

引擎给中间件下达的命令不同，参数也不一样，也很简单易懂，就不一一列举。
- 退出：

只有一种情况：引擎调用 MMM\_PP\_CLOSE 接口后。

FM 的中间件是跟 USB 音箱等公用的。但 FM 用的部分相对简单。

- 中间件是怎么被调用的

中间件提供一组 API 函数接口(即 MMM\_PP\_OPEN 等)，同时有自己的线程。当引擎调用 API 函数时，会导致中间件自身的线程被挂起。

在 ADDA 通道时，当引擎发送了 OPEN 指令后，中间件的线程启动。

当引擎发送了 PLAY 指令后，DSP 启动。

此时 DSP 会发来 NEED\_INIT\_PARA，中间件线程收到后，才使能 ADC。这是为了避免开始的杂音。

除了这个处理外，中间件线程就没有什么别的事情要处理了。

## 23.6 驱动详解

- 作用：
  - 实现对 FM 硬件操作接口的封装，使上层应用不需关心 FM 硬件操作的细节，只需调用相关接口就可实现对应功能。
  - 实现部分代码对各款 FM 模组的通用性。
  - 实现 FM 驱动接口的易添加，易修改，以及 FM 硬件相关项的易配置（比如模拟 I2C 的 GPIO Pin 配置等）。
- 入口参数：

驱动向引擎提供一些函数，功能不一样，参数也不一样，也很简单易懂，就不一一列举。
- 退出：

只有一种情况：引擎卸载驱动。

驱动是一组函数库，它没有自己的线程，它被引擎的线程所调用，是在引擎的线程空间中跑的，它用的也是引擎的栈空间。

FM 驱动设计时，主要的设计原则有：

- 各套 FM 驱动向上层提供统一的接口，这样，上层应用就不需要关心具体使用的 FM 模组。如下：

```
/*FM 驱动对外接口函数*/
fm_driver_operations fm_drv_op =
{
 (fm_op_func) sFM_Init, //初始化
 (fm_op_func) sFM_Standby, //standby
 (fm_op_func) sFM_SetFreq, //设置频率
 (fm_op_func) sFM_GetStatus, //获取状态
 (fm_op_func) sFM_Mute, //静音
 (fm_op_func) sFM_Search, //软件搜台
 (fm_op_func) sFM_HardSeek, //硬件搜台
 (fm_op_func) sFM_SetBand, //设置波段
 (fm_op_func) sFM_SetThrod, //设置搜台门限
 (fm_op_func) sFM_BreakSeek, //中断搜索
 (fm_op_func) sFM_GetHardSeekflag, //获取硬件搜台的是否结束
 (fm_op_func) sFM_GetBand, //获取当前波段
 (fm_op_func) sFM_GetFreq, //获取当前播放频率
 (fm_op_func) sFM_GetIntsity, //获取当前信号强度
 (fm_op_func) sFM_GetAnten, //是否有天线
 (fm_op_func) sFM_GetStereo, //是否立体音
 (fm_op_func) sFM_Debug, //调试接口， 便于方案公司调试用
};
```

- 将各款 FM 模组可以公用的代码分离出来，独立实现。比如 rom\_I2C.c 是 I2C 操作代码，已经固化；rcode\_fm\_op\_entry.c 定义 FM 驱动接口及 IO 配置；bank\_a\_fm\_init.c 是驱动初始化和退出函数等。
- 各款模组针对驱动接口，分别实现。并考虑将比较常用的接口实现放在常驻空间。
- 由于系统并未给 FM 驱动分配代码和数据空间，驱动中代码和数据空间的分配如下：除了固化代码外，常驻代码和 codec 空间复用。因为 FM 引擎后台需要的数据空间不多，所以可从引擎的数据空间中分配一块作为 FM 驱动的数据空间。

```
link_base.xn

/*FM 引擎 DATA,BSS*/
SRAM_FM_ENG_RDATA_ADDR = SRAM_AP_BACK_DATA_ADDR;
SRAM_FM_ENG_RDATA_SIZE = 0x300;

/*FM DRIVE*/
```



```

SRAM_FM_DRV_RCODE_ADDR = SRAM_AP_FRONT_BASAL_RCODE_ADDR;
SRAM_FM_DRV_RCODE_SIZE = 0x800;
SRAM_FM_DRV_RDATA_ADDR = SRAM_FM_ENG_RDATA_ADDR +
SRAM_FM_ENG_RDATA_SIZE;
SRAM_FM_DRV_RDATA_SIZE = SRAM_AP_BACK_DATA_SIZE - SRAM_FM_ENG_RDATA_SIZE;

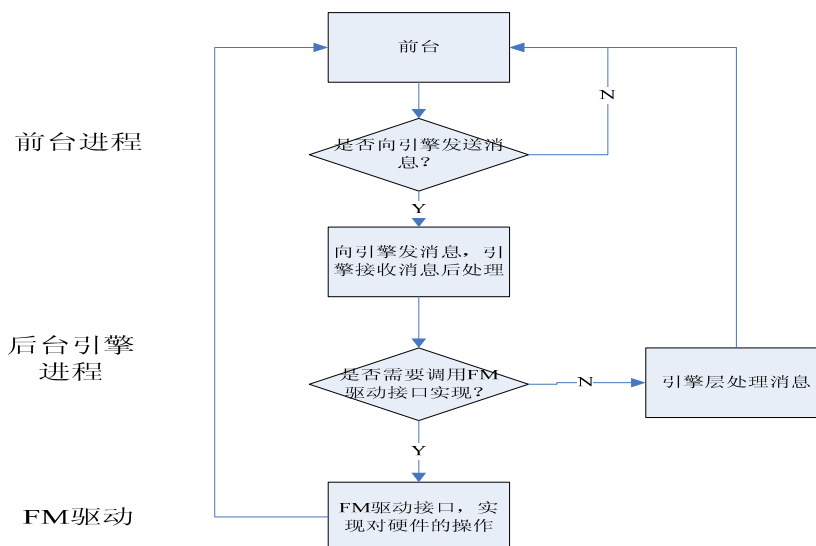
fm_driver.xn

SRAM_TEXT_ADDR = SRAM_FM_DRV_RCODE_ADDR;
RCODE_TEXT_ADDR = RCODE_ADDR_BASE + SRAM_TEXT_ADDR;
RCODE_SIZE = SRAM_FM_DRV_RCODE_SIZE;

SRAM_DATA_ADDR = SRAM_FM_DRV_RDATA_ADDR;
RDATA_DATA_ADDR = RDATA_ADDR_BASE + SRAM_DATA_ADDR;
DATA_BSS_SIZE = SRAM_FM_DRV_RDATA_SIZE;

```

## 23.6.1 总体设计



FM 驱动是 FM 硬件操作层，需要时，调用 `sys_drv_install(..."drv_fm.drv")` 进行装载，此时会将驱动常驻代码以及初始化数据搬入内存，并执行驱动初始化函数。

驱动装载完成后，对驱动的使用完全由上层控制，当前台需要操作 FM 模组时，可通过向 FM 引擎发送消息，来调用 FM 驱动的接口，实现硬件操作。

比如：前台播放场景下，需要步进调节频率，此时前台更换到新的频率后，向 FM 引擎发送 `MSG_FMENGINE_SETFREQ_SYNC` 消息，并将新的频率值传递下去，则引擎收到消息后，会调用 FM 驱动的接口 `fm_set_freq`，将频率值通过 I2C 写到模组寄存器，实现对频率的调整。

## 23.6.2 FM 驱动的模块划分

按照上述 FM 驱动设计原则，将 FM 驱动的模块划分如下：

| 模块名称     | 功能简述                                                                           | 对应文件                |
|----------|--------------------------------------------------------------------------------|---------------------|
| 驱动初始化模块  | 驱动的装载初始化以及卸载函数，为方便各模组公用。这两个函数主要给 FM 模组配置 CLOCK。                                | bank_a_fm_init.c    |
| I2C 硬件模块 | 主要是使用模拟 I2C 接口，实现向 FM 模组寄存器批量写入数据或读取数据的功能。此模块代码已经固化，为支持模拟 IO 口的配置，相关 IO 由传参输入。 | rom_I2C.c           |
| 接口及配置模块  | 配置模拟 IO 口，列出 FM 驱动向上提供的接口表                                                     | rcode_fm_op_entry.c |

以上三个模块为各个 FM 模组共用模块

|            |                                                 |                  |
|------------|-------------------------------------------------|------------------|
| 常驻代码模块     | FM 驱动比较常用的接口实现，放在驱动常驻空间。                        | rcode_fm_deal.c  |
| 驱动初始化模块    | 驱动的装载初始化以及卸载函数，为方便各模组公用。这两个函数主要给 FM 模组配置 CLOCK。 | bank_a_fm_init.c |
| 驱动 BANK 模块 | FM 驱动其他的接口实现，放在 BANK 空间。                        | bank_b*.c        |

## 23.6.3 FM 驱动的硬件接口设计

FM 驱动的硬件接口部分，主要是模拟 I2C 的 PIN 配置，以及模拟 I2C 的代码实现。

由于模拟 I2C 的代码需要固化，而 IO 口要可配，因此，I2C 操作的相关函数，其 IO 口是由传参确定的，另外，为适应不同模组的 I2C 传输速率要求，操作延时也传参确定。相关配置如下：

```
/* TWI 所用模拟 GPIO 信息结构*/
gpio_init_cfg_t gpio_twi[2] =
{
 { GPIOAINEN, GPIOAOUTEN, GPIOADAT, GPIO_SCL_BIT },
 { GPIOAINEN, GPIOAOUTEN, GPIOADAT, GPIO_SDA_BIT }
};
```

```
/* TWI 操作延时*/
uint8 delay_twi = PARAM_DELAY_TWI;
```

I2C 固化代码向上提供若干接口，对所有模组可以通用：

```
uint8 TWI_Trans_Bytes(uint8 *buf, uint8 address, uint8 length, gpio_init_cfg_t* gpio, uint8* delay);
uint8 TWI_Recev_Bytes(uint8 *buf, uint8 address, uint8 length, gpio_init_cfg_t* gpio, uint8* delay);
void TWI_Start(gpio_init_cfg_t* gpio, uint8* delay);
void TWI_Stop(gpio_init_cfg_t* gpio, uint8* delay);
void TWI_WriteByte(uint8 dat, gpio_init_cfg_t* gpio, uint8* delay);
uint8 TWI_ReadByte(gpio_init_cfg_t* gpio, uint8* delay);
void TWI_Init(gpio_init_cfg_t* gpio, uint8* delay);
void TWI_Exit(gpio_init_cfg_t* gpio);
void TWI_SendAck(uint8 ack, gpio_init_cfg_t* gpio, uint8* delay);
uint8 TWI_GetAck(gpio_init_cfg_t* gpio, uint8* delay);
```

## 23.6.4 FM 驱动的应用接口设计

FM 驱动向上提供的接口，是供 FM 引擎调用，实现硬件操作功能。FM 驱动的应用接口，应充分考虑到应用需实现的功能，并实现各模组向上接口的统一，使 FM 驱动的底层硬件操作，对于上层应用来说是透明的。

## 23.6.5 FM 驱动提供的统一接口及每个宏定义接口的介绍

FM 驱动用到的相关宏定义如下：

```
/* 模拟 TWI 总线 GPIO 配置宏*/
#define TWI_SCL_BIT 31
#define TWI_SDA_BIT 15
#define GPIO_SCL_BIT (0x00000001<<TWI_SCL_BIT)
#define GPIO_SDA_BIT (0x00000001<<TWI_SDA_BIT)
```

FM 驱动向上提供的接口介绍：

- int sFM\_Init(radio\_band\_e band, uint8 level, uint32 freq)  
FM 模组初始化
- int sFM\_Standby(void\* null1, void\* null2, void\* null3)  
FM 模组进入 Standby
- int sFM\_SetFreq(uint32 freq, void\* null2, void\* null3)  
设置一个频率开始播放
- int sFM\_GetStatus(void \* pstruct\_buf, uint8 mode, void\* null3)  
获取当前模组的状态信息

- `int sFM_Mute(FM_MUTE_e mode, void* null2, void* null3)`  
模组静音或者解除静音模式
- `int sFM_Search(uint32 freq, uint8 direct, void* null3)`  
软件搜台操作，即逐个设置频率，判断是否有效电台
- `int sFM_HardSeek(uint32 freq, uint8 direct, void* null3)`  
启动硬件搜台操作，即设置好起始和结束频率，由模组硬件执行搜台
- `int sFM_SetBand(radio_band_e band, void* null2, void* null3)`  
设置电台波段
- `int sFM_SetThrod(uint8 level, void* null2, void* null3)`  
设置搜台门限值
- `int sFM_BreakSeek(void* null1, void* null2, void* null3)`  
退出硬件搜台操作。
- `int sFM_GetHardSeekflag(void* flag, void* null2, void* null3)`  
获取硬件搜台是否完成的标记
- `int sFM_GetBand(void* band, void* null2, void* null3)`  
获取当前波段信息
- `int sFM_GetFreq(void* freq, void* null2, void* null3)`  
获取当前频率信息
- `int sFM_GetIntsity(void* intensity, void* null2, void* null3)`  
获取当前电台的信号强度
- `int sFM_GetAnten(void* antenna, void* null2, void* null3)`  
获取天线（耳机）连接状态
- `int sFM_GetStereo(void* stereo, void* null2, void* null3)`  
获取当前电台的立体声信息。
- `int sFM_Debug(void* null1, void* null2, void* null3)`  
调试用。

### 23.6.6 FM 驱动的数据流图

FM 驱动的数据空间比较充裕，基本上都是常驻数据，主要关注两个数组 `WriteBuffer` 和 `ReadBuffer`，这是和 I2C 操作相关的数据 `buffer`。全局数据的含义在代码中已有详细描述，此处不再赘述。

### 23.6.7 FM 驱动的内存分配说明

- 系统未给 FM 驱动分配专门的内存空间，考虑到 FM 驱动执行时，没有 music 播放等，因此：FM 驱动常驻代码空间和 codec 空间复用。
- FM 驱动常驻的数据空间，可从引擎数据空间中划分出来，供 FM 驱动使用。因为 FM 引擎数据量较少，

不需要的那么多数据空间。

## 23.6.8 FM 驱动的的修改指南

更换 FM 的 IC 之后,需要针对新的 FM IC,修改 FM 驱动的硬件接口部分,并根据硬件设计修改对应的 GPIO。

步骤 1: 根据硬件设计, 修改 GPIO 配置宏。

步骤 2: 各模组共有代码, 以及驱动接口部分, 可以不需改动。按照驱动接口表, 逐一实现新模组的驱动接口即可。

## 23.6.9 FM 驱动的配置说明

FM 驱动的配置, 主要是配置模拟 I2C 的 GPIO, 以及 I2C 操作的延时。前者通过 TWI.h 中

```
/* 模拟 TWI 总线 GPIO 配置宏*/
#define TWI_SCL_BIT 31
#define TWI_SDA_BIT 15
#define GPIO_SCL_BIT (0x00000001<<TWI_SCL_BIT)
#define GPIO_SDA_BIT (0x00000001<<TWI_SDA_BIT)
```

以及 rcode\_fm\_op\_entry.c 中

```
/* TWI 所用模拟 GPIO 信息结构*/
gpio_init_cfg_t gpio_twi[2] =
{
 { GPIOAINEN, GPIOAOUTEN, GPIOADAT, GPIO_SCL_BIT },
 { GPIOAINEN, GPIOAOUTEN, GPIOADAT, GPIO_SDA_BIT }
};
```

即可修改为 GPIOA 或者 GPIOB 的任意 IO 口。

后者通过 fm\_drv.h 中的

```
#define PARAM_DELAY_TWI 20
```

即可修改。

## 23.6.10 前台是怎么样设置频率的

以前台设置 FM 的频率为例。

- 前台最终都会执行到 radio\_set\_freq
- radio\_set\_freq 发送 MSG\_FMENGINE\_SETFREQ\_SYNC 消息到引擎;
- 引擎的 fmengine\_set\_freq\_sync-> fm\_set\_freq
- 驱动的 sFM\_SetFreq-> QND\_TuneToCH

- 这里面有很多读写寄存器的操作，操作的就是 FM 芯片的寄存器。但本质是它都是通过一组 I2C 的接口如：TWI\_Trans\_Bytes 等来实现的。

### 23.6.11 如何增加一个 FM 驱动的应用接口

增加一个 FM 驱动的应用接口，按照如下步骤实现即可：

1. 在 rcode\_fm\_op\_entry.c 的 fm\_drv\_op 中，增加接口函数。
  2. 在 fm\_interface.h 的 fm\_driver\_operations 中，相应位置添加新的接口成员。
  3. 在 fm\_interface.h 的 fm\_cmd\_e 中，相应位置添加新的接口命令。
  4. 在 fm\_interface.h 中，添加新的宏定义，即上层应用调用驱动时的函数名。
- 接口定义好后，在驱动中实现接口功能。

## 24 API 介绍

### 24.1 KERNEL API

操作系统接口按照不同的功能模块分为：时间管理、设备驱动管理、虚拟文件系统管理、VM、中断管理、AP 管理、字符设备驱动、调频管理等。

#### 24.1.1 时间管理 API

##### 24.1.1.1 sys\_set\_irq\_timer1

- 功能描述：用于注册毫秒级的定时器 timer1，单位为 1ms。
- 函数原型：int8 sys\_set\_irq\_timer1(void\* time\_handle, uint32 ms\_count)
- 输入参数描述：

|             |          |
|-------------|----------|
| time_handle | 定时器回调函数  |
| ms_count    | 定时周期(ms) |
- 输出参数描述：

|     |         |                  |
|-----|---------|------------------|
| 0~5 | success | 定时器索引号。最多 6 个定时器 |
| -1  | failed  |                  |
- 说明：bank 代码，禁止在中断调用；与 sys\_del\_irq\_timer1 配合使用

##### 24.1.1.2 sys\_del\_irq\_timer1

- 功能描述：删除 Timer1 定时器，单位为 10ms。
- 函数原型：int sys\_del\_irq\_timer1(unsigned int timer)
- 输入参数描述：

|       |        |
|-------|--------|
| timer | 定时器索引号 |
|-------|--------|
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：bank 代码，禁止在中断调用；与 sys\_set\_irq\_timer1 配合使用

### 24.1.1.3 sys\_udelay

- 功能描述：微秒级延时，CPU 空等，不释放。
- 函数原型：void sys\_udelay(uint32 us)
- 输入参数描述：  
us            延时时间
- 输出参数描述：无
- 说明：

### 24.1.1.4 sys\_mdelay

- 功能描述：毫秒级延时，CPU 空等，不释放。
- 函数原型：void sys\_mdelay(uint32 ms)
- 输入参数描述：  
ms            延时时间
- 输出参数描述：无
- 说明：
- 

### 24.1.1.5 sys\_get\_ab\_timer

- 功能描述：获取绝对时间，单位 ms，毫秒精度，count 模块实现。
- 函数原型：uint32 sys\_get\_ab\_timer(void)
- 输入参数描述：无
- 输出参数描述：返回绝对时间 ms
- 说明：

### 24.1.1.6 sys\_set\_time

- 功能描述：设置日历的时分秒。RTC 模块实现。
- 函数原型：void sys\_set\_time(time\_t \*time)
- 输入参数描述：



time 设置时间参数

- 输出参数描述：无
- 说明：

### 24.1.1.7 sys\_get\_time

- 功能描述：获取日历的时分秒。RTC 模块实现。
- 函数原型：void sys\_get\_time(time\_t \*time)
- 输入参数描述：无
- 输出参数描述：

time 获取的时间
- 说明：

### 24.1.1.8 sys\_set\_date

- 功能描述：设置日历的年月日。RTC 模块实现。
- 函数原型：void sys\_set\_date (date\_t \*date)
- 输入参数描述：

date 设置日期
- 输出参数描述：

0 success

-1 failed
- 说明：如果输入日期格式不对，接口会返回失败

### 24.1.1.9 sys\_get\_date

- 功能描述：获取日历的年月日。RTC 模块实现。
- 函数原型：void sys\_get\_date (date\_t \*date)
- 输入参数描述：无
- 输出参数描述：

date 获取的日期
- 说明：

### 24.1.1.10 sys\_set\_alarm\_time

- 功能描述：设置闹钟的时间。RTC 模块实现。
- 函数原型：void sys\_set\_alarm\_time (time\_t \*time)
- 输入参数描述：  
time 闹钟时间
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用

### 24.1.1.11 sys\_get\_alarm\_time

- 功能描述：获取闹钟的时间。RTC 模块实现。
- 函数原型：void sys\_get\_alarm\_time (time\_t \*time)
- 输入参数描述：无
- 输出参数描述：  
time 获取的闹钟时间
- 说明：

### 24.1.1.12 sys\_os\_time\_dly

- 功能描述：用于挂起线程睡眠，释放 CPU 控制权，延时一定时间。
- 函数原型：void sys\_os\_time\_dly(uint16 ticks)
- 输入参数描述：  
ticks 延时时间节拍数，单位 10ms
- 输出参数描述：无
- 说明：

### 24.1.1.13 sys\_os\_time\_dly\_resume

- 功能描述：恢复延时的任务。
- 函数原型：uint8 sys\_os\_time\_dly\_resume(uint8 prio)
- 输入参数描述：  
prio 任务优先级

- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

### 24.1.1.14 sys\_sleep

- 功能描述：用于挂起线程睡眠，释放 CPU 控制权，秒级延时。
- 函数原型：void sys\_sleep(uint32 s)
- 输入参数描述：

|   |           |
|---|-----------|
| s | 延时时间，单位 s |
|---|-----------|
- 输出参数描述：无
- 说明：

### 24.1.1.15 sys\_usleep

- 功能描述：挂起线程睡眠，释放 CPU 控制权，微级延时。
- 函数原型：void sys\_usleep(uint32 us)
- 输入参数描述：

|    |            |
|----|------------|
| us | 延时时间，单位 us |
|----|------------|
- 输出参数描述：无
- 说明：

### 24.1.1.16 sys\_get\_delay\_val

- 功能描述：用于获取延时时数值。
- 函数原型：int sys\_get\_delay\_val(uint32 delay\_ms uint32 div\_val)
- 输入参数描述：

|          |            |
|----------|------------|
| delay_ms | 延时时间，单位 ms |
| div_val  | 时钟分频比      |
- 输出参数描述：

|       |
|-------|
| 延时时数值 |
|-------|
- 说明：bank 代码，禁止在中断调用

### 24.1.1.17 sys\_us\_timer\_start

- 功能描述：用于初始微秒计时器。
- 函数原型：void sys\_us\_timer\_start(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：

### 24.1.1.18 sys\_us\_timer\_break

- 功能描述：用于获取微秒计时器值，微秒级误差
- 函数原型：uint32 sys\_us\_timer\_break(void)
- 输入参数描述：无
- 输出参数描述：  
微秒时间值
- 说明：与 sys\_us\_timer\_start 配合使用，计算过程中不能有调频动作，否则计时不准。

### 24.1.1.19 sys\_reset\_timer

- 功能描述：用于重置系统的定时计数器
- 函数原型：void sys\_reset\_timer(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用

## 24.1.2 设备驱动管理 API

### 24.1.2.1 sys\_drv\_install

- 功能描述：装载驱动。
- 函数原型：int sys\_drv\_install (uint8 drv\_type, void \*drv\_para, char\* drv\_name)

- 输入参数描述：
  - drv\_type          驱动类型索引，详见 drv\_type\_t 说明
  - drv\_para          传给驱动参数
  - drv\_name          驱动文件名，如 card.drv
- 输出参数描述：
  - 0                  success
  - 1                 failed
- 说明：bank 代码，禁止在中断调用；与 sys\_drv\_uninstall 一定要匹配使用。

### 24.1.2.2 sys\_drv\_uninstall

- 功能描述：卸载驱动。
- 函数原型：int sys\_drv\_uninstall(uint8 drv\_type)
- 输入参数描述：
  - drv\_type          驱动类型索引，详见 drv\_type\_t 说明
- 输出参数描述：
  - 0                  success
  - 1                 failed
- 说明：bank 代码，禁止在中断调用；与 sys\_drv\_install 一定要匹配使用。

### 24.1.2.3 sys\_get\_drv\_install\_info

- 功能描述：获取驱动安装信息，低 8 个比特表示该种驱动被安装的次数，次低 8 个比特表示该种驱动的工作模式。务必保证多次安装时的工作模式是一样的。在双应用场景，两个应用安装的驱动都应该是 MODE\_NORMAL。。
- 函数原型：int sys\_get\_drv\_install\_info(uint8 drv\_type)
- 输入参数描述：
  - drv\_type          驱动类型索引，详见 drv\_type\_t 说明
- 输出参数描述：驱动安装信息
- 说明：bank 代码，禁止在中断调用

### 24.1.2.4 sys\_detect\_disk

- 功能描述：检测存储介质是否存在。

- 函数原型: `int sys_detect_disk(uint8 drv_type)`
- 输入参数描述:
  - `drv_type` 驱动类型索引, 详见 `drv_type_t` 说明
- 输出参数描述:
  - 0 存在
  - 1 不存在
- 说明: bank 代码, 禁止在中断调用

### 24.1.2.5 sys\_set\_drv\_setting

- 功能描述: 设备 card 驱动配置, 如 card 驱动能力等。
- 函数原型: `int sys_set_drv_setting(card_pm_cfg_t *set_info)`
- 输入参数描述:
  - 设置信息内容 `card_pm_cfg_t`
- 输出参数描述:
  - 0 成功
  - 1 失败
- 说明: bank 代码, 禁止在中断调用

### 24.1.2.6 sys\_set\_drv\_ops

- 功能描述: 用于重定位驱动的 ops 地址
- 函数原型: `int sys_set_drv_ops(uint8 drv_type, uint32 ops_addr)`
- 输入参数描述:
  - `drv_type`: 驱动类型索引, 详见 `drv_type_t` 说明
  - `ops_addr`: 驱动地址表
- 输出参数描述:
  - 0 成功
  - 1 失败
- 说明: bank 代码, 禁止在中断调用

## 24.1.3 VFS 管理 API

### 24.1.3.1 sys\_mount\_fs

- 功能描述：挂载设备的文件系统
- 函数原型：int sys\_mount\_fs(uint8 drv\_type, uint8 disk, uint8 partition\_num)
- 输入参数描述：

|               |                         |
|---------------|-------------------------|
| drv_type      | 驱动类型索引，详见 drv_type_t 说明 |
| disk          | 设备盘符，如 H,U 等            |
| partition_num | 分区号，默认为 0               |
- 输出参数描述：

|     |                                                                     |
|-----|---------------------------------------------------------------------|
| 0~2 | success, vfs_mount_t 数据结构的索引。以后对该用户驱动的文件系统进行操作都需要把这个索引作为第一个参数传给 vfs |
| -1  | failed                                                              |
- 说明：bank 代码，禁止在中断调用；装载 FS 前，一定要先装载存储驱动。

### 24.1.3.2 sys\_unmount\_fs

- 功能描述：卸载已挂载的文件系统。
- 函数原型：int sys\_unmount\_fs(int8 vfs\_mount\_index)
- 输入参数描述：

|                 |                      |
|-----------------|----------------------|
| vfs_mount_index | vfs_mount_t 数据结构的索引。 |
|-----------------|----------------------|
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：bank 代码，禁止在中断调用；

### 24.1.3.3 sys\_get\_fat\_type\_after\_mount

- 功能描述：获取挂载后的文件系统的类型。
- 函数原型：uint8 sys\_get\_fat\_type\_after\_mount(uint32 vfs\_mount\_index)
- 输入参数描述：

|                 |                      |
|-----------------|----------------------|
| vfs_mount_index | vfs_mount_t 数据结构的索引。 |
|-----------------|----------------------|
- 输出参数描述：

|   |             |
|---|-------------|
| 0 | FAT16/FAT32 |
|---|-------------|

1 exFAT

- 说明：bank 代码，禁止在中断调用；

### 24.1.3.4 sys\_format\_disk

- 功能描述：指定文件系统类型，格式化指定设备分区。
- 函数原型：int sys\_format\_disk(uint8 dry\_type, uint8 partition\_num, uint8 fat\_type)
- 输入参数描述：

|               |                                                |
|---------------|------------------------------------------------|
| dry_type      | 存储驱动类型                                         |
| partition_num | 分区号，默认为 0                                      |
| fat_type      | FS 类型，FORMAT_FAT32, FORMAT_EXFAT, FORMAT_FAT16 |
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：bank 代码，禁止在中断调用；

## 24.1.4 SDFS 管理 API

### 24.1.4.1 sys\_sd\_fopen

- 功能描述：打开一个 sd 区的文件。
- 函数原型：sd\_file\_t \*sys\_sd\_fopen (char \* filename)
- 输入参数描述：

|          |                                   |
|----------|-----------------------------------|
| filename | 文件名 8+3, xxx.yyy 格式字符串，一定能够是 0 结尾 |
|----------|-----------------------------------|
- 输出参数描述：

|     |        |
|-----|--------|
| 非 0 | 文件句柄   |
| 0   | failed |
- 说明：bank 代码，禁止在中断调用；最多支持 8 个 sdfs 句柄。

### 24.1.4.2 sys\_sd\_fclose

- 功能描述：关闭 SDFS 句柄。
- 函数原型：int sys\_sd\_fclose (sd\_file\_t \*fp)
- 输入参数描述：



- |    |      |
|----|------|
| fp | 文件句柄 |
|----|------|
- 输出参数描述:

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明:

### 24.1.4.3 sys\_sd\_fseek

- 功能描述: 在已经打开的文件内 seek 到某个位置。
- 函数原型: `int sys_sd_fseek (sd_file_t *fp, uint8 whence, int32 offset)`
- 输入参数描述:

|          |                                                                                                                                                                            |          |   |     |          |   |     |          |   |     |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|---|-----|----------|---|-----|----------|---|-----|
| fp       | 文件句柄                                                                                                                                                                       |          |   |     |          |   |     |          |   |     |
| whence   | seek 方向,                                                                                                                                                                   |          |   |     |          |   |     |          |   |     |
|          | <table border="0"><tr><td>SEEK_SET</td><td>0</td><td>只能正</td></tr><tr><td>SEEK_CUR</td><td>1</td><td>正/负</td></tr><tr><td>SEEK_END</td><td>2</td><td>只能正</td></tr></table> | SEEK_SET | 0 | 只能正 | SEEK_CUR | 1 | 正/负 | SEEK_END | 2 | 只能正 |
| SEEK_SET | 0                                                                                                                                                                          | 只能正      |   |     |          |   |     |          |   |     |
| SEEK_CUR | 1                                                                                                                                                                          | 正/负      |   |     |          |   |     |          |   |     |
| SEEK_END | 2                                                                                                                                                                          | 只能正      |   |     |          |   |     |          |   |     |
| offset   | 偏移值                                                                                                                                                                        |          |   |     |          |   |     |          |   |     |
- 输出参数描述:

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明:

### 24.1.4.4 sys\_sd\_ftell

- 功能描述: 获取文件当前读指针位置。
- 函数原型: `int sys_sd_ftell (sd_file_t *fp)`
- 输入参数描述:

|    |      |
|----|------|
| fp | 文件句柄 |
|----|------|
- 输出参数描述: 当前读指针的位置
- 说明:

### 24.1.4.5 sys\_sd\_fread

- 功能描述：读取已经打开的 SD 区文件的内容。
- 函数原型：int32 sys\_sd\_fread(sd\_file\_t \*fp, void \*buf, uint32 len)
- 输入参数描述：
  - fp 文件句柄
  - buf 读取文件内容到该 buffer 内
  - len 要求读取的长度，单位 byte
- 输出参数描述：当前读指针的位置  
真实读取的长度，
  - 1 failed
- 说明：

### 24.1.4.6 sys\_base\_set\_info

- 功能描述：用于 sd 区配置信息
- 函数原型：sys\_base\_set\_info(void \*info, uint32 type)
- 输入参数描述：
  - info: 配置信息
  - type: 设置类型
- 输出参数描述：
  - 0 成功
  - 1 失败
- 说明：bank 代码，禁止在中断调用

### 24.1.4.7 get\_fw\_info

- 功能描述：获取 SD 区指定位置数据
- 函数原型：int get\_fw\_info (uint8 \*buf, uint32 info\_addr, uint32 info\_len)
- 输入参数描述：
  - ptr\_fw\_info: 存放信息内容
  - info\_addr: sd 位置
  - info\_len: 数据长度(字节)
- 输出参数描述：

0            success  
-1          failed

- 说明：

## 24.1.5 VRAM 读写 API

### 24.1.5.1 sys\_vm\_read

- 功能描述：读 VM 区数据，字节单位读
- 函数原型：int sys\_vm\_read(void \*buf, uint32 address, uint32 len)
- 输入参数描述：

|         |              |
|---------|--------------|
| buf     | 读的目标内存地址     |
| address | 读的 VM 地址     |
| len     | 读的长度，单位 byte |
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

### 24.1.5.2 sys\_vm\_write

- 功能描述：写入 VRAM 的数据，字节单位写入
- 函数原型：int sys\_vm\_write(void \*buf, uint32 address, uint32 len)
- 输入参数描述：

|         |             |
|---------|-------------|
| buf     | 数据内存地       |
| address | 写入的 VM 地址   |
| len     | 写入数据长度，字节单位 |
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

## 24.1.6 中断管理 API

### 24.1.6.1 sys\_request\_irq

- 功能描述：注册中断。
- 函数原型：int sys\_request\_irq(uint32 irq\_type, void \*handle)
- 输入参数描述：

|          |                     |
|----------|---------------------|
| irq_type | 中断索引号，详见 irq_type_t |
| handle   | 中断回调函数              |
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：bank 代码，禁止在中断调用；与 sys\_free\_irq 配对使用

### 24.1.6.2 sys\_free\_irq

- 功能描述：注销中断。
- 函数原型：void sys\_free\_irq(uint32 irq\_type)
- 输入参数描述：

|          |                     |
|----------|---------------------|
| irq_type | 中断索引号，详见 irq_type_t |
|----------|---------------------|
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用；与 sys\_request\_irq 配对使用

### 24.1.6.3 sys\_local\_irq\_save

- 功能描述：关中断。
- 函数原型：uint32 sys\_local\_irq\_save(void)
- 输入参数描述：无
- 输出参数描述：返回当前中断配置，必须保存。
- 说明：与 sys\_local\_irq\_restore 配对使用

### 24.1.6.4 sys\_local\_irq\_restore

- 功能描述：恢复中断。
- 函数原型：void sys\_local\_irq\_restore(uint32 irq\_save)
- 输入参数描述：  
irq\_save 之前函数 sys\_local\_irq\_save 保存的状态值
- 输出参数描述：无
- 说明：与 sys\_local\_irq\_save 配对使用

### 24.1.6.5 sys\_set\_mpu\_irq

- 功能描述：注册 mpu 模块中断方式。
- 函数原型：int32 sys\_set\_mpu\_irq(sys\_mpu\_param\_t \*param)
- 输入参数描述：  
param: 注册参数类型，见定义 sys\_mpu\_param\_t
- 输出参数描述：中断注册 id 号，-1 表示失败
- 说明：bank 代码，禁止在中断调用；与 sys\_del\_mpu\_irq 配对使用

### 24.1.6.6 sys\_del\_mpu\_irq

- 功能描述：注销 mpu 中断。
- 函数原型：void sys\_del\_mpu\_irq(id)
- 输入参数描述：  
id 中断注册 id 号
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用；与 sys\_set\_mpu\_irq 配合使用

### 24.1.6.7 sys\_request\_dsp\_irq

- 功能描述：请求 DSP 协议中断
- 函数原型：int32 sys\_request\_dsp\_irq(uint8 in\_use, uint8 \*cmd\_context)
- 输入参数描述：  
in\_use 中断类型，0~3  
cmd\_context 中断协议信息，默认 0

- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：bank 代码，禁止在中断调用；

### 24.1.6.8 sys\_respond\_dsp\_cmd

- 功能描述：回应 DSP 服务请求，表示服务已完成，DSP 可以开始工作
- 函数原型：int32 sys\_respond\_dsp\_cmd(void)
- 输入参数描述：
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

## 24.1.7 AP 管理 API

### 24.1.7.1 sys\_exece\_ap

- 功能描述：装载 ap。
- 函数原型：int sys\_exece\_ap(char \*name,uint32 type,void \*argc)
- 输入参数描述：

|      |                |
|------|----------------|
| name | 应用的文件名 XXX.AP  |
| type | 应用类型，是否为引擎进程   |
| arg  | 传给 ap main 的参数 |
- 输出参数描述：0-表示成功，-1 表示失败
- 说明：bank 代码，禁止在中断调用；sys\_exece\_ap 与 sys\_free\_ap 配对使用

### 24.1.7.2 sys\_free\_ap

- 功能描述：卸载 ap。
- 函数原型：void sys\_free\_ap(uint32 type)

- 输入参数描述：  
    type       应用类型，是否为引擎进程
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用；sys\_exece\_ap 与 sys\_free\_ap 配对使用

### 24.1.7.3 sys\_load\_codec

- 功能描述：装载 codec。
- 函数原型：int sys\_load\_codec(char \*name, uint32 type)
- 输入参数描述：  
    name       codec 名字  
    type       是否为引擎进程加载的 codec 驱动
- 输出参数描述：  
    0           success  
    -1          failed
- 说明：bank 代码，禁止在中断调用；sys\_load\_codec 与 sys\_free\_codec 配对使用

### 24.1.7.4 sys\_free\_codec

- 功能描述：卸载 codec。
- 函数原型：void sys\_free\_codec(uint32 type)
- 输入参数描述：  
    type       是否为引擎进程加载的 codec 驱动
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用；sys\_load\_codec 与 sys\_free\_codec 配对使用

### 24.1.7.5 sys\_load\_mmm

- 功能描述：装载中间件。
- 函数原型：int sys\_load\_mmm(char \*name, uint32 type)
- 输入参数描述：  
    name       中间件名字  
    type       是否为引擎进程加载的 mmm 驱动
- 输出参数描述：

0            success  
-1           failed

- 说明：bank 代码，禁止在中断调用；sys\_load\_mmm 与 sys\_free\_mmm 配对使用

### 24.1.7.6 sys\_free\_mmm

- 功能描述：装载中间件。
- 函数原型：void sys\_free\_mmm(uint32 type)
- 输入参数描述：  
type        是否为引擎进程加载的 mmm 驱动
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用；sys\_load\_mmm 与 sys\_free\_mmm 配对使用

### 24.1.7.7 sys\_load\_dsp\_codec

- 功能描述：加载 DSP 库。
- 函数原型：int sys\_load\_dsp\_codec(char \*name, uint32 type)
- 输入参数描述：  
name            库名字  
type            库类型
- 输出参数描述：  
0            success  
-1           failed
- 说明：bank 代码，禁止在中断调用；sys\_load\_dsp\_codec 与 sys\_free\_dsp\_codec 配对使用

### 24.1.7.8 sys\_free\_dsp\_codec

- 功能描述：卸载 DSP 库。
- 函数原型：void sys\_free\_dsp\_codec(uint32 type)
- 输入参数描述：  
type        库类型
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用；sys\_load\_dsp\_codec 与 sys\_free\_dsp\_codec 配对使用



## 24.1.8 调频管理 API

### 24.1.8.1 sys\_adjust\_clk

- 功能描述：用于调整系统 MISP 和 DSP 频率。
- 函数原型：uint32 sys\_adjust\_clk(uint32 freq, uint32 type)
- 输入参数描述：
  - freq 频率值，低 8 位为 MISP 频率，高 8 位为 DSP 频率
  - type 调频类型，无用，默认为 0
- 输出参数描述：返回调频前的频率值，低 8 位为 MISP 频率，高 8 位为 DSP 频率。
- 说明：bank 代码，禁止在中断调用。

### 24.1.8.2 sys\_adjust\_asrc\_clk

- 功能描述：用于调整系统 ASRC 频率。
- 函数原型：uint32 sys\_adjust\_asrc\_clk(uint32 freq)
- 输入参数描述：
  - freq 频率值
- 输出参数描述：返回调频前的 ASRC 频率值。
- 说明：bank 代码，禁止在中断调用。

### 24.1.8.3 sys\_request\_clkadjust

- 功能描述：注册调频回调函数。
- 函数原型：int8 sys\_request\_clkadjust(void \*call\_back, uint32 pll\_range)
- 输入参数描述：
  - call\_back：回调函数入口
  - pll\_range：限制 PLL 频率范围，低 16 位为最小值，高 16 位为最大值；若值为 0 表示不限制 PLL 频率
- 输出参数描述：
  - 0~5 调频数组的管理索引号
  - 1 failed

- 说明：bank 代码，禁止在中断调用；

#### 24.1.8.4 sys\_free\_clkadjust

- 功能描述：注销调频回调函数。
- 函数原型：int sys\_free\_clkadjust(int8 id)
- 输入参数描述：

|    |             |
|----|-------------|
| id | 调频数组的管理索引号。 |
|----|-------------|
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：bank 代码，禁止在中断调用；

#### 24.1.8.5 sys\_lock\_adjust\_freq

- 功能描述：锁定系统当前的 MISP&DSP 频率，不接受其他频率的调节
- 函数原型：void sys\_lock\_adjust\_freq(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用；

#### 24.1.8.6 sys\_unlock\_adjust\_freq

- 功能描述：解锁系统的 MISP&DSP 频率调节
- 函数原型：void sys\_unlock\_adjust\_freq(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用；

## 24.1.9 内存管理 API

### 24.1.9.1 sys\_malloc

- 功能描述：申请内存空间。
- 函数原型：void\* sys\_malloc(uint32 size)
- 输入参数描述：内存地址，为 0 表示失败  
size 申请的字节大小
- 输出参数描述：  
0 success  
-1 failed
- 说明：sys\_malloc 与 sys\_free 配对使用

### 24.1.9.2 sys\_free

- 功能描述：释放申请内存。
- 函数原型：int sys\_free(void\* addr)
- 输入参数描述：  
addr 内存的指针变量的地址。
- 输出参数描述：  
0 success  
-1 failed
- 说明：sys\_malloc 与 sys\_free 配对使用。

## 24.1.10 消息处理 API

### 24.1.10.1 sys\_mq\_send

- 功能描述：发送消息。
- 函数原型：int mq\_send(uint8 queue\_id, void \*msg)
- 输入参数描述：  
queue\_id 消息队列索引号。

- |         |         |
|---------|---------|
| msg     | 消息内容指针  |
| 输出参数描述: |         |
| 0       | success |
| -1      | failed  |
- 说明:

### 24.1.10.2 sys\_mq\_receive

- 功能描述: 接收消息。
- 函数原型: `int sys_mq_receive (uint8 queue_id, void *msg)`
- 输入参数描述:

|          |          |
|----------|----------|
| queue_id | 消息队列索引号。 |
| msg      | 消息内容指针   |
- 输出参数描述:

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明:

### 24.1.10.3 sys\_mq\_flush

- 功能描述: 清空消息队列。
- 函数原型: `int sys_mq_flush (uint8 queue_id)`
- 输入参数描述:

|          |          |
|----------|----------|
| queue_id | 消息队列索引号。 |
|----------|----------|
- 输出参数描述:

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明:

### 24.1.10.4 sys\_mq\_traverse

- 功能描述: 查询/获取 消息, 获取后的消息不会被删除
- 函数原型: `int sys_mq_traverse(uint8 queue_id, void* msg, uint32 msg_index)`
- 输入参数描述:

|           |         |
|-----------|---------|
| queue_id: | 消息类型 id |
|-----------|---------|

msg: 存放消息内容, 若为 NULL 则只查询消息总数, 不获取消息内容

msg\_index: 获取指定的消息序号

- 输出参数描述: 消息池中消息的总数, 为-1 表示操作失败
- 说明:

## 24.1.11 共享查询机制 API

### 24.1.11.1 sys\_share\_query\_creat

- 功能描述: 创建共享内存查询管理队列, 并返回可被写入的内存地址。
- 函数原型: `void* sys_share_query_creat(int8 query_id, uint8 *mem_addr, uint16 size)`
- 输入参数描述:
  - query\_id: 队列 ID
  - mem\_addr: 内存地址
  - size: 内存大小
- 输出参数描述: 可写入的内存地址, NULL -为创建失败
- 说明: bank 代码, 禁止在中断调用

### 24.1.11.2 sys\_share\_query\_destroy

- 功能描述: 删除已创建的队列。
- 函数原型: `int sys_share_query_destroy(int8 query_id)`
- 输入参数描述:
  - query\_id: 队列 ID
- 输出参数描述:

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明: bank 代码, 禁止在中断调用;

### 24.1.11.3 sys\_share\_query\_read

- 功能描述: 获取共享内存消息内容
- 函数原型: `int sys_share_query_read(int8 query_id, uint8 *read_addr)`
- 输入参数描述:

query\_id: 队列 ID

read\_addr: 存入消息内容

- 输出参数描述:

0 success

-1 failed

- 说明:

### 24.1.11.4 sys\_share\_query\_update

- 功能描述: 更新已写入共享内存消息内容, 并返回可被写入的内存地址

- 函数原型: void\* sys\_share\_query\_update(int8 query\_id)

- 输入参数描述:

query\_id: 队列 ID

- 输出参数描述: 可被写入的内存地址, NULL-为更新失败

- 说明:

### 24.1.12 共享内存机制 API

#### 24.1.12.1 sys\_shm\_query\_creat

- 功能描述: 创建共享内存空间

- 函数原型: int sys\_shm\_creat(int8 shm\_id, uint8 \*shm\_addr, uint16 shm\_size)

- 输入参数描述:

shm\_id: 共享内存标识 ID

shm\_addr: 内存地址

shm\_size: 内存大小

- 输出参数描述:

0 success

-1 failed

- 说明: bank 代码, 禁止在中断调用

### 24.1.12.2 sys\_shm\_destroy

- 功能描述：删除共享内存空间
- 函数原型：int sys\_shm\_destroy(int8 shm\_id)
- 输入参数描述：  
shm\_id: 共享内存标识 ID
- 输出参数描述：  
0 success  
-1 failed
- 说明：bank 代码，禁止在中断调用

### 24.1.12.3 sys\_shm\_mount

- 功能描述：请求共享内存地址
- 函数原型：uint8\* sys\_shm\_mount(int8 shm\_id)
- 输入参数描述：bank 代码，禁止在中断调用  
shm\_id: 共享内存标识
- 输出参数描述：共享内存地址，NULL-为请求失败
- 说明：bank 代码，禁止在中断调用

## 24.1.13 工作&信息配置 API

### 24.1.13.1 sys\_enter\_high\_powered

- 功能描述：启动系统电气特性 性能最优模式
- 函数原型：void sys\_enter\_high\_powered(int up\_type)
- 输入参数描述：  
up\_type : 提升类型，1--VDD, 2--VCC, 3--(VCC + VDD)
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用

### 24.1.13.2 sys\_exit\_high\_powered

- 功能描述：退出系统电气特性 性能最优模式
- 函数原型：void sys\_exit\_high\_powered(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用

### 24.1.13.3 sys\_set\_hosc\_param

- 功能描述：设置高频电容参数值
- 函数原型：void sys\_set\_hosc\_param(uint16 param)
- 输入参数描述：  
param 电容参数值
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用

### 24.1.13.4 sys\_set\_sys\_info

- 功能描述：设置系统信息
- 函数原型：int sys\_set\_sys\_info(void \*info, uint32 sys\_info\_type)
- 输入参数描述：  
sys\_info\_type: 信息类型，见 sys\_info\_type\_e 定义
- 输出参数描述：0 表示成功，-1 表示失败
- 说明：bank 代码，禁止在中断调用

### 24.1.13.5 sys\_get\_sys\_info

- 功能描述：获取系统信息
- 函数原型：int sys\_get\_sys\_info(void \*info, uint32 sys\_info\_type)
- 输入参数描述：  
sys\_info\_type: 信息类型，见 sys\_info\_type\_e 定义
- 输出参数描述：0 表示成功，-1 表示失败
- 说明：bank 代码，禁止在中断调用



### 24.1.13.6 sys\_random

- 功能描述：获取系统的随机数据
- 函数原型：uint32 sys\_random(void)
- 输入参数描述：
- 输出参数描述：  
随机数 范围 0~0xFFFFFFFF
- 说明：

### 24.1.13.7 sys\_read\_c0count

- 功能描述：获取系统的 cpu 计数器值
- 函数原型：uint32 sys\_read\_c0count(void)
- 输入参数描述：
- 输出参数描述：  
计数器值 范围 0~0xFFFFFFFF
- 说明：

## 24.1.14 系统调试 API

### 24.1.14.1 sys\_cpu\_monitor\_start

- 功能描述：启动 cpu 使用占空比监控
- 函数原型：void sys\_cpu\_monitor\_start(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用

### 24.1.14.2 sys\_cpu\_monitor\_end

- 功能描述：结束 cpu 使用占空比监控，查看结果
- 函数原型：void sys\_cpu\_monitor\_end(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用

### 24.1.14.3 sys\_dsp\_print

- 功能描述：打印 dsp 缓存数据
- 函数原型：void sys\_dsp\_print(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：bank 代码，禁止在中断调用

### 24.1.14.4 sys\_irq\_print

- 功能描述：用于系统中断打印
- 函数原型：void sys\_irq\_print(char \*str, uint32 data, uint8 mode)
- 输入参数描述：
  - str:字符串
  - data:数据
  - mode:打印模式，0-只打印字符串，1-只打印数据，2-打印字符串+数据
- 输出参数描述：无
- 说明：只允许在中断服务函数中使用

## 24.2 LIBC API

libc 接口按照不同的功能模块分为：线程管理、进程管理、信号量等。

## 24.2.1 线程管理 API

### 24.2.1.1 libc\_pthread\_create

- 功能描述：创建线程。
- 函数原型：int libc\_pthread\_create(pthread\_param\_t \*pthread\_param, INT8U prio, uint8 process\_descr\_index)
- 输入参数描述：

|                     |                                        |
|---------------------|----------------------------------------|
| pthread_param       | 线程创建数据结构，详见 pthread_param_t            |
| prio                | 线程优先级                                  |
| process_descr_index | CREATE_NOT_MAIN_THREAD，表示创建非 main 主线程。 |
- 输出参数描述：

|     |                          |
|-----|--------------------------|
| 0~5 | success 定时器索引号。最多 6 个定时器 |
| -1  | failed                   |
- 说明：因为 Main 主线程由运行时库创建，所以调用此接口一般为非主线程。

### 24.2.1.2 libc\_pthread\_exit

- 功能描述：线程退出。
- 函数原型：void libc\_pthread\_exit(void)
- 输入参数描述：无
- 输出参数描述：无
- 说明：Main 中如果没有调用 libc\_pthread\_exit 接口退出将会退出所属进程；如果最后一个线程调用该接口后也会退出所属进程。

## 24.2.2 进程管理 API

### 24.2.2.1 libc\_exit

- 功能描述：进程退出。
- 函数原型：void libc\_exit(int8 exitval)
- 输入参数描述：

|         |                                  |
|---------|----------------------------------|
| exitval | 退出进程时传递的参数，将会由 waitpid 接口获得该参数的值 |
|---------|----------------------------------|

- 输出参数描述：无
- 说明：Main 中如果没有调用 pthread\_exit 接口退出，将会调用该接口。

### 24.2.2.2 libc\_waitpid

- 功能描述：等待其他进程退出。
- 函数原型：int libc\_waitpid(int8 \*stat\_loc, int options)
- 输入参数描述：

|          |                                |
|----------|--------------------------------|
| stat_loc | 获得 exit 中传递的参数                 |
| options  | 指示是否阻塞等待，WNOHANG 表示不阻塞返回，其他会阻塞 |
- 输出参数描述：

|                              |            |
|------------------------------|------------|
| 0~3                          | 表示退出的进程索引号 |
| WAITPID_ONLY_PROCESS_MANAGER | 表示没有其他进程存在 |
| WAITPID_NO_PROCESS_EXIT      | 表示没有其他进程退出 |
- 说明：

### 24.2.2.3 libc\_get\_process\_struct

- 功能描述：申请进程。
- 函数原型：int libc\_get\_process\_struct(void)
- 输入参数描述：无
- 输出参数描述：

|     |       |
|-----|-------|
| 0~3 | 进程索引号 |
| -1  | fail  |
- 说明：

### 24.2.2.4 libc\_free\_process\_struct

- 功能描述：释放进程。
- 函数原型：int libc\_free\_process\_struct(int8 index)
- 输入参数描述：

|       |       |
|-------|-------|
| index | 进程索引号 |
|-------|-------|
- 输出参数描述：无

- 说明：

## 24.2.3 信号量 API

### 24.2.3.1 libc\_sem\_init

- 功能描述：信号量初始化。
- 函数原型：int libc\_sem\_init(os\_event\_t \*sem, unsigned int value)
- 输入参数描述：

|       |                   |
|-------|-------------------|
| sem   | 信号量，详见 os_event_t |
| value | 资源值               |
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

### 24.2.3.2 libc\_sem\_wait

- 功能描述：阻塞等待资源。
- 函数原型：int libc\_sem\_wait(os\_event\_t \*sem, unsigned short timeout)
- 输入参数描述：

|         |                        |
|---------|------------------------|
| sem     | 信号量                    |
| timeout | 等待时间，10 毫秒为单位，0 表示无限等待 |
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

### 24.2.3.3 libc\_sem\_trywait

- 功能描述：尝试获得资源。
- 函数原型：int libc\_sem\_trywait(os\_event\_t \*sem)

- 输入参数描述：  
sem 信号量
- 输出参数描述：  
0 success  
-1 failed
- 说明：

#### 24.2.3.4 libc\_sem\_post

- 功能描述：释放资源。
- 函数原型：int libc\_sem\_post(os\_event\_t \*sem)
- 输入参数描述：  
sem 信号量
- 输出参数描述：  
0 success  
-1 failed
- 说明：

#### 24.2.3.5 libc\_sem\_destroy

- 功能描述：销毁资源。
- 函数原型：int libc\_sem\_destroy (os\_event\_t \*sem)
- 输入参数描述：  
sem 信号量
- 输出参数描述：  
0 success  
-1 failed
- 说明：

### 24.2.4 互斥量 API

### 24.2.4.1 libc\_pthread\_mutex\_init

- 功能描述：互斥量初始化。
- 函数原型：int libc\_pthread\_mutex\_init(os\_event\_t \* mutex)
- 输入参数描述：

|       |     |
|-------|-----|
| mutex | 互斥量 |
|-------|-----|
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

### 24.2.4.2 libc\_pthread\_mutex\_lock

- 功能描述：互斥量加锁。
- 函数原型：int libc\_pthread\_mutex\_lock(os\_event\_t \* mutex)
- 输入参数描述：

|       |     |
|-------|-----|
| mutex | 互斥量 |
|-------|-----|
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

### 24.2.4.3 libc\_pthread\_mutex\_unlock

- 功能描述：互斥量解锁。
- 函数原型：int libc\_pthread\_mutex\_unlock(os\_event\_t \* mutex)
- 输入参数描述：

|       |     |
|-------|-----|
| mutex | 互斥量 |
|-------|-----|
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

#### 24.2.4.4 libc\_pthread\_mutex\_trylock

- 功能描述：不阻塞地尝试互斥量加锁。
- 函数原型：int libc\_pthread\_mutex\_trylock(os\_event\_t \* mutex)
- 输入参数描述：

|       |     |
|-------|-----|
| mutex | 互斥量 |
|-------|-----|
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

#### 24.2.4.5 libc\_pthread\_mutex\_destroy

- 功能描述：销毁互斥量。
- 函数原型：int libc\_pthread\_mutex\_destroy (os\_event\_t \* mutex)
- 输入参数描述：

|       |     |
|-------|-----|
| mutex | 互斥量 |
|-------|-----|
- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |
- 说明：

### 24.2.5 条件变量 API

#### 24.2.5.1 libc\_pthread\_cond\_init

- 功能描述：条件变量初始化。
- 函数原型：int libc\_pthread\_cond\_init(os\_event\_t \* cond , uint32 init\_value)
- 输入参数描述：

|            |        |
|------------|--------|
| cond       | 条件量    |
| init_value | 变量初始化值 |
- 输出参数描述：

|   |         |
|---|---------|
| 0 | success |
|---|---------|



-1            failed

- 说明：

### 24.2.5.2 libc\_pthread\_cond\_wait

- 功能描述：等待条件变量，可以定时，也可以无限时等待。
- 函数原型：int libc\_pthread\_cond\_wait(os\_event\_t \*cond, os\_event\_t \*mutex, unsigned short timeout)
- 输入参数描述：

|         |                        |
|---------|------------------------|
| cond    | 条件量                    |
| mutex   | 互斥量                    |
| timeout | 等待时间，10 毫秒为单位，0 表示无限等待 |

- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |

- 说明：

### 24.2.5.3 libc\_pthread\_cond\_signal

- 功能描述：等待条件变量，可以定时，也可以无限时等待。
- 函数原型：int libc\_pthread\_cond\_signal (os\_event\_t \*cond)
- 输入参数描述：

|      |     |
|------|-----|
| cond | 条件量 |
|------|-----|

- 输出参数描述：

|    |         |
|----|---------|
| 0  | success |
| -1 | failed  |

- 说明：

### 24.2.5.4 libc\_pthread\_cond\_destroy

- 功能描述：注销条件变量。
- 函数原型：int libc\_pthread\_cond\_destroy (os\_event\_t \*cond)
- 输入参数描述：

|      |     |
|------|-----|
| cond | 条件量 |
|------|-----|

- 输出参数描述：

|   |         |
|---|---------|
| 0 | success |
|---|---------|

-1 failed

- 说明：

## 24.2.6 字符串操作 API

### 24.2.6.1 libc\_memcpy

- 功能描述：内存拷贝。
- 函数原型：void libc\_memcpy(void \*dst, void \*src, uint32 byte\_size)
- 输入参数描述：

|           |      |
|-----------|------|
| dst       | 目标地址 |
| src       | 源地址  |
| byte_size | 字节长度 |
- 输出参数描述：无
- 说明：

### 24.2.6.2 libc\_memset

- 功能描述：内存赋值。
- 函数原型：void libc\_memset(void \*dst, int8 value, uint32 byte\_size)
- 输入参数描述：

|           |      |
|-----------|------|
| dst       | 目标地址 |
| value     | 初始值  |
| byte_size | 字节长度 |
- 输出参数描述：无
- 说明：

### 24.2.6.3 libc\_memcmp

- 功能描述：内存比较。
- 函数原型：int libc\_memcmp(const void \*cs, const void \*ct, unsigned int count)
- 输入参数描述：

|           |      |
|-----------|------|
| cs        | 目标地址 |
| ct        | 源地址  |
| byte_size | 字节长度 |

- 输出参数描述:

|    |       |
|----|-------|
| 0  | 相等    |
| <0 | cs<ct |
| >0 | cs>ct |

- 说明:

#### 24.2.6.4 libc\_strlen

- 功能描述: 获取字符串长度, 不包括末尾的结束符。

- 函数原型: `int libc_strlen (const char * s)`

- 输入参数描述:

|   |         |
|---|---------|
| s | 字符串起始地址 |
|---|---------|

- 输出参数描述: 返回字符串长度

- 说明:

#### 24.2.6.5 libc\_strncat

- 功能描述: s2 指向的字符串中的字符复制到 s1 所指向的字符串后面, 最多可以复制 n 个字符。返回值是和 s1 相同的值, 即结果字符串。

- 函数原型: `char *libc_strncat (char *s1, const char *s2, unsigned int n)`

- 输入参数描述:

|    |         |
|----|---------|
| s1 | 目标字符串指针 |
| s2 | 源字符串指针  |
| n  | 字节长度    |

- 输出参数描述: 返回目标字符串指针

- 说明:

#### 24.2.6.6 libc\_strncmp

- 功能描述: 比较 s1 和 s2 所指向字符串的前 n 个字符, 结束符后面的字符会被忽略。

- 函数原型: `int libc_strncmp(const char * cs, const char * ct, unsigned int count)`

- 输入参数描述：

|       |         |
|-------|---------|
| cs    | 目标字符串指针 |
| ct    | 源字符串指针  |
| count | 比较字节长度  |
- 输出参数描述：

|    |       |
|----|-------|
| 0  | 相等    |
| <0 | cs<ct |
| >0 | cs>ct |
- 说明：

### 24.2.6.7 libc\_strncpy

- 功能描述：从 s2 中复制最多 n 个字符到 s1 字符串中。返回值和第一个自变量的值相同。其中 s1 和 s2 所指向的内存区域不能有重叠的地方。
- 函数原型：char \*libc\_strncpy (char \*s1, const char \*s2, unsigned int n)
- 输入参数描述：

|    |         |
|----|---------|
| s1 | 目标字符串指针 |
| s2 | 源字符串指针  |
| n  | 字节长度    |
- 输出参数描述：返回目标字符串指针
- 说明：

### 24.2.6.8 libc\_strlenuni

- 功能描述：获取 unicode 字符串的长度。
- 函数原型：int libc\_strlenuni (int8 \*s1)
- 输入参数描述：

|    |       |
|----|-------|
| s1 | 字符串指针 |
|----|-------|
- 输出参数描述：返回目标字符串长度
- 说明：

### 24.2.6.9 libc\_strncpyuni

- 功能描述：从 s2 中复制最多 n 个 unicode 字符到 s1 字符串中。返回值和第一个自变量的值相同。其中 s1 和 s2 所指向的内存区域不能有重叠的地方。

- 函数原型: `char *libc_strncpyuni (int8 *s1, int8 *s2, unsigned int n)`
- 输入参数描述:

|                 |         |
|-----------------|---------|
| <code>s1</code> | 目标字符串指针 |
| <code>s2</code> | 源字符串指针  |
| <code>n</code>  | 字节长度    |
- 输出参数描述: 返回目标字符串指针
- 说明:

### 24.2.6.10 libc\_itoa

- 功能描述: 数值转字符。
- 函数原型: `void libc_itoa(uint32 num, uint8 *str, uint8 counts)`
- 输入参数描述:

|                     |         |
|---------------------|---------|
| <code>num</code>    | 数值      |
| <code>str</code>    | 存放结果内存  |
| <code>counts</code> | 转换的字节长度 |
- 输出参数描述: 返回目标字符串指针
- 说明:

## 24.2.7 Uart print API

### 24.2.7.1 libc\_print

- 功能描述: 打印字符串, 数值。
- 函数原型: `void libc_print(unsigned char* s, unsigned int Data, unsigned char mode)`
- 输入参数描述:

|                   |                                                   |
|-------------------|---------------------------------------------------|
| <code>s</code>    | 字符串地址                                             |
| <code>Data</code> | 数值                                                |
| <code>mode</code> | 打印模式,<br>0, 为仅打印字符串;<br>1, 仅打印数值;<br>2, 字符串和数值都打印 |
- 输出参数描述: 无

- 说明：最多打印 16 个字符。

## 24.3 文件系统 API

文件系统为上层应用访问 FLASH，CARD 等存储设备提供接口，应用能在 CARD 或者 FLASH 上创建文件、读写文件并且可以在 CARD 和 FLASH 之间执行拷贝数据。

表格 3-2-1： 文件系统接口一览表

| 函数类别   | 函数列表                | 功能              |
|--------|---------------------|-----------------|
| 目录操作接口 | vfs_cd              | 切换目录            |
|        | vfs_dir             | 查找目录或文件         |
|        | vfs_make_dir        | 创建目录            |
|        | vfs_file_open       | 打开文件            |
|        | vfs_file_create     | 创建文件            |
|        | vfs_file_close      | 关闭文件            |
| 文件操作接口 | vfs_file_get_size   | 获取文件大小          |
|        | vfs_get_time        | 获取文件时间信息        |
|        | vfs_file_seek       | 文件定位            |
|        | vfs_file_tell       | 获取文件位置          |
|        | vfs_file_read       | 读文件             |
|        | vfs_file_write      | 写文件             |
|        | vfs_set_time        | 设置文件时间信息        |
|        | vfs_file_attralter  | 设置文件目录属性        |
|        | vfs_file_dir_offset | 记录或设置文件位置信息     |
|        | vfs_file_dir_remove | 删除文件或目录（目录必须为空） |
| 公共操作接口 | vfs_get_err_info    | 获取错误信息          |
|        | vfs_file_dir_exist  | 判断文件目录是否存在      |
|        | vfs_file_rename     | 重命名文件目录         |
|        | vfs_get_name        | 获取文件/目录名        |
|        | vfs_get_space       | 获取磁盘空间          |
|        | vfs_create_volume   | 创建磁盘卷标          |

需要说明的是：fat,fat32 文件/目录名少于 11 个字符的时候为 8.3 格式即 11 个 byte,大写,不足补 20h,超过 11 个字符，则需要转换为 unicode 编码，以 0xFFFE 开头，即长名，exfat 均为 unicode 编码，没有长名短名之分。

## 24.3.1 目录操作接口

### 24.3.1.1 vfs\_cd

- 功能描述：根据用户输入将当前目录指向当前目录的子目录，父目录或直接返回根目录。
- 函数原型：bool vfs\_cd(vfs\_mount\_t \*p\_vfs\_mount, uint8 mode, uint8\* ptr\_input\_name)
- 输入参数描述：

|                |                                                               |
|----------------|---------------------------------------------------------------|
| p_vfs_mount    | mount 返回的文件系统索引                                               |
| mode           |                                                               |
| CD_UP          | 改变当前目录到上一级父目录，目录项指针指向目录起始位置；                                  |
| CD_BACK        | 改变当前目录到上一级父目录，目录项指针指向之前 CD 进去的子目录；                            |
| CD_ROOT        | 改变当前目录到根目录；                                                   |
| CD_SUB         | 改变当前目录到当前目录项对应的子目录，此时参数 c 起作用。                                |
| ptr_input_name | 子目录名的 buffer 指针；为 NULL 的时候默认为进入当前目录。前两个字符为"0xfffe"则表示通过长名切换目录 |

- 输出参数描述：

1      success  
0      failed

- 说明：

### 24.3.1.2 vfs\_dir

- 功能描述：在当前目录按各种方式检索目录或文件
- 函数原型：uint32 vfs\_dir(vfs\_mount\_t \*p\_vfs\_mount, uint8 mode, uint8\* ptr\_input\_name, uint32 ext\_name\_bitmap)
- 输入参数描述：

|                |                                                       |
|----------------|-------------------------------------------------------|
| p_vfs_mount    | mount 返回的文件系统索引                                       |
| mode           | 各种 dir 的模式。                                           |
| DIR_NEXT       | 向后 dir；                                               |
| DIR_PREV       | 向前 dir；                                               |
| DIR_HEAD       | 从目录首向后 dir；                                           |
| DIR_TAIL       | 从目录尾向前 dir；                                           |
| ptr_input_name | 用来存放输入文件名的 buffer 指针。如指针为 NULL，则直接从当前目录项开始 dir 或不起作用。 |

`ext_name_bitmap` 如有效内存地址，则表示输入扩展名字符串的地址；如不是则表示要 `dir` 的 `bitmap`。此参数如最高位为 1 表示传入的内存地址，否则则是 `bitmap` 值。

- 输出参数描述：
  - >0 success – 实际的扩展名
  - 0 failed
- 说明

### 24.3.1.3 vfs\_make\_dir

- 功能描述：在当前目录下生成一个用户程序指定目录名的子目录。
- 函数原型：bool vfs\_make\_dir(vfs\_mount\_t \*p\_vfs\_mount, uint8\* ptr\_input\_name)
- 输入参数描述：
  - `p_vfs_mount` mount 返回的文件系统索引
  - `ptr_input_name` 要创建的目录名指针
- 输出参数描述：
  - 1 success
  - 0 failed
- 说明：

## 24.3.2 文件操作接口

### 24.3.2.1 vfs\_file\_open

- 功能描述：根据用户输入的文件名在当前目录中打开一个已存在的文件。
- 函数原型：uint32 vfs\_file\_open(vfs\_mount\_t \*p\_vfs\_mount, uint8\* ptr\_input\_name, uint8 mode)
- 输入参数描述：
 

|                              |                                                            |
|------------------------------|------------------------------------------------------------|
| <code>vfs_mount</code>       | mount 返回的文件系统索引                                            |
| <code>ptr_input_name</code>  | 待打开文件的文件名的输入指针；为 <code>null</code> 表示直接打开当前目录项指向的文件        |
| <code>mode</code>            | 打开方式。                                                      |
| <code>R_NORMAL_SEEK</code>   | 普通 seek 模式                                                 |
| <code>R_FAST_SEEK</code>     | 快速 seek 模式( <code>R_FAST_SEEK</code> 每次打开后需要先 seek 到文件末尾。) |
| <code>OPEN_MODIFY</code>     | 修改模式（不允许跨越文件大小写）                                           |
| <code>LARGE_FILE_SEEK</code> | 大文件 seek 模式（大文件打开可能较慢）                                     |
| <code>OPEN_RECOVER</code>    | 恢复模式（允许恢复写后没有正常 close 的文件，操作方式见说明）                         |



- 输出参数描述：  
    >0     success – 文件句柄  
    0     failed
- 说明：  
    在当前系统下只支持同时打开 4 个句柄，最多支持两个快速 seek，只支持 1 个写或修改,写和修改都以 OPEN\_MODIFY 方式打开。  
    恢复未正常关闭文件操作：  
    1、写文件过程中记下文件 size：  
    2、重现上电后以 OPEN\_RECOVER 打开文件（需保证上电到打开文件期间没有进行磁盘写相关操作）  
    3、通过写文件接口设置要恢复的文件大小 size，buffer 可以直接填 NULL；  
    调用关闭文件接口，如果返回正确，则文件恢复成功。

### 24.3.2.2 vfs\_file\_create

- 功能描述：根据用户输入的文件名创建一个文件,用户可获得当前操作文件的句柄
- 函数原型：     uint32    vfs\_file\_create(vfs\_mount\_t     \*p\_vfs\_mount,     uint8\*  
                  ptr\_input\_name,uint32 lentgh)
- 输入参数描述：  
    p\_vfs\_mount         mount 返回的文件系统索引  
    ptr\_input\_name     为将要创建的文件的文件名字符串指针。  
    lentgh             创建空文件的长度，为 0 表示创建空文件
- 输出参数描述：  
    >0     success – 文件句柄  
    0     failed
- 说明：  
    如输入长度不为 0，将先分配簇链，但文件内容不确定。

### 24.3.2.3 vfs\_file\_close

- 功能描述：关闭文件,用户输入需要操作文件的句柄
- 函数原型：bool vfs\_file\_close(vfs\_mount\_t \*p\_vfs\_mount, uint32 fhandle)
- 输入参数描述：  
    p\_vfs\_mount         mount 返回的文件系统索引  
    fhandle             文件句柄

- 输出参数描述:

- 1 success

- 0 failed

- 说明:

### 24.3.2.4 vfs\_file\_get\_size

- 功能描述: 取文件的长度, 字节为单位

- 函数原型: `bool vfs_file_get_size(vfs_mount_t *p_vfs_mount, uint32* output_length, uint8* ptr_file, uint8 mode)`

- 输入参数描述:

- `p_vfs_mount` mount 返回的文件系统索引

- `output_length` 返回的文件长度 (以字节为单位)

- `ptr_file` 待获取长度的文件句柄或文件名输入指针

- `mode` 表示参数 `ptr_file` 的意义。为 0, 表示 `ptr_file` 为文件句柄; 为 1 表示 `ptr_file` 为文件名指针, `ptr_file` 为 null 表示当前目录项指向的文件。

- 输出参数描述:

- 1 success

- 0 failed

- 说明:

- 虽然 exfat 支持超过 4GB 的文件, 但我们方案一般没有操作这么大的文件, 所以以 4 字节表示文件长度。

### 24.3.2.5 vfs\_get\_time

- 功能描述: 获取文件的创建时间或修改时间

- 函数原型: `bool vfs_get_time(vfs_mount_t *p_vfs_mount, file_time_t* ptr_output_time, uint8* ptr_input_name, uint8 type)`

- 输入参数描述:

- `p_vfs_mount` mount 返回的文件系统索引

- `ptr_output_time` 时间输出 buf 指针

- `ptr_input_name` 待获取时间的文件的文件名输入指针; 为 null 表示当前目录项指向的文件

- `type` FS\_TIME\_CREATE 获取创建时间;

FS\_TIME\_MODIFY 获取修改时间。

- 输出参数描述：
  - 1 success
  - 0 failed
- 说明：输出的时间格式见关键数据结构

### 24.3.2.6 vfs\_file\_seek

- 功能描述：

定位文件的字节偏移，实现：

  - a.根据相对文件首的偏移量，实现用户程序的顺序读和随机读。
  - b.实现从当前位置往文件首或文件尾偏移，实现用户程序的顺序读和随机读。
  - c.根据相对文件首的偏移量，实现用户程序的对指定位置数据的修改。

配合 read 支持用户程序顺序读数据，快进快退以及随机定位读数据，另外可以方便实现回写修改已经生成的文件。
- 函数原型：bool vfs\_file\_seek(vfs\_mount\_t \*p\_vfs\_mount, int32 offset, uint8 type, uint32 fhandle)
- 输入参数描述：

|             |                                           |
|-------------|-------------------------------------------|
| p_vfs_mount | mount 返回的文件系统索引                           |
| offset      | 对应 SEEK 偏移量，范围是 2GB                       |
| type        | 对应 SEEK 类型                                |
| SEEK_SET    | 从文件首往文件尾定位，offset 为正数                     |
| SEEK_CUR    | 从当前位置往文件头或尾定位；正数表示向文件尾 seek，负数表示向文件头 seek |
| SEEK_END    | 从文件尾往文件首定位，offset 为负数                     |
- 输出参数描述：
  - 1 success
  - 0 failed
- 说明：

### 24.3.2.7 vfs\_file\_tell

- 功能描述：取当前读写操作的指针，指针是指相对文件头的字节偏移量。读数据时用户调用该函数记录 AB 点，配合 seek 和 read 实现数据的 AB 读取。写数据时，支持用户程序修改已生成的文件。
- 函数原型：bool vfs\_file\_tell(vfs\_mount\_t \*p\_vfs\_mount, uint32\* ptr\_offset, uint32 fhandle)

- 输入参数描述：

|             |                    |
|-------------|--------------------|
| p_vfs_mount | mount 返回的文件系统索引    |
| ptr_offset  | 文件当前读写位置相对文件头的偏移量。 |
| fhandle     | 对应文件操作句柄           |
- 输出参数描述：

|   |         |
|---|---------|
| 1 | success |
| 0 | failed  |
- 说明：

### 24.3.2.8 vfs\_file\_read

- 功能描述：从文件的当前位置读数据
- 函数原型：uint32 vfs\_file\_read(vfs\_mount\_t \*p\_vfs\_mount, uint8\* ptr\_data\_buffer, uint32 byte\_count, uint32 fhandle)
- 输入参数描述：

|                 |                    |
|-----------------|--------------------|
| p_vfs_mount     | mount 返回的文件系统索引    |
| ptr_data_buffer | 读操作数据输出 buffer 的指针 |
| byte_count      | 读的字节数。             |
| fhandle         | 操作文件的句柄            |
- 输出参数描述：

|    |            |
|----|------------|
| >0 | 实际读到的字节数   |
| 0  | read error |
- 说明：  
读完后，文件指针为当前指针加上读的字节数

### 24.3.2.9 vfs\_file\_write

- 功能描述：在当前文件位置写入数据
- 函数原型：uint32 vfs\_file\_write(vfs\_mount\_t \*p\_vfs\_mount, uint8\* ptr\_data\_buffer, uint32 byte\_count, uint32 fhandle)
- 输入参数描述：

|                 |                 |
|-----------------|-----------------|
| p_vfs_mount     | mount 返回的文件系统索引 |
| ptr_data_buffer | 写数据 buffer 的指针  |
| byte_count      | 要写的字节数。         |

fhandle                      操作文件的句柄

- 输出参数描述:

>0                      实际写入的字节数  
0                      write error

- 说明:

写完后，文件指针为当前指针加上写的字节数，写过程中不支持 seek 操作。

当参数 ptr\_data\_buffer 为 NULL 和 byte\_count 为 0 时，会启用 block 功能，flash 会进行 merge 操作。此操作供录音使用，可减小录音杂音。

### 24.3.2.10 vfs\_set\_time

- 功能描述: 设置文件的创建时间或修改时间

- 函数原型: bool vfs\_set\_time(vfs\_mount\_t \*p\_vfs\_mount, file\_time\_t\* ptr\_input\_time, uint8\* ptr\_input\_name, uint8 type)

- 输入参数描述:

p\_vfs\_mount                      mount 返回的文件系统索引  
ptr\_input\_time                      时间输入 buf 指针;  
ptr\_input\_name                      待设置时间的文件的文件名输入指针; 为 null 表示  
                                        前目录项指向的文件  
type                      FS\_TIME\_CREATE    设置创建时间  
                                        FS\_TIME\_MODIFY    设置修改时间。

- 输出参数描述:

1                      success  
0                      failed

- 说明: 时间输入格式见关键数据结构

### 24.3.2.11 vfs\_cut\_file\_tail

- 功能描述: 关闭文件之前去掉文件的尾部的部分数据

- 函数原型: bool vfs\_cut\_file\_tail(vfs\_mount\_t \*p\_vfs\_mount, uint32 cut\_length, fhandle\_t \*ptr\_file)

- 输入参数描述:

p\_vfs\_mount                      mount 返回的文件系统索引  
cut\_length                      要丢弃的长度，以字节为单位，数值最好应该是 512 的整数倍;  
ptr\_file                      操作文件的句柄

- 输出参数描述:

1                      success

0                      failed

- 说明： 该接口必须在文件关闭之前调用。

### 24.3.2.12 vfs\_file\_divided

- 功能描述： 将两个文件分割
- 函数原型：bool vfs\_file\_divided(vfs\_mount\_t \*p\_vfs\_mount, uint8\* source\_file\_name, uint8\* new\_file\_name, uint32 divided\_length)
- 输入参数描述：

|                  |                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------|
| p_vfs_mount      | mount 返回的文件系统索引                                                                                  |
| source_file_name | 源文件，也就是被分割文件的文件名；                                                                                |
| new_file_name    | 分割产生的新文件的文件名                                                                                     |
| divided_length   | 将原文件分割的长度，如，一个文件的长度为 4096 byte, divided_length 为 1024，那么源文件会被分割为长度为 1024 的文件，而新的文件的长度就是 3072byte |
- 输出参数描述：

|   |         |
|---|---------|
| 1 | success |
| 0 | failed  |
- 说明： 由于文件系统是以簇为单位的，所以文件系统分割有误差。若分割点在一个簇的位置是在簇的 1/2 之前的话，那么该簇的所有数据将会被分割到新文件中去，若分割位置在簇的 1/2 处之后，则该簇的所有数据将会分割到源文件中。

### 24.3.2.13 vfs\_file\_insert

- 功能描述： 将一个文件的数据插入到另外一个文件
- 函数原型：bool vfs\_file\_insert(vfs\_mount\_t \*p\_vfs\_mount, fhandle\_t \*fhandle, uint8\* insert\_file\_name)
- 输入参数描述：

|                  |                                |
|------------------|--------------------------------|
| p_vfs_mount      | mount 返回的文件系统索引                |
| fhandle          | 源文件的文件句柄，插入时需要将文件的当前位置设置到要插入点； |
| insert_file_name | 要插入文件的文件名，该文件被插入源文件后目录项会被删除    |
- 输出参数描述：

|   |         |
|---|---------|
| 1 | success |
| 0 | failed  |
- 说明： 由于文件系统以簇为单位进行操作，所以插入有误差。插入文件最后一个簇未写满，剩余的部分我们会自动填零。

## 24.3.3 公共操作接口

### 24.3.3.1 vfs\_file\_attralter

- 功能描述：获取或修改文件的属性。
- 函数原型：uint8 vfs\_file\_attralter(vfs\_mount\_t \*p\_vfs\_mount, uint8 attr, uint8\* ptr\_file, uint8 mode)
- 输入参数描述：

|             |                                                                                                                               |
|-------------|-------------------------------------------------------------------------------------------------------------------------------|
| p_vfs_mount | mount 返回的文件系统索引                                                                                                               |
| attr        | ATTR_READ_ONLY (0x01) 只读文件<br>ATTR_HIDDEN (0x02) 隐藏文件<br>ATTR_SYSTEM (0x04) 系统文件<br>ATTR_ARCHIVE (0x20) 存档文件<br>为 NULL 表示获取属性 |
| ptr_file    | 待获取或修改属性的文件句柄或文件名输入指针；                                                                                                        |
| mode        | 表示参数 ptr_file 的意义。为 0，表示 ptr_file 为文件句柄；为 1，表示 ptr_file 为文件名指针，ptr_file 为 null 表示当前目录项指向的文件或目录。                               |

- 输出参数描述：  
>0 success – 返回要获取或设置的属性值  
0 failed
- 说明：

### 24.3.3.2 vfs\_file\_dir\_offset

- 功能描述：获取或设置当前目录项的位置信息
- 函数原型：bool vfs\_file\_dir\_offset(vfs\_mount\_t \*p\_vfs\_mount, pdir\_layer\_t\* ptr\_layer, pfile\_offset\_t\* ptr\_file\_offset, uint8 mode)
- 输入参数描述：

|             |                                       |
|-------------|---------------------------------------|
| p_vfs_mount | mount 返回的文件系统索引                       |
| ptr_layer   | 存储要获取或设置的目录层级信息 buffer 指针；为 NULL 表示当前 |
| ptr_file    | 存储要获取或设置的文件位置信息 buffer 指针；            |
| mode        | 操作方式：0 表示获取，1 表示设置                    |
- 输出参数描述：

|   |         |
|---|---------|
| 1 | success |
|---|---------|

0                      failed

- 说明：  
ptr\_layer 和 pfile\_offset 不能同时为 NULL。

### 24.3.3.3 vfs\_file\_dir\_remove

- 功能描述：在当前目录下删除一个用户程序指定目录或文件，删除目录时要求目录为空。
- 函数原型：bool vfs\_file\_dir\_remove(vfs\_mount\_t \*p\_vfs\_mount, uint8\* ptr\_input\_name, uint8 type)
- 输入参数描述：

|                |                                        |
|----------------|----------------------------------------|
| p_vfs_mount    | mount 返回的文件系统索引                        |
| ptr_input_name | 要删除的目录或文件名；如为 null 则直接删除当前目录项指向的目录或文件； |
| type           | 要操作的类型：0 为目录；1 为文件                     |
- 输出参数描述：

|   |         |
|---|---------|
| 1 | success |
| 0 | failed  |
- 说明：

### 24.3.3.4 vfs\_get\_err\_info

- 功能描述：获取文件系统出错信息
- 函数原型：uint8 vfs\_get\_err\_info(vfs\_mount\_t \*p\_vfs\_mount)
- 输入参数描述：

|             |                 |
|-------------|-----------------|
| p_vfs_mount | mount 返回的文件系统索引 |
|-------------|-----------------|
- 输出参数描述：

|   |                                                    |
|---|----------------------------------------------------|
| 0 | -- no error                                        |
| 1 | 磁盘读写错误                                             |
| 2 | 磁盘写保护                                              |
| 3 | 磁盘未格式化                                             |
| 4 | 文件操作超出文件边界,目录操作超出目录边界                              |
| 5 | 文件操作的目标文件,目录操作的目录项不存在                              |
| 6 | 表示文件操作时没有磁盘空间,不能写数据或者扩展目录；目录操作时没有磁盘空间,不能扩展目录,新建子目录 |
| 7 | 文件操作时根目录目录项满                                       |



8 删除目录时返回,表示删除的目录非空

- 说明:

### 24.3.3.5 vfs\_file\_dir\_exist

- 功能描述: 判断当前目录是否有指定的子目录或文件
- 函数原型: `uint32 vfs_file_dir_exist(vfs_mount_t *p_vfs_mount, uint8* ptr_input_name, uint8 type)`
- 输入参数描述:

|                             |                      |
|-----------------------------|----------------------|
| <code>p_vfs_mount</code>    | mount 返回的文件系统索引      |
| <code>ptr_input_name</code> | 要判断的文件或目录名指针         |
| <code>type</code>           | 要判断的类型: 0 为目录; 1 为文件 |
- 输出参数描述:

|                    |                |
|--------------------|----------------|
| <code>&gt;0</code> | 文件或目录存在, 返回首簇号 |
| <code>0</code>     | 文件或目录不存在       |
- 说明:

### 24.3.3.6 vfs\_file\_rename

- 功能描述: 重命名文件
- 函数原型: `bool vfs_file_rename(vfs_mount_t *p_vfs_mount, uint8* ptr_new_name, uint8* ptr_file, uint8 mode)`
- 输入参数描述:

|                           |                                                                                                                                                        |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>p_vfs_mount</code>  | mount 返回的文件系统索引                                                                                                                                        |
| <code>ptr_new_name</code> | 新文件名字符串指针                                                                                                                                              |
| <code>ptr_file</code>     | 待重命名的文件句柄或文件名指针;                                                                                                                                       |
| <code>mode</code>         | 表示参数 <code>ptr_file</code> 的意义。为 0, 表示 <code>ptr_file</code> 为文件句柄; 为 1, 表示 <code>ptr_file</code> 为文件名指针, <code>ptr_file</code> 为 null 表示当前目录项指向的文件或目录 |
- 输出参数描述:

|                |         |
|----------------|---------|
| <code>1</code> | success |
| <code>0</code> | failed  |
- 说明:

### 24.3.3.7 vfs\_get\_name

- 功能描述：取当前的文件名（优先返回长名，没有长名则返回短名）或后缀名
- 函数原型：`uint16 vfs_get_name(vfs_mount_t *p_vfs_mount, uint8* ptr_output_name, uint16 longname_length)`
- 输入参数描述：

|                              |                                        |
|------------------------------|----------------------------------------|
| <code>p_vfs_mount</code>     | mount 返回的文件系统索引                        |
| <code>ptr_output_name</code> | 用来存放输出文件名的 buffer 指针。                  |
| <code>longname_length</code> | 输入为要获取的长名字符数（包括长名标记和结束符），如为 0 表示获取后缀名。 |
- 输出参数描述：

|    |                                                                                          |
|----|------------------------------------------------------------------------------------------|
| >0 | success – 实际返回的获取到的文件名字符数。为取长名时返回实际函数输出的长名字符的个数（包括长名标记和结束符 0x0000），为短名时返回 11；获取后缀名时返回 3。 |
| 0  | failed                                                                                   |
- 说明：

文档中所有文件名和目录名如短名则指大写的 8+3 字符数组格式,占用 11 个 byte,不足补 20h; 输入的文件名长度包括长名标记和结束符共 2 个字符,所以如要获取长名的 6 个字符,则输入 `longname_length` 为 8,且输出 buffer 至少要 2\*8=16 字节。

### 24.3.3.8 vfs\_get\_space

- 功能描述：获取磁盘分区空间，根据输入参数不同选择要求返回磁盘分区总的扇区数还是剩余扇区数
- 函数原型：`bool vfs_get_space(vfs_mount_t *p_vfs_mount, uint32* ptr_sector_count, uint8 type)`
- 输入参数描述：

|                               |                                                   |
|-------------------------------|---------------------------------------------------|
| <code>p_vfs_mount</code>      | mount 返回的文件系统索引                                   |
| <code>ptr_sector_count</code> | 返回分区的扇区数                                          |
| <code>type</code>             | 0 表示调用将返回表示当前磁盘分区总空间的扇区数；<br>1 表示返回当前磁盘分区剩余空间的扇区数 |
- 输出参数描述：

|   |      |
|---|------|
| 1 | 获取成功 |
| 0 | 获取失败 |
- 说明：

### 24.3.3.9 vfs\_create\_volume

- 功能描述：创建卷标
- 函数原型：bool vfs\_create\_volume(vfs\_mount\_t \*p\_vfs\_mount, uint8\* ptr\_input\_name)
- 输入参数描述：

|                |                 |
|----------------|-----------------|
| p_vfs_mount    | mount 返回的文件系统索引 |
| ptr_input_name | 卷标名字符串指针        |
- 输出参数描述：

|   |         |
|---|---------|
| 1 | success |
| 0 | failed  |
- 说明：

fat,fat32 的卷标名为 11 个字符，大写，不足十一个字符补 20h，exfat 卷标名为 11 个 unicode 编码的字符。

### 24.3.3.10 vfs\_jump\_to\_direntry

- 功能描述：根据提供的目录信息，直接跳转到该目录的起始位置
- 函数原型：bool fat\_jump\_to\_direntry(vfs\_mount\_t \*vfs\_mount, uint32 mode, uint8\* direntry\_info,uint8\* reserve)
- 输入参数描述：

|               |                               |
|---------------|-------------------------------|
| p_vfs_mount   | mount 返回的文件系统索引               |
| mode          | 0x80000000-从尾往前检索；其他值：从头往后检索； |
| direntry_info | 需要跳转到的目录信息                    |
| reserve       | 保留，暂时无用                       |
- 输出参数描述：

|   |         |
|---|---------|
| 1 | success |
| 0 | failed  |

### 24.3.3.11 vfs\_dir\_current\_entry\_file

- 功能描述：通过后缀检索定位到文件位置
- 函数原型：uint32 fat\_dir\_current\_entry\_file(vfs\_mount\_t \*vfs\_mount, uint32 mode, uint8\* input\_name, uint32 ext\_name\_bitmap)
- 输入参数描述：

|             |                                                                                                                |
|-------------|----------------------------------------------------------------------------------------------------------------|
| p_vfs_mount | mount 返回的文件系统索引                                                                                                |
| mode        | 0x80000000-从尾往前检索；其他值：从头往后检索；如果为 0 或 0x80000000 时，表示检索定位到第一个配置后缀的文件的位置。<br>其他值:mode=n 则表示检索定位到第 N 个配置后缀的文件的位置。 |

ptr\_input\_name 存储检索定位到的文件位置信息。

ext\_name\_bitmap 如是有有效内存地址，则表示输入扩展名字符串的地址；如不是则表示要 dir 的 bitmap。此参数如最高位为 1 表示传入的内存地址，否则则是 bitmap 值，具体的 bitmap 对应的意义请参照 vfs\_interface.h 第 43 行下的定义

● 输出参数描述：

非 0 success  
0 failed

### 24.3.3.12 vfs\_file\_move

● 功能描述：删除或增加目录项

● 函数原型：bool vfs\_file\_move(vfs\_mount\_t \*vfs\_mount, void \*fat\_direntry, uint8 \*ptr\_file\_name, uint8 mode)

● 输入参数描述：

p\_vfs\_mount mount 返回的文件系统索引  
fat\_direntry 目录项信息指针  
ptr\_file\_name 待增加或删除的目录名字符串指针；当 mode==0 时，如果为 null 表示前目录项指向的文件；否则参数无效  
mode 0-删除目录项；1-增加目录项

● 输出参数描述：

1 success  
0 failed

## 24.4 AUDIO DEVICE API

AUDIO DEVICE 驱动为上层应用访问 AD/DA/PA 等模块提供接口，通过调用这些接口能实现声音输出、监听和声音采集等功能，同时增加对硬件 asrc 的配置、开关、和采样率转换等功能。

表格 5-1: AUDIO DEVICE 操作命令一览表

| Command |               | 函数功能说明    |
|---------|---------------|-----------|
| 音频输出    | ENABLE_PA     | 打开 PA     |
|         | DISABLE_PA    | 关闭 PA     |
|         | SET_PA_VOLUME | 设置 PA 的音量 |
|         | GET_PA_VOLUME | 获取 PA 的音量 |
|         | ENABLE_DAC    | 打开 DAC    |

| Command |                  | 函数功能说明                          |
|---------|------------------|---------------------------------|
|         | DISABLE_DAC      | 关闭 DAC                          |
|         | SET_DAC_RATE     | 设置 DAC 的采样率                     |
|         | GET_DAC_RATE     | 关闭 DAC 的采样率                     |
|         | ENABLE_AIN_OUT   | 打开 AA 通路开关                      |
|         | DISABLE_AIN_OUT  | 关闭 AA 通路开关                      |
| 音频输入    | ENABLE_AIN       | 打开 FM 或者 MIC 或者 LINEIN 的 AIN 输入 |
|         | DISABLE_AIN      | 关闭 FM 或者 MIC 或者 LINEIN 的 AIN 输入 |
|         | SET_AIN_GAIN     | 设置 AIN 输入的增益                    |
|         | ENABLE_ADC       | 打开 ADC                          |
|         | DISABLE_ADC      | 关闭 ADC                          |
|         | SET_ADC_RATE     | 设置 ADC 采样率                      |
| ASRC    | CONFIG_ASRC      | 配置 asrc                         |
|         | CLOSE_ASRC       | 关闭 asrc                         |
|         | SET_ASRC_RATE    | 设置 asrc 采样率转换的抽样比               |
| 数字音效    | SET_EFFECT_PARAM | 设置音效参数                          |
|         | GET_FEATURE_INFO | 获取音效处理信息                        |

## 24.4.1 音频输出

### 24.4.1.1 ENABLE\_PA

- 功能描述：打开 PA/I2S/SPDIF 等音频输出模块
- 函数原型：int32 enable\_pa(uint32 ddv\_sel, uint32 pa\_swing, uint32 aout\_type)
- 输入参数描述：
  - ddv\_sel 类型相关的参数选择、内部 pa 用于选择直驱非直驱；对于 i2s、spdif 用于选 mfp 组
  - pa\_swing 内部 pa 使用：bit6 用于选择输出峰值 bit6(1<<6)2.8pp, bit6(0<<6)1.6pp
  - aout\_type 用于选择音频输出的类型，见结构体 aout\_type\_e
- 输出参数描述：
  - 0 success
  - 1 failed
- 说明：
 

进入音乐应用只需打开一次，退出时关闭即可。对于内部 pa 来说第一次打开的时间花费的时间会多一些，或者有轻微杂音。

### 24.4.1.2 DISABLE\_PA

- 功能描述：关闭 PA/I2S/SPDIF 等音频输出模块
- 函数原型：int32 disable\_pa(uint32 aout\_type, void \*null2, void \*null3)
- 输入参数描述：  
aout\_type 用于选择音频输出的类型，见结构体 aout\_type\_e
- 输出参数描述：  
0 success  
-1 failed
- 说明：退出音乐时关闭。

### 24.4.1.3 SET\_PA\_VOLUME

- 功能描述：设置内部 pa 音量
- 函数原型：int32 set\_pa\_volume(uint32 vol\_hard, void \*null2, void \*null3)
- 输入参数描述：  
vol\_hard 设置到寄存器的音量值（0-40）
- 输出参数描述：  
0 success  
-1 failed
- 说明：

### 24.4.1.4 GET\_PA\_VOLUME

- 功能描述：获取内部 pa 音量
- 函数原型：int32 get\_pa\_volume(void \*null1, void \*null2, void \*null3)
- 输入参数描述：
- 输出参数描述：  
音量值（0-40）
- 说明：

### 24.4.1.5 ENABLE\_DAC

- 功能描述：打开 dac 模块
- 函数原型：int32 enable\_dac(uint32 src\_type, uint32 dac\_chan, void \*null3)
- 输入参数描述：  
src\_type      选择 dac fifo 的输入源，见结构体 dac\_fifo\_in\_e  
dac\_chan      需要使能的 dac 的通道，见结构体 dac\_chenel\_e
- 输出参数描述：  
0              success  
-1             failed
- 说明：

### 24.4.1.6 DISABLE\_DAC

- 功能描述：关闭 dac 模块
- 函数原型：int32 disable\_dac(uint32 dac\_chan, void \*null2, void \*null3)
- 输入参数描述：  
dac\_chan      需要使能的 dac 的通道，见结构体 dac\_chenel\_e
- 输出参数描述：  
0              success  
-1             failed
- 说明：

### 24.4.1.7 SET\_DAC\_RATE

- 功能描述：设置 dac 采样率
- 函数原型：int32 set\_dac\_rate(uint32 dac\_rate, uint32 chanel\_no, void \*null3)
- 输入参数描述：  
dac\_rate      采样率值（单位 khz）  
chanel\_no    未使用
- 输出参数描述：  
0              success  
-1             failed
- 说明：

### 24.4.1.8 GET\_DAC\_RATE

- 功能描述：设置 dac 采样率
- 函数原型：int32 get\_dac\_rate(void \*null1, void \*null2, void \*null3)
- 输入参数描述：
- 输出参数描述：  
当前 dac 采样率值（单位 khz）
- 说明：

### 24.4.1.9 ENABLE\_AIN\_OUT

- 功能描述：打开 AA 通道
- 函数原型：int32 enable\_ain\_out(uint32 out\_mode, void \*null2, void \*null3)
- 输入参数描述：  
out\_mode 当前模拟输入，见结构体 mmm\_ai\_type\_t
- 输出参数描述：  
0 success  
-1 failed
- 说明：

### 24.4.1.10 ENABLE\_AIN\_OUT

- 功能描述：关闭 AA 通道
- 函数原型：int32 disable\_ain\_out(uint32 out\_mode, void \*null2, void \*null3)
- 输入参数描述：  
out\_mode 当前模拟输入，见结构体 mmm\_ai\_type\_t
- 输出参数描述：  
0 success  
-1 failed
- 说明：



## 24.4.2 音频输入

### 24.4.2.1 ENABLE\_AIN

- 功能描述：使能模拟输入
- 函数原型：int32 enable\_ain(uint32 ain\_type, uint32 ain\_gain, void \*null3)
- 输入参数描述：
  - ain\_type 模拟输入选择，见结构体 mmm\_ai\_type\_t
  - ain\_gain 模拟输入的增益
    - 0x0: -12dB
    - 0x1: -3dB
    - 0x2: 0dB
    - 0x3: 1.5dB
    - 0x4: 3.0dB
    - 0x5: 4.5dB
    - 0x6: 6.0dB
    - 0x7: 7.5db
- 输出参数描述：
  - 0 success
  - 1 failed
- 说明：

### 24.4.2.2 DISABLE\_AIN

- 功能描述：关闭模拟输入
- 函数原型：int32 disable\_ain(uint32 ain\_type, void \*null2, void \*null3)
- 输入参数描述：
  - ain\_type 模拟输入选择，见结构体 mmm\_ai\_type\_t
- 输出参数描述：
  - 0 success
  - 1 failed
- 说明：

### 24.4.2.3 SET\_AIN\_GAIN

- 功能描述：设置模拟输入的增益
- 函数原型：int32 set\_ain\_gain(uint32 ain\_type, uint32 ain\_gain, void \*null3)
- 输入参数描述：  
ain\_type 模拟输入选择，见结构体 mmm\_ai\_type\_t  
ain\_gain 模拟输入的增益  
0x0: -12dB  
0x1: -3dB  
0x2: 0dB  
0x3: 1.5dB  
0x4: 3.0dB  
0x5: 4.5dB  
0x6: 6.0dB  
0x7: 7.5db
- 输出参数描述：  
0 success  
-1 failed
- 说明：

### 24.4.2.4 ENABLE\_ADC

- 功能描述：使能 ADC
- 函数原型：int32 enable\_adc(uint32 src\_type, uint32 adc\_gain, void \*null3)
- 输入参数描述：  
src\_type adc fifo 的输出选择，见结构 adc\_fifo\_out\_e  
adc\_gain adc 的增益  
0x0: 0dB  
0x1: 3dB  
0x2: 6dB  
0x3: 9dB  
0x4: 12dB  
0x5: 15dB  
0x6: 18dB  
0x7: 21db

0x8: 24dB

0x9: 27dB

0xa: 30dB

0xb: 33dB

0xc: 36dB

0xd: 39dB

0xe: 42dB

0xf: 45dB

- 输出参数描述:  
0            success  
-1          failed
- 说明:

### 24.4.2.5 DISABLE\_ADC

- 功能描述: 关闭 ADC
- 函数原型: `int32 disable_adc(void* null, void* null, void *null3)`
- 输入参数描述:
- 输出参数描述:  
0            success  
-1          failed
- 说明:

### 24.4.2.6 SET\_ADC\_RATE

- 功能描述: 设置 ADC 采样率
- 函数原型: `int32 set_adc_rate(uint32 adc_rate, void *null2, void *null3)`
- 输入参数描述:  
adc\_rate    adc 采样率 (单位 khz)
- 输出参数描述:  
0            success  
-1          failed
- 说明:

## 24.4.3 ASRC 配置

### 24.4.3.1 CONFIG\_ASRC

- 功能描述：配置 ASRC
- 函数原型：int32 config\_asrc(uint32 nTestSel, uint8 asrc\_mode)
- 输入参数描述：  
nTestSel asrc 的类型选择 -头文件// ram select table  
asrc\_mode asrc 参数选择
- 输出参数描述：  
0 success  
-1 failed
- 说明：

### 24.4.3.2 CLOSE\_ASRC

- 功能描述：关闭 ASRC
- 函数原型：int32 audio\_device\_close\_asrc(uint32 nTestSel)
- 输入参数描述：  
nTestSel asrc 的类型选择 -头文件// ram select table
- 输出参数描述：  
0 success  
-1 failed
- 说明：

### 24.4.3.3 SET\_ASRC\_RATE

- 功能描述：配置 ASRC 抽样比
- 函数原型：int32 set\_asrc\_rate(uint32 asrc\_rate, uint32 chanel\_no, uint32 asrc\_offset)
- 输入参数描述：  
asrc\_rate asrc 输入采样率

channel\_no asrc 通道 0—通道 out0; 1—通道 out1; 2—通道 in  
asrc\_offset 微调的偏移

- 输出参数描述:  
0 success  
-1 failed
- 说明:

## 24.4.4 数字音效

### 24.4.4.1 SET\_EFFECT\_PARAM

- 功能描述: 设置音效参数
- 函数原型: `int32 set_effect_param(uint32 set_type, void* param_ptr, void* null3)`
- 输入参数描述:  
set\_type 设置类型, 见结构体 `set_effect_type_e`  
param\_ptr 参数指针, 与 set\_type 相关
- 输出参数描述:  
0 success  
-1 failed
- 说明:

### 24.4.4.2 GET\_FEATURE\_INFO

- 功能描述: 获取音效处理信息
- 函数原型: `int32 get_feature_info(uint32 set_type, void* info_ptr, void* null3)`
- 输入参数描述:  
set\_type 设置类型, 见结构体 `set_effect_type_e`  
param\_ptr 参数指针, 与 set\_type 相关
- 输出参数描述:  
0 success  
-1 failed

- 说明：