US212A 方案——

应用程序设计指南

最新版本：1.0

2013-5-27

1 声 明

Circuit diagrams and other information relating to products of Actions Semiconductor Company, Ltd. (“Actions”) are included as a means of illustrating typical applications. Consequently, complete information sufficient for construction is not necessarily given. Although the information has been examined and is believed to be accurate, Actions makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and disclaims any responsibility for inaccuracies. Information in this document is provided solely to enable use of Actions’ products. The information presented in this document does not form part of any quotation or contract of sale. Actions assumes no liability whatsoever, including infringement of any patent or copyright, for sale and use of Actions’ products, except as expressed in Actions’ Terms and Conditions of Sale for. All sales of any Actions products are conditional on your agreement of the terms and conditions of recently dated version of Actions’ Terms and Conditions of Sale agreement Dated before the date of your order.

This information does not convey to the purchaser of the described semiconductor devices any licenses under any patent rights, copyright, trademark rights, rights in trade secrets and/or know how, or any other intellectual property rights of Actions or others, however denominated, whether by express or implied representation, by estoppel, or otherwise.

Information Documented here relates solely to Actions products described herein supersedes, as of the release date of this publication, 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 right to make changes to specifications and product descriptions at any time without notice. Contact your Actions sales representative to obtain the latest specifications before placing your product order. Actions product may contain design defects or errors known as anomalies or errata which may cause the products functions to deviate from published specifications. Anomaly or “errata” sheets relating to currently characterized anomalies or errata are available upon request. Designers must not rely on the absence or characteristics of any features or 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’ 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

版 权 所 有 侵 权 必 究

customer.

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. The word "ACTIONS", the Actions' LOGO, whether used separately and/or in combination, is trademark of Actions Semiconductor Company, Ltd., Names and brands of other companies and their products that may from time to time descriptively appear in this product data sheet are the trademarks of their respective holders; no affiliation, authorization, or endorsement by such persons is claimed or implied except as may be expressly stated therein.

ACTIONS DISCLAIMS AND EXCLUDES ANY AND ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY AND ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND AGAINST INFRINGEMENT AND THE LIKE, AND ANY AND ALL WARRANTIES ARISING FROM ANY COURSE OF DEALING OR USAGE OF TRADE.

IN NO EVENT SHALL ACTIONS BE RELIABLE FOR ANY DIRECT, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES; OR FOR LOST DATA, PROFITS, SAVINGS OR REVENUES OF ANY KIND; REGARDLESS OF THE FORM OF ACTION, WHETHER BASED ON CONTRACT; TORT; NEGLIGENCE OF ACTIONS OR OTHERS; STRICT LIABILITY; BREACH OF WARRANTY; 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.

支持：

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

2 目 录

1 声 明 2
2 目 录 4
3 引言 24
3.1 版本历史 24
3.2 编写目的 24
3.3 术语和缩写词 24
3.4 用户手册导读 25
4 开发环境 28
4.1 开发环境的搭建 28
4.1.1 Cygwin 的安装 28
4.1.1.1 全新安装 cygwin 28
4.1.1.2 卸载 cygwin 31
4.1.2 SDE 工具链的安装 32
4.1.3 Firmware Develop Kit 的安装 32
4.1.4 UI Editor 工具和菜单设计工具 32
4.1.5 量产工具的安装 33
4.1.6 如何编译 AP、生成固件并烧录到小机运行 33
4.2 UI 模拟器 34
4.2.1 UI 模拟器的作用 34
4.2.2 UI 模拟器工程简介 35
4.2.3 UI 模拟器的开发环境和使用方法 35
4.2.3.1 创建工程 35
4.2.3.2 设置工程 38
4.2.3.3 添加目录和文件 42
4.2.3.4 编译工程 47
4.2.4 如何为新建的 AP 搭建模拟器的开发环境 48
4.2.5 如何用模拟器进行 Debug 48

版 权 所 有 侵 权 必 究

4.2.5.1 调试前的设置 48
4.2.5.2 启动调试（F5） 52
4.2.5.3 设置断点（F9） 53
4.2.5.4 单步调试（F10，查看变量和函数调用关系） 54
4.2.5.5 进入函数体调试（F11） 55
4.3 调试方法介绍 57
4.3.1.1 UDI 和 Insigh 57
4.3.1.2 串口打印 60
4.3.1.3 UI 模拟器 61
5 case 基本结构及开发 61
5.1 Case 运行环境 62
5.1.1 PSP 模块概述 62
5.1.1.1 基本概念 62
5.1.1.2 PSP 接口说明书 63
5.1.1.3 任务调度机制 64
5.1.1.4 BANK 机制 65
5.1.1.5 API 机制 67
5.1.1.6 VFS 机制 68
5.1.2 解码编码概述 70
5.1.2.1 解码编码总述 70
5.1.2.2 装载与卸载 70
5.1.2.3 解码编码接口说明书 71
5.1.3 enhanced 模块概述 71
5.1.3.1 enhanced 模块总述 71
5.1.3.2 Enhanced 接口说明书 72
5.1.4 模块调用关系 72
5.2 代码目录导航 74
5.2.1 case 目录导航 74
5.2.2 case_simulator 目录导航 79
5.2.3 psp_rel 目录导航 80

版 权 所 有 侵 权 必 究

5.3 应用管理器 ap_manager........81
5.3.1 ap_manager 的地位与作用........81
5.3.2 Ap_manager 的设计要点........81
5.3.3 Ap_manager 入口模块........82
5.3.4 Ap_manager 消息循环模块........83
5.3.5 Ap_manager 的空间分配........85
5.4 开机与关机 ap_config........85
5.4.1 ap_config 的地位与作用........86
5.4.2 ap_config 的设计要点........86
5.4.3 ap_config 的开机流程........86
5.4.4 ap_config 的关机流程........87
5.5 common 设计与使用........88
5.5.1 AppLib 设计与使用........89
5.5.1.1 AppLib 的功能概述........89
5.5.1.2 应用程序（进程）管理........89
5.5.1.3 消息通信管理........90
5.5.1.4 应用级定时器管理........93
5.5.2 Common_func 设计与使用........95
5.5.2.1 Common_func 的功能概述........95
5.5.2.2 字符串处理........95
5.5.2.3 路径记忆........96
5.5.2.4 配置项解释........98
5.5.2.5 可配置化菜单解释........99
5.5.2.6 声音输出管理........99
5.5.2.7 按键映射处理........100
5.5.2.8 按键消息预处理........101
5.5.3 Common_ui 设计与使用........102
5.5.3.1 菜单列表控件........103
5.5.3.2 文件浏览控件........110
5.5.3.3 删除文件控件........113
5.5.3.4 对话框控件........114

版 权 所 有 侵 权 必 究

5.5.3.5 USB 连接对话框 118
5.5.3.6 参数设置框控件 119
5.5.3.7 音量条控件 122
5.5.3.8 文本阅读框控件 122
5.5.3.9 状态栏控件 125
5.5.3.10 动画显示控件 127
5.5.3.11 按键锁控件 128
5.5.3.12 屏幕保护 129
5.5.3.13 关机对话框 129
5.5.4 Common_misc 设计与使用 130
5.5.4.1 系统定时器 131
5.5.4.2 应用睡眠 131
5.5.4.3 默认消息处理 132
5.5.4.4 应用私有消息预处理 133
5.5.5 Common 的空间分配 134
5.5.5.1 数据空间 134
5.5.5.2 代码空间 134
5.5.5.3 空间分配说明 135
5.5.5.4 链接脚本模板 136
5.6 前台应用设计与开发 136
5.6.1 前台应用的组成结构 136
5.6.1.1 应用组成部分 136
5.6.1.2 应用基本架构 137
5.6.1.3 应用主体介绍 138
5.6.1.4 如何使用 Common 139
5.6.1.5 应用映像文件结构 139
5.6.2 前台应用的内存空间 141
5.6.2.1 常驻代码空间 142

版 权 所 有 侵 权 必 究

5.6.2.2 常驻数据空间 142

5.6.2.3 BANK 代码及 BANK 数据空间 142

5.6.2.4 运行栈空间 142

5.6.2.5 堆空间 143

5.6.2.6 VRAM 空间 143

5.6.3 应用程序的启动 143

5.6.4 应用程序的退出 148

5.6.5 前台应用的开发指南 152

5.6.5.1 前台 AP 开发流程 152

5.6.5.2 UI 显示开发详解 153

5.6.5.3 消息通信开发详解 155

5.6.5.4 应用级定时器开发详解 159

5.6.5.5 可配置化菜单开发详解 159

5.6.6 应用程序 makefile 与 xn 脚本编写 159

5.6.6.1 应用程序 makefile 脚本 159

5.6.6.2 应用程序 xn 脚本 160

5.6.7 如何增加一个前台应用 164

5.6.7.1 user1 演示目录说明 164

5.6.7.2 user1 应用概要设计 165

5.6.7.3 步骤 1：搭建应用基本架构 166

5.6.7.4 步骤 2：开发播放场景 168

5.6.7.5 步骤 3：添加新 AP 到 case 中 173

5.6.7.6 步骤 4：在 UI simulator 上调试播放场景 174

5.6.7.7 步骤 5：开发菜单场景 177

5.6.7.8 步骤 6：在 UI simulator 上调试菜单场景 178

5.6.7.9 步骤 7：上板调试 user1 AP 178

5.7 后台应用设计与开发 178

5.7.1 后台应用的组成结构 179

5.7.1.1 应用组成部分 179

5.7.1.2 应用主体介绍 179
5.7.2 后台应用的内存空间 180
5.7.2.1 常驻代码空间 180
5.7.2.2 常驻数据空间 180
5.7.2.3 BANK 代码及 BANK 数据空间 180
5.7.2.4 运行栈空间 180
5.7.2.5 堆空间 181
5.7.2.6 VRAM 空间 181
5.7.3 引擎的互斥处理流程 181
5.7.4 如何增加一个后台应用 182
5.7.4.1 user1_engine 演示目录说明 182
5.7.4.2 user1_engine 应用概要设计 182
5.7.4.3 步骤 1：搭建应用基本架构 182
5.7.4.4 步骤 2：在 case 添加 user1_engine AP 183
5.7.4.5 步骤 3：在 UI simulator 上调试消息通信等 183
5.7.4.6 步骤 4：上板调试 183
5.8 多线程设计与开发 184
5.8.1 多线程架构 184
5.8.1.1 创建子线程 184
5.8.1.2 销毁子线程 184
5.8.1.3 典型多线程场景 184
5.8.2 多线程开发 185
5.9 驱动程序设计与开发 185
5.9.1 驱动程序的组成结构 186
5.9.1.1 驱动程序工程 186
5.9.1.2 驱动程序映像文件 186
5.9.2 驱动程序的内存空间 188
5.9.3 驱动程序的启动和退出 189
5.9.4 驱动程序接口详解 190
5.9.4.1 驱动统一入口 190

版 权 所 有 侵 权 必 究

5.9.4.2 驱动接口表 ... 191
5.9.4.3 内部接口声明和定义 ... 192
5.9.4.4 外部接口命令号和宏定义 ... 192
5.9.4.5 外部接口调用流程 ... 193
5.9.4.6 修改驱动外部接口 ... 194
5.9.5 驱动 makefile 与 xn 脚本编写 ... 194
5.9.5.1 驱动 makefile 脚本 ... 194
5.9.5.2 驱动 xn 脚本 ... 195
5.9.6 如何增加一个驱动程序 ... 198
5.10 内存空间分配调整指南 ... 198
5.10.1 Case 内存分配图 ... 198
5.10.2 分配说明 ... 200
5.10.3 调整指南 ... 200
6 case 驱动程序详解 ... 201
6.1 KEY 驱动设计 ... 202
6.1.1 需求概述及设计原则 ... 202
6.1.2 KEYBOARD 原理 ... 202
6.1.2.1 ADC 按键的原理 ... 202
6.1.2.2 GPIO 按键的原理 ... 203
6.1.2.3 IR 按键的设计原理 ... 203
6.1.3 KEY 驱动功能模块 ... 203
6.1.3.1 按键扫描及消息发送 ... 204
6.1.3.2 充电控制 ... 204
6.1.3.3 KEY 驱动初始化及卸载 ... 205
6.1.3.4 获取按键映射表地址 ... 205
6.1.4 KEY 驱动的外部接口设计 ... 206
6.1.5 KEY 驱动的内存分配说明 ... 207
6.1.6 KEY 驱动修改指南 ... 207
6.1.6.1 修改物理按键及按键映射表 ... 207
6.1.6.2 修改按键生命周期 ... 208
6.2 LCD 驱动设计 ... 208

版 权 所 有 侵 权 必 究

6.2.1 需求概述及设计原则......208
6.2.2 LCD 驱动功能模块......209
6.2.2.1 LCD 模组功能......209
6.2.2.2 LCD 硬件初始化......210
6.2.2.3 LCD 基本硬件功能......210
6.2.3 LCD 驱动的外部接口设计......211
6.2.4 LCD 驱动的内存分配说明......214
6.2.5 LCD 驱动的修改指南......215
6.2.5.1 替换新的 LCD 屏......215
6.2.5.2 LCD 相关 GPIO 修改......215
6.2.5.3 如何修改屏尺寸......216
6.2.6 LCD 驱动的功能配置......217
6.3 UI 驱动设计......218
6.3.1 需求概述及设计原则......218
6.3.2 显示系统的优化......218
6.3.3 UI 驱动功能模块......220
6.3.3.1 总体架构......220
6.3.3.2 功能模块......221
6.3.3.3 Style 文件打开与关闭......223
6.3.3.4 控件显示及获取属性......223
6.3.3.5 设置语言类型......224
6.3.3.6 字符编码转换......224
6.3.3.7 图形处理......225
6.3.3.8 图片显示......225
6.3.3.9 字符串显示......226
6.3.4 UI 驱动的外部接口设计......227
6.3.5 关键的数据结构......233
6.3.5.1 显示 PicBox 的私有数据......234
6.3.5.2 显示 TextBox 的私有数据......234
6.3.5.3 显示 TimeBox 的私有数据......235

版权所有 侵权必究

6.3.5.4 显示 NumBox 的私有数据 237
6.3.5.5 显示 ProgressBar 的私有数据 238
6.3.5.6 显示 ListBox 的私有数据 239
6.3.5.7 显示 DialogBox 的私有数据 240
6.3.5.8 显示 ParamBox 的私有数据 241
6.3.5.9 显示字符串数据结构 243
6.3.6 控件显示流程 244
6.3.6.1 PicBox 的显示流程 244
6.3.6.2 ListBox 的显示流程 244
6.3.6.3 TextBox 的显示流程 244
6.3.6.4 SliderBar 的显示流程 244
6.3.6.5 ProgressBar 的显示流程 244
6.3.6.6 TimeBox 的显示流程 244
6.3.6.7 DialogBox 的显示流程 244
6.3.6.8 NumBox 的显示流程 244
6.3.7 字符串显示 244
6.3.7.1 设计概述 244
6.3.7.2 显示 BUFFER 的选择 245
6.3.7.3 字符串显示处理的设计 246
6.3.7.4 字符串向上滚屏的实现 246
6.3.7.5 泰语特殊处理 247
6.3.7.6 阿拉伯特殊处理 247
6.3.8 UI 驱动的内存分配说明 249
6.3.9 多种语言支持 249
6.3.9.1 字符编码与字库 250
6.3.9.2 需求与设计要点 252
6.3.9.3 资源字符串 253
6.3.9.4 显示流程 253

版 权 所 有 侵 权 必 究

6.3.9.5 添加语言 254
6.3.10 多字库/字体支持 255
6.3.10.1 字库格式 255
6.3.10.2 字库配置说明 255
6.4 黑白屏 **UI** 驱动设计 **257**
6.4.1 需求概述及设计原则 257
6.4.2 修改指南 258
6.5 **Welcome** 驱动设计 **260**
6.5.1 Welcome 的定义和作用 260
6.5.2 Welcome 的硬件接口 260
6.5.3 Welcome 的启动和业务流程 261
6.5.4 如何修改 Welcome 的界面 262
7 界面设计与开发 264
7.1 引进可配置化方法 **264**
7.2 可配置化 **UI** **264**
7.2.1 UI Editor 概述 265
7.2.1.1 基本概念 265
7.2.1.2 控件基本组成 265
7.2.1.3 UI Editor 工具布局 269
7.2.1.4 工具菜单说明 270
7.2.1.5 基本操作 270
7.2.1.6 工程与风格 271
7.2.1.7 子场景模板 272
7.2.2 UI Editor 工作步骤 272
7.2.2.1 步骤 1：新建 AP 工程 273
7.2.2.2 步骤 2：设置资源 274
7.2.2.3 步骤 3：编辑场景 UI 276
7.2.2.4 步骤 4：生成结果 277
7.2.2.5 步骤 5：调试 UI 277
7.2.3 UI Editor 控件配置 277
7.2.3.1 PicBox 控件 277

版 权 所 有 侵 权 必 究

7.2.3.2 TextBox 控件 ... 280

7.2.3.3 NumBox 控件 ... 283

7.2.3.4 TimeBox 控件 ... 287

7.2.3.5 ProgressBar 控件 ... 290

7.2.3.6 ListBox 控件 ... 294

7.2.3.7 ParamBox 控件 ... 295

7.2.3.8 SliderBar 控件 ... 295

7.2.3.9 DialogBox 控件 ... 295

7.2.3.10 AttributeBox 控件 ... 295

7.2.4 Common 工程说明 ... 296

7.3 可配置化菜单 ... 296

7.3.1 需求概述 ... 296

7.3.2 设计与实现要点 ... 297

7.3.2.1 菜单资源项 ... 297

7.3.2.2 mcg 文件格式 ... 299

7.3.3 可配置化菜单开发流程 ... 301

7.3.3.1 步骤 1：生成 *.sty 和 *_res.h ... 302

7.3.3.2 步骤 2：构造菜单资源项 ... 302

7.3.3.3 步骤 3：Make 并打包为*.ap ... 302

7.3.3.4 步骤 4：配置菜单树并生成 *.mcg ... 302

7.3.3.5 步骤 5：调试菜单 ... 305

7.3.4 可配置化菜单修改指南 ... 307

7.3.4.1 增加/删除入口菜单 ... 307

7.3.4.2 增加/删除菜单项 ... 307

7.3.4.3 使用 Firmware Develop Kits 工具 ... 307

8 ap_music 应用 ... 309

8.1 需求概述 ... 310

8.2 总体架构设计 ... 311

8.3 Music UI 应用的设计 ... 311

版 权 所 有 侵 权 必 究

8.3.1 Music UI 应用的功能模块划分 311
8.3.2 与其他模块的同步和交互......313
8.3.3 ap_music 应用依赖库及其接口说明 314
8.4 ap_music 应用的业务流程......314
8.4.1 ap_music 应用的总流程 314
8.4.2 初始化模块的流程 315
8.4.3 场景调度总流程图 317
8.4.4 列表场景流程图 320
8.4.4.1 menu 控件部分 322
8.4.4.2 列表控件部分 324
8.4.4.3 专辑图片控件 325
8.4.5 播放场景流程图 326
8.4.6 设置场景流程图 331
8.4.7 书签模块处理流程 333
8.4.8 应用退出模块流程图......338
8.5 播放一个音频文件的接口调用系列和顺序说明......**338**
8.6 如何减少一种音频格式的支持 **345**
8.7 歌词的解析和显示是如何实现的......**345**
8.8 music 是如何获取和显示专辑图片的 **347**
8.9 music 应用支持哪些类型文件的 **ID3** 显示......**349**
8.10 小机开机如何实现 **music** 断点播放 **350**
8.11 music 如何选择某个专辑，某个歌手的所有专辑播放 **352**
8.12 如何定位专辑，歌手等播放列表......**352**
8.13 music 遇到格式不支持文件是如何处理的 **353**
8.14 music 前后台读写 **VRAM** 有何注意事项......**354**
8.15 music 正在播放时长按 **play** 键关机的流程需要注意什么 **355**
8.16 播歌过程中 **VRAM** 读写有何注意事项 **355**
8.17 audible 如何实现专辑图片和章节的切换......**356**
8.18 audible 如何实现断点续播功能 **357**
8.19 audible 的 **pos** 文件是如何保存和读取的......**358**
8.20 audible 是如何读取和保存激活文件的 **358**

版 权 所 有 侵 权 必 究

8.21 如何在列表控件嵌套菜单项......358
8.22 列表控件的嵌套菜单激活项是怎么计算的......359
8.23 列表和菜单控件是如何实现路径记忆的......359
8.24 music 函数在头文件声明时为什么要加__FAR__......360
9 music_engine 引擎......362
9.1 需求概述......362
9.2 总体设计......362
9.3 音乐引擎的业务流程......363
9.3.1 音乐引擎的总体流程......363
9.3.2 音乐引擎的状态处理流程......364
9.3.3 音乐引擎的消息处理流程......367
9.4 与其他模块的同步和交互......368
9.5 应用依赖库及其接口说明......370
9.6 如何增加一条引擎消息......370
10 ap_record 应用......371
10.1 需求概述......371
10.2 总体架构设计......371
10.2.1 总体架构图......371
10.2.2 功能模块划分......372
10.3 与其它应用的同步和交互......372
10.4 应用依赖库及其接口......372
10.5 应用的业务流程......374
10.5.1 应用的总流程和场景调度流程......374
10.5.2 主菜单场景流程图......374
10.5.3 录音场景流程图......375
10.6 如何实现录音监听功能......375
10.7 录音命令的调用系列和顺序说明......377
11 ap_picture 应用......378
11.1 需求概述......378
11.2 总体架构设计......378
11.2.1 总体架构图......378

11.2.2 功能模块划分 379
11.3 与其它应用的同步和交互 **379**
11.4 应用依赖库及其接口 **379**
11.5 应用的业务流程 **379**
11.5.1 应用的总流程和场景调度流程 380
11.5.2 文件列表场景流程图 381
11.5.3 图片预览场景流程图 382
11.5.4 图片播放场景流程图 383
11.5.5 弹出菜单场景流程图 385
11.5.6 图片菜单场景流程图 385
11.6 **图片解码的数据流图** **386**
11.7 **如何修改图片预览的行和列的数量** **388**
12 **ap_video** 应用 **388**
12.1 需求概述 **388**
12.2 总体架构设计 **388**
12.2.1 总体架构图 388
12.2.2 功能模块划分 389
12.3 与其它应用的同步和交互 **389**
12.4 应用依赖库及其接口 **389**
12.5 **应用的业务流程** **389**
12.5.1 应用的总流程和场景调度流程 390
12.5.2 文件列表场景流程图 391
12.5.3 视频播放场景流程图 392
12.5.4 弹出菜单场景流程图 393
12.5.5 视频菜单场景流程图 394
12.6 视频解码的数据流图 **394**
12.7 **video** 是如何实现显示视频同时显示进度条等 **UI** 界面的 **395**
12.8 **video** 如何优化播放过程中显示性能 **396**
12.9 视频播放过程中如何实现全屏到进度条，音量条的显示 **396**
12.10 视频菜单配置文件为何编译模式不同 **398**
13 **ap_radio** 应用 **399**
13.1 需求概述 **399**

版 权 所 有 侵 权 必 究

13.2 总体架构设计 399
13.2.1 总体架构图 399
13.2.2 功能模块划分 400
13.3 与其它应用的同步和交互 401
13.4 应用依赖库及其接口 402
13.5 应用的业务流程 402
13.5.1 应用的总流程和场景调度流程 402
13.5.2 电台播放场景流程图 404
13.6 如何增加预置电台的数量 406
14 fm_engine 引擎 408
14.1 需求概述 408
14.2 总体设计 408
14.3 FM 引擎的业务流程 409
14.3.1 FM 引擎的总体流程 409
14.3.2 FM 引擎的消息处理流程 410
14.4 与其他模块的同步和交互 410
14.5 应用依赖库及其接口说明 411
14.6 如何增加一条引擎消息 411
15 FM 驱动程序 411
15.1 需求概述及设计原则 411
15.2 总体设计 412
15.2.1 FM 驱动在系统结构中的位置 413
15.2.2 FM 驱动的模块划分 413
15.3 FM 驱动的硬件接口设计 414
15.4 FM 驱动的应用接口设计 415
15.5 FM 驱动提供的统一接口及每个宏定义接口的介绍 416
15.6 FM 驱动的数据流图 417
15.7 FM 驱动的内存分配说明 417
15.8 FM 驱动的的修改指南 418
15.9 FM 驱动的配置说明 419
15.10 如何增加一个 FM 驱动的应用接口 419

版 权 所 有 侵 权 必 究

16 ap_mainmenu 应用 419
16.1 需求概述 419
16.2 总体架构设计 420
16.2.1 总体架构图 420
16.2.2 功能模块划分 420
16.3 应用的生命周期 421
16.3.1 应用的启动 421
16.3.2 应用的退出 421
16.4 与其它应用的同步和交互 421
16.5 应用依赖库及其接口 421
16.6 应用的业务流程 422
16.6.1 应用的总流程和场景调度流程 423
16.6.2 桌面场景流程图（mainmenu_desktop） 424
16.6.3 弹出菜单场景流程图（option_menulist） 424
16.7 如何增加一个应用的入口 424
16.8 如何删除一个应用的入口 425
17 ap_browser 应用 426
17.1 需求概述 426
17.2 总体架构设计 426
17.2.1 总体架构图 426
17.2.2 功能模块划分 426
17.3 与其它应用的同步和交互 427
17.4 应用依赖库及其接口 427
17.5 应用的总流程和场景调度流程 428
17.6 browser 是如何保存进入和退出 ap 的 428
17.7 browser 如何实现删除整个目录操作 429
17.8 如何进入指定目录浏览文件 430
17.9 如何备份和恢复目录项 431
18 ap_udisk 应用 432
18.1 需求概述 432
18.2 总体架构设计 432

版 权 所 有 侵 权 必 究

18.2.1 总体架构图 432
18.2.2 功能模块划分 433
18.3 与其它应用的同步和交互 **433**
18.3.1 应用的启动 433
18.3.2 应用的退出 433
18.4 应用依赖库及其接口 **434**
18.5 应用的业务流程 **434**
18.6 主模块设计说明 **434**
18.6.1 模块描述 434
18.6.2 模块功能 434
18.6.3 场景调度流程 435
18.6.4 消息处理流程图 435
18.6.5 充电流程图 436
18.7 初始化模块设计说明 **436**
18.7.1 模块描述 436
18.7.2 模块功能 436
18.7.3 流程逻辑 437
18.8 退出模块设计说明 **438**
18.8.1 模块描述 438
18.8.2 模块功能 438
18.8.3 流程逻辑 438
18.9 显示模块设计说明 **438**
18.9.1 模块描述 438
18.9.2 模块功能 438
18.9.3 流程逻辑 438
18.10 如何修改 USB 设备的属性 **439**
18.11 如何实现只充电而不上报 USB 盘符 **439**
18.12 如何实现没插卡的时候，不上报卡盘符 **439**
19 **ap_playlist 应用** **439**
19.1 需求概述 **439**
19.2 总体架构设计 **441**
19.2.1 总体架构图 441
19.2.2 功能模块划分 442
19.3 PLAYLIST 应用的生命周期 **442**

版 权 所 有 侵 权 必 究

19.3.1 应用的启动 442
19.3.2 应用的退出 443

19.4 应用依赖库及其接口说明 **444**

19.5 PLAYLIST 应用的内存空间分配说明 **444**

19.6 应用中 **RAM** 空间的划分 **446**

19.7 PLAYLIST 应用的业务流程 **450**
19.7.1 应用的总流程 450
19.7.2 初始化模块的流程 453
19.7.3 遍历文件方式 454
19.7.4 获取文件信息存储 456
19.7.5 列表排序 457
19.7.6 应用退出模块流程图 467

19.8 Playlist 的数据结构说明 **467**
19.8.1 结构体数据结构 467
19.8.2 列表 LIB 文件结构 468

19.9 如何在列表上增加或者删除一种文件格式 **471**

19.10 如何去掉 **VIDEO** 或者 **AUDIBLE** 的播放列表 **472**

19.11 如何过滤掉录音应用产生的文件 **473**

20 ap_setting 应用 **473**

20.1 需求概述 **473**

20.2 总体架构设计 **473**
20.2.1 总体架构图 473
20.2.2 功能模块划分 474

20.3 与其它应用的同步和交互 **474**

20.4 应用依赖库及其接口 **475**

20.5 应用的业务流程 **475**

20.6 配置数据的数据流图 **476**

20.7 关键的数据结构 **476**

20.8 如何增加一个配置项 **478**

21 ap_tools 应用 **479**

21.1 需求概述 **479**

21.2 总体架构设计 **479**

21.2.1 总体架构图 479
21.2.2 功能模块划分 479

21.3 与其它应用的同步和交互 480

21.4 应用依赖库及其接口....480

21.5 应用的业务流程....480
21.5.1 应用的总流程和场景调度流程 480
21.5.2 工具列表场景流程图 483
21.5.3 日历场景流程图 484
21.5.4 秒表场景流程图 485

21.6 如何增加一组 alarm 486

22 ap_alarm 应用 486

22.1 需求概述 486

22.2 闹钟的设计原理和启动 486

22.3 总体架构设计 487
22.3.1 总体架构图 487
22.3.2 功能模块划分 487

22.4 与其它应用的同步和交互 488

22.5 应用依赖库及其接口....488

22.6 应用的业务流程....488
22.6.1 应用的总流程和场景调度流程 488
22.6.2 超时退出之后的流程图 489

22.7 关键的数据结构....489

23 ap_ebook 应用 491

23.1 需求概述 491

23.2 总体架构设计 491
23.2.1 总体架构图 491
23.2.2 功能模块划分 492

23.3 与其它应用的同步和交互 493

23.4 应用依赖库及其接口....493

23.5 应用的业务流程....494
23.5.1 应用的总流程和场景调度流程 494
23.5.2 文件列表场景流程图 496

版 权 所 有 侵 权 必 究

23.5.3 阅读场景流程图 497
23.5.4 弹出菜单场景流程图 499
23.6 如何修改阅读场景每页的行和列的数量 500

版 权 所 有 侵 权 必 究

3 引言

3.1 版本历史

日 期	版本号	注释	作者
2012-08-02	0.1	建立初始版本	Paul Chen
2013-05-27	1.0	第一次发布文档	蔡李镇

3.2 编写目的

详细描述 US212A 方案的基本结构，应用程序开发，case 驱动程序开发；详细描述各个 case 驱动程序的设计和实现，各个应用程序的设计和实现。用于指导 Case 详细设计和开发，同时给二次开发和代码维护提供参考。

3.3 术语和缩写词

缩写、术语	解 释
US212A	指的是适用于 ATJ212X IC 的设计方案总成，包括软件平台及其对应的开发工具，硬件设计等。
OS	Operation System
应用（Application，AP）	指的是能够被任务管理器调度和执行的应用程序单元。
应用场景	指的是某个具体的应用的某个具体的界面。
前台应用	又称为 UI 应用，指的是需要展现界面的应用，该类应用程序，往往在界面消失之后，应用也已经退出。本文档后面统一称为前台应用
后台应用	又成为引擎应用，指的是无需展现界面，但完成特定的控制流程，并由任务管理器调度执行的应用程序。本文档后面统一称为后台应用。

版 权 所 有 侵 权 必 究

消息循环	指的是一个不断的接收消息和处理消息的循环，而且循环的退出条件，往往是因为处理某个具体的消息。
VRAM	虚拟 RAM 的缩写。是在 FLASH 的系统区划分出来的一个按照绝对地址读写的 FLASH 区域，每个 AP 分别有 1K 的空间，主要用于存储各个 AP 需要保存到 FLASH，供后续启动使用的配置数据。系统专门提供了相关接口，应用只需要传入各个 AP 对应的 VRAM 区的相对地址值，就可以直接读取和写入。
ID3	位于一个 mp3 文件的开头或末尾的若干字节内，附加了关于该 mp3 的歌手，标题，专辑名称，年代，风格等信息
Album art	专辑图片
Lrc	歌词的一种文件格式
playlist	音乐文件播放列表
favorlist	音乐收藏夹
mcg 文件	指的是存储了菜单配置信息的文件
Sty 文件	指的是存储了相关资源的及其属性的文件，比如字符串资源，图片资源及其位置，大小等信息。

3.4 用户手册导读

本文档结构上可以分成 6 个部分，如下图所示：

1 声 明
2 目 录
⊟3 引言
3.1 版本历史
3.2 编写目的
3.3 术语和缩写词
3.4 用户手册导读
⊟4 开发环境
⊞4.1 开发环境的搭建
⊞4.2 UI 模拟器
⊞4.3 调试方法介绍
⊟5 case 基本结构及开发
⊞5.1 Case 运行环境
⊞5.2 代码目录导航
⊞5.3 应用管理器 ap_manager
⊞5.4 开机与关机 ap_config
⊞5.5 common 设计与使用
⊞5.6 前台应用设计与开发
⊞5.7 后台应用设计与开发
⊞5.8 多线程设计与开发
⊞5.9 驱动程序设计与开发
⊟6 case 驱动程序详解
⊞6.1 KEY 驱动设计
⊞6.2 LCD 驱动设计
⊞6.3 UI 驱动设计
⊞6.4 Welcome 驱动设计
⊟7 界面设计与开发
7.1 引进可配置化方法
⊞7.2 可配置化 UI
⊞7.3 可配置化菜单
⊞8 ap_music 应用
⊞9 music_engine 引擎
⊞10 ap_record 应用
⊞11 ap_picture 应用
⊞12 ap_video 应用
⊞13 ap_radio 应用
⊞14 fm_engine 引擎
⊞15 FM 驱动程序
⊞16 ap_mainmenu 应用
⊞17 ap_browser 应用
⊞18 ap_udisk 应用
⊞19 ap_playlist 应用
⊞20 ap_setting 应用
⊞21 ap_tools 应用
⊞22 ap_alarm 应用
⊞23 ap_ebook 应用

1 2 3 4 5 6

第一部分：引言

包括版本历史，编写目的，术语和缩略词，以及用户手册导读。

第二部分：开发环境

介绍开发环境的搭建，一些工具的使用方法，以及调试方法。UI Editor 工具和菜单配置工具的介绍放在第五部分，即界面设计与开发，中进行重点介绍。

第三部分：case 基本结构及开发

介绍 case 的基本结构原理，以及应用程序与驱动程序的设计与开发。

case 的基本结构原理包括：Case 的运行环境、代码目录导航、应用管理器、开机与关机、应用程序基本接口模块 common 的设计与使用。

版 权 所 有 侵 权 必 究

这部分文档应该详细阅读，读者通过这部分文档，基本上能全面完整的理解 US212A，对开发健壮高效、概念统一的 Case 有非常大的帮助。

第四部分：cae 驱动程序详解

介绍 3 个与 case 关系密切的驱动程序：KEY 驱动、LCD 驱动、UI 驱动，这 3 个驱动实现用户的 I/O 功能，是不可缺少的 case 基础驱动。

另外，由于 Welcome 驱动与 LCD 驱动关系较大，所以也放在这部分讲解。

FM 驱动不是 case 的基本驱动，只是作为收音机功能的一部分，是一个可选的外部模组的驱动程序，所以不在这一部分中介绍，而放在收音机应用的后面。

第五部分：界面设计与开发

介绍可配置化 UI 和可配置化菜单的设计与开发，包括工具使用介绍和配套代码开发等。

第六部分：应用程序详解

介绍各个应用程序的设计和开发要点。

版权所有 侵权必究

4 开发环境

4.1 开发环境的搭建

4.1.1 Cygwin 的安装

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。

4.1.1.1 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）。

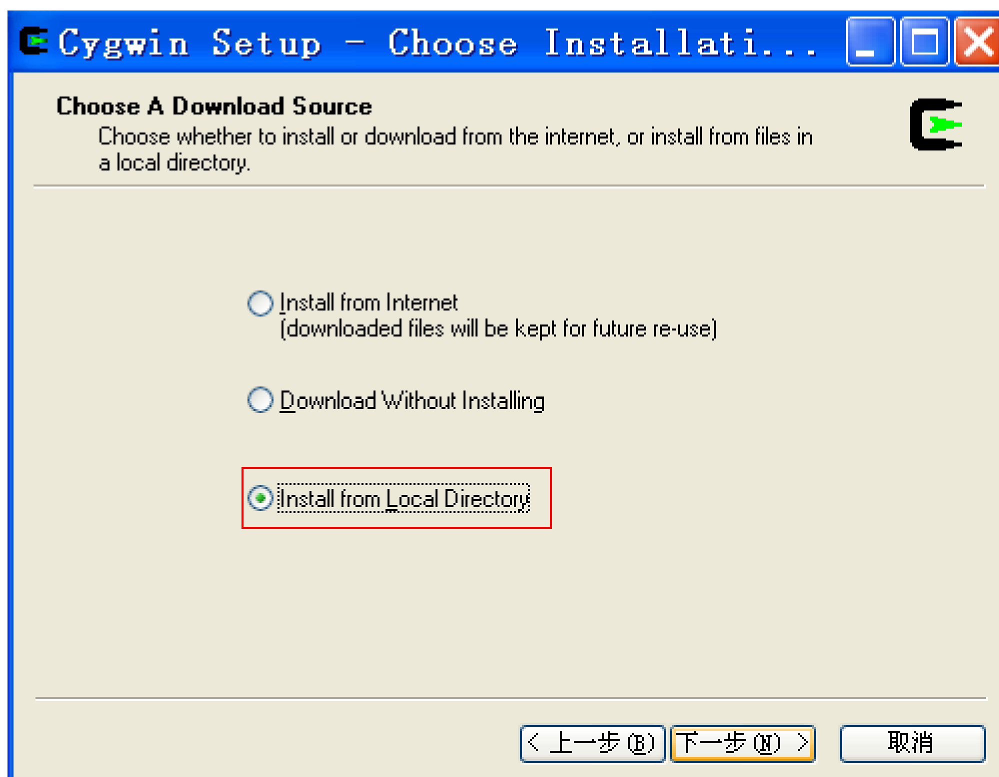
3）cygwin 安装到根目录下，不可安装到类似 D:\Program Files\cygwin 的路径。

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；

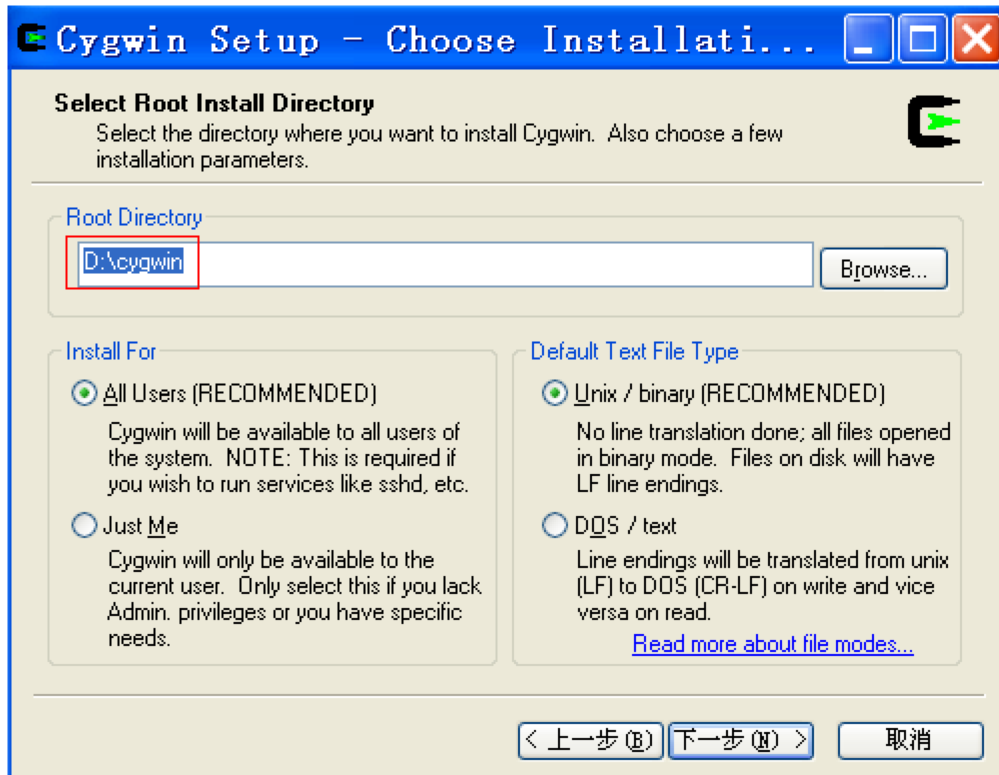
将服务器的 tidy 版本 cygwin 拷贝到本地目录 D:\MIPS_TOOLS\cygwin_full，此路径就是下面步骤 4）的路径。

1）双击 setup.exe，按下一步。

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

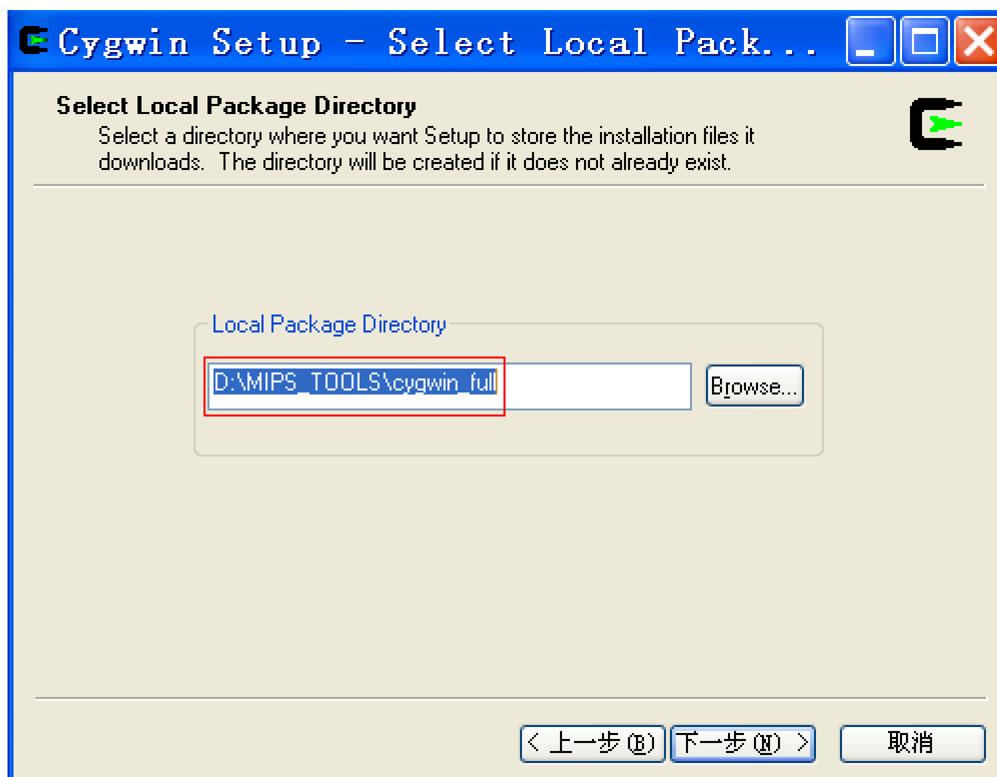


3）选择安装路径 D:\Cygwin，按下一步；

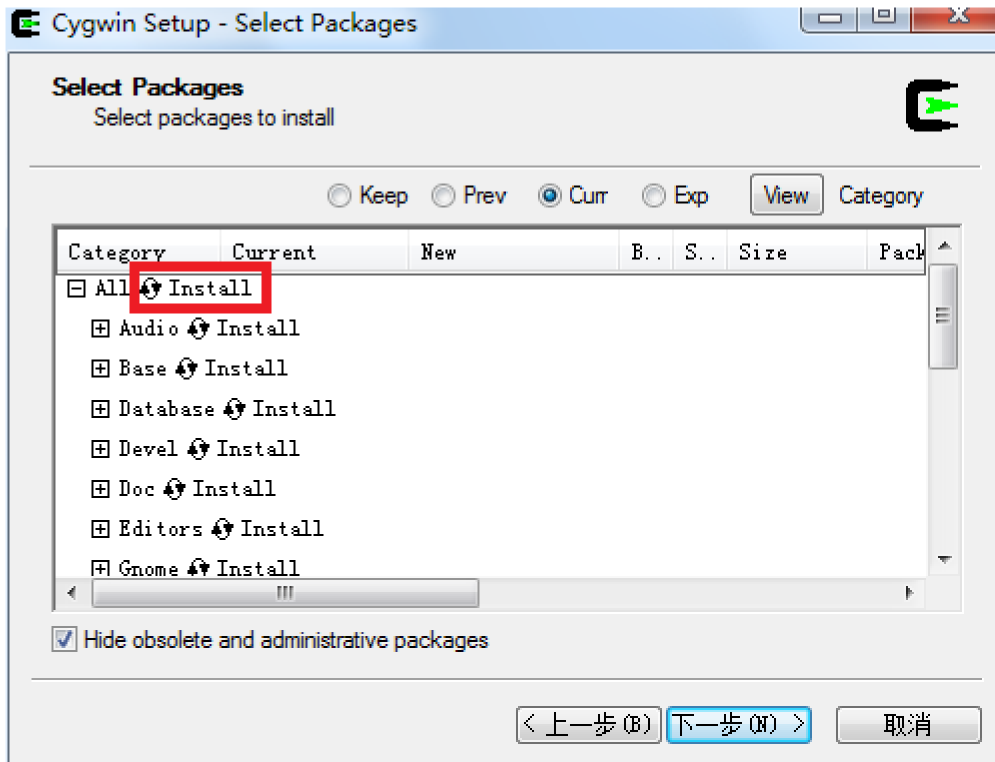


版 权 所 有 侵 权 必 究

4）选择源安装程序的路径，按下一步。



5）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。



6）后面等待安装结束就可以了（可能需要十几分钟才装完）。

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。

7）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 QAC 失败）。

8）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”。

4.1.1.2 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）

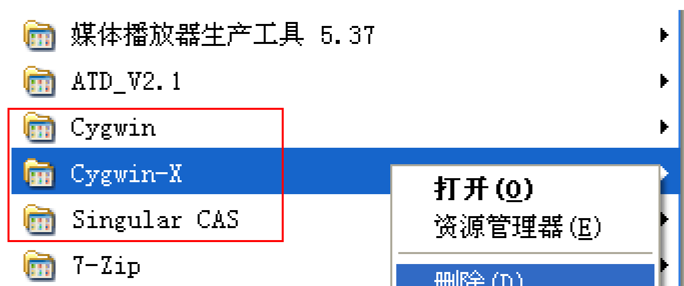
1）按照第 1 节的 1）至 4）步操作，在第 1 节第 5）步的界面，红色框处于 Uninstall 时，按下一步进行卸载，直到完成。

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）。

版 权 所 有 侵 权 必 究



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"

4.1.2 SDE 工具链的安装

步骤如下（下面的命令行，可以直接 copy）：

1）将安装文件放在目录 D:\cygwin\tmp\PN0016-06.61-2B-MIPSSW-MSDE-v6.06.01.tgz。

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）解压安装文件“gzip -dc /tmp/PN00116-06.61-2B-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。

4.1.3 Firmware Develop Kit 的安装

按照默认安装完成即可，无特殊配置。

4.1.4 UI Editor 工具和菜单设计工具

UI Editor 工具无需安装，直接打开使用即可。

版 权 所 有　侵 权 必 究

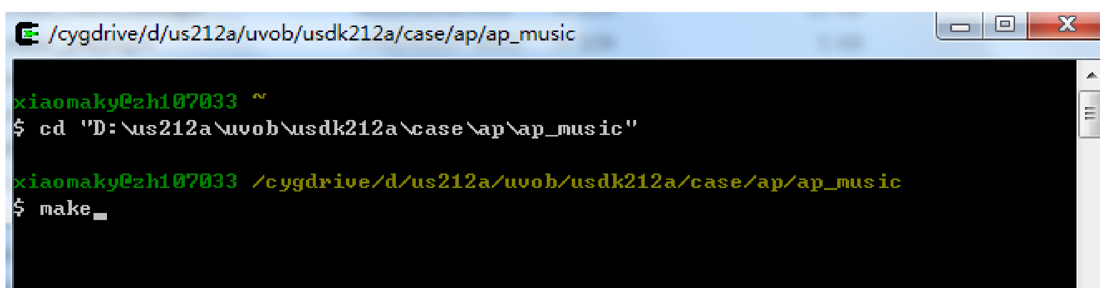
菜单编辑可以使用 FW Modify 工具。

4.1.5 量产工具的安装

按照默认安装完成即可，无特殊配置。

4.1.6 如何编译 AP、生成固件并烧录到小机运行

一、编译 AP：在各 AP 目录下，有 makefile 文件，cd 到相应 AP 目录下，直接 make 即可。



也可以执行\usdk212a\case\cfg\makefile，批量执行所有 AP 的 make。

二、生成固件：
执行\usdk212a\case\fwpkg\buildfw.bat，即可生成固件。

三、下载固件到小机：
1、小机以 ADFU 模式或者 U 盘模式连接 PC；
2、打开量产工具；
3、选择固件，选择相应的下载配置选项，下载；
4、等待下载完成，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；

4.2 UI 模拟器

4.2.1 UI 模拟器的作用

UI 模拟器是一种不依赖于开发板，能够独立在 PC 上进行嵌入式程序开发的工具，它主要是为了改善现有嵌入式软件编制过程中开发效率低下，调试手段单一，过分依赖开发板的状况。它不属于硬件模拟范畴，是对嵌入式软件环境的模拟，只要是与硬件无关的代码，都可以使用 ui 模拟器来开发和调试。

模拟器在 Windows 200/Windows XP、VC++6.0 环境下进行开发，在符合 ANSI C 规范的条件下，case 下所有 AP 以及 psp 下部分代码在 VC6 中进行编译后即可运行，得到的效果与开发板一致。

使用模拟器主要有以下 2 大优势：

1. 提高开发效率

   在开发板上开发程序，需要在 Cygwin 环境下编译，然后生成固件并下载到开发板上验证程序，这个过程需要上分钟的时间，如果固件越大，烧写固件的时间就更长，而在 pc 环境下，只需要编译和运行就能看到结果，速度大为提升；

2. 方便跟踪调试

版 权 所 有 侵 权 必 究

常见开发模式下，需要调试时，大都通过串口或者 USB 口与 PC 端通讯，由 PC 端软件显示、控制、调试等方式进行。往往为了确定是哪一段代码出了问题，需要多次、多位置添加调试代码以显示多个变量的值，一个简单的 bug 可能会耗上很多的时间来跟踪；

在模拟器下，可以很方便的设置断点，单步跟踪，查看变量值、函数调用层次等，很容易检查出内存溢出，空指针操作，变量类型不匹配等常见错误；

同时为了某些特殊需要还专门编写了辅助调试工具，可以及时查看当前界面的效果。

4.2.2 UI 模拟器工程简介

4.2.3 UI 模拟器的开发环境和使用方法

4.2.3.1 创建工程

每个 AP 应用模块或者驱动，都需要创建对应的 VC 工程。

下面以主界面应用模块（mainmenu.ap）为例，说明一下工程的创建步骤。

1．打开 Visual C++ 6.0，选择子菜单 File→New：

版 权 所 有 侵 权 必 究

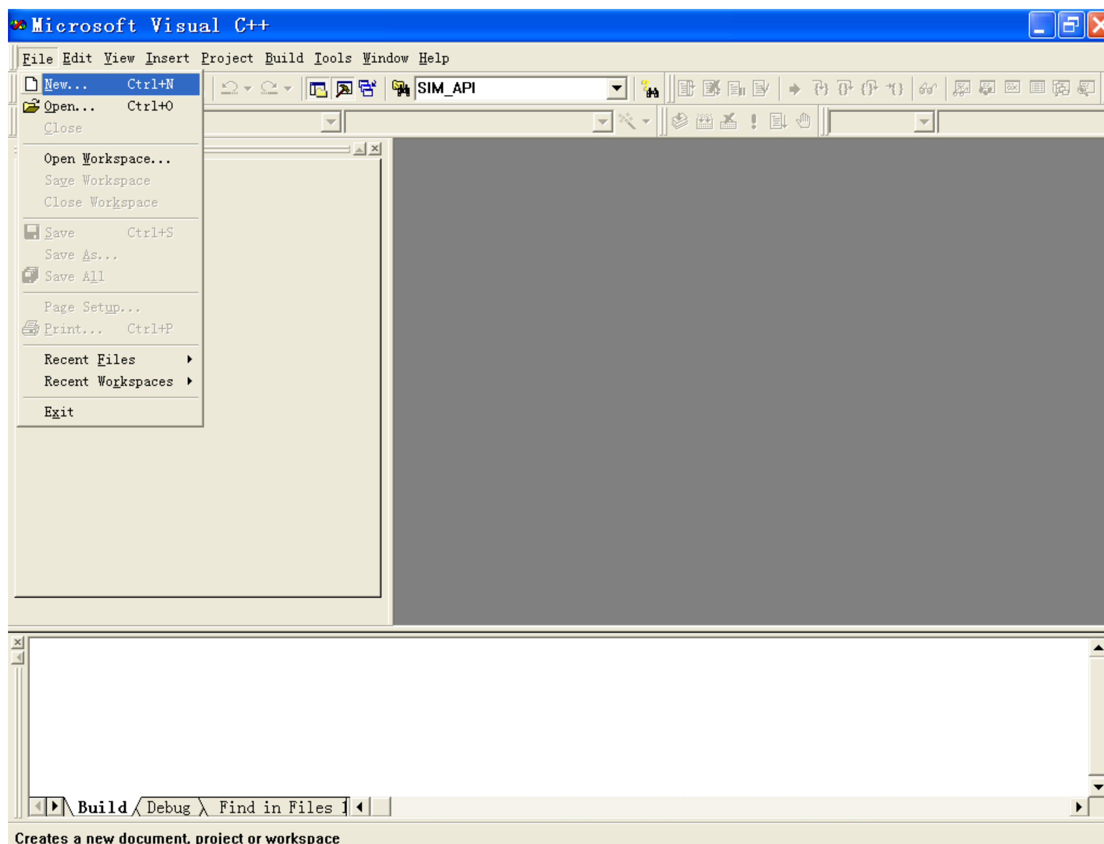


图 **31-1** 创建工程

2．点击<Projects>页面，选择 Win 32 Dynamic-Link Library，在 Project name 处输入工程名，在 location 处选择模拟器目录下 simulator\Apps\路径，然后点击[OK]按钮：

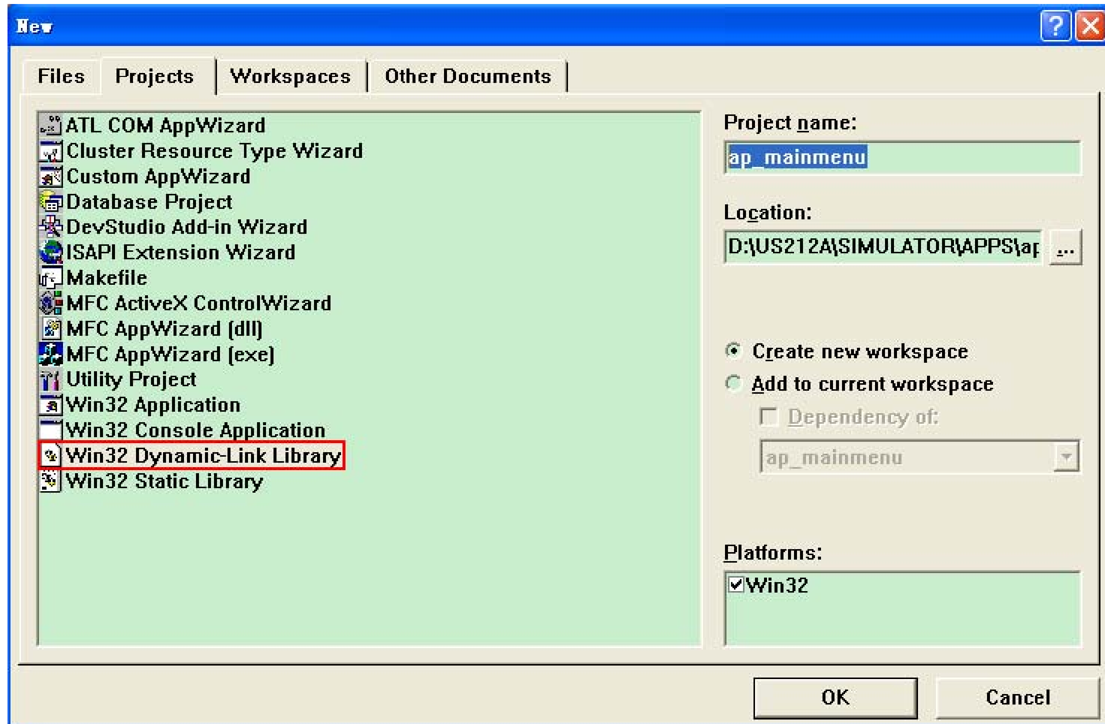


图 31-2 输入工程名

3．选择 An empty DLL project，然后点击 Finish 完成基本工程建立。

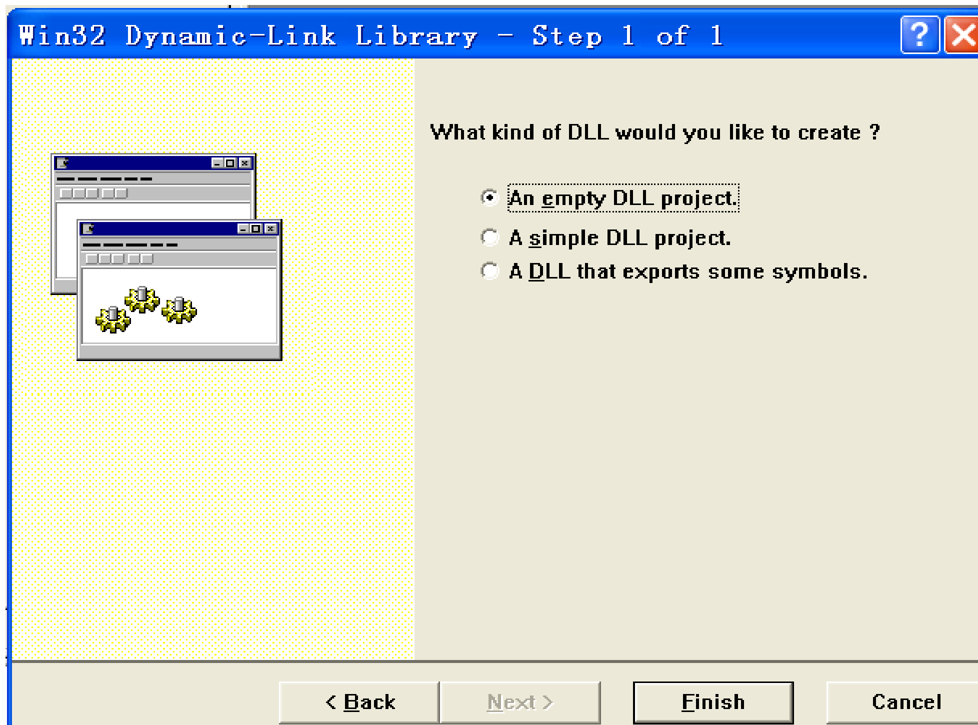


图 31-3 选择空工程

版 权 所 有 侵 权 必 究

4.2.3.2 设置工程

1．对着工程名单击鼠标右键，弹出菜单，选择 Setting 项。也可以选择菜单 Project 下的 Setting 或热键 Alt+F7：

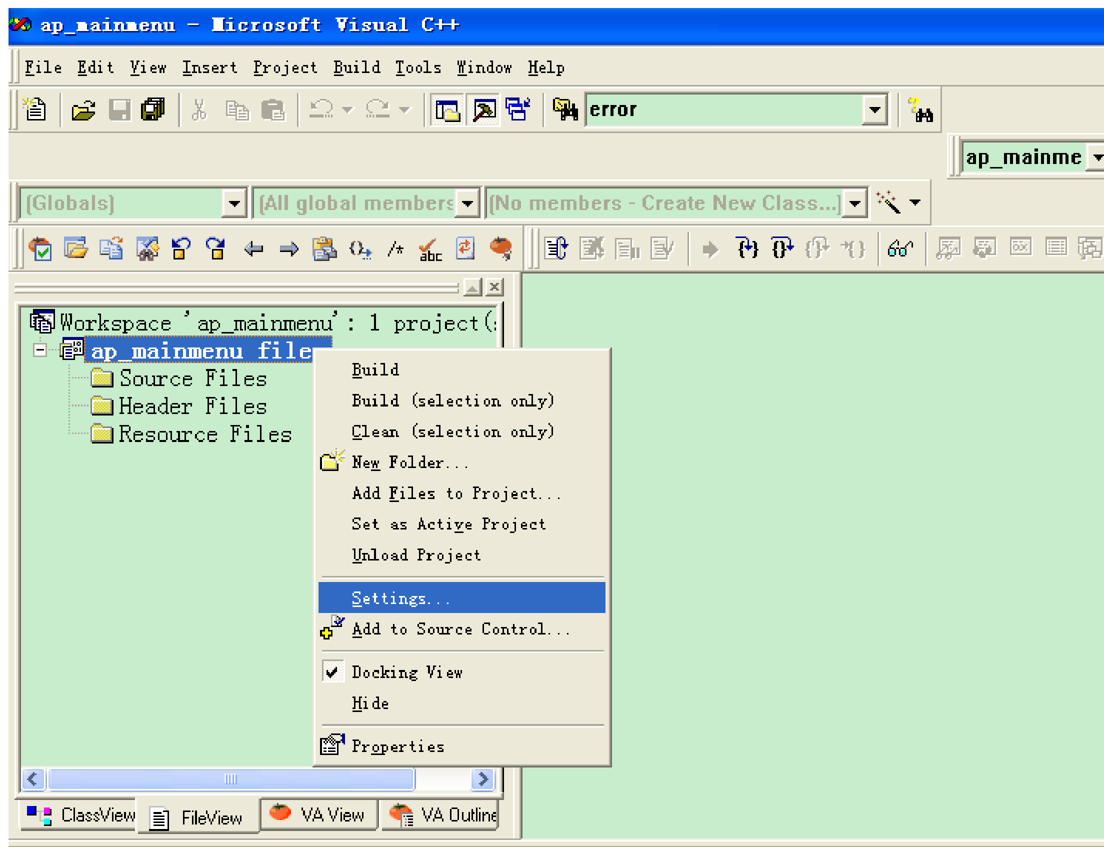


图 31-4 设置工程

2．选择<Debug>页，在 Executable for debug session:栏输入“..\..\bin\Debug\Simulator.exe”，在 Working directory:栏输入“..\..\bin\Debug\”：

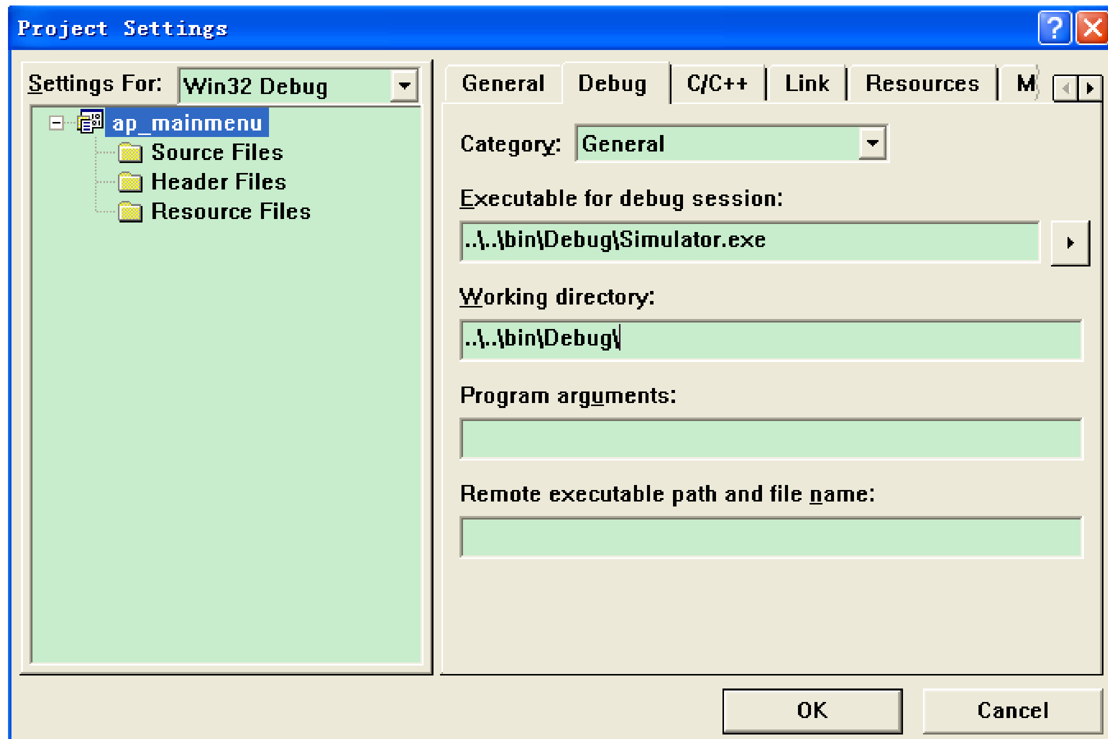
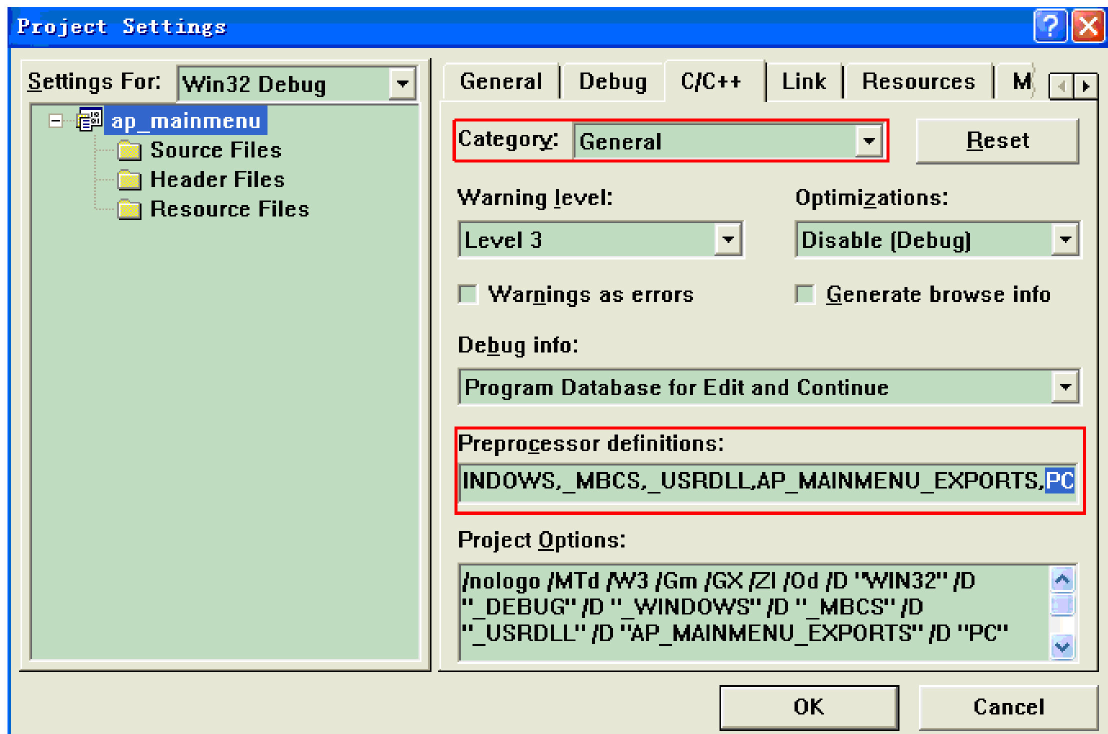


图 31-5 输入 debug 目录

3．选择 C/C++页，在 Category 栏选择 General，在 Preprocessor definitions:栏添加宏定义“**PC**”,PC 是为模拟器和小机环境差异而设。



版 权 所 有 侵 权 必 究

图 31-6 添加预处理宏

4．Category 栏选择 Code Generation，Use run-time library 栏选择 Debug Multithreaded DLL，Struct member alignment 栏选择 1 Byte：

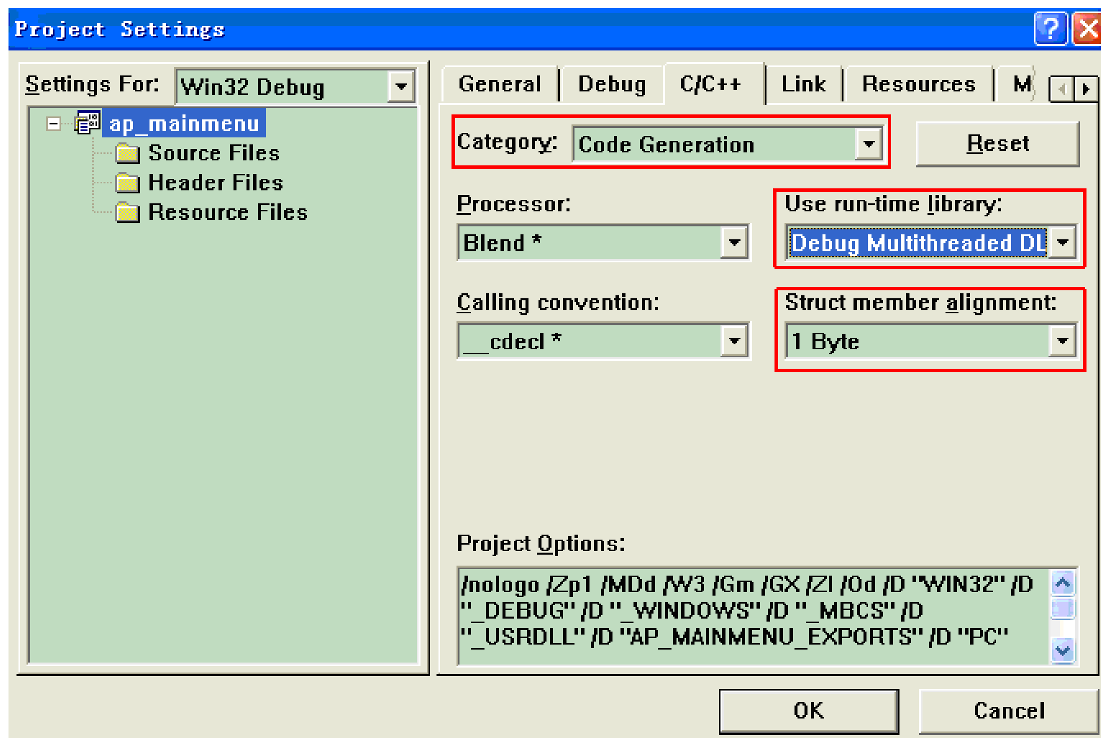


图 31-7 选择单字节对齐

5．Category 栏选择 Precompiled Headers，选择 Not using precompiled headers：

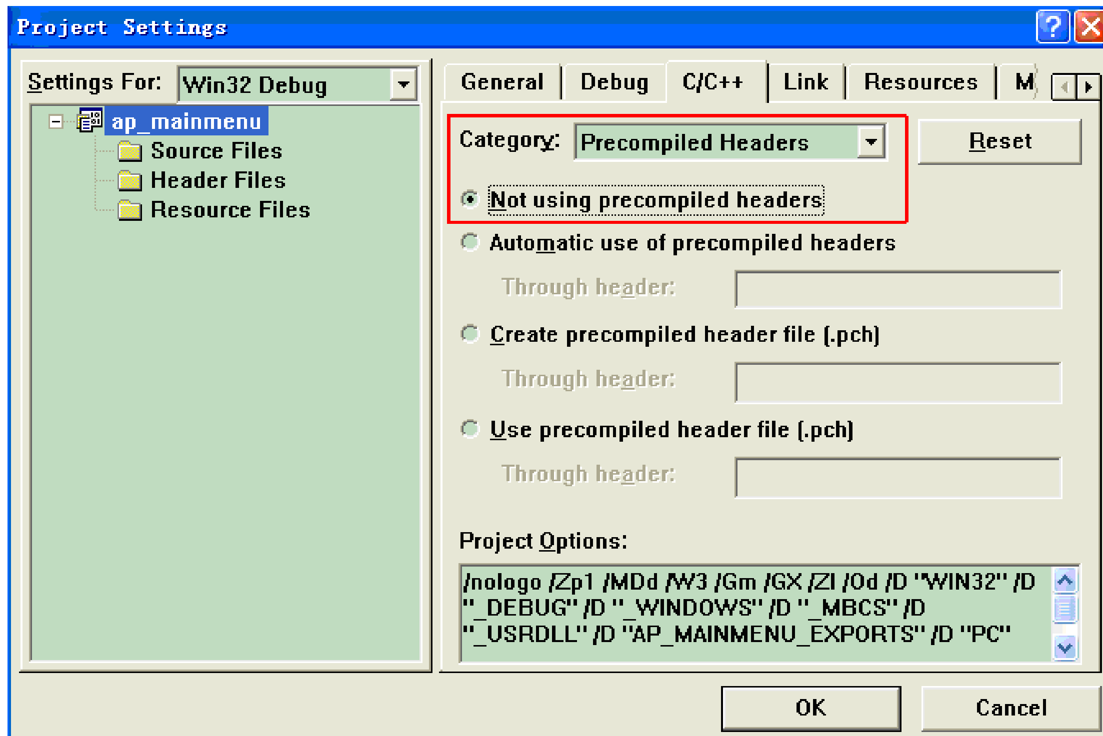
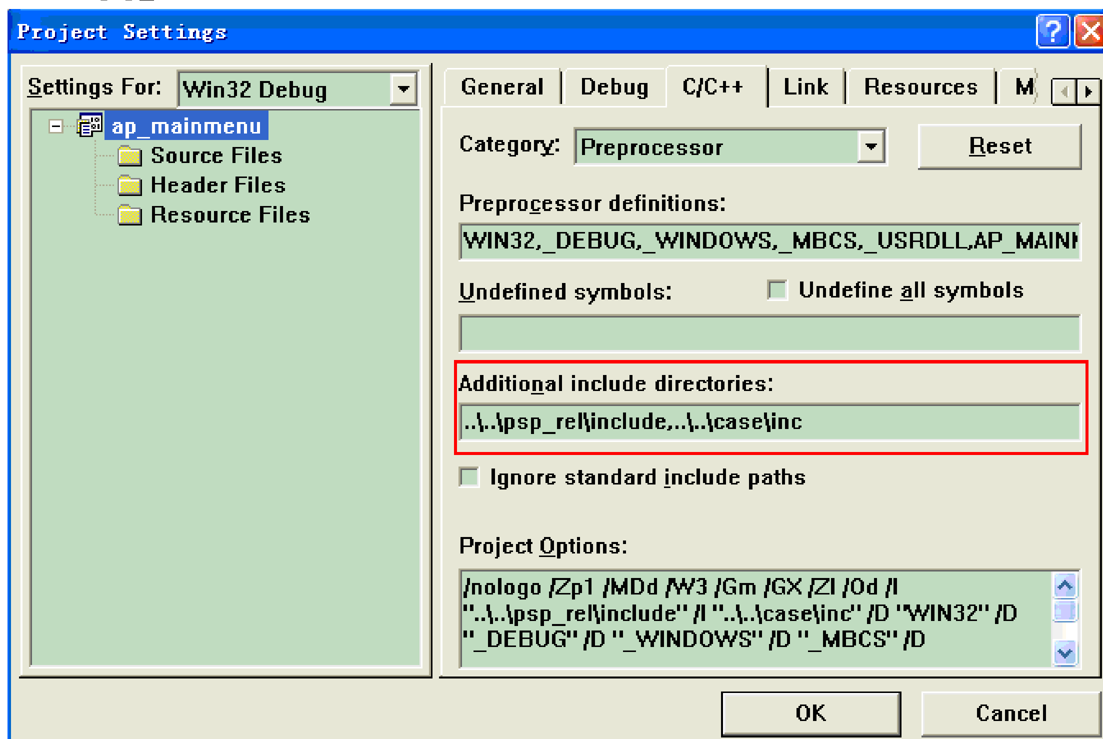


图 31-8 不使用预编译头

6．选择 Preprocessor，在 Additional include directories 输入：“..\..\..\psp_rel\inc,..\..\..\case\inc”：



版 权 所 有 侵 权 必 究

图 31-9 输入 include 路径

7. 选择 Link 页，在 Output file name:栏输入“..\..\bin\Debug\mainmenu.ap，在 Object/library modules 栏 输 入 “..\..\bin\debug\SimKernel.lib ..\..\bin\debug\SimBase.lib ..\..\bin\debug\comlib.lib”：

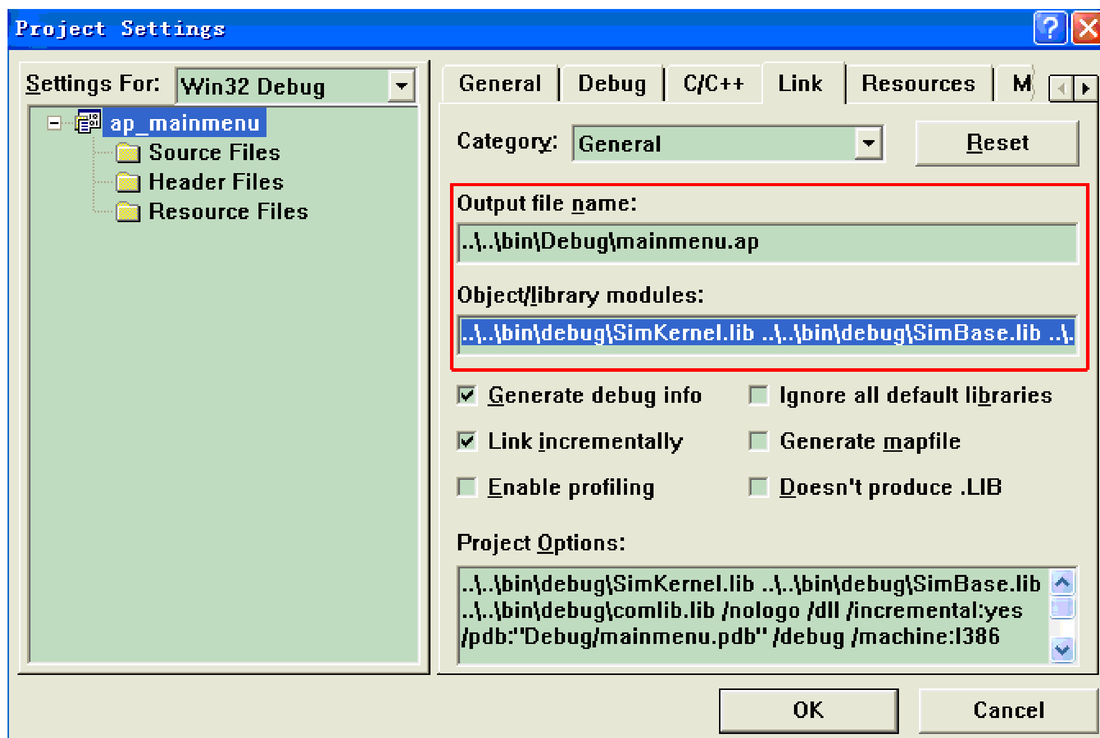


图 31-10 输入 output file 名

4.2.3.3 添加目录和文件

1．对着文件夹 Source Files 点鼠标右键，选 Add Files to Folder：

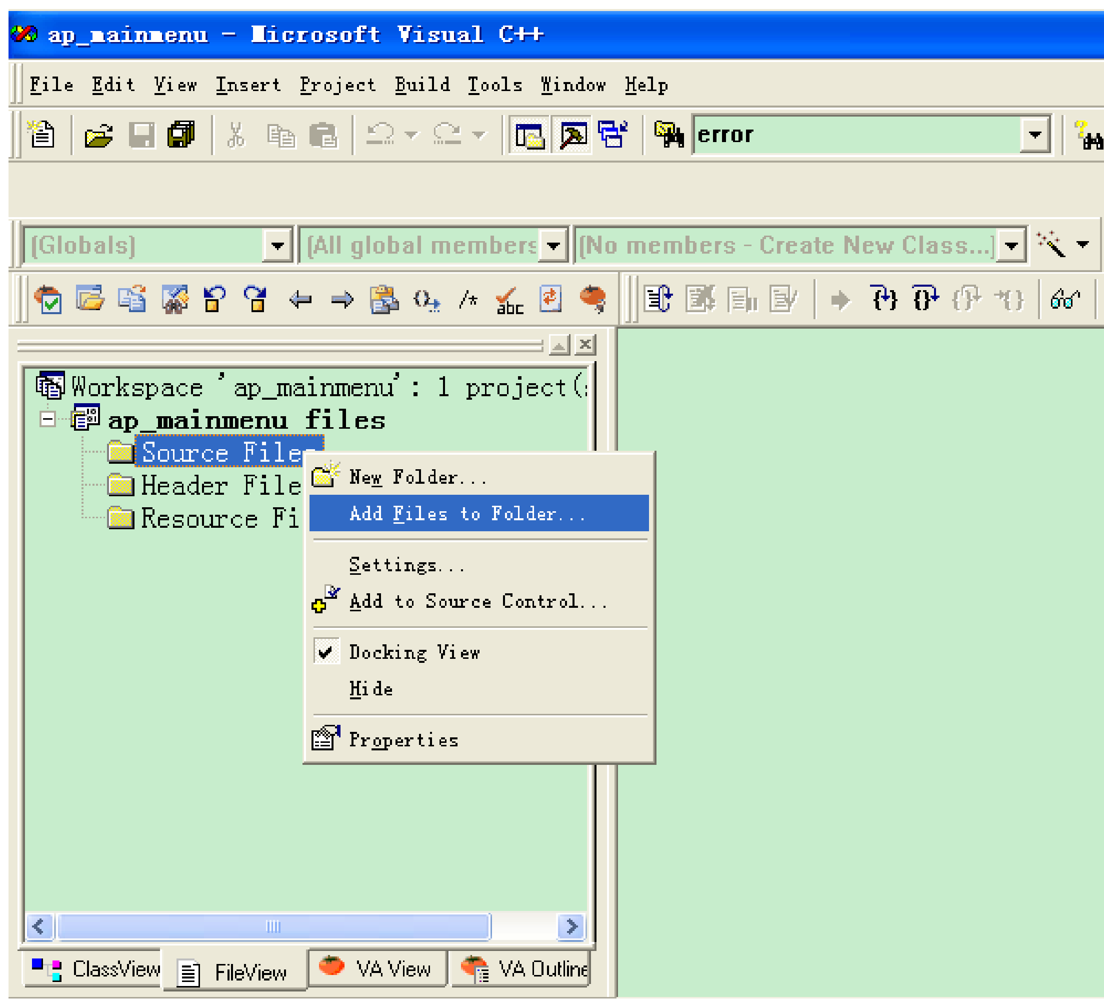


图 31-11 往工程添加文件

2．把 AP 目录（.\ case\ap\ap_mainmenu）下的 C 文件添加到文件夹中：

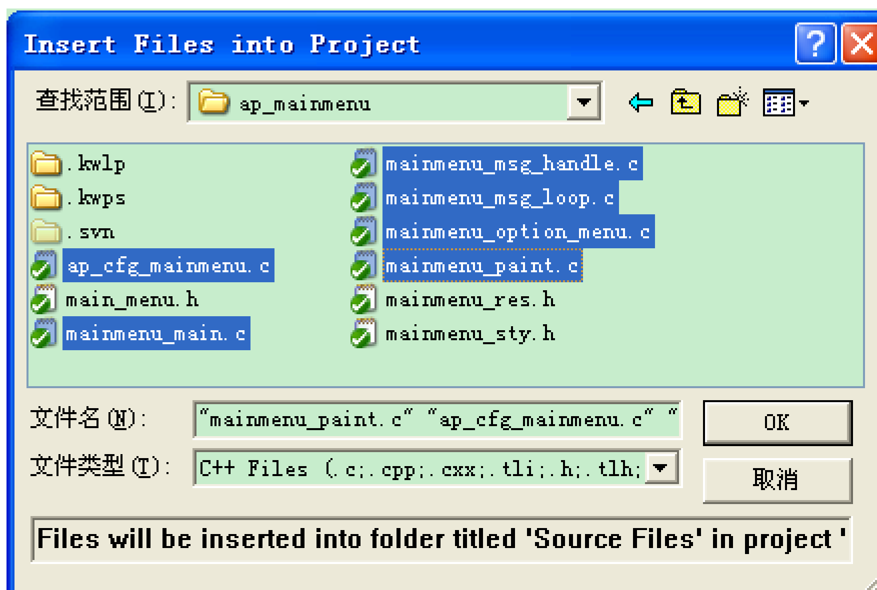


图 31-12 选择添加的文件

3．在 Source Files 文件夹标志上点右键，选择“New Folder”：

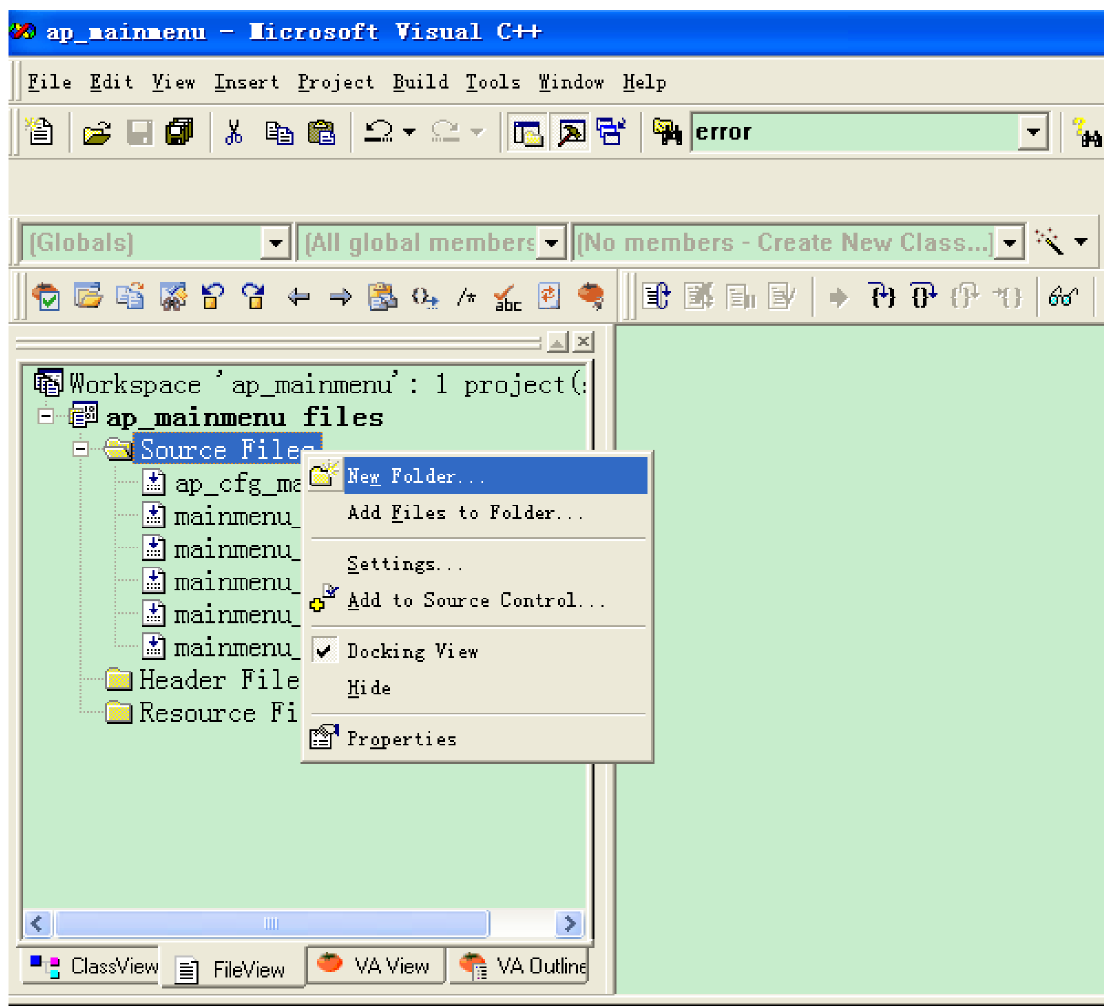


图 31-13 建立 common 文件夹

在弹出的 New Folder 窗口的 Name of the new folder 栏输入 ForSim：

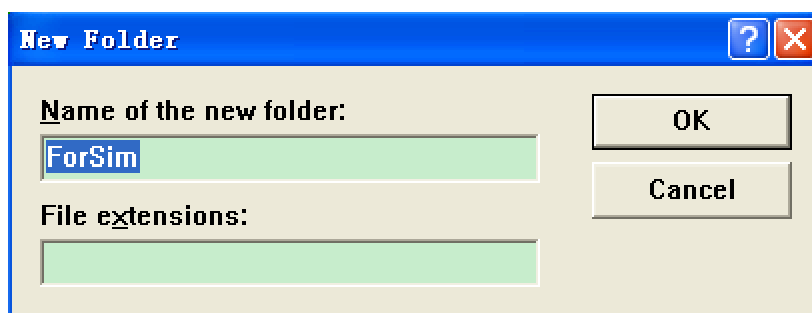


图 31-14 输入 common

在工程窗口 ForSim 文件夹处点右键，选择"Add Files to Folder"：

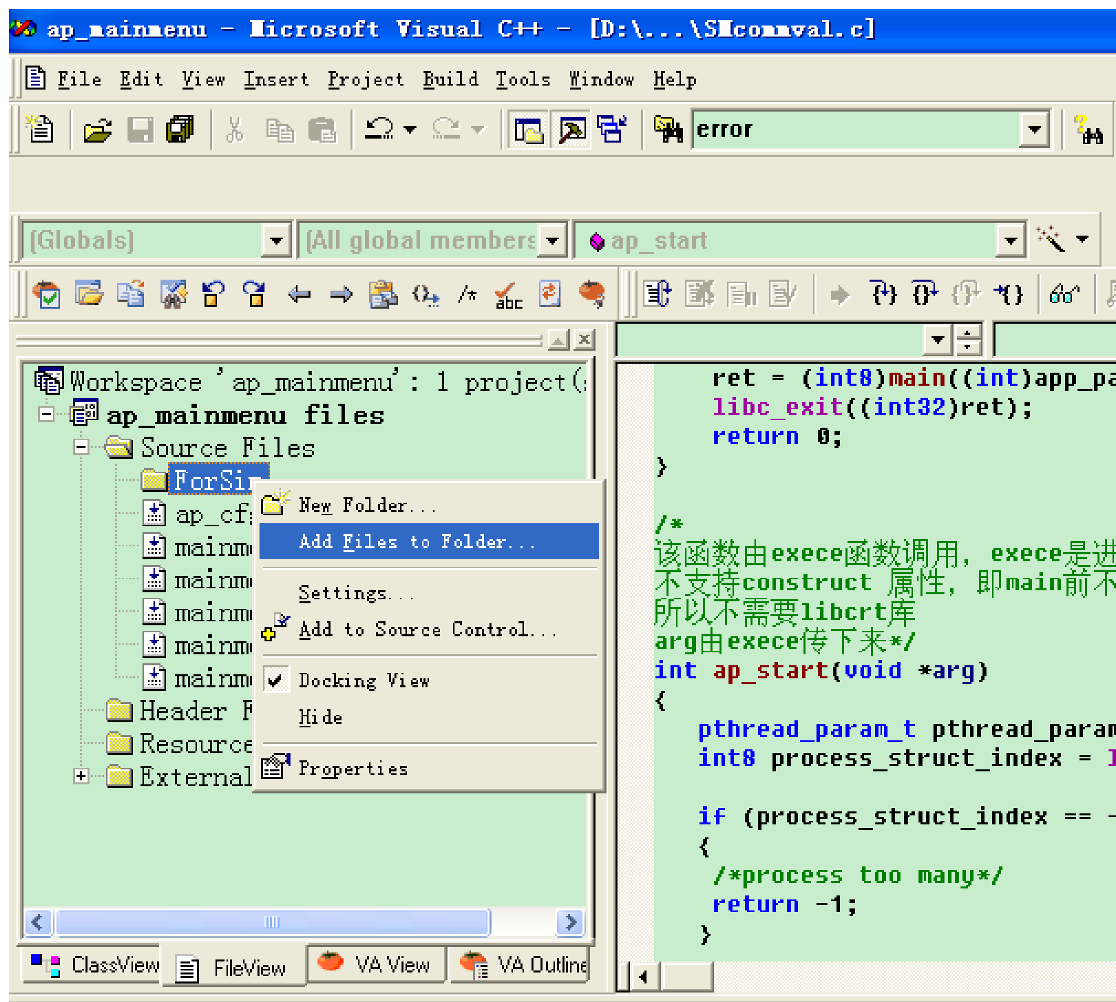


图 31-15 添加文件到 common

将 SMcommval.c 添加到 ForSim 下，SMcommval.c 是用来模拟 ap 的运行时库，每个 ap 都需要一个，可以使用相同的文件：

版 权 所 有 侵 权 必 究

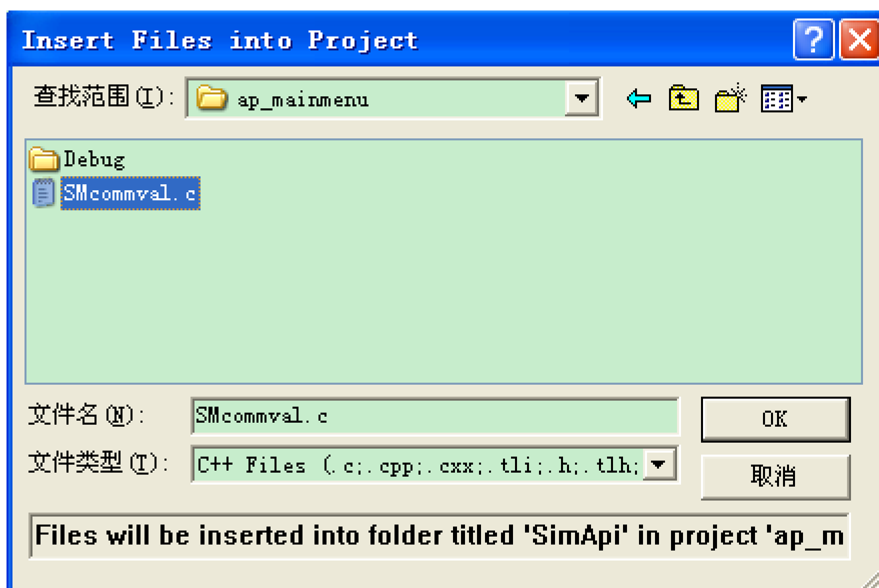


图 31-16 添加 **SMcommval.c** 到 **ForSim**

4.2.3.4 编译工程

选择菜单 Build，选择 Rebuild All，编译没有错误，即可开始运行调试程序了。

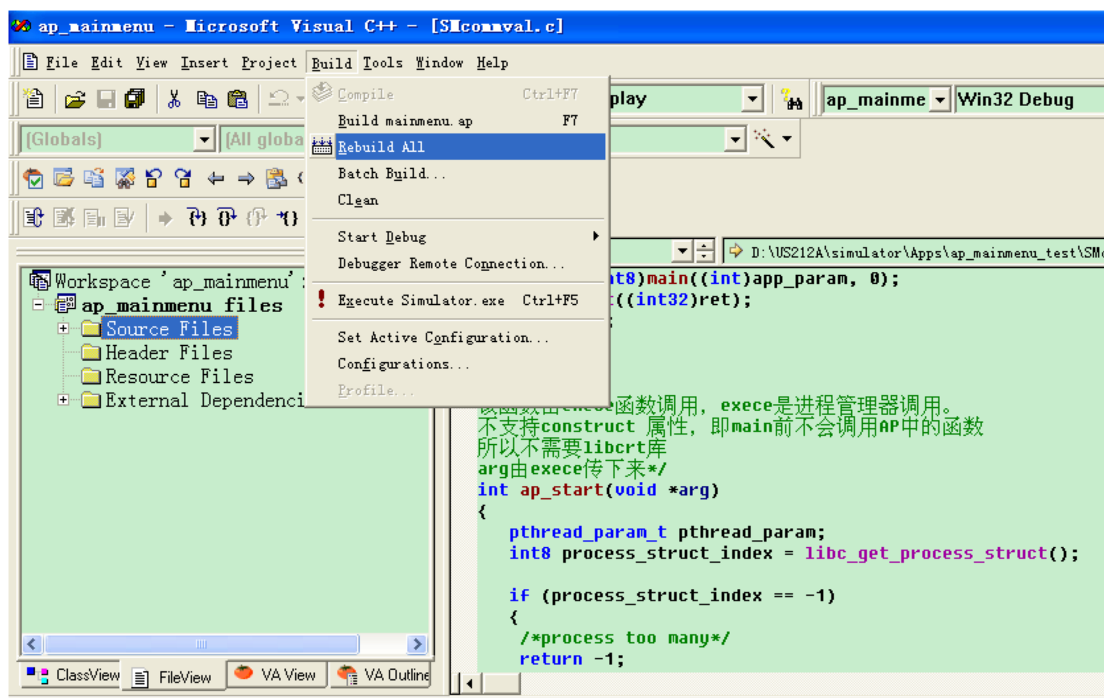


图 31-17 编译工程

4.2.4 如何为新建的 AP 搭建模拟器的开发环境

可以按照上一节介绍的模拟器使用方法，先创建一个新的工程，然后按照说明配置工程的设置项，添加这个工程的源文件，然后编译。

也可以用一个现有的 ap 工程，直接修改，例如以 mainmenu_ap 为模板建立一个 record_ap 步骤如下：

1. 复制 mainmenu_ap 目录并更名为 record_ap，将目录里的 mainmenu.def，mainmenu.dsp amainmenu.dsw 更名为 record.def，record.dsp，record.dsw，然后用 UltraEdit 打开 record.dsp，将里面的 mainmenu 字符串修改为 record；
2. 打开 record.dsw，将原来添加的文件全部删除，然后添加这个驱动需要编译的文件，编译即可。

4.2.5 如何用模拟器进行 Debug

4.2.5.1 调试前的设置

若当前 workspace 有多个工程，则要将待调试的工程设为活动工程，这样设置的断点才有效。方法如下：

设要调试的是 ap_maimenu 工程，在 VC 菜单栏上选择 Project→Set Active Project。 或者选中待设的工程，点右键，选“Set as Active Project”：

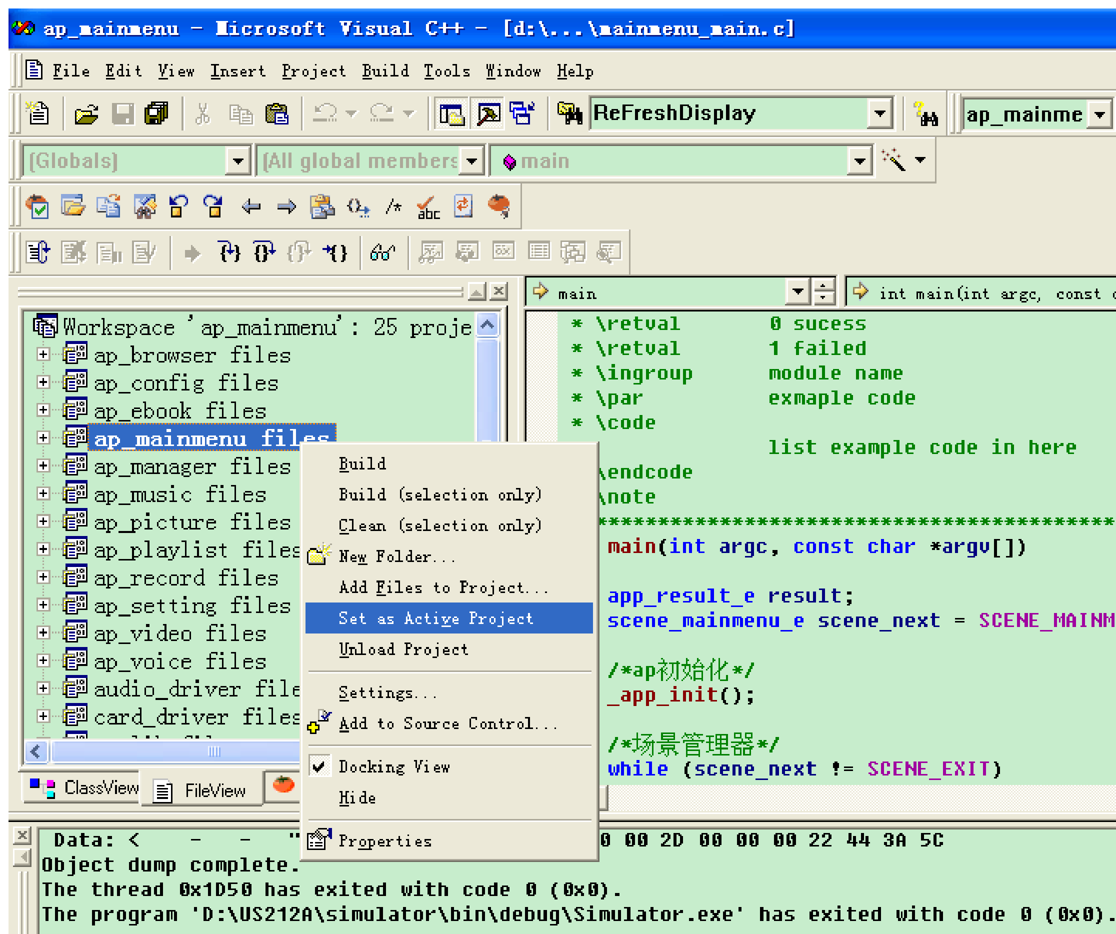


图 31-18 设置活动工程

为方便在显示活动工程，可 VC 菜单栏选择 Tools→Options，在弹出窗口中选“Format”项，在 Category 栏选 Workspace Window，在 Font 栏选“隶书”，然后按 OK 按钮：

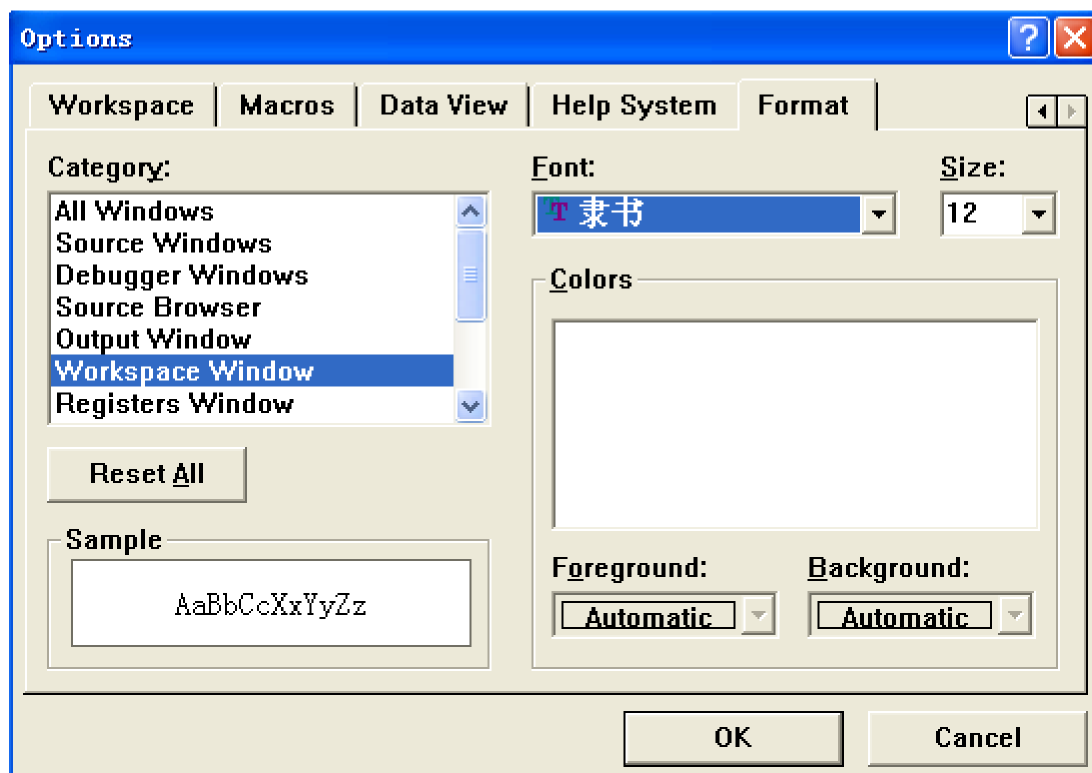


图 31-19 设置 Workspace Window 字体

这样，活动的工程(ap_playlist)将加粗显示，容易辨认：

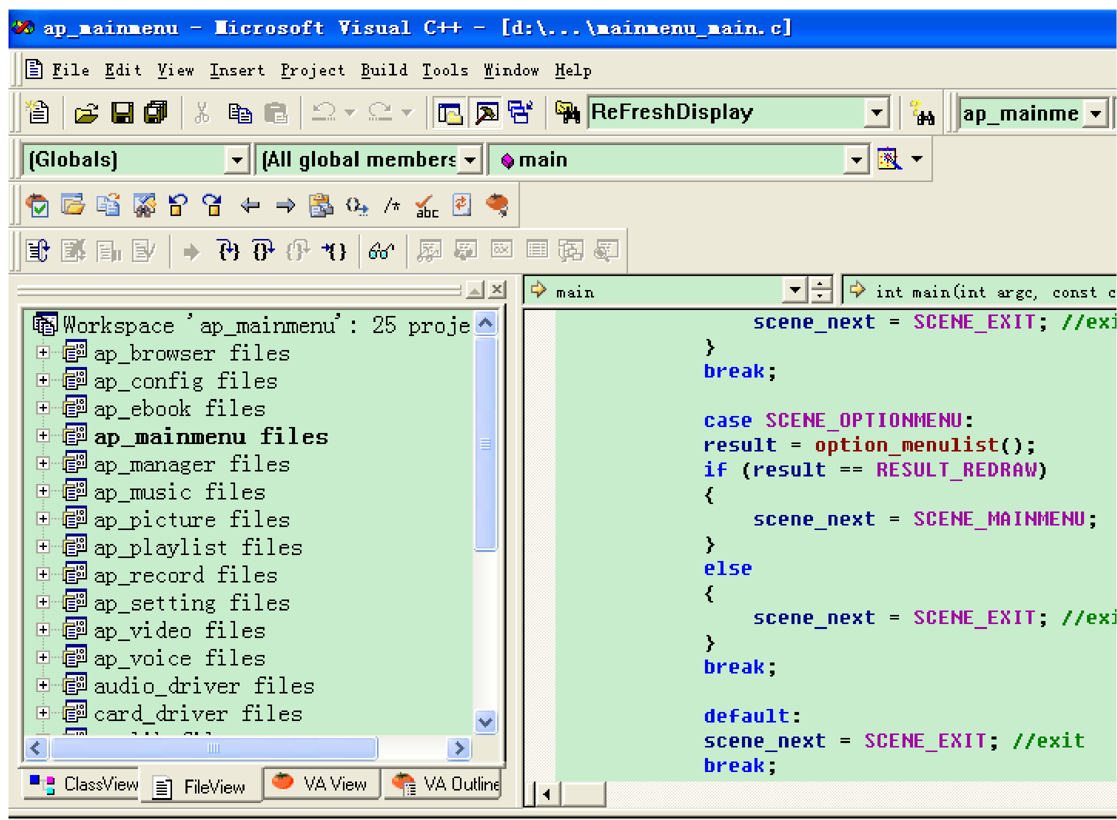


图 31-20 活动工程显示

设好活动工程后，最好将活动工程 Rebuild All 一下，方法是选择菜单 Build →Rebuild All:

版 权 所 有 侵 权 必 究

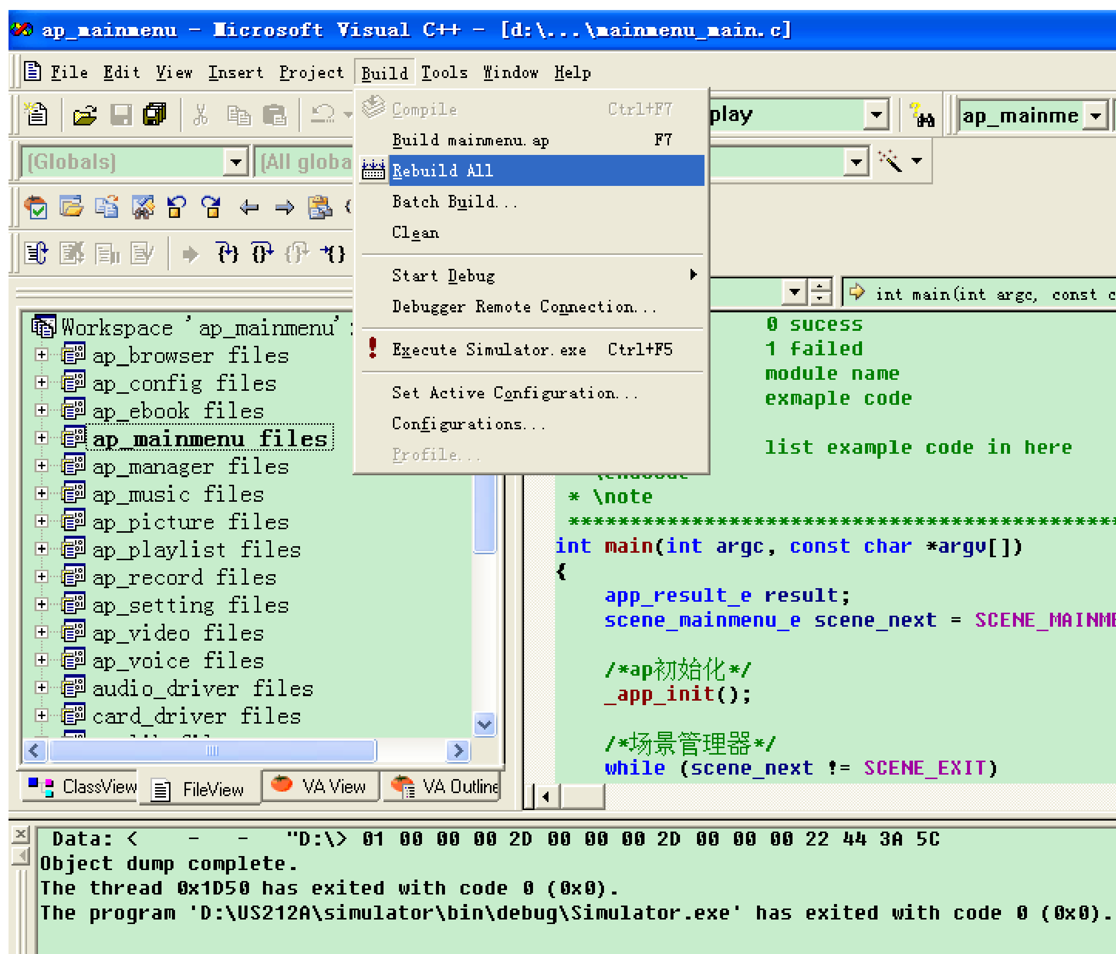


图 31-21 Rebuild All

Rebuild All 无误后，按 F5 开始调试。

4.2.5.2 启动调试（F5）

将待调试工程设为活动工程并编译、链接无误后，按 F5 开始调试，程序进入调试状态。程序将一直往下运行，碰到断点（如果有）则停下来：

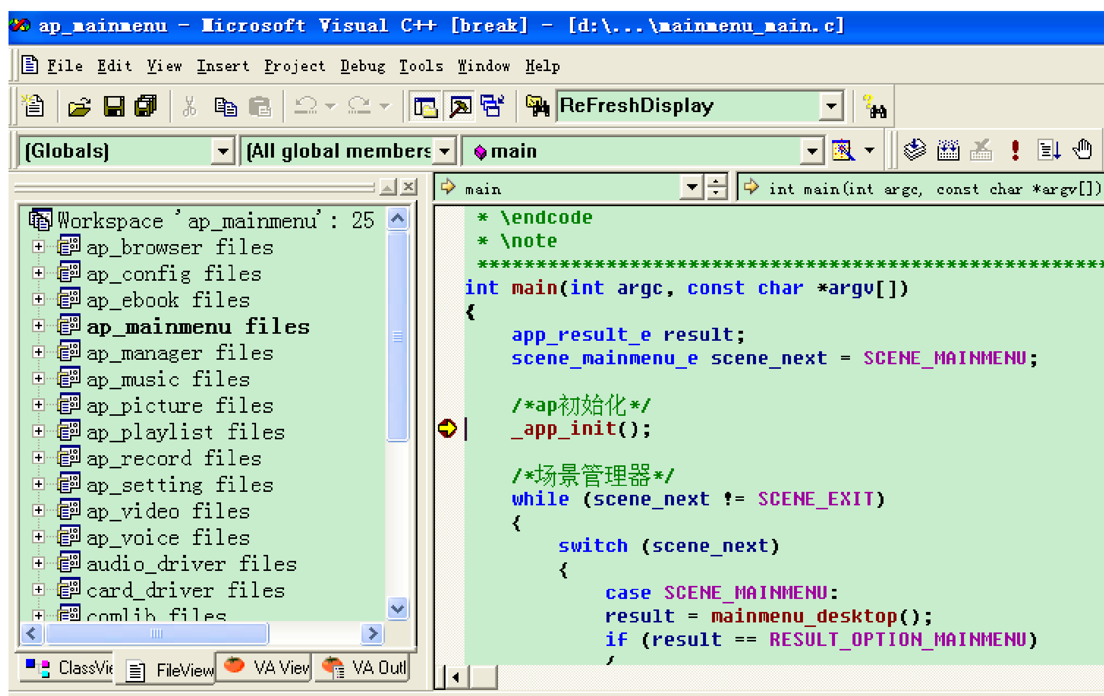


图 31-22 程序运行到断点

程序停在断点处后，可按 F10 或 F11 进行单步跟踪，也可按 F5 经过断点一直往后执行，直到碰到下一个断点再停下。

4.2.5.3 设置断点（F9）

设置断点是为了实时查看变量值和程序走向。可在按 F5 开始调试前设断点，也可在调试进行中设。将光标移至 VC 代码窗口的某一代码行，按 F9 设置断点。设置好断点后，断点所在行会有红色标记，如图 9-6 所示：

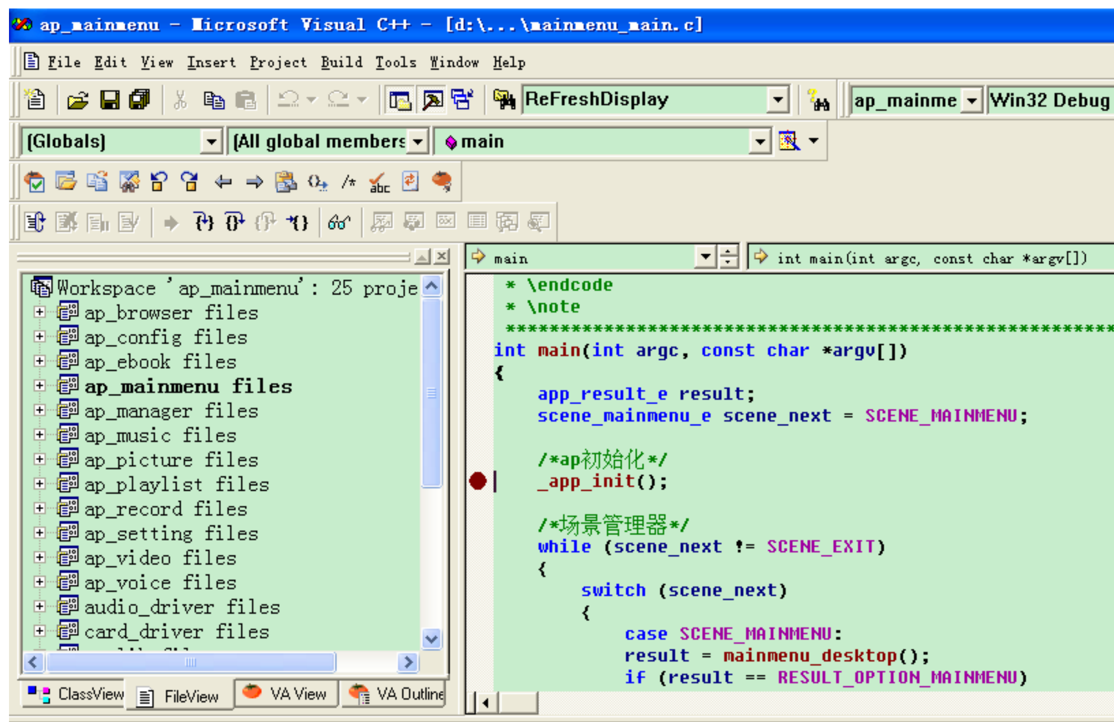


图 31-23 断点显示

4.2.5.4 单步调试（F10，查看变量和函数调用关系）

当程序在断点处停下来后，按 F10 让程序一条一条地执行，碰到函数则也当作一条语句执行，不进入函数内部。

可以选菜单 View→Debug Windows→Call Stack 查看各函数调用层次关系；可在右下窗口的“Name”栏输入程序中的变量名，其值将显示于“Value”栏。

版 权 所 有 侵 权 必 究

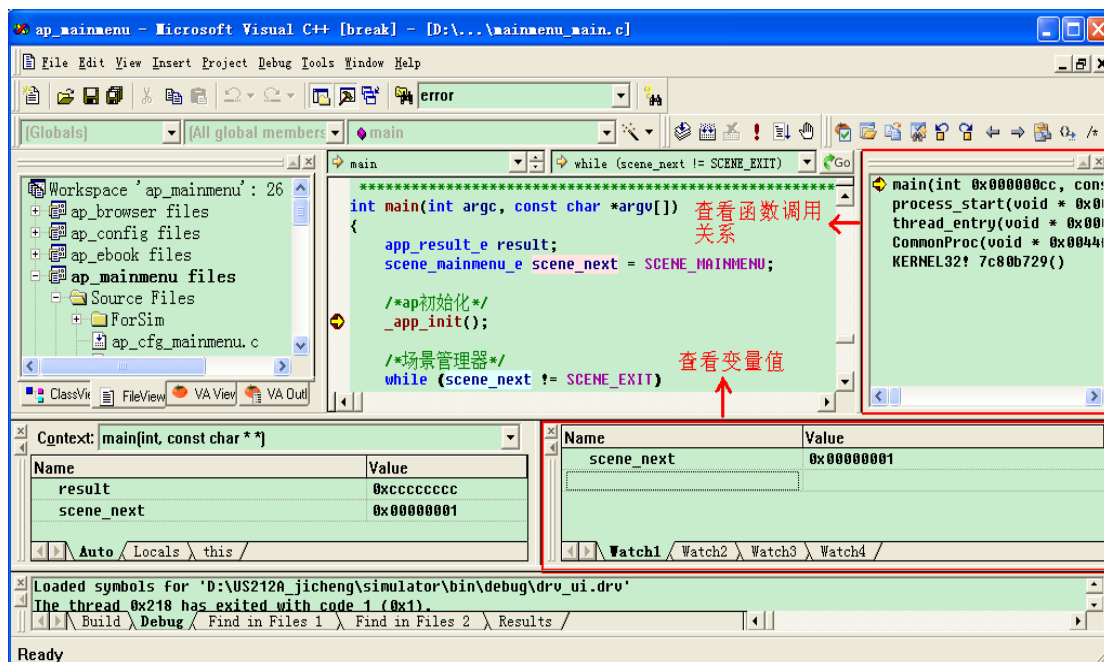


图 31-24 看堆栈和变量

4.2.5.5 进入函数体调试（F11）

当程序单步执行到某函数语句上时，按 F11 可以进入函数体跟踪；若按 F10 则执行其下一句，不进入函数体。如函数执行到_app_init 处：

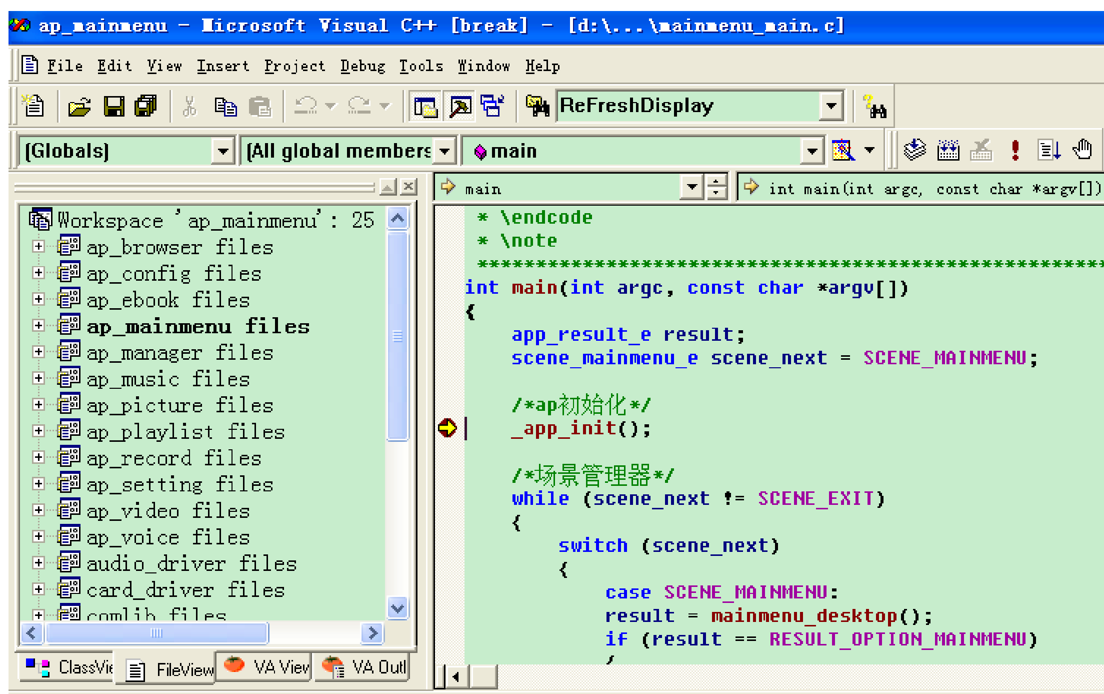


图 31-25 单步执行

此时若按 F11，程序跳转到函数体的入口处：

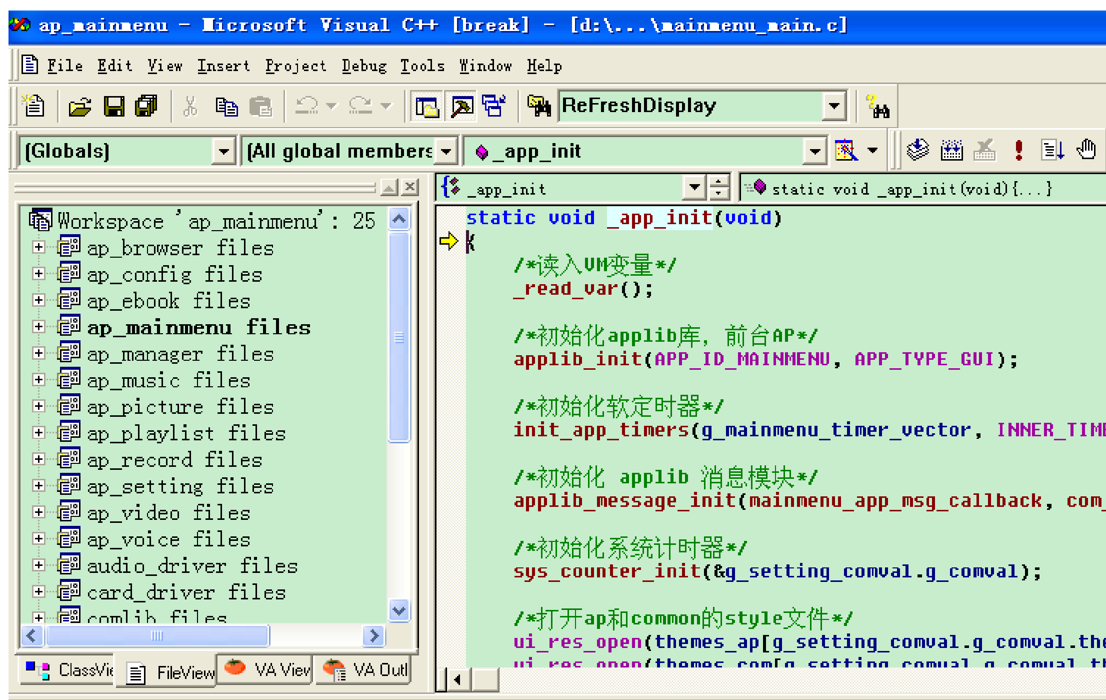


图 31-26 F11 进函数体

若按的是 F10，则程序跳到函数下一句运行：

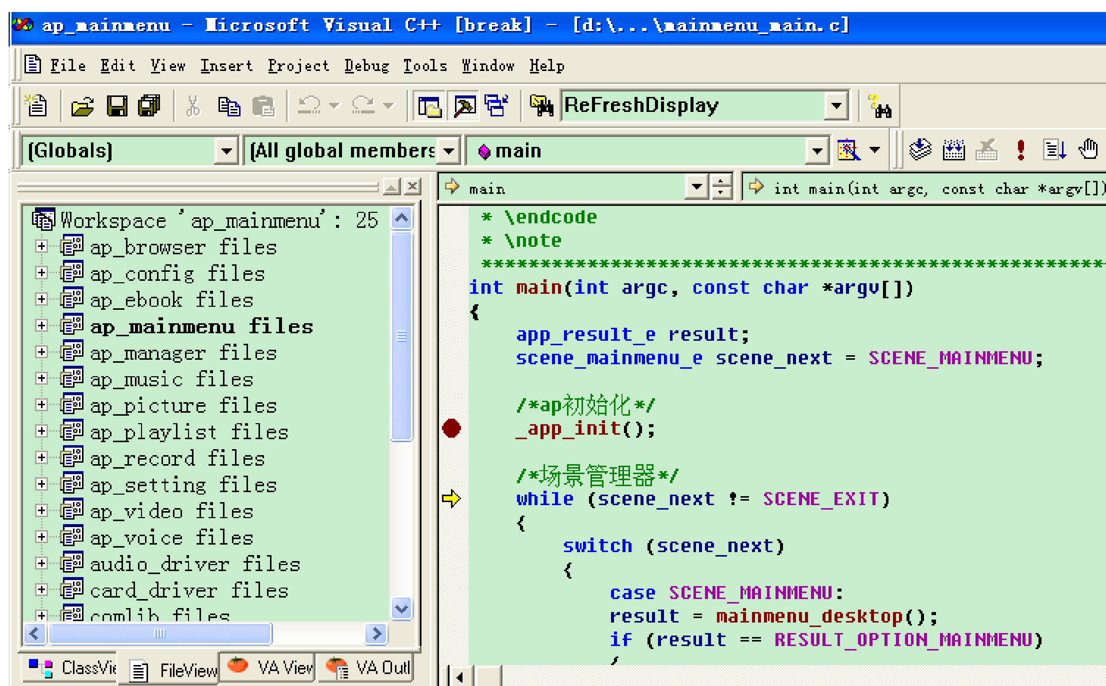


图 31-27 F10 跳过函数

注：按 F11 能进入函数体调试的前提是，函数体所在工程（工程 A）必须在当前 workspace 中。若工程 A 非正在调试的工程（工程 B），则要将工程 A 加入当前 workspace，并让

版权所有 侵权必究

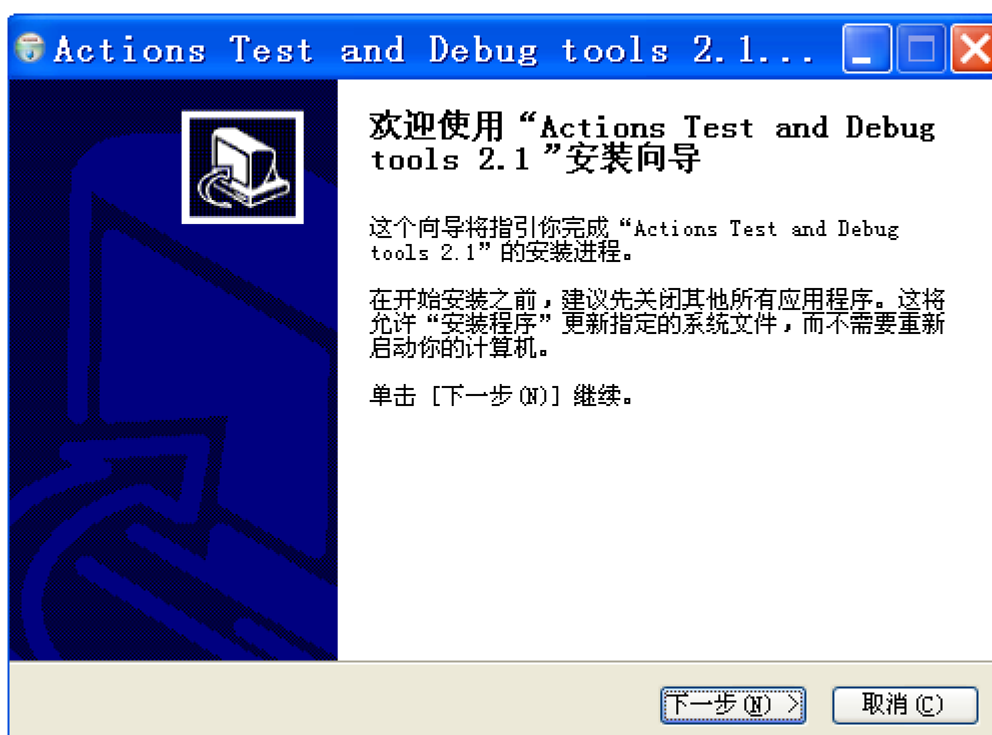
工程 B link 工程 A，link 方法见 7.2，然后重新调试。

4.3 调试方法介绍

4.3.1.1 UDI 和 Insigh

一、Actions Debug tool 安装

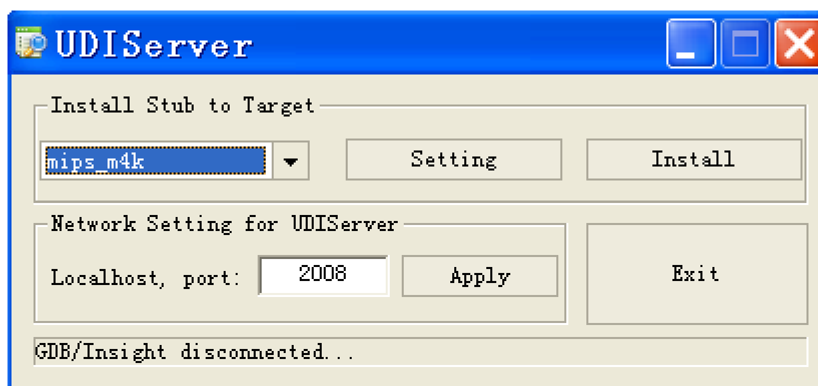
打开 Actions Debug tool 安装包，运行 ATD_V2.1_Setup.exe，如图所示：



继续按提示默认安装即可。

二、连接 UDI

插上 UDI，会提示 ADFU 连接，如果未安装 ADFU 驱动，请先安装 ADFU 驱动；然后打开 UDIServer 工具，install M4K 驱动

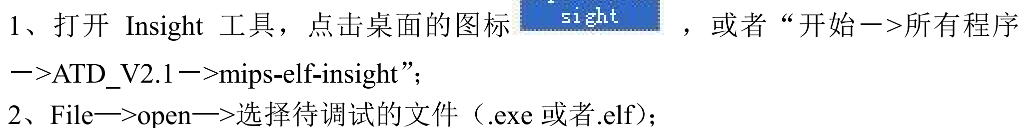


成功后，图标会显示为

将 UDI 连接到小机，正确连接后，UDI 的“DEM”指示灯会亮；

三、insight 调试

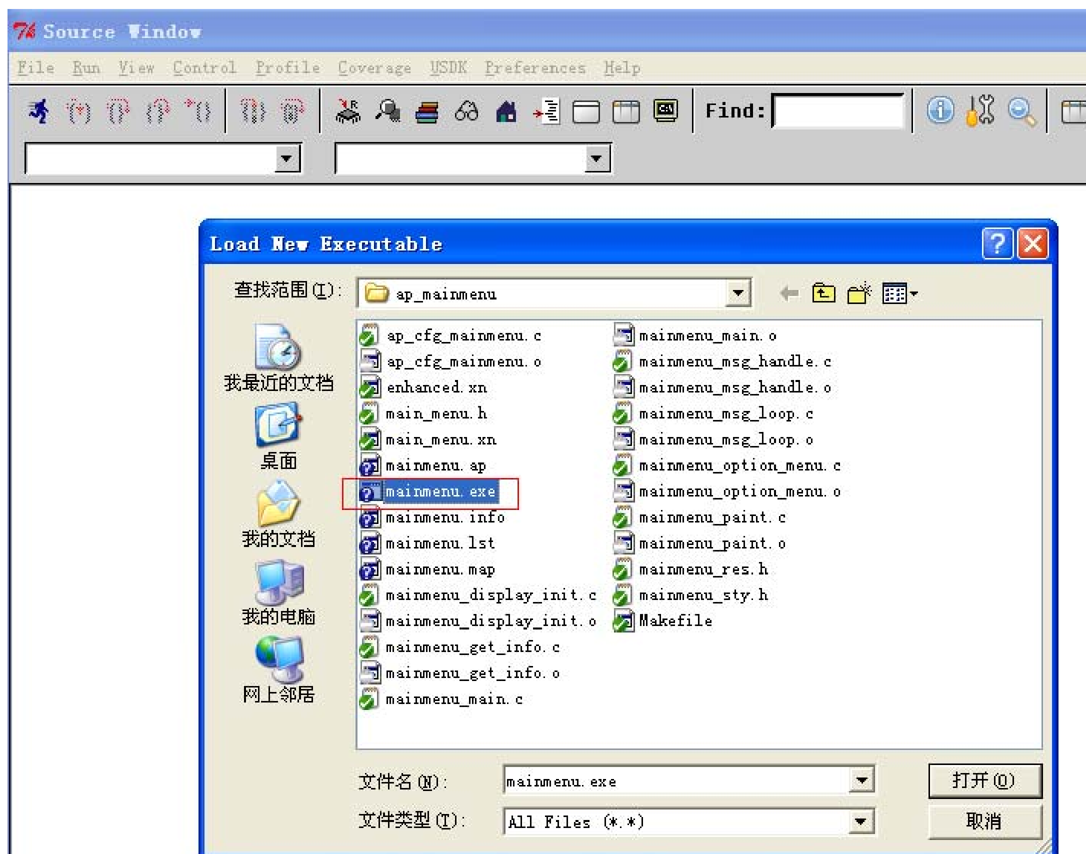
UDI 调试时，调试方法如下：



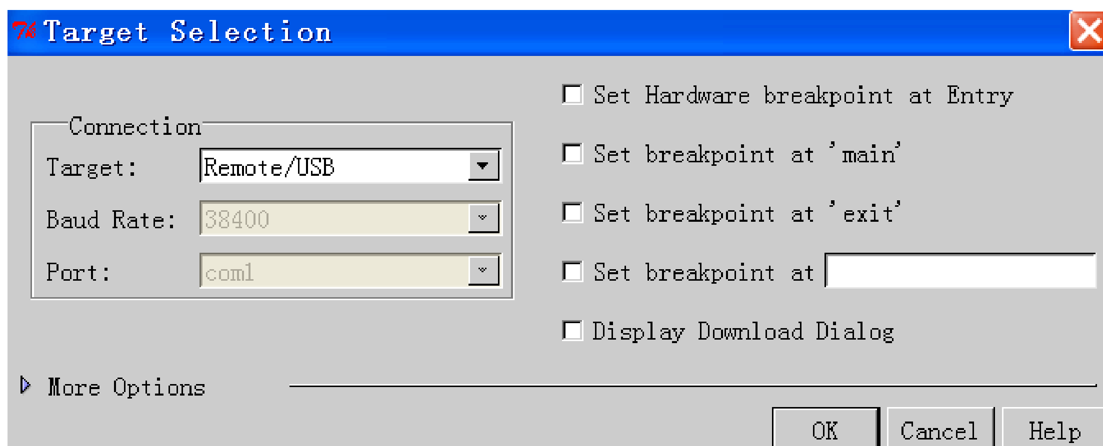
1、打开 Insight 工具，点击桌面的图标

，或者“开始—>所有程序—>ATD_V2.1—>mips-elf-insight”；

2、File—>open—>选择待调试的文件（.exe 或者.elf）；



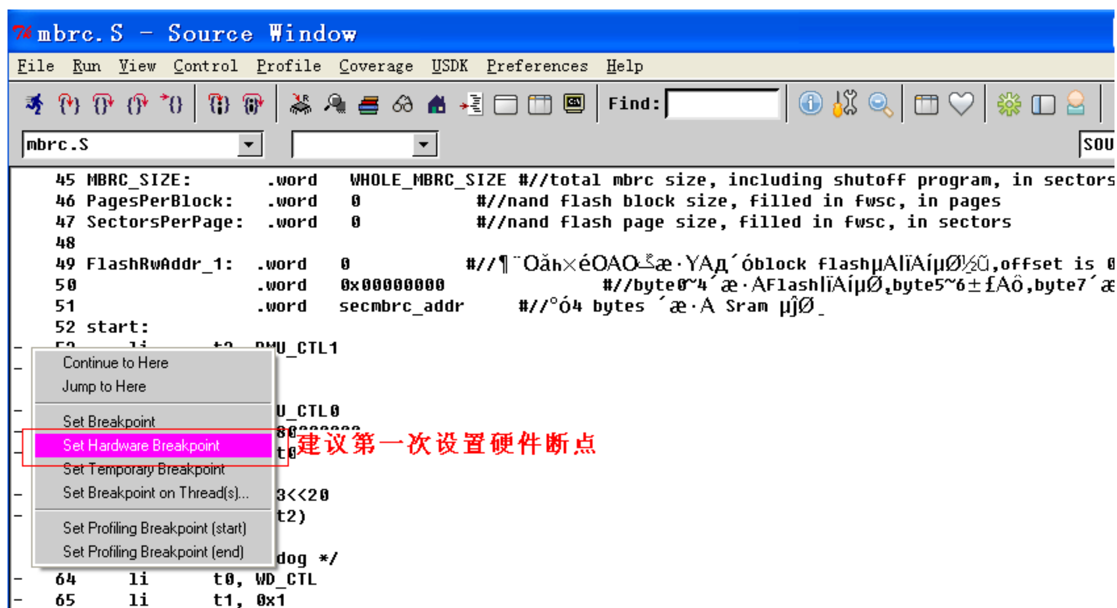
3、如果是第一次使用该工具，请查看设置，File—>Targer Settings，确认设置如下：



4、Run—>Connect to target；成功后会提示“Success Connect”；

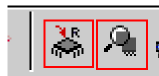
5、设置 BreakPoint，在代码行前，点击右键“Set Hardware Breakpoint”，建议第一次设置硬件断点；如果确定“从当前位置到断点之间执行的所有（包含子函数）”代码已经在内存中，可设置软件断点，建议都使用硬件断点（最多只能设置 4 个硬件断点）。

版 权 所 有 侵 权 必 究



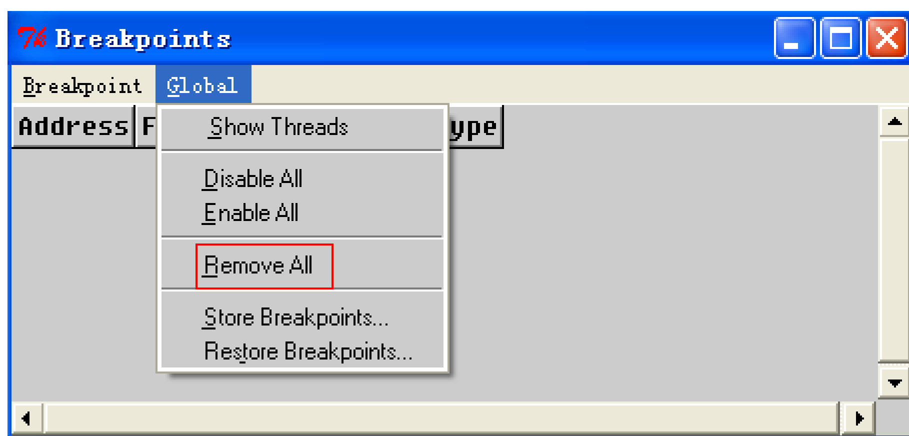
6、点击“continue”的图标 ，程序会运行到设置的断点处。

7、继续进行相关调试；

1）Insight 工具可以查看查看 register 和 memory，

2）每次重新开始调试，需要设置 register PC 为 0xBFC00000，系统从此地址开始启动；

3）在菜单 View　Breakpoints 中可以查看当前断点，如果需要清除断点，可以右键点击某个断点，选择 remove；如果想清除所有断点，可以在 Global　Remove All



4.3.1.2　串口打印

一、在需要打印的地方，使用函数 libc_print (unsigned char* s, unsigned int Data, unsigned char mode)

版 权 所 有　侵 权 必 究

s　　　　　　　　字符串地址
Data　　　　　　　数值
mode　　　　　　　打印模式，
　　　　　　　　　0，为仅打印字符串；
　　　　　　　　　1，仅打印数值；
　　　　　　　　　2，字符串和数值都打印
注意：一次最多打印 16 字符。

二、硬件 UART_TX 使用可使用 GPIOA17 或者 GPIOA21，可以根据实际方案设计，在 manager_main 中配置 UART_TX 使用的 GPIO。
比如，使用 GPIOA21，配置如下

```
*(uint32 *)(0x9fc19a88) = 0xFFFF1FFF;//GPIO A21
*(uint32 *)(0x9fc19a8c) = 0x00006000; //GPIO A21
```

使用 GPIOA17，配置如下

```
*(uint32 *)(0x9fc19a88) = 0xFFFFFFC7;//GPIO A17
*(uint32 *)(0x9fc19a8c) = 0x00000020; //GPIO A17
```

其中，0x9fc19a88--uart_tx_gpio_cfg_mask，0x9fc19a8c--uart_tx_gpio_cfg_value，实际上就是设置 MFP_CTL1。

三、打开串口打印软件，设置：
波特率——115200
数据位——8
停止位——1
校验位——none
流控制——none

4.3.1.3 UI 模拟器

UI 模拟器是一个 VC++ 工程，可以借助 VC++ 强大的开发调试平台进行调试。
具体使用方法请参考 UI 模拟器 一节。

5 case 基本结构及开发

首先介绍 case 运行环境，然后在此基础上展开讲解 case 基本结构及开发，给应用程序和 case 驱动程序开发人员一个统一全面的认识与开发能力。

版 权 所 有 侵 权 必 究

5. 1 Case 运行环境

Case 运行环境指的是 case 中的应用程序和 case 驱动程序所依赖的平台环境，Case 运行环境为它们提供了一个稳定、高效、功能强大、易于开发的基本架构。

Case 运行环境包括以下几个模块：

- PSP，包括 OS kernel、平台基本函数库 libc、平台基本驱动。其中平台基本驱动包括文件系统驱动、外部存储设备驱动如 nandflash、card、uhost 等
- 驱动统一入口 api.a
- 解码编码 DSP 模块
- 通用用户模块如 ID3 解析、歌词解析、文件选择器

5.1.1 PSP 模块概述

5.1.1.1 基本概念

OS Kernel

OS Kernel 的核心是实时操作系统 UCOS II，实现了实时抢占式的任务调度、任务间通信等核心功能；另外，OS Kernel 还实现了消息通信管理、API 管理、bank 管理、AP 管理、驱动管理、异常和中断管理、定时延时服务、等等。

应用、进程、线程、任务及其优先级

对于一般应用程序开发工程师来说，他所开发的对象就是应用。在 US212A 平台上，为了支持真正意义上的双应用场景，比如边看图片边播放音乐，也就是前台应用加后台应用，我们将应用分为 3 种类型，即前台应用（或称为 UI 应用）、后台应用（或称为引擎应用）及应用管理器。其中应用管理器是负责创建和回收前台应用和后台应用的特殊应用程序。

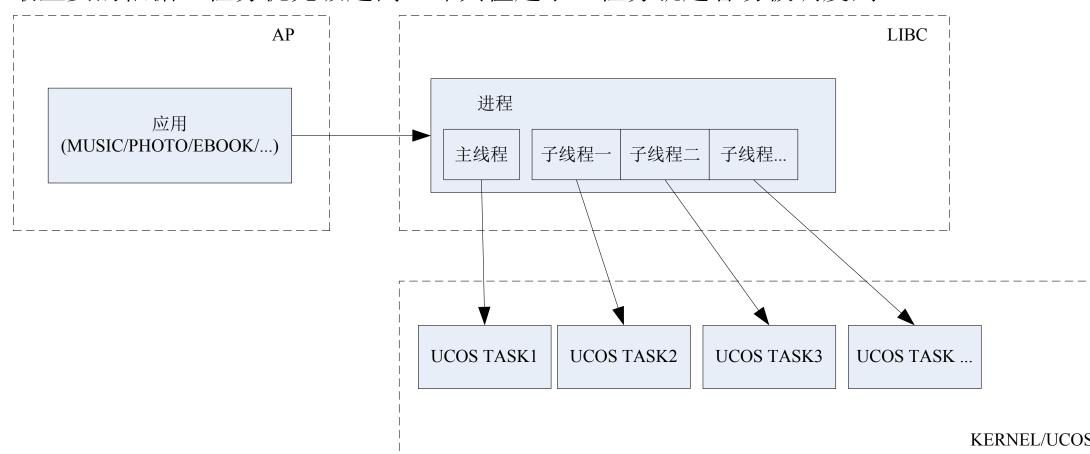
应用在 PSP 实现为进程，即拥有独占的代码和数据空间的主体，引进进程概念，是为了后面的多线程。

为了对应用进行分层设计与实现，比如图片应用分为与用户交互部分和底层图片解码部分、我们引进了**线程**的概念，并且把线程封装为 UCOS II 的任务概念。也就是说，每个应用/进程，可以分为多个线程实现，并最终在 UCOS II 核心由多个任务并行运行。这些线程

版 权 所 有 侵 权 必 究

共享进程所拥有的代码空间和数据空间，其中应用加载时由系统创建的线程叫主线程，其它线程叫子线程，是通过 libc 提供的线程管理接口维护的。对于线程，有一点必须特别注意的是，就是线程需要自己的栈空间，所以每个应用的子线程的数目是有限制的，目前设计为 1 个子线程。

任务是 UCOS 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。



驱动程序

驱动程序是对一些独立的公用模块的封装格式，比如文件系统驱动、外部存储设备驱动。另外，我们方便管理，我们把 kernel 和 libc 也封装为驱动程序。

对 Case 开发工程师来说，最关心的驱动程序是 Case 驱动程序，这些驱动是 Case 相关的，即为了适应不同的 Case ，可能需要修改到 Case 驱动。这些驱动包括 UI 驱动、LCD 驱动、KEY 驱动、Welcome 驱动以及 FM 驱动等。

basal & codec

US212A 的解码编码的 basal & codec 的链接机制同以前的方案有很大的区别，以前都是直接通过静态链接的方式进行连接；而 US212A 则把 basal & codec 分别独立打包为 *.al 格式的映像文件，当需要使用 basal & codec 时使用系统 API 将其加载进来，并封装为编码解码子线程，然后通过 basal & codec 命令式 API 接口对编码解码子线程间接访问控制。

basal 也称为 mmm，即多媒体中间件模块，是 ap 与 codec 交互的桥梁，它将 codec 的一些细节封装起来，以命令式 API 提供给 ap 调用，便于 ap 理解和使用。

5.1.1.2 PSP 接口说明书

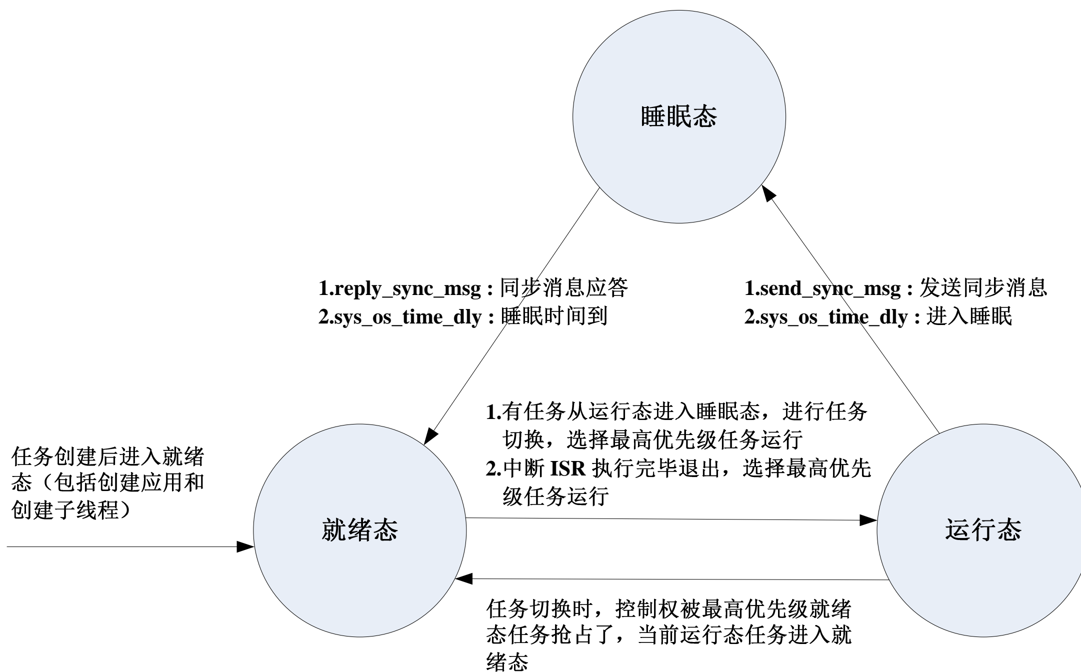
PSP 接口使用说明请参考《us212a_psp 接口说明书.chm》。

5.1.1.3 任务调度机制

US212A 的内核是 UCOS II，其基本单元是任务，任务的状态有 3 种：

- 睡眠态
- 就绪态
- 运行态

任务的这 3 中状态的转换关系如下图所示：



说明：

- 优先级在 case 中统一分配，提供，分配情况如下所示：

任务名称/类型	任务优先级（值越小优先级越高）	说明
后台解码子线程	AP_BACK_HIGH_PRIO **3**	比如 music 解码
后台应用主线程	AP_BACK_LOW_PRIO **4**	比如 music engine
前台解码/编码子线程	AP_FRONT_HIGH_PRIO **6**	比如 picture 解码、video 解码、record 编码
前台应用主线程	AP_FRONT_LOW_PRIO **7**	比如 picture 前台应用

版 权 所 有 侵 权 必 究

应用管理器	AP_PROCESS_MANAGER_PRIO **13**	即 ap_manager
IDLE 任务	OS_LOWEST_PRIO **15**	其他任务都没就绪时运行 IDLE 任务

- 由上表也可以看出，系统将解码/编码模块封装为线程，这样做可以进一步增强任务并行性，可以在解码/编码的同时接受用户的按键操作，比如对解码时间较长的对象取消。
- 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，我们要求每个消息处理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。
- Kernel 设计了一个 10ms 的 timer tick 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。

5.1.1.4 BANK 机制

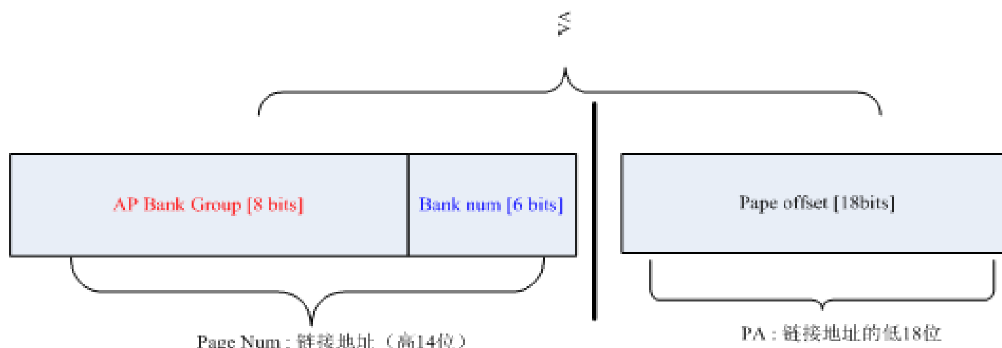
US212A 同以前的方案一样，都是内存紧缺型系统，其物理内存即 SRAM 大小都只有几十到一二百 KB，很明显是无法把整个驱动和应用加载到内存并常驻的，所以我们只好扩展物理内存空间，使用地址空间足够大的虚拟内存空间，并使用 BANK 机制对驱动和应用实现透明处理。

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 US212A 中，BANK 机制是一种完全的硬件机制，使用 MIPS 平台的 TLB miss 机制实现。

由于应用程序和驱动程序的物理内存空间分配有所区别，US212A 对这二者的 BANK 设计也是不同的。

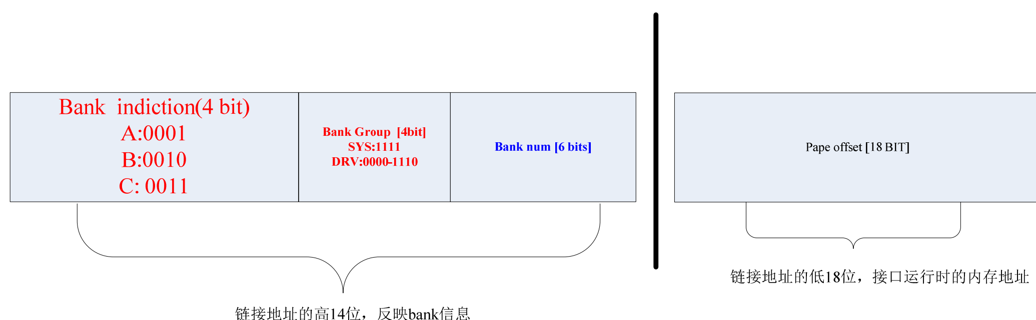
AP BANK

版 权 所 有 侵 权 必 究



- ❖ 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。

DRV BANK



- ❖ 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

版 权 所 有 侵 权 必 究

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。

5.1.1.5 API 机制

US212A 驱动程序 API 机制和 9X 以前的方案有较大的区别。因为 UA212A 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，包括 ap bank 和 bank a/b/c 。所以，US212A 的驱动程序 API，不再需要考虑 bank 机制，bank 这个概念对 API 来说完全透明。

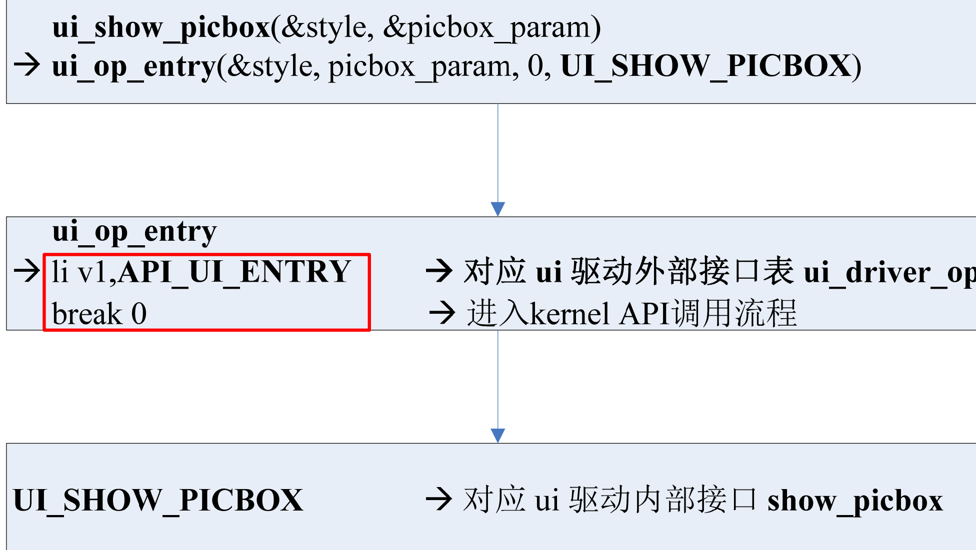
US212A 驱动程序 API 由 4 部分组成：

- 由 kernel 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。
- 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 kernel 驱动管理器，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 ui 驱动的 ui_show_picbox 为例）

版 权 所 有 侵 权 必 究

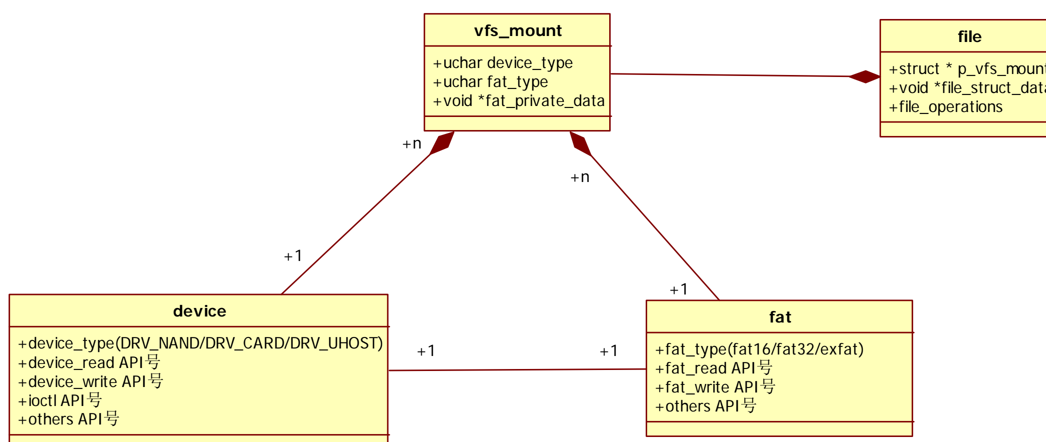


5.1.1.6 VFS 机制

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
- 各种具体的驱动对文件系统也是透明的，即文件系统无需关心具体的驱动，其与驱动的交互交由虚拟文件系统的块设备操作层(block layer)处理

US212A 的 VFS 设计并实现如下：



1. 每种设备驱动对应 device 结构。
2. 每种文件系统对应 fat 数据结构。

版 权 所 有　侵 权 必 究

3. 文件系统和设备驱动结合的数据结构是 **vfs_mount**，即管理具体设备以及部署其上的具体的文件系统。GL5110 每种设备都只有一份驱动存在于内存中，即使两个应用都同时访问这个设备。每种设备都对应于一种特定的具体的文件系统。当两个应用都通过同一个文件系统访问同一个设备时(听歌和看图片都是同一个盘)，为了让文件系统像存储驱动一样表现为一种纯粹的驱动，需要将每种应用对应的文件系统私有数据提取出来，放到 vfs_mount 结构中。例如，音乐应用的 file_location 在某个子目录中，而图片应用的 file_location 在根目录，那两个应用在调用 file_dir_next 接口获取下一个文件时将需要根据这些私有数据去操作，如文件系统当前访问的簇号等。VFS 层不关心这些私有数据，只会用一个 void*型的成员去记录，并由文件系统操作接口更新。明显，这些私有数据会是全局变量，空间将是在文件系统模块中分配，并把地址赋值给 vfs_mount 的 fat_private_data 成员。因此可以看到，设备和文件系统是 1 对 1 的关系，而 device 和 vfs_mount 是 1 对 N 的关系(两个应用都访问同一个设备时是 1 对 2)；同理 fat 跟 vfs_mount 也是 1 对 N 的关系。
4. GL5110 为了节省内存的考虑删除所有缓存数据结构，因此也需要寻找一种方法让应用能够透明编程，即应用也不需要关心底层具体的文件系统接口。因为 vfs_mount 数据结构有保存具体的文件系统信息，所以设备在挂载文件系统时可以返回 vfs_mount 的索引，应用在后续所有调用文件系统的接口中都应该加入这个索引。这样 VFS 层是可以根据索引去获得具体的文件系统接口的。
5. 设备挂载文件系统调用 drv_mount，这个函数中根据设备类型 DRV_XXX 获得设备的操作接口，并检测设备的数据上部署的是哪种有效的文件系统，并根据文件系统类型加载具体的文件系统驱动，并完成 vfs_mount 数据结构的填充，即绑定设备驱动和具体的文件系统，最后返回 vfs_mount 的索引。
6. GL5110 同样需要要达到文件系统开发不关心底层驱动的目标。因此 GL5110 会保留简单的 block layer。在文件系统的接口中都会有一个参数是设备驱动类型 device_type，文件系统接口最后通过 block layer 调用底层驱动。Block layer 会根据 device_type 去调用对应的驱动接口。
7. file 数据结构用于文件操作，如 read、write 或者 seek 等操作。vfs_mount 是 file 的成员，因为文件操作跟文件系统和底层驱动也有关系，其在 sys_open 时完成赋值。另外，每个文件句柄必然会包括文件指针等元素，这部分成员的空间由具体文件系统提供，并指示到 file 的数据成员 file_struct_data 中。

版权所有 侵权必究

5.1.2 解码编码概述

在 PSP 模块概述中的基本概念中，我们已经简单介绍了 basal & codec，这里将接下来说明整个平台的解码编码模块的概况。

5.1.2.1 解码编码总述

编码解码由 basal & codec 两部分组成，ap 通过 basal 对应的命令式 API 接口对 basal 间接访问控制，再通过 basal 来访问控制 codec。

US212A 平台下，basal 与对应的命令式 API 接口对应如下：

Basal(mmm)	命令式 API
音频解码 mmm_mp.al	int mmm_mp_cmd(void *, mmm_mp_cmd_t, unsigned int)
音频编码 mmm_mr.al	int mmm_mr_cmd(void *, mmm_mr_cmd_t, unsigned int)
图片解码 mmm_id.al	int mmm_id_cmd(void *id_handle, mmm_id_cmd_t cmd,unsigned int param)
视频解码 mmm_vp.al	int mmm_vp_cmd(void *vp_handle, unsigned int cmd, unsigned int param)

具体命令请参考《us212a_解码编码接口说明书.chm》。

US212A 平台支持双应用场景，比如前台看图片后台听音乐，这就需要同时支持前台 basal & codec 以及 后台 basal & codec ，所以系统提供的装载与卸载接口都会提供一个参数表示是否后台的 basal & codec ，必须正确填写，否则系统会出错。

basal & codec 同样支持 BANK 机制，完全由系统分配和控制，ap 用户无须关心。

basal & codec 的编码解码子线程是与对应的 ap 处于不同的链接模块的，不能共享代码和数据。

5.1.2.2 装载与卸载

basal 装载与卸载，由 ap 调用：

```
int sys_load_mmm(char *mmm_name, bool is_back_ap);
void sys_free_mmm(bool is_back_ap);
```

版 权 所 有 侵 权 必 究

codec 装载与卸载，由 basal 调用：

int sys_load_codec(char *codec_name, bool is_back_ap);

void sys_free_codec(bool is_back_ap);

以 ap_picture 为例，basal 装载与卸载，以及解码编码子线程的创建与销毁，basal 命令式 API 调用的完整流程如下：

- 加载图片解码 basal (image decode)

sys_load_mmm("mmm_id.al", FALSE);

- 创建图片解码子线程

mmm_id_cmd(&picture_handle, MMM_ID_OPEN, 0);

- 调用命令式 API

mmm_id_cmd(picture_handle, MMM_ID_SET_FILE, & g_picture_userinfor);

- 销毁图片解码子线程

mmm_id_cmd(picture_handle, MMM_ID_CLOSE, 0);

- 卸载图片解码 basal

sys_free_mmm(FALSE);

5.1.2.3 解码编码接口说明书

解码编码接口使用说明请参考《us212a_解码编码接口说明书.chm》。

5.1.3 enhanced 模块概述

enhanced 模块是相对独立于具体应用的中间层，位于 ap 与 psp 之间，用来对一些较大的较为固定的功能进行封装，以便管理和使用。

5.1.3.1 enhanced 模块总述

US212A 的 enhanced 模块包括 3 个部分：

- fsel 文件选择器：以应用视图操作文件系统、播放列表和收藏夹，进行目录浏览、文件选择等操作；US212A 支持双应用场景，比如前台看图片后台听音乐，所以可能会前后台都需要一个独立的 fsel 模块；所以 fsel 在设计时按照场景及功能进行划分，具体在下面详细说明。
- id3 解析模块：以统一的接口解析获取多媒体文件的标签信息，并消除各种文字编

版 权 所 有 侵 权 必 究

码的差异性。

- lrc 歌词解析模块：按照歌词时间戳排序，以时间戳为关键字获取歌词，lrc 歌词解析模块不再与字符显示的概念相关，ap 层获取到一行歌词后，根据显示区域自行实现歌词字符串显示。

enhanced 模块为所有前台应用和后台应用共用，为了更方便 ap 使用，我们以场景和功能为依据，提供多个链接脚本模板给应用直接使用，应用只需要在 makefile 中把适合的 xn 文件添加到 LD_SCRIPT 参数值列表中即可。

链接脚本模板存放在目录 psp_rel\usermodule\enhanced\case_link_xn 下面。

enhanced 模块划分及链接脚本模板列举如下：

Enhanced 模块	**模块描述**	**链接脚本**
Fsel-目录浏览	实现文件系统、播放列表、收藏夹的目录和文件浏览	bs_link.xn
Fsel-文件选择	实现文件系统、播放列表、收藏夹的文件选择	fsel_link.xn
id3 解析	实现各种编码类型的多媒体文件的标签信息的获取	id3_link.xn
歌词解析	实现 lrc 歌词文件的解析	lrc_link.xn

说明：case_link_xn 目录下其他链接脚本是以上几个脚本的组合或变体，是为了更加适合具体的应用使用而进行合并及修改而成。

5.1.3.2 Enhanced 接口说明书

Enhanced 接口使用说明请参考《us212a_enhanced 接口说明书.chm》。

5.1.4 模块调用关系

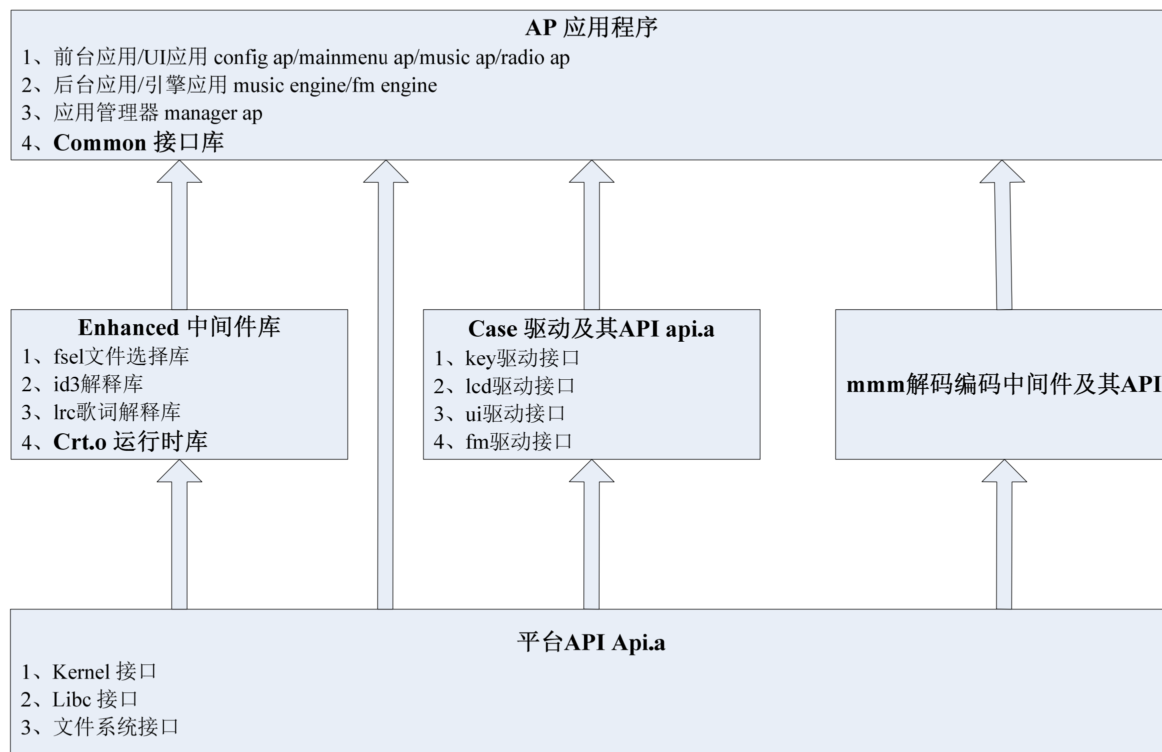
Case 运行环境以及 Case 本身，概括起来可以分为以下几个模块：

1. AP 应用程序
2. Case 驱动及其 API
3. Enhanced 中间件库及其他接口
4. 解码编码模块及其 API

版 权 所 有 侵 权 必 究

5. PSP 模块及其 API

这些模块之间的调用关系如下图所示：



说明：

- Case 驱动是与具体的 case 相关的驱动，代码放在 case/drv 目录下，AP 通过 Api.a 静态库中对应的 *_op_entry.o 接口进行调用。
- Common 接口库是 AP 的一些公共功能的封装的集合，代码放在 case/common 目录下，独立编译，以 *.o 这种 ELF 文件给 AP 链接。
- Crt.o 运行时库是一个特殊的库文件，AP 加载后会先运行 Crt.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。
- Mmm 解码编码中间件
- 平台 API 接口是独立于具体方案的功能接口，以 Api.a 静态库中对应的 *_op_entry.o 接口方式对外提供系统服务等。

版权所有 侵权必究

5.2 代码目录导航

US212A 的发布包中的代码目录如下所示，包括 case、case_simulator、psp_rel 等，在下面将简单介绍这 3 部分，以让我们对方案的代码目录有一个大致了解，方便后面展开说明。

名称	大小	类型	修改日期
case		文件夹	2012-12-28 9:05
case_simulator		文件夹	2012-12-27 11:43
psp_rel		文件夹	2012-12-28 10:46
read me.txt	1 KB	文本文档	2012-9-20 14:21

5.2.1 case 目录导航

case 目录即 US212A 方案的 AP 和 case 驱动的源代码等的工作目录。

名称	大小	类型	修改日期
ap		文件夹	2012-12-28 10:48
cfg		文件夹	2012-12-28 9:04
drv		文件夹	2012-12-28 9:04
fwpkg		文件夹	2012-12-28 9:04
inc		文件夹	2012-12-28 9:05
lib		文件夹	2012-12-28 9:04
resource		文件夹	2012-12-27 11:43
tools		文件夹	2012-12-27 11:43
build_sty.bat	5 KB	MS-DOS 批处理文件	2012-12-25 11:43

case 目录各部分说明如下表所示：

目录	说明
ap	AP 和 common 源代码及编译链接脚本
cfg	Case 编译链接公共文件
drv	Case 类驱动，包括 fm 驱动， key 驱动，lcd 驱动，ui 驱动，以及 welcome 驱动
fwpkg	Case 固件打包相关文件
inc	Case 公共头文件
lib	Common obj 文件，供 AP 直接链接
resource	AP 和 common UI Editor 工程及其资源文件
tools	Case 辅助开发工具，包括 UI Editor 工具，fw modify 工具

build_sty.bat	从新生成所有 AP 和 Common 的 UI Editor 工程的*.sty 文件

case/ap 目录展开，看到 US212A 方案所有的 AP 和 common 的目录。

名称	大小	类型	修改日期
ap_alarm		文件夹	2012-12-28 9:02
ap_browser		文件夹	2012-12-28 9:02
ap_config		文件夹	2012-12-28 9:02
ap_ebook		文件夹	2012-12-28 9:02
ap_mainmenu		文件夹	2012-12-28 9:02
ap_manager		文件夹	2012-12-28 9:02
ap_music		文件夹	2012-12-28 9:02
ap_picture		文件夹	2012-12-28 9:02
ap_playlist		文件夹	2012-12-28 9:02
ap_radio		文件夹	2012-12-28 9:03
ap_record		文件夹	2012-12-28 9:03
ap_setting		文件夹	2012-12-28 9:03
ap_tools		文件夹	2012-12-28 9:03
ap_udisk		文件夹	2012-12-28 9:03
ap_upgrade		文件夹	2012-12-28 9:03
ap_video		文件夹	2012-12-28 9:03
common		文件夹	2012-12-28 9:03
fm_engine		文件夹	2012-12-28 9:03
music_engine		文件夹	2012-12-28 9:03

case/ap 目录下各 AP 及 common 说明如下表所示：

目录	说明
ap_manager	应用管理应用，创建和回收其他 AP
ap_config	开机与关机应用
ap_mainmenu	主应用，提供可视化 AP 入口
ap_music	音乐应用前台，与 Music Engine 配合实现完全的音乐功能
ap_picture	图片应用
ap_video	视频应用
ap_record	录音应用
ap_radio	收音应用前台，与 FM Engine 配合实现完全的收音功能
ap_ebook	电子书应用
ap_browser	浏览器应用，选择音频，视频，图片，电子书等文件进行播放
ap_playlist	播放列表创建 AP，创建和更新播放列表
ap_tools	工具应用，提供秒表、日历、闹钟、数据交互 等功能

版 权 所 有 侵 权 必 究

ap_alarm	闹钟应用
Ap_setting	设置应用，用于配置系统参数
ap_udisk	U 盘应用，作为 MSC 设备与 PC 连接并进行数据传输
ap_upgrade	U 盘自动升级
music_engine	音乐应用后台引擎，受 Music AP 控制实现音乐功能
fm_engine	收音应用后台引擎，受 Radio AP 控制实现收音功能
common	应用程序公共功能模块，在下表进一步展开介绍

case/ap/common 目录展开如下图所示。

名称	大小	类型	修改日期
applib		文件夹	2012-12-28 9:03
common_func		文件夹	2012-12-28 9:03
common_misc		文件夹	2012-12-28 9:03
common_ui		文件夹	2012-12-28 9:03
data		文件夹	2012-12-28 9:03
common_aphead.xn	2 KB	XN 文件	2012-12-25 11:42
common_engine.xn	4 KB	XN 文件	2012-12-25 11:42
common_front.xn	13 KB	XN 文件	2012-12-25 11:42
common_front_no_selector.xn	12 KB	XN 文件	2012-12-25 11:42
Makefile	1 KB	文件	2012-12-25 11:42

case/ap/common 目录下各部分说明如下表所示：

目录	说明
applib	应用基本功能接口库，包括应用（进程）管理接口、应用级定时器管理接口、消息通信管理接口
Common_func	应用公共子功能接口库，包括字符串处理（包括格式化字符串，省略显示形式，短名加点，显示调试信息），路径记忆，配置项解释，配置菜单解释，按键映射处理，声音输出管理，按键消息预处理，应用消息预处理，等等
Common_misc	应用公共子功能杂项功能接口库，包括系统定时器，应用睡眠，默认消息处理，屏幕方向设置，背光亮度映射，电池电量映射，闹钟消息处理，应用私有消息预处理，天线检测处理，等等
Common_ui	应用公共控件，包括菜单列表，文件浏览器及删除文件，对话框，参数设置框，状态栏，动画显示，按键锁，屏保，关机对话框，USB 连接对话框，文本显示框，音量条，等等

版 权 所 有 侵 权 必 究

data	Common 模块全局数据，其中 applib_globe_data.c 是所有应用共享的全局数据
*.xn 和 makefile	Common 模块的 make 脚本及供 AP 链接用的链接脚本，其中 common_aphead.xn 里面的输出段是要直接添加到 AP 自己的链接脚本去的；common_engine.xn 是后台应用的链接脚本；common_front.xn 是功能齐全的前台应用的链接脚本；common_front_no_selector.xn 是不需要用到文件选择器的前台应用的链接脚本

case/inc 目录展开如下图所示：

名称	大小	类型	修改日期
alarm_common.h	3 KB	C Header file	2012-12-25 11:43
app_msg.h	20 KB	C Header file	2013-2-27 14:01
applib.h	18 KB	C Header file	2012-12-25 11:43
case_include.h	2 KB	C Header file	2012-12-25 11:43
case_type.h	16 KB	C Header file	2012-12-27 19:42
common_func.h	12 KB	C Header file	2012-12-25 11:43
common_res.h	13 KB	C Header file	2012-12-25 11:43
common_sty.h	6 KB	C Header file	2012-12-25 11:43
common_ui.h	22 KB	C Header file	2012-12-27 19:42
config_id.h	3 KB	C Header file	2012-12-25 11:43
display.h	33 KB	C Header file	2012-12-25 11:43
fm_interface.h	4 KB	C Header file	2012-12-25 11:43
fmengine.h	4 KB	C Header file	2012-12-25 11:43
gui_msg.h	4 KB	C Header file	2012-12-25 11:43
I2C.h	2 KB	C Header file	2012-12-25 11:43
key_interface.h	2 KB	C Header file	2012-12-25 11:43
lang_id.h	2 KB	C Header file	2012-12-25 11:43
lcd_driver.h	11 KB	C Header file	2012-12-25 11:43
music_common.h	6 KB	C Header file	2012-12-25 11:43
setting_common.h	6 KB	C Header file	2012-12-25 11:43
vm_def.h	4 KB	C Header file	2012-12-25 11:43

case/inc 目录下各部分说明如下表所示：

目录	说明
app_msg.h	应用私有消息、系统消息、gui 映射事件等定义头文件
applib.h	Common->applib 相关接口和类型头文件
case_include.h	Case 总头文件，包含其他头文件
Case_type.h	应用类型头文件，包括应用 ID，app_result_e 等
Common_func.h	Common->common_func 相关接口和类型头文件

Common_res.h 和 Common_sty.h	Common 可配置化 UI 工程生成头文件
Common_ui.h	Common->common_ui 和 common_misc 相关接口和类型头文件
Config_id.h	固件配置化配置项 ID 定义头文件
Display.h Fm_interface.h Key_interface.h Lcd_driver.h	驱动外部接口和类型头文件
gui_msg.h	按键消息相关定义头文件
setting_common.h	Case 环境变量结构体定义头文件

case/fwpkg 目录展开如下图所示：

名称	大小	类型	修改日期
ap		文件夹	2012-12-28 9:04
drv		文件夹	2012-12-28 9:04
font		文件夹	2012-12-28 9:04
mcg		文件夹	2012-12-28 9:04
sty		文件夹	2012-12-28 9:05
alarm1.mp3	8 KB	MP3 Format Sound	2012-12-25 11:43
build_mcg.bat	1 KB	MS-DOS 批处理文件	2012-12-25 11:43
buildfw.bat	1 KB	MS-DOS 批处理文件	2012-12-25 11:43
config.bin	2 KB	BIN 文件	2012-12-25 11:43
config.spc	10 KB	PKCS #7 证书	2012-12-25 11:43
config.txt	5 KB	文本文档	2012-12-25 11:43
fmtool.cfg	1 KB	Microsoft Offic...	2012-12-25 11:43
fwimage.cfg	5 KB	Microsoft Offic...	2012-12-27 19:42
legal.txt	4 KB	文本文档	2012-12-25 11:43
m_type.txt	1 KB	文本文档	2012-12-25 11:43
welcome.res	40 KB	RES File	2012-12-25 11:43

case/fwpkg 目录下各部分说明如下表所示：

目录	说明
ap	应用映像文件 *.ap
Drv	Case 驱动影响文件 *.drv
font	字库文件及 codepage 转换表
Mcg	应用可配置化菜单配置文件 *.mcg
Sty	应用可配置化 UI 配置文件 *.sty

Alarm1.mp3	闹钟默认闹铃 mp3 文件
Build_mcg.bat	应用可配置化菜单配置文件更新批处理，当字符串 ID 或者菜单回调函数地址发生变化时执行更新
Buildfw.bat	固件生成批处理
Config.bin 、config.spc、config.txt	固件配置化 bin 文件、固件配置化规格文件（modify 工具的规格书）、固件配置化脚本，要修改固件配置，比如默认值，通过修改 config.txt 脚本，然后执行 genconfig.bat 批处理更新；要添加新的配置项，必须同时修改 config.txt 和 config.spc 文件
Fwimage.cfg	固件打包配置脚本
Legal.txt 和 m_type.txt	供文本显示框直接显示的文本文件，前者是法律信息，后者是 mp3 类型
Welcome.res	欢迎界面资源图片数据文件

case/tools 目录展开如下图所示：

名称	大小	类型	修改日期
Gen_config		文件夹	2012-12-27 11:43
UI-EDITOR		文件夹	2012-12-27 11:43
msxml.msi	5,166 KB	Windows Install...	2012-12-25 11:43
setting.ini	1 KB	配置设置	2012-12-25 11:43
TreeLayer.exe	992 KB	应用程序	2012-12-25 11:43

case/tools 目录下各部分说明如下表所示：

目录	说明
Gen_config	应用配置脚本解释工具，更新配置脚本后，运行 genconfig.bat 即可
UI-Editor	应用可配置化 UI 编辑工具，目录有个简单的说明文件 ui_editor 工具使用说明.doc
TreeLayer.exe	应用可配置化菜单编辑工具

5.2.2 case_simulator 目录导航

case_simulator 目录即 UI Simulator 工具的工作目录。

版 权 所 有 侵 权 必 究

名称	大小	类型	修改日期
Apps		文件夹	2012-12-27 11:43
bin		文件夹	2012-12-27 11:43
CaseDrivers		文件夹	2012-12-27 11:43

case_simulator 目录下各部分说明如下表所示：

目录	说明
Apps	UI simulator 各 AP 工程
bin	UI simulator 各种目标文件和辅助工具
CaseDrivers	Case 层驱动工程，包括 key 驱动，lcd 驱动，UI 驱动等工程

5.2.3 psp_rel 目录导航

psp_rel 目录即平台相关目标模块、目标库、enhanced 模块源码及目标库、平台工具、以及头文件等。

名称	大小	类型	修改日期
bin		文件夹	2012-12-28 9:17
cfg		文件夹	2012-12-28 10:47
include		文件夹	2012-12-28 9:17
lib		文件夹	2012-12-28 9:17
tools		文件夹	2012-12-27 11:45
usermodule		文件夹	2012-12-28 9:17

pep_rel 目录下各部分在下面的表格简单说明。

目录	说明
bin	AFI.bin，是 psp 最终用于链接成固件的目标文件
cfg	Psp 编译链接公共文件
include	Psp 公共头文件
lib	Psp 可链接目标文件 obj 集合，包括 usb，enhanced 等
tools	Psp 相关辅助开发工具
usermodule	中间件源代码，包括文件选择器，歌词解释器，ID3 解释器等

5.3 应用管理器 ap_manager

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。

5.3.1 ap_manager 的地位与作用

US212A 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 US212A 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。

设备上电后，boot 启动，加载 kernel 并运行，kernel 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。

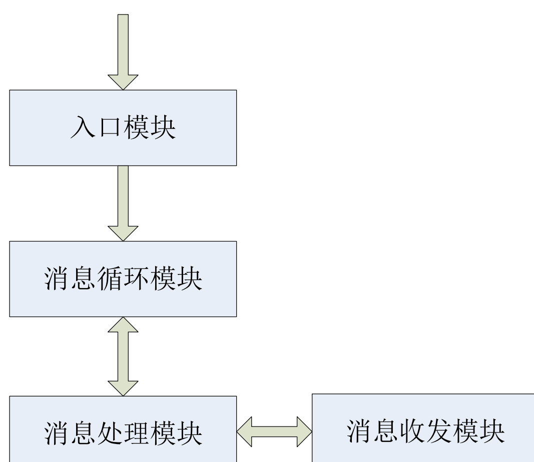
Ap_manager 作为应用程序的调度核心，具体的功能如下：

1. 开机时创建 ap_config ，执行开机流程。
2. 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。
3. 接收前台应用创建后台应用的同步消息，同步创建后台应用。
4. 接收前台应用杀死后台应用的同步消息，同步杀死后台应用。
5. 关机处理，杀死当前前台应用和后台应用，然后创建 ap_config，执行关机流程。
6. 进入 Udisk 应用，杀死当前前台应用和后台应用，然后创建 Udisk 应用。

5.3.2 Ap_manager 的设计要点

下图说明各个模块的调用关系，应用从入口模块进来，进行必要的初始化动作并创建 ap_config 后，就进入消息循环模块循环查询私有消息，如果获取到消息，由消息处理模块处理，此时需要消息的同步或者交互，需要借助消息收发模块实现。消息处理模块属于消息循环模块的一部分，而消息收发模块只是一些消息通信接口的集合。

版 权 所 有 侵 权 必 究



应用的启动：由系统创建，系统启动加载 OS 后，就会创建 ap_manager 应用。

应用的退出：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，其设计要点如下：

Ap_manager 避免调用 Common 的接口，我们把一些必须的接口在 ap_manager 重新实现以更加适应 ap_manager 代码空间非常局限的现实条件，这些接口包括：applib_init 、open_config_file 、get_config_default 、send_async_msg 、broadcast_msg 、reply_sync_msg 。

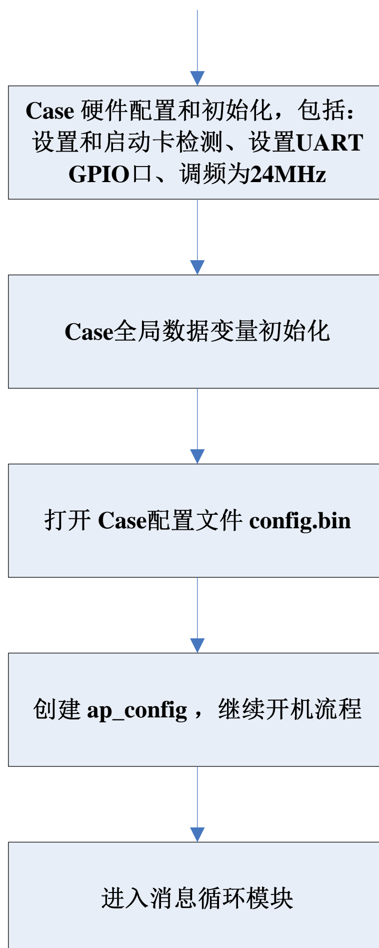
注意：send_async_msg 的实现是不安全的，它发送给后台应用消息时，只可以正确处理 music_engine 和 fm_engine，所以如果方案添加了新的后台应用，那么必须修改 send_async_msg 的实现，使之能够正确发送给新增的后台应用消息。

Ap_manager 只会在系统启动时创建一次，所以其入口模块也就是在整个系统生命周期中只会执行一次，所以我们把一些只需要执行一次的动作放在该模块，包括 Case 硬件配置和初始化、Case 全局数据变量初始化、打开 Case 配置文件 config.bin 等。

5.3.3 Ap_manager 入口模块

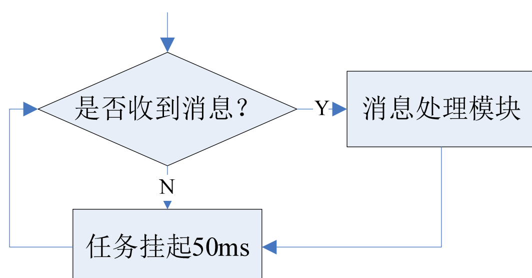
Ap_manager 应用创建后，进入入口模块，该模块负责 Case 硬件配置和初始化、Case 全局数据变量初始化、打开 Case 配置文件 config.bin、创建 ap_config 以继续开机流程，其流程如下图所示：

版 权 所 有 侵 权 必 究



5.3.4 Ap_manager 消息循环模块

Ap_manager 的消息循环模块功能简单，主流程如下：



说明：

1. 因为 ap_manager 只需要接受并处理发给 ap_manager 的消息，所以我们直接调用 sys_mq_receive(MQ_ID_MNG, &pri_msg) 接口从 ap_manager 的私有消息队列接收消息。

2. 因为 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，并且每次消息循环都会挂起 50ms，减轻多任务系统的调度压力。

消息循环收到消息后进行消息分发处理，我们处理的消息包括：

MSG_CREAT_APP：异步创建应用，此消息是异步消息，收到此消息后需要等待发送此消息的应用退出后，才创建需要创建的应用。

MSG_CREAT_APP_SYNC：同步创建应用，此消息是前台应用创建后台引擎使用，是同步消息，如果创建成功，回复 MSG_REPLY_SUCCESS，如果创建失败，回复 MSG_REPLY_FAILED。

对于需要创建何种应用和应用的入口参数，有以下约定：

私有消息内容 msg_apps_t->content->data[4] 的第一个字节为应用的 ID，第二个字节为入口的参数。

MSG_KILL_APP_SYNC：用于杀死后台的同步消息，因为内存的限制，目前方案只允许有 1 个后台运行，对于空间冲突的应用场景，需要发送此消息来杀死后台。

MSG_POWER_OFF：关机消息，用户按下关机按键，或者电池电量不足，应用会发出此消息，manager 收到此消息，需要查询后台的状态，然后创建 config 应用进行关机，关机前后台的类型和状态需要通过应用的入口参数传给 config 应用（因为目前的规格是如果关机时存在后台引擎，那么下次开机需要恢复该后台引擎对应的前台应用，是否播放与关机前的播放状态一致，如果不存在后台引擎，下次开机回到 mainmenu 应用，并且应用图标停留在第一个应用，所以 config 应用需要知道关机时后台的信息，但是在 config 运行时，后台已经被杀死了，无法得知状态，因此需要 manager 通过参数告知）。

MSG_USB_TRANS：进入 u 盘的消息，用户通过 usb 线连接电脑并确认需要进入 u 盘后，由 common 发出此消息，此时 manager 同样需要查询后台的状态，并将后台的类型和状态通过应用的入口参数传给 config 应用，因为目前方案规格是进入 u 盘后如果没有更新过文件，退出 u 盘后，如果之前后台引擎存在，则恢复该后台引擎对应的前台应用，是否播放与进入 u 盘前的播放状态一致，如果不存在后台引擎，退出后回到 mainmenu 应用，如果进入 u 盘有更新过文件，由 u 盘创建播放列表的应用，创建完后回到 mainmenu 应用，所以 u 盘应用需要知道关机时后台的信息，但是在 config 运行时，后台已经被杀死了，无法得知状态，因此需要 manager 通过参数告知。

版权所有 侵权必究

另外，当 manager 收到 MSG_POWER_OFF 和 MSG_USB_TRANS 的消息，会给所有应用发送 MSG_APP_QUIT 的消息，要求所有应用都在收到这个私有消息时，都退出自身应用。

5.3.5 Ap_manager 的空间分配

Ap_manager 是 3 种应用程序中的一种，所以它拥有自己一套独立空间，空间分配如下：

Case全局数据空间

0x9fc19f80-0x9fc19fff 共 128B，该全局数据空间是所有 AP 共享的，只会在 ap_manager 中初始化一次，主要用来存放 Common 模块的一些 case 全局数据。

Ap_manager 独占数据空间

0x9fc1adc4-0x9fc1d000 ，与常驻代码空间 0xbfc1ad00-0xbfc1adc3 合起来共计 0x100 字节。

常驻代码空间

0xbfc1ad00-0xbfc1adc3 ，与常驻数据空间 0x9fc1adc4-0x9fc1d000 合起来共计 0x100 字节。

Bank UI 段空间

(0x7f**0000+0x1ae00) - (0x7f**0000+0x1afff) 共 0x200 字节，是 ap_manager 唯一 bank 段。

Ap_manager 数据空间和常驻代码空间都比较局限，所以要精简消息处理主循环的业务功能，尽量把功能封装为函数，放在 Bank UI 段中；而在数据空间上，如果实在不够用，可以使用 Case 全局数据空间剩余部分。

5.4 开机与关机 ap_config

Ap_config 是一个特殊的前台应用，是 Case 基本结构中必不可少的一部分，所以把 ap_config 放在这里介绍。

版 权 所 有 侵 权 必 究

5.4.1 ap_config 的地位与作用

ap_config 应用负责开机和关机，开机需要安装应用需要的驱动，根据断电时的记录创建开机后启动的应用，同时清除菜单记录的路径，如果是第一次上电，还需要创建播放列表；关机需要将关机前运行的应用信息保存到 vram。

5.4.2 ap_config 的设计要点

从方案的角度上看，ap_config 开机的关键在于决定应该去到哪个应用及进入应用的模式，这需要考虑以下几个要素：

- 上一次关机时的状态
- 触发开机的事件

而对于关机主要考虑是否连接着 usb 线或充电适配线，如果是则需要处理充电。

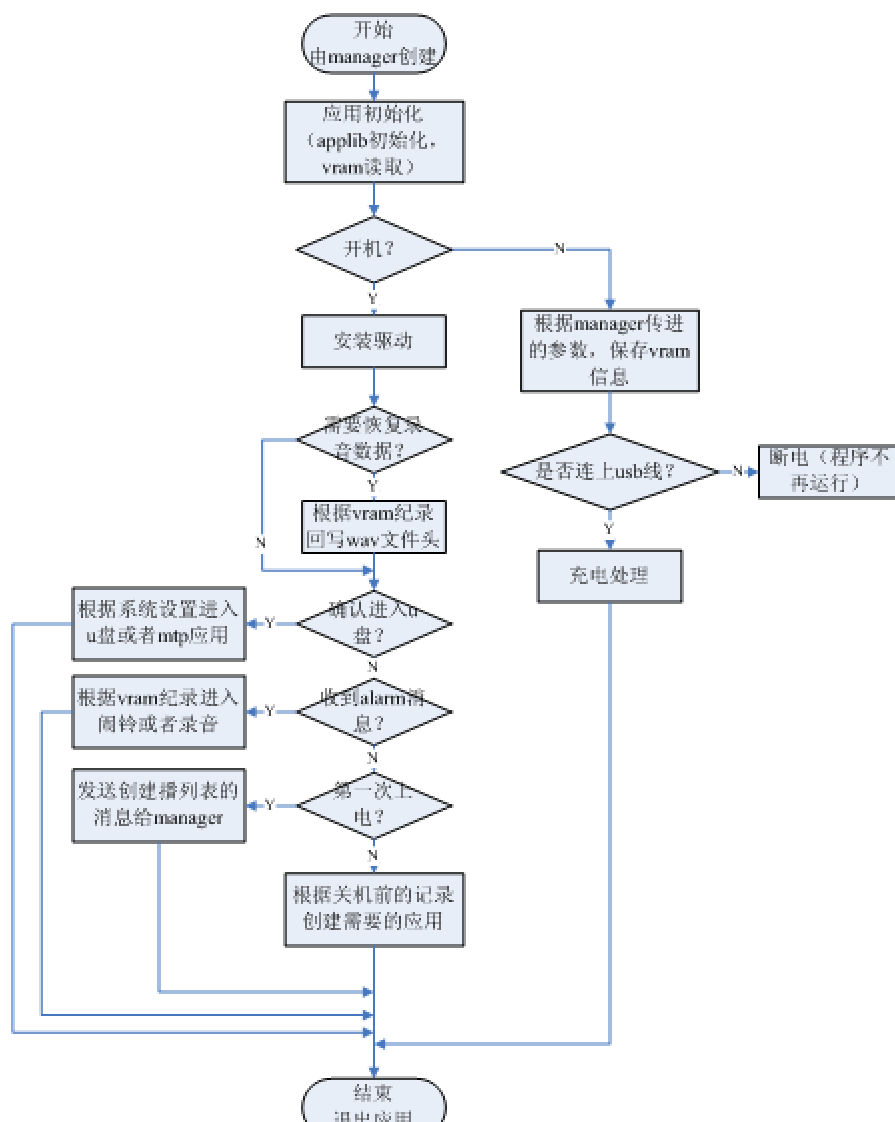
5.4.3 ap_config 的开机流程

触发开机的事件包括：

- 长按开机键
- Alarm 定时触发
- 插入 usb 线或者充电适配线

设备上电后，boot 启动，加载 kernel 并运行，kernel 加载 ap_manager 并运行，再由 ap_manager 创建 ap_config 开机，进入如下图开机流程：

版 权 所 有 侵 权 必 究



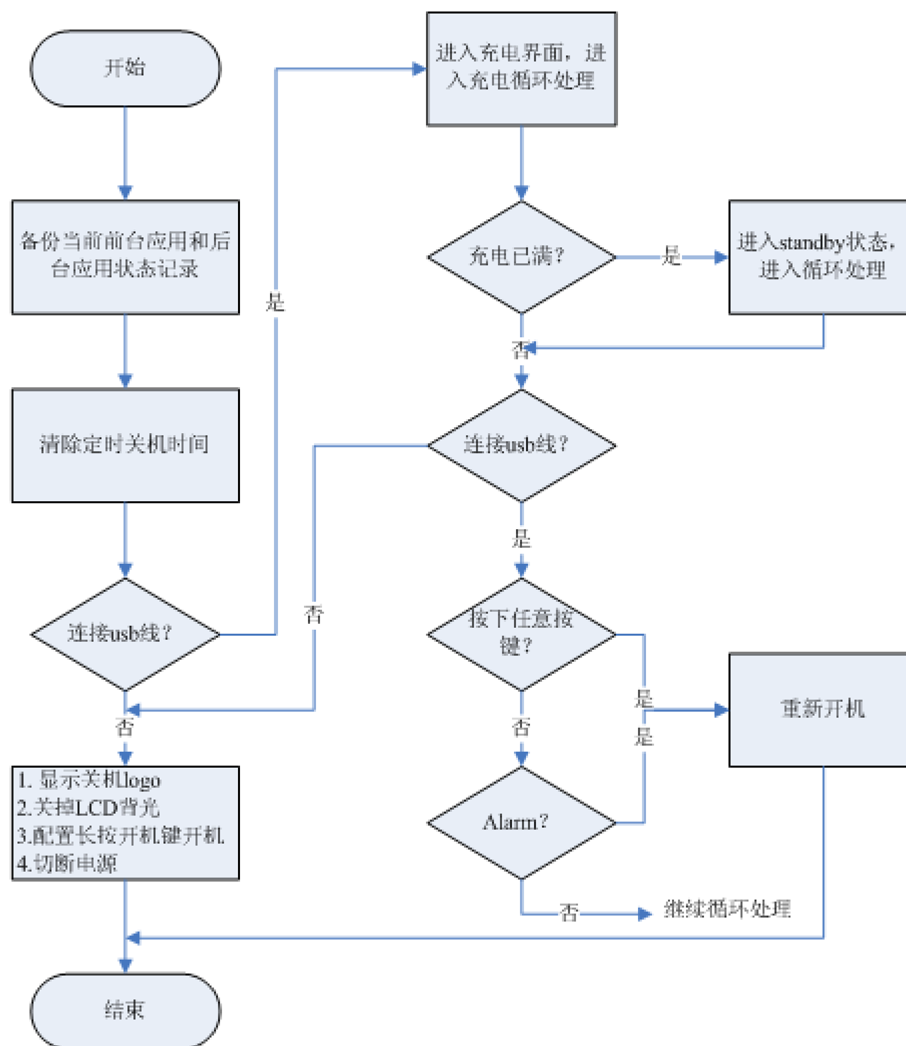
5.4.4 ap_config 的关机流程

触发关机的事件包括：

- 低电：当其它前台应用接收到低电消息 MSG_LOW_POWER 进入关机流程：广播 MSG_POWER_OFF 消息，ap_manager 收到该消息后，接管关机控制权，杀死前台应用和后台应用，并以关机模式创建 ap_config ，即 sys_exece_ap("config.ap", 0, 1 | ((uint32)last_ap_id << 8) | ((uint32)last_engine_state << 16))。
- 长按关机键，进入关机对话框，如果保持按住关机键 3 秒钟，就进入关机流程，流

程与低电一致。

进入 ap_config 关机后，流程如下图所示：



5.5 common 设计与使用

common 是 case 架构的基础组成部分。common 配合 psp 为应用程序提供一个易于协调管理和开发的基本环境，并且保证 case 在非业务功能上达成一致，有助于实现方案概念完整性。

为了达成以上目标，common 设计并实现了以下几个功能接口库：

- 应用基本功能接口库 applib
- 应用公共子功能基本接口库 common_func
- 应用公共控件 common_ui
- 应用公共子功能杂项接口库 common_misc

本文档并不过多介绍具体的接口，我们将重点介绍各接口的地位和作用，以及使用要点。接口介绍详细内容请参考 《us212a_common 接口说明书.chm》。

5.5.1 AppLib 设计与使用

5.5.1.1 AppLib 的功能概述

应用基本功能接口库 applib，包括应用程序（进程）管理、应用级定时器管理、消息通信管理。

这些接口与应用的基本架构有很大的关系。

5.5.1.2 应用程序（进程）管理

US212A 是一个抢占式多任务调度系统，可以同时存在 3 个应用，即进程管理器 ap_manager，前台应用和后台应用，其中前台应用和后台应用可能包含 2 个线程，即 2 个任务。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、前台应用或后台应用
mq_id	进程私有消息队列 ID
其他	保留，暂时不用

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。

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。

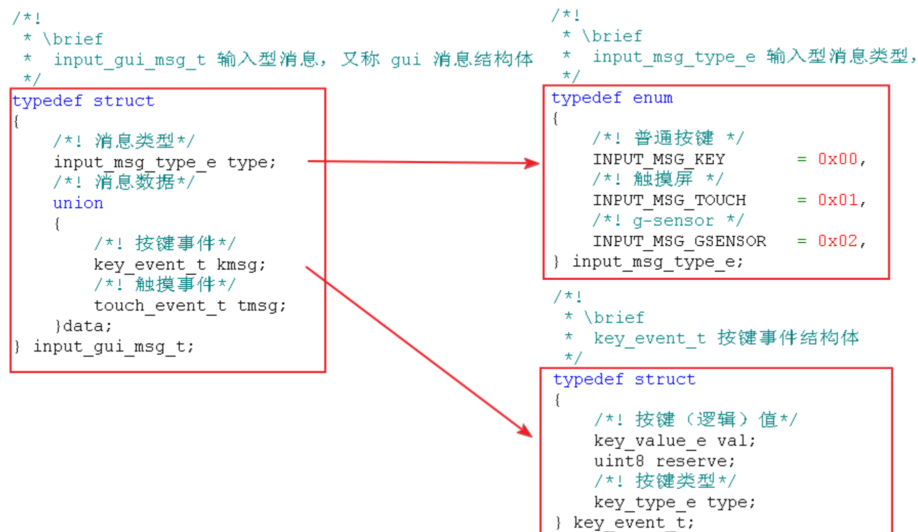
5.5.1.3 消息通信管理

在多任务调度系统中，应用（进程）间的通信是非常重要的。US212A 选择了消息通信作为应用（进程）间的通信方式。

另外，炬力以前的单任务方案，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，在 US212A 上，我们还是保留这样的机制。所以，US212A 的消息通信，是在以前的单任务方案的基础上进行扩展实现的。

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

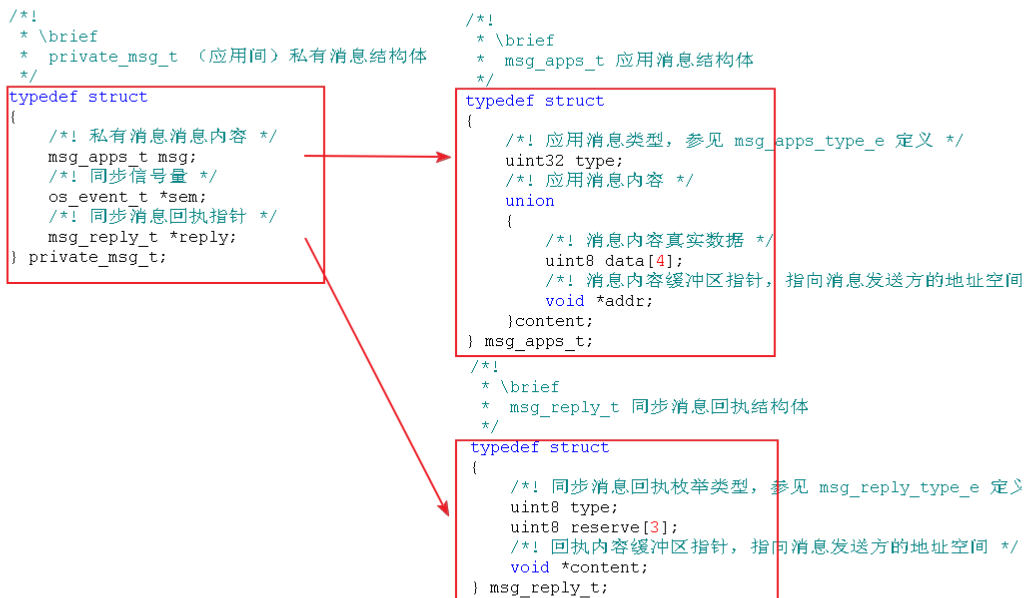
- gui 消息队列：用来存储按键消息、触摸屏消息和 gsensor 消息等，系统只有 1 个 gui 消息队列。gui 消息队列只有前台应用才会访问到，是用户与前台应用交互的通道。



- 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，同样，系统也只有 1 个系统消息队列。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给前台应用和后台应用，并且一般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系统 kernel 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。

```
/*!
 * \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 1 个私有消息队列，所以系统共有 3 个私有消息队列。



消息通信模块初始化：应用启动并且注册和初始化后，必须调用接口 bool **applib_message_init**(app_msg_dispatch msg_dispatch) 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和 gui 消息队列（只有前台应用）清空，并且把应用的私有消息分发函数注册到 common 模块，以便公共控件场景能像应用场景一样地处理应用私有消息。

按键驱动检测到按键按下，并且按照按键生命周期发送 KEY_TYPE_DOWN、KEY_TYPE_LONG、KEY_TYPE_HOLD 和 KEY_TYPE_SHORT_UP 消息。按键消息通过接口 bool **post_key_msg**(input_gui_msg_t *input_msg) 发送到 gui 消息队列。

系统消息则由 kernel 或驱动检测到某个系统事件，直接调用接口 int **mq_send**(uint8

queue_id, void *msg, void* null2) 发送消息。

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件事件，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。*广播消息时，如果某个应用不想接收某些类型的消息，那么可以设置过滤掉；如果某个应用想独占，也就是不想被其他应用处理这些消息，那么可以设置抓取；这就是广播的过滤和抓取机制。*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：US212A 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。

消息接收则每种消息队列我们都提供一个接口，这些接口都是常驻内存的：

- 按键消息接收：bool **get_gui_msg**(input_gui_msg_t *input_msg)

版 权 所 有 侵 权 必 究

- 系统消息接收：bool **get_sys_msg**(sys_msg_t *sys_msg)
- 前台应用私有消息接收：bool **get_app_msg**(private_msg_t *private_msg)
- 后台应用私有消息接收：bool **get_app_msg_for_engine**(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 bool **reply_sync_msg**(private_msg_t *private_msg) 实现。

更多关于消息通信说明请参考 消息通信开发详解 一节。

5.5.1.4 应用级定时器管理

我们在 US212A 上设计并实现了应用级定时器，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。

应用级定时器工作原理：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。

应用级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，接口声明如下。

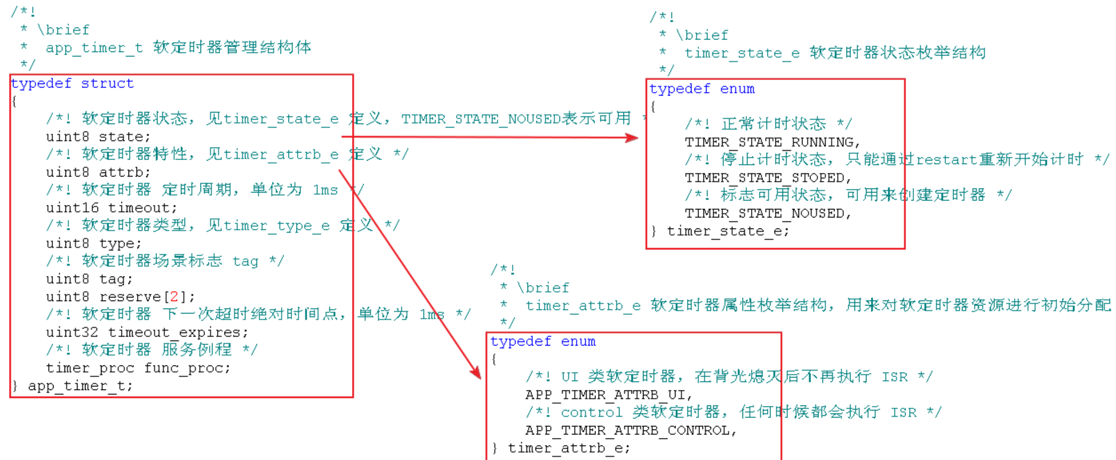
- 创建周期性定时器：int8 **set_app_timer**(timer_attrb_e attrb, uint16 timeout, timer_proc func_proc);
- 创建单发定时器：int8 **set_single_shot_app_timer**(timer_attrb_e attrb, uint16 timeout, timer_proc func_proc);
- 修改定时器周期：bool **modify_app_timer**(int8 timer_id, uint16 timeout);
- 停止计时：bool **stop_app_timer**(int8 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);

其中 timer_proc 定义为： typedef void (*timer_proc)(void);

另外，应用级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。

应用级定时器还有一个很特别的地方，就是其定时器数据空间是开在当前应用程序的内存空间上的，这样也就不占用系统有限的数据空间。



应用级定时器模块初始化：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。

应用级定时器设计重点难点说明：

1. 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。

2. 定时器属性：分为 UI 类定时器和 CONTROL 类定时器，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。

3. 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。具体使用如下：

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）0x80 ~ 0xfe，common 控件场景。

3）0x00 ~ 0x7f，ap 自定义场景，包括自定义控件场景，其中 0x00 是应用主场景，应用启动后就把场景 tag 初始化为 0x00。

4）进入一个场景，要先调用下面语句切换场景 tag 到当前场景对应唯一 tag 。

版 权 所 有　侵 权 必 究

```
uint8 tag_backup;
tag_backup = get_app_timer_tag();
change_app_timer_tag(APP_TIEMR_TAG_XXX);
```

5）退出一个场景，要恢复之前的场景 tag。

```
change_app_timer_tag(tag_backup);
```

更多关于应用级定时器说明请参考 应用级定时器开发详解 一节。

5.5.2 Common_func 设计与使用

5.5.2.1 Common_func 的功能概述

应用公共子功能基本接口库 common_func，包括字符串处理（包括格式化字符串，省略显示形式，短名加点，内码名字转为 UNICODE 码，显示调试信息），路径记忆，配置项解释，可配置化菜单解释，声音输出管理，按键映射处理，按键消息预处理，等等。

5.5.2.2 字符串处理

字符串处理和显示对带显示屏的方案来说是一种基本功能，为了减轻应用程序的负担，也为了保证所有应用程序的字符串处理和显示实现一致，common 在设计和实现时收集并提取出以下字符串处理和显示接口。

1.格式化字符串：字符串显示中，有一种情况是要在字符串资源（后面称为字符串模板）中插入一个文件名之类的可变字符串。类似于格式化打印显示接口一样，我们要求字符串模板中包含某特殊符号或组合以指代需要替换为可变字符串，然后提供要插入的字符串内容，把两者格式化合并为最终用于显示的字符串内容。

接口声明为：bool **com_string_format**(string_desc_t *pattern, string_desc_t *strings)

2.省略显示形式：在列表显示中，有时要求把超出显示区域的列表项以省略显示形式进行显示。我们在 ui 驱动中实现了这样的接口，但是对应用程序来说不是很容易使用，所以又在 common 封装了一次。

接口声明为：uint16 **com_ellipsis_to_longname**(uint8 *long_name, uint8 *ellipsis_name, uint16 ellipsis_len)

3.短名加点：把 8+3 短名转换为带点号文件名方式，比如 ABC　　　　mp3，转换为 ABC.mp3。

接口声明为：void **com_dot_to_shortname**(uint8 *short_name)

版 权 所 有　侵 权 必 究

4. 内码名字转为UNICODE码：US212A创建文件和文件夹需要使用UNCIODE码名字，但UNICODE码名字写起来很容易出错，所以我们提供内码名字转为UNICODE码的接口。

接口声明为：void **com_ansi_to_unicode**(uint8 *file_name)

5.显示调试信息：我们在开发调试过程中，如果需要显示某个变量的值什么的，可以通过该接口进行显示。

接口声明为：void **com_dump_debug**(uint16 x, uint16 y, uint8 *format, uint32 value)

5.5.2.3 路径记忆

我们在进行菜单列表或文件列表浏览时，往往希望退出后下次能够沿上一次退出的路径一步一步再次进入，这就需要路径记忆了，这样的操作体验会很流畅。

我们先对路径记忆进行定义：

1） 路径记忆需要把每一级列表的现场都记忆下来，也就是列表从第几项开始显示，哪一项是激活项。

2） 文件列表浏览的路径记忆的层次需要达到9级以上，具体由文件系统和文件选择器决定的。而菜单列表的路径记忆层次在 8 级以内。

3） 应用独立保存路径记忆，每个应用最多保存 4 个文件列表浏览的路径记忆，以实现多磁盘路径记忆；每个应用最多保存 8 个菜单列表路径记忆，每课菜单树使用一个路径记忆。

路径记忆数据结构如下：

```
/*!
 * \brief
 *  history_index_t 路径记忆索引结构体
 */
typedef struct
{
    /*! 路径类型，0表示菜单列表，1表示文件浏览 */
    uint8 type;
    /*! 应用ID，取值范围：0 ~ 31 */
    uint8 app_id;
    /*! 路径记忆ID，取值范围：0 ~ 7(3) */
    uint8 path_id;
    /*! 当前层级，取值范围：0 ~ 7(15) */
    uint8 layer;
} history_index_t;
```

```
/*!
 * \brief
 *  history_item_t 路径记忆项结构体
 */
typedef struct
{
    /*! 当前显示列表首项编号，0xffff表示无效 */
    uint16 top;
    /*! 激活项在当前显示列表中的位置，0xffff表示无效 */
    uint16 list_no;
} history_item_t;
```

其中 history_index_t 用来定位路径记忆项，history_item_t 是真正存储在 VRAM 的路径记忆项。整个 case 的路径记忆是这样存储在 VRAM 中的：

菜单列表的路径记忆同文件列表结构是一样的，区别在于菜单列表每个路径记忆的 History_item_t 项目为 8 项，并且每个应用程序的菜单列表路径记忆总数为 8 个。

由于在列表浏览中，路径是会频繁变化的，如果每次变化都更新一次路径记忆到 VRAM，那么对 VRAM 写的压力会比较大，并且 VRAM 写速度比较慢，对列表浏览的性能也会产生一定的影响。所以，我们采用了缓冲的方式来处理列表浏览过程中的路径记忆，浏览过程中并不会更新路径记忆，只有在退出列表浏览时才 Update 到 VRAM 中。

在某些特殊情况下，我们会把路径记忆给清除掉。

1）开机时，有些方案会把菜单列表路径记忆全部清除掉；创建播放列表，会把文件列表路径记忆全部清除掉，可以调用 **com_clear_all_history** 接口。

2）当某应用某磁盘被破坏掉了，可以把该磁盘的文件列表路径记忆清除掉，可以调用 **com_clear_app_history** 接口。

故路径记忆模块提供了以下功能接口：

1）获取当前层级的路径记忆项：

bool **com_get_history_item**(history_index_t *index, history_item_t *history)

版 权 所 有 侵 权 必 究

该接口第一次调用时会把整个路径记忆加载到缓冲区，以后就直接从缓冲区读取。

2）设置当前层级的路径记忆项：

bool **com_set_history_item**(history_index_t *index, history_item_t *history)

该接口仅仅是把路径记忆更新到缓冲区中。

3）更新路径记忆到 VRAM：

bool **com_update_path_history**(history_index_t *index)

该接口只在退出列表浏览时才调用，调用完该接口，缓冲区会被释放掉。

4） 清除所有菜单列表/文件列表路径记忆：

bool **com_clear_all_history**(uint8 type)

5）清除指定应用指定磁盘的文件列表路径记忆：

bool **com_clear_app_history**(uint8 type, uint8 app_id, uint8 disk)

5.5.2.4 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 3 中结构体中的一种：

```
/*!
 * \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：

版 权 所 有 侵 权 必 究

uint16 **com_get_config_default**(uint16 config_id)

对于 case 配置项更详细的介绍请参考。

5.5.2.5 可配置化菜单解释

可配置化菜单解释基本功能接口包括：

1. 打开可配置化菜单文件，该接口一般在应用初始化时调用：

bool **com_open_confmenu_file**(const char* file_name)

2. 关闭可配置化菜单文件，该接口一般在应用退出时调用：

bool **com_close_confmenu_file**(void)

3. 读取可配置化菜单文件菜单列表头：

bool **com_get_confmenu_title**(uint8 menu_id, menu_title_data_t* confmenu_title)

4. 读取可配置化菜单文件指定菜单列表的菜单项：

bool **com_get_confmenu_item**(menu_title_data_t* confmenu_title, uint16 item_index, menu_item_data_t *confmenu_item)

5. 获取可配置化菜单文件指定菜单列表中某个菜单项的索引号：

uint16 **com_get_confmenu_active**(menu_title_data_t* confmenu_title, uint16 key_str)

uint16 **com_get_menu_active**(uint8 menu_id, uint16 key_str)

更多关于可配置化菜单说明请参考 菜单列表控件 和 可配置化菜单 一节。

5.5.2.6 声音输出管理

涉及声音输出的有 4 个方面：音频输出与否（包括音乐播放、视频播放和 FM 收音）、按键音开关、录音（暂时关闭按键音）和外放喇叭使能与否。为了协调处理好这 4 者的开和关，因此设计并实现了声音输出管理。另外，由于 PA 模拟模块的开启要消耗几百个 ms，为了最大限度减少 PA 的开启和关闭，也要求进行统一声音输出管理。

对应用来说，应用知道什么时候有音频输出，按键音开关是一个菜单选项，所以我们需要提供如下接口：

1. 按键音开和关：

bool **com_set_sound_keytone**(uint8 kt_enable)

2. 音频输出控制：

bool **com_set_sound_out**(bool sound_out, soundout_state_e state, void *sound_func)

版 权 所 有 侵 权 必 究

音频输出有 4 中控制方式和状态，即开始，暂停，恢复，停止，为了减少 PA 的开启和关闭，暂停时并不会关闭 PA，这样在后面的恢复，就能快速的恢复断点播放。

3. 音量设置接口，各个应用统一使用该接口设置音量，以统一维护当前音量级别：

bool **com_set_sound_volume**(uint8 volume)

4. 录音声音控制接口：

bool **com_set_sound_record**(bool on_off, uint8 type)

5.5.2.7 按键映射处理

按键映射就是把按键消息映射为某个事件，以实现消息处理循环以事件驱动机制进行。按键映射的关键就是要正确定义好按键映射表。

按键映射分成两部分，即用户按键映射表和快捷键映射表。快捷键映射表是 case 默认处理的按键消息集合，它的优先级比用户按键映射表低。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用户按键映射表中，也可以把该快捷键映射为 MSG_NULL 来取消该快捷键。

按键消息映射表结构如下：

```
/*!
 * \brief
 *  key_map_t 按键（gui）消息I
 */
typedef struct
{
    /*! 源按键事件 */
    key_event_t key_event;
    /*! 映射 gui 事件 */
    msg_apps_type_e event;
} key_map_t;
```

```
/*!
 * \brief
 *  key_event_t 按键事件结构体
 */
typedef struct
{
    /*! 按键（逻辑）值*/
    key_value_e val;
    uint8 reserve;
    /*! 按键类型*/
    key_type_e type;
} key_event_t;
```

```
/*!
 * \brief
 *  key_type_e 按键类型
 */
typedef enum
{
    /*! 没有按键*/
    KEY_TYPE_NULL        = 0x0000,
    /*! 按键按下*/
    KEY_TYPE_DOWN        = 0x2000,
    /*! 按键长按*/
    KEY_TYPE_LONG        = 0x1000,
    /*! 按键保持*/
    KEY_TYPE_HOLD        = 0x0800,
    /*! 按键短按弹起*/
    KEY_TYPE_SHORT_UP    = 0x0400,
    /*! 按键长按弹起*/
    KEY_TYPE_LONG_UP     = 0x0200,
    /*! 按键保持弹起*/
    KEY_TYPE_HOLD_UP     = 0x0100,
    /*! 任意按键类型 */
    KEY_TYPE_ALL         = 0x3f00,
} key_type_e;
```

```
/*!
 * \brief
 *  dialog_key_map_list: 对话框控件按键映射表
 */
const key_map_t dialog_key_map_list[] =
{
    /*! PREV 按键转换为 PREV_BUTTON 事件 */
    {{KEY_PREV, 0, KEY_TYPE_DOWN | KEY_TYPE_LONG | KEY_TYPE_HOLD}, EVENT_DIALOG_PREV_BUTTON},
    /*! NEXT 按键转换为 NEXT_BUTTON事件 */
    {{KEY_NEXT, 0, KEY_TYPE_DOWN | KEY_TYPE_LONG | KEY_TYPE_HOLD}, EVENT_DIALOG_NEXT_BUTTON},
    /*! 短按KEY_PLAY 按键转换为 CONFIRM事件 */
    {{KEY_PLAY, 0, KEY_TYPE_SHORT_UP}, EVENT_DIALOG_CONFIRM},
    /*! 短按KEY_VOL 按键转换为 CANCEL事件 */
    {{KEY_VOL, 0, KEY_TYPE_SHORT_UP}, EVENT_DIALOG_CANCEL},
    /*! 短按KEY_MODE 按键转换为 CANCEL事件 */
    {{KEY_MODE, 0, KEY_TYPE_SHORT_UP}, EVENT_DIALOG_CANCEL},

    /*! 结束标志 */
    {{KEY_NULL, 0, KEY_TYPE_NULL}, MSG_NULL},
};
```

说明：

1. 按键消息映射表中的按键消息类型以 类型或 填写，在匹配按键消息时，当前按键消息的类型与该类型按位与，结果非 0 就认为匹配。

2. 按键消息映射表必须以 KEY_NULL 映射项为结束，按键映射对表的匹配就是以找到按键消息值为 KEY_NULL 的映射项为结束条件。

用户根据具体的用户场景抽取用户关心的按键消息映射为业务事件。没有列入到用户按键消息映射表，并且也不在快捷键映射表中的按键消息，会被该用户场景忽略掉，不做任何响应。

按键映射接口如下：

bool **com_key_mapping**(input_gui_msg_t *input_msg, msg_apps_type_e *gui_event, const key_map_t *key_map_list)

接口使用注意：

调用该接口时，往往会发生 BANK 切换，而该接口的参数 key_msp_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。

5.5.2.8 按键消息预处理

版 权 所 有 侵 权 必 究

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、恢复背光、按键音等等。我们可以通过在前台应用的 gui 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 gui 消息接收接口，优化方案架构。

按键消息预处理设计和实现的功能包括：

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，我们在该勾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)，之后勾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。

2. 退出屏保，恢复背光。

3. 按键音响应，只对 KEY_DOWN 消息响应。

4. 按键锁处理，如果当前按键锁处于锁住状态，那么勾函数会把其他非开锁按键消息转换为 KEY_LOCKHOLD 消息，用来提示按键锁已锁的状态。

5. 清零空闲计时器，从新开始计时背光变暗定时等。

按键消息预处理接口如下：

bool **com_gui_msg_hook**(input_gui_msg_t *input_msg)

该接口返回 FALSE 时表示按键消息被过滤掉。

该接口在 **get_gui_msg** 函数中接收到 gui 消息时调用。

5.5.3 Common_ui 设计与使用

应用公共控件 common_ui，包括菜单列表控件，文件浏览器，删除文件控件，对话框，USB 连接对话框，参数设置框，音量条，文本显示框，状态栏，动画显示，按键锁，屏幕保护，关机对话框，等等。

公共控件对方案实现一致的操作方式，实现概念一致性的目标是至关重要的。

对每一种公共控件的说明，我们基本按照以下几个方面以此进行：

- 控件的定义
- 控件的特性及设计，包括内部关键数据结构
- 控件场景 UI 模板说明

版 权 所 有 侵 权 必 究

- 控件的接口设计及说明
- 控件使用要点

大部分公共控件设计并实现为一个控件场景，实现与用户进行 I/O 交互。

5.5.3.1 菜单列表控件

菜单列表控件提供一种方式让用户从几个并行的选项中选取其中一项，可以用来选择执行命令操作，可以用来设置非数值型和非线性数值型参数，等等。

控件的特性及设计

1.控件特性

在 US212A 方案中，菜单是一个非常重要的概念，因为我们引进了以下特性：

1. 可配置化菜单，一种可视化的菜单列表构造实现方法，使菜单列表的构造变得更加简单和灵活。

2. 使用大量的菜单回调函数，以一种强大的菜单列表架构为 AP 开发工程师省掉菜单场景开发的细节，工程师们可以集中精力关注菜单回调函数的编写。每个菜单项可以挂载 3 个回调函数，即菜单即时响应函数，嵌入快捷响应函数，以及确定执行函数，这 3 个函数在某些条件下都可为空。回调函数与菜单项相对于菜单列表的索引号无关，仅与菜单项本身有关。

3. 支持灵活的动态菜单，根据一定的条件，在进入子菜单，返回父菜单，以及重绘当前菜单列表都可以实现动态更新，动态菜单必须实现为子入口菜单。这里动态的意思是在构造的菜单树中，动态菜单并不属于菜单树的一部分，而是在运行过程中临时替换菜单树的某个菜单列表头节点。

4. 支持菜单项/标题内容可变，使得菜单列表适用范围更广。

2.控件基本概念

1. 激活项 active：高亮显示的菜单项，激活项属于当前列表页面中的一项，用户通过上下键切换激活项。

2. 选中项 selected：在单选参数菜单列表中，上一次选择了的菜单项，不一定会出现在当前列表页面上，选中项在 UI 上一般体现为在末尾提示选中图标，如上面控件模板所示。

3. 入口菜单：入口菜单是一个菜单树的总入口，每个菜单场景可以设计多个入口菜单，根据某些条件进行选择；并且入口菜单也可以作为其他入口菜单的子菜单树，从而共灵活呈现菜单树。

4. 菜单列表：菜单列表是指当前层级菜单列表页面。

5. 菜单项、叶子菜单项和菜单列表头：菜单项是菜单列表的节点，叶子菜单项是没有下一级菜单的菜单项，菜单列表头是有下一级菜单的菜单项。

3.控件内部关键数据结构

```
//小机内部使用的数据结构，由pc菜单配置工具生成
/*!
 * \brief
 *  menu_title_data_t 菜单头数据结构
 */
typedef struct
{
    /*! 菜单头资源字符串ID */
    uint16 str_id;
    /*! 当前菜单的菜单项个数 */
    uint16 count;
    /*! 菜单item data 开始地址 */
    uint16 offset;
    /*! 默认的菜单活动项，通过工具设置 */
    uint16 active_default;
    /*! 当前菜单的上一级菜单的索引，如果当前菜单为入口根菜单，此项为0x7fff */
    uint16 father_index;
    /*! 当前菜单列表对应的根菜单项在父菜单中的编号 */
    uint16 father_active;
} menu_title_data_t;

/*!
 * \brief
 *  menu_item_data_t 菜单项数据结构
 */
typedef struct
{
    /*! bit<15>表示菜单是否为radio按钮，0表示否，1表示是
     *  bit<14-0>表示菜单项类型：=0x7fff，表示叶子菜单；其他值表示根菜单，值表示指向的菜单头索引值
     */
    uint16 child_index;
    /*! 菜单项资源字符串ID */
    uint16 str_id;
    /*! 菜单显示字符副ID，在主ID相同的情况下，模拟器上需要使用此ID找到各个回调函数的地址 */
    uint16 str_id_sub;
    uint16 reserve;
    /*! 菜单项确定执行回调函数
     *  对于菜单头项，返回值不处理，返回 RESULT_NULL 即可；而叶子菜单项则需要谨慎选择返回值
     */
    menu_cb_func menu_func;
    /*! 即时叶子菜单回调函数，非NULL表示该叶子菜单为即时叶子菜单，NULL表示不是即时叶子菜单 */
    menu_cb_leaf callback;

    /*! PHILIPS特性支持，菜单项option回调函数 */
    menu_cb_option menu_option;
} menu_item_data_t;
```

版 权 所 有 侵 权 必 究

```
/*!
 * \brief
 *  menulist_control_t 菜单列表控制结构体
 */
typedef struct
{
    /*! 默认菜单style类型，即调用 gui_menulist 时传递进来的style_id */
    uint16 style_id;
    /*! 当前入口菜单栈指针 */
    uint8 stack_pointer;
    /*! 当前级菜单每页项数 */
    uint8 one_page_count;
    /*! 当前菜单列表控制结构体 */
    menu_com_data_t *menu_com;
    /*! 路径记忆ID */
    uint8 path_id;
    /*! 路径记忆层级ID */
    uint8 layer_no;
    /*! 当前级菜单列表总项数 */
    uint16 total;
    /*! 当前显示的菜单列表首项序号 */
    uint16 top;
    /*! 当前显示的菜单列表尾项序号 */
    uint16 bottom;
    /*! 当前激活项位置 */
    uint16 list_no;
    /*! 当前选中项，仅对 RADIO 菜单有效，select为-1表示非 RADIO 菜单 */
    uint16 select;
    /*! 当前激活项序数 */
    uint16 active;
    /*! 之前激活项序数 */
    uint16 old;
    /*! 菜单项更新模式，参见 list_update_mode_e 定义 */
    list_update_mode_e update_mode;
    /*! 列表UI绘制模式，参见 list_draw_mode_e 定义 */
    list_draw_mode_e draw_mode;
    /*! 列表需要更新菜单头数据标志 */
    bool enter_menu;
} menulist_control_t;
```

4.关键设计要素——菜单回调函数

在说明菜单回调函数之前，必须先说明回调函数返回值类型和回调函数输出参数。

1. 回调函数返回值：在 gui_menulist / gui_menulist_simple 场景内部，规定了回调函数的返回值及其处理方式：

返回值	说明
RESULT_MENU_EXIT	退出菜单列表控件，转换为 RESULT_REDRAW 返回

RESULT_MENU_PARENT	退回上一级，如果当前层级为根，退出菜单列表控件，转换为 RESULT_REDRAW 返回
RESULT_MENU_CUR	重新进入当前层级菜单列表，在执行确定函数后要进入动态菜单时返回
RESULT_MENU_SON	进入下一级菜单列表
RESULT_MENU_REDRAW_PARENT	重绘菜单列表后返回上一级，比如选择风格后
RESULT_MENU_REDRAW	重绘菜单列表，用于嵌套调用 menulist_simple 返回
RESULT_REDRAW	按返回键返回上一级，或正常退出菜单列表控件等返回 AP，重绘 UI，如果回到场景调度循环，通常忽略即可
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

2. 回调函数输出参数结构体如下：

```
/*!
 * \brief
 *  menu_title_action_t 根菜单项确定函数返回的执行结构体
 */
typedef struct
{
    /*! 返回本级菜单style_id, 可以更换ui_menulist传进来的默认style_id进行
     *  显示，-1表示沿用默认style_id
     */
    uint16 style_id;
    /*! 返回动态菜单入口ID，-1表示采用菜单头默认的根菜单ID */
    uint16 menu_entry;
    /*! 菜单标题（更准确说是菜单描述符）ID，-1表示采用菜单头字符串ID */
    uint16 title_id;
    /*! 返回激活项多国语言字符串ID，-1表示采用默认激活项 */
    uint16 str_id;
    /*! 返回动态菜单项字符串指针，NULL表示无效，并且不用理会source_id的值 */
    uint8 *ret_str;
    /*! 返回菜单配置工具中配置字符串ID，指向动态菜单项，-1表示菜单标题 */
    uint16 source_id;
} menu_title_action_t;
```

回调函数输出参数在传递给回调函数之前，会被初始化为无效状态，即 ret_str 初始化为 NULL，其他初始化为-1。

下面分别对 3 种回调函数展开说明：

菜单即时响应函数

函数原型：typedef void (*menu_cb_leaf)(void)

该函数用于在切换参数型菜单列表时，即时响应以提供预览功能。该函数不带输出参数，也不返回结果，所以用法很简单。

嵌入快捷响应函数

函数原型：typedef app_result_e (*menu_cb_option)(void)

该函数用于在菜单列表浏览中无须退出菜单场景就可以快捷处理某些事情，在 EVENT_MENULIST_ENTER_OPTION 事件响应。该函数返回结果，所以必须慎重选择返回结果类型。嵌入快捷响应函数的使用情景可以参考确定执行函数的，只是不能使用输出参数。

确定执行函数

函数原型：typedef app_result_e (*menu_cb_func)(void *param)

该函数在菜单列表层级切换以及重新加载当前层加菜单列表时调用。该函数既返回结果，又带输出参数，这两者的配合适用于各种情景。接下来会列举主要使用情景。

情景一：选择菜单列表头，正常进入下一级菜单列表

这种情景，如果下一级菜单列表是一般菜单列表，即没有动态菜单，不需要指定选中项，没有菜单项内容可变，也不需要更换控件模板，那么直接使 menu_cb_func 为 NULL 就可以了；也可以实现一个只需返回 RESULT_MENU_SON 的 menu_cb_func 函数。

情景二：选择菜单列表头，进入动态菜单

这种情景必须仔细构造输出参数 menu_title_action_t ，构造例子如下：

```
app_result_e sample_dynamic_menu(void * title_action)
{
    uint16 active_id;
    menu_title_action_t* menu_title = (menu_title_action_t*) title_action;

    //如果条件满足则进入动态菜单，否则正常进入下一级
    if (g_sample_dynamic == 1)
    {
        menu_title->menu_entry = 1;
    }

    return (app_result_e) RESULT_MENU_SON; //进入下一级
}
```

情景三：选择菜单列表头，指定选中项

这种情景必须仔细构造输出参数 menu_title_action_t ，方法跟情景二一致。即把 str_id 成员赋值为选中项的字符串资源 ID，从确定执行函数返回来后由菜单解释器从菜单列表中匹配得到索引号。

情景四：选择菜单列表头，修改菜单项/标题内容

这种情景必须仔细构造输出参数 menu_title_action_t ，方法跟情景二一致。即把 ret_str

版 权 所 有 侵 权 必 究

赋值为某个缓冲区地址，并且选择 source_id ，-1 表示修改标题内容，菜单项字符串 ID 则表示修改对应菜单项内容。

情景五：选择菜单列表头，修改菜单控件模板

这种情景必须仔细构造输出参数 menu_title_action_t ，方法跟情景二一致。即把 style_id 修改为指定的菜单控件模板 ID，比如 MENULIST_TITLE 。

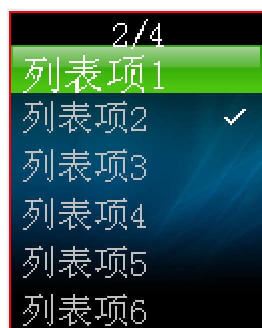
情景六：选择叶子菜单项

选择叶子菜单，执行确定执行函数，执行完业务处理后，要慎重选择返回结果。另外，可以通过输出参数 menu_title_action_t 赋值 str_id 为某个菜单项字符串资源 ID 来指定退出后菜单列表的激活项。

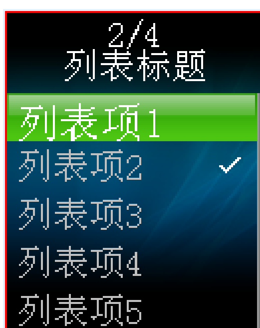
以上情景可以综合使用，比如选择菜单列表头，既进入动态菜单，也指定选中项，还修改菜单控件模板等。以上例子菜单列表头的确定执行函数也适用于首层菜单回调函数，即当入口菜单需要根据某些条件进行非一般处理，可以也应该使用以上例子的方式进行。

控件场景 UI 模板

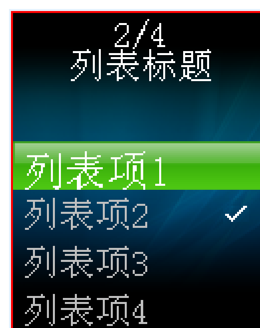
Common UI Editor 工程为菜单列表提供了丰富的控件模板，如下图所示：



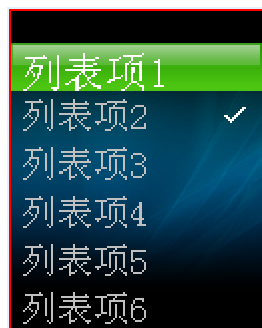
MENULIST



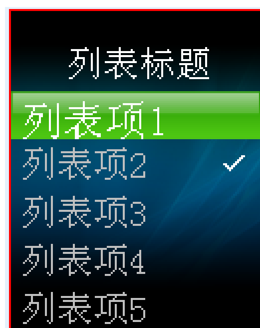
MENULIST_TITLE



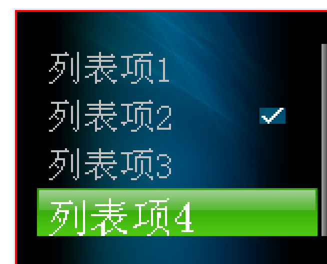
MENULIST_TITLE2



MENULIST_OPTION



MENULIST_OPTION_TITLE



MENULIST_OPTION_V

如果以上控件模板没有合适的，用户可自己在 Common 工程创建自己的控件模板。关于控件模板的进一步介绍，请参考 可配置化 UI 一节。

控件接口设计

app_result_e **gui_menulist** (uint16 style_id, menu_com_data_t *menu_com)：完全功能菜单列表控件接口。

app_result_e **gui_menulist_simple** (uint16 style_id, menu_com_data_t *menu_com)：不带路径记忆功能，不支持动态菜单功能的简化版菜单列表控件，这种菜单列表属于命令型菜单列表，它只会有一层菜单，选择了叶子菜单项后执行完立即退出。

```
/*!
 * \brief
 *  menu_com_data_t 菜单列表控件初始化结构体
 */
typedef struct
{
    /*! 应用ID */
    uint8 app_id;
    /*! 入口菜单ID */
    uint8 menu_entry;
    /*! 路径记忆ID */
    uint8 path_id;
    /*! 浏览模式，0表示从入口菜单0层，1表示直接跳到之前退出时最深层次 */
    uint8 browse_mode;
    /*! 首层菜单回调函数 */
    menu_cb_func menu_func;
} menu_com_data_t;
```

接口返回参数如下表所示：

返回值	说明
RESULT_REDRAW	正常退出菜单列表控件，重绘 UI，如果回到场景调度循环，通常忽略即可
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

控件使用说明

1. 退回上一级菜单执行函数：有时候我们需要在退回到上一级菜单时执行某些处理，比如某些即时响应菜单，在切换菜单项时就已经执行了设置，但是我们要求在取消返回时能够还原设置。为此，我们提供了以下 inline 接口：

static inline void **gui_menulist_set_back2parent**(back_to_parent func, uint8 *arg_addr)

该 inline 函数向菜单列表注册了退回上一级菜单时的执行函数以及条件参数。设置之后，只要从当前菜单列表退出（退回上一级菜单或者退出菜单列表场景），都会判断条件参数的值 ***(uint8 *)(arg_addr)** 是否非 0，如果是则执行该函数。

2. 菜单场景嵌套：有时候我们需要在菜单场景中嵌套显示另一个菜单，但是 gui_menulist 菜单场景比较复杂，栈控件消耗较多，并且嵌套场景使用了相同的应用级定时器场景 tag，会造成一些混乱。后面我们分析发现，嵌套菜单一般都会比较简单，可以理解为一个不需要路径记忆的单层快捷菜单，所以我们开发了另一个菜单场景 **gui_menulist_simple** 来支持嵌套菜单，它与 **gui_menulist** 使用不同的应用级定时器场景 tag，不会造成混乱。

3. 菜单列表控件已经实现了路径记忆，如果进入下一级菜单列表时指定了选中项，并且在菜单列表中匹配成功，那么就按照指定的选中项为激活项显示菜单列表，否则就从 VRAM /缓冲区中读取路径记忆来显示菜单列表。

更多关于可配置化菜单说明请参考 可配置化菜单 和 可配置化菜单开发详解 一节。

5.5.3.2 文件浏览控件

文件浏览器提供一种可视化方式来浏览选择文件和目录，其中文件概念延伸到收藏夹，播放列表等非文件系统组织形式的文件选项。文件浏览器需要文件系统和文件选择器的支持，在使用前必须保证文件系统和文件选择器已经初始化了。

控件的特性及设计

1.控件内部关键数据结构

版 权 所 有 侵 权 必 究

```
/*!
 * \brief
 *  dir_control_t 文件浏览器控制结构体
 */
typedef struct
{
    /*! 正在浏览磁盘盘符，DISK_C表示主盘，DISK_H表示卡盘 */
    uint8 dir_disk;
    /*! 文件选择（来源）类型，见 fsel_type_e 定义 */
    uint8 file_source;
    /*! 目录模板每页项数 */
    uint8 one_page_count;
    uint8 reserve;
    /*! 当前文件浏览器控制结构体 */
    dir_com_data_t *dir_com;
    /*! 项目列表有效区域top项序号，相对整个列表的序号，值从1开始 */
    uint16 top;
    /*! 项目列表有效区域bottom项序号，相对整个列表的序号，值从1开始 */
    uint16 bottom;
    /*! 项目列表项总数 */
    uint16 list_total;
    /*! 当前激活文件信息 */
    /*! 当前文件序号，相对整个列表的序号，值从1开始 */
    uint16 list_no;
    /*! 当前列表的激活项，值从1 ~ one_page_count*/
    uint16 active;
    /*! 当前列表旧的激活项，值从1 ~ one_page_count */
    uint16 old;
    /*! 文件列表更新模式，参见 list_update_mode_e 定义 */
    list_update_mode_e update_mode;
    /*! 列表控件绘制模式，参见 list_draw_mode_e 定义 */
    list_draw_mode_e draw_mode;
    /*! 列表需要更新目录头数据标志 */
    bool enter_dir;
} dir_control_t;
```

控件场景 UI 模板

Common UI Editor 工程为文件浏览控件提供了 2 个控件模板，如下图所示：

DIRLIST FILELIST

如果以上控件模板没有合适的，用户可自己在 Common 工程创建自己的控件模板。关于控件模板的进一步介绍，请参考 可配置化 UI 一节。

控件接口设计

app_result_e **gui_directory** (uint16 style_id, file_path_info_t *path_info, dir_com_data_t *dir_com)

```
/*!
 * \brief
 *  dir_com_data_t 文件浏览器控件初始化结构体
 */
typedef struct
{
    /*!
     * 浏览模式，0表示从路径的第一级目录浏览，1表示直接跳到路径指向目录浏览；
     * bit7 为1表示允许根目录没有文件项而又菜单项的浏览；
     */
    uint8 browse_mode;
    /*! 文件浏览菜单列表的列表项数目，一般为1 */
    uint8 menulist_cnt;
    /*! 当前浏览目录树根节点层级，默认为0；比如为1，则从第1级子目录返回时要退出浏览器 */
    uint8 root_layer;
    /*! 是否删除成功后返回，TRUE表示是，FALSE表示否 */
    bool del_back;

    /*! option子菜单回调函数 */
    list_cb_option list_option_func;
    /*! 文件浏览器菜单列表 */
    list_menu_t *menulist;
} dir_com_data_t;
```

```
/*!
 * \brief
 *  list_menu_t 文件浏览器菜单参数数据结构
 */
typedef struct
{
    /*! 层级条件，-1表示不理会层级条件 */
    uint8  layer;
    /*! 应用ID */
    uint8  app_id;
    /*! 列表菜单入口ID */
    uint16 list_menu;
} list_menu_t;
```

接口返回值如下表所示：

返回值	说明
RESULT_XXX_PLAY	选择某类解码型文件进行播放返回
RESULT_CONFIRM	确定选择文件夹等返回
RESULT_REDRAW	按返回键逐级列表返回 AP，重绘 UI
RESULT_DIR_ERROR_ENTER_DIR	错误返回，进入目录错误，包括 root
RESULT_DIR_ERROR_NO_FILE	错误返回，根目录下没有文件和文件夹
RESULT_DIR_ERROR_SETLOC	错误返回，顶层不是根目录情况下，set location 失败
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

版 权 所 有 侵 权 必 究

控件使用说明

1. 支持在指定层级文件列表前面添加菜单列表，比如在所有歌曲列表开头添加“随机播放”菜单，可以更人性化实现随机选择歌曲播放功能。一个文件浏览控件中支持嵌套多个菜单列表。

2. 快捷菜单：在文件浏览时，按下菜单键，可以弹出文件处理快捷菜单，用于支持删除文件、删除文件夹、复制/粘贴文件等功能。

3. 文件浏览控件已经实现了路径记忆，从 VRAM/缓冲区 中读取路径记忆来显示文件列表。

5.5.3.3 删除文件控件

文件浏览器需要文件系统和文件选择器的支持，在使用前必须保证文件系统和文件选择器成功初始化了。

控件的特性及设计

删除文件控件实际上并不是一个原子控件，而是由其他控件组合而成的特殊场景。它需要完成的任务包括：定位文件/文件夹、确认是否删除文件/文件夹、UI 显示正在删除文件/文件夹、提示文件已删除。

它使用情景有 2 种：

1. 删除正在播放的文件：这种情景已经解决了定位文件的任务，只需执行后续 3 个任务。

2. 从文件浏览开始删除文件：需要借助文件浏览器的功能定位目标文件/文件夹，然后再执行后续 3 个任务。这种情景下的删除文件可以实现为文件浏览的一个快捷功能。

不论是哪种情景，删除文件时都需要保证删除文件/文件夹的安全性，即如果目标按键正在播放，那么要先停止播放，释放播放器对目标文件的控制权，然后再执行删除。

控件接口设计

app_result_e **gui_delete** (file_path_info_t *path_info, del_com_data_t *del_com)

版 权 所 有 侵 权 必 究

```
/*!
 * \brief
 *  del_com_data_t 文件（文件夹，列表）删除控件初始化结构体
 */
typedef struct
{
    /*! 删除对象名称，Unicode编码必须有0xfeff前缀 */
    uint8 *filename;
    /*! 删除文件或文件夹序号，为文件同文件夹总和中的序号 */
    uint16 del_no;
    /*! 删除模式，0表示删除文件，1表示删除文件夹或列表，见 del_mode_e 定义 */
    uint8 del_mode;
    /*! 是否删除正在播放文件，TRUE表示是，FALSE表示否（仅对删除文件有效） */
    bool del_playing;
    /*! 是否删除文件夹或列表本身，只对删除文件夹或列表有效 */
    bool del_self;
    uint8 reserve[3];
    /*! delete 对话框中选择"是"后执行的函数，为NULL表示无需执行 */
    void (*del_func)(void);
} del_com_data_t;
```

说明：

1. del_func 函数正是为了解决删除文件/文件夹的安全性而增加的回调函数。我们可以在 del_func 中释放对目标文件的控制权。

2. filename 只是用作 UI 显示，而非定位目标文件/文件夹。

3. del_no 指向目标文件在当前文件列表中的位置，不为 0 表示用 del_no 定位目标文件，为 0 表示用 path_info 定位目标文件。

接口返回值如下表所示：

返回值	说明
RESULT_REDRAW	询问是否删除对话框取消返回 AP，重绘 UI
RESULT_DELETE_FILE	删除文件成功返回
RESULT_DELETE_DIRLIST	删除文件夹或列表成功返回
RESULT_DELETE_DIRLIST_NOSELF	RESULT_DELETE_DIRLIST_NOSELF 删除文件夹或列表（不删除自身）成功返回
RESULT_ERROR	RESULT_ERROR 删除文件失败返回
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

5.5.3.4 对话框控件

对话框是一种交互式的消息提示控件，弹出一个窗口提供用户发生了某件事，或者询问用户下一步如何操作等，然后一般会等待用户回答后根据用户的选择进行下一步处理。

控件的特性及设计

首先，我们会从业务逻辑的角度上把对话框分成 4 类：

1. 有按钮选择的对话框，在有需要询问用户，请求用户作出选择时使用。

2. 没有按钮，提示信息后，需要用户按任意按键才能返回，这种对话框可以用在一些提示内容较长的情况下，比如，对一个名词进行解释。

3. 没有按钮，提示信息后，自动延时 2 秒钟后返回，这是为了给用户足够的时间看清楚提示信息。

4. 没有按钮，提示信息后立即返回，这样应用即给用户做了提示且一直保持到下次刷屏，又不耽误后续业务处理的时间。这在提示之后还要进行较长时间业务处理的情景下使用。

```
/*!
 * \brief
 *  dialog_type_e 对话框类型枚举类型
 */
typedef enum
{
    /*! 含有按键选择的对话框 */
    DIALOG_BUTTON_INCLUDE        = 0x00,
    /*! 不含按键的简单提示对话框，弹出后按任意按键退出 */
    DIALOG_BUTTON_ANYKEY         = 0x01,
    /*! 不含按键的简单提示对话框，弹出后等待2秒钟后退出 */
    DIALOG_INFOR_WAIT            = 0x02,
    /*! 不含按键的简单提示对话框，弹出后立即退出 */
    DIALOG_INFOR_IMMEDIATE       = 0x03,
} dialog_type_e;
```

而在 UI 设计上，我们参考了 Windows 风格的对话框设计，设计细节如下：

1. 对于有按钮选择的对话框，我们提供了以下几种默认按钮组合供用户选择，在实现时，每个按钮都会对应一个事件（为了方便对对话框返回值进行处理，我们一般把 NO 、Cancel 等看成取消返回事件 RESULT_REDRAW ）。

1） Yes 和 No。

2） OK 和 Cancel。

3） Retry 和 Cancel。

4） Abort、Retry 和 Ignore。

5） Yes、No 和 Cancel。

按钮结构体如下所示：

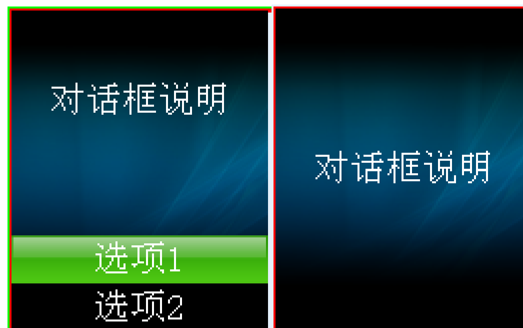
```
/*!
 * \brief
 *  button_list_item_t 按键列表项结构体
 */
typedef struct
{
    /*! 按键总数 */
    uint8 button_cnt;
    /*! 按键-消息映射数组，最多3个按键 */
    struct
    {
        /*! 按键资源字符串ID */
        uint16 str_id;
        /*! 按键资源字符串ID */
        uint16 result;
    } buttons[3];
} button_list_item_t;
const button_list_item_t button_list[] =
{
    {2, {{S_BUTTON_OK, RESULT_DIALOG_OK}, {S_BUTTON_CANCEL, RESULT_REDRAW}, \
        {0, 0}}},
    {3, {{S_BUTTON_ABORT, RESULT_DIALOG_ABORT}, {S_BUTTON_RETRY, RESULT_DIALOG_RETRY}, \
        {S_BUTTON_IGNORE, RESULT_REDRAW}}},
    {3, {{S_BUTTON_YES, RESULT_DIALOG_YES}, {S_BUTTON_NO, RESULT_DIALOG_NO}, \
        {S_BUTTON_CANCEL, RESULT_REDRAW}}},
    {2, {{S_BUTTON_YES, RESULT_DIALOG_YES}, {S_BUTTON_NO, RESULT_REDRAW}, \
        {0, 0}}},
    {2, {{S_BUTTON_RETRY, RESULT_DIALOG_RETRY}, {S_BUTTON_CANCEL, RESULT_REDRAW}, \
        {0, 0}}},
    {2, {{0, 0}, {0, 0}, {0, 0}}}
};
/*!
 * \brief
 *  button_type_e 对话框按键组合类型枚举类型
 */
typedef enum
{
    /*! 确定和取消 */
    BUTTON_OKCANCEl              = 0x00,
    /*! 终止、重试和忽略 */
    BUTTON_ABORTRETRYIGNORE      = 0x01,
    /*! 是、否和取消 */
    BUTTON_YESNOCANCEL           = 0x02,
    /*! 是和否 */
    BUTTON_YESNO                 = 0x03,
    /*! 重试和取消 */
    BUTTON_RETRYCANCEL           = 0x04,
    /*! 没有按钮 */
    BUTTON_NO_BUTTON             = 0x05,
} button_type_e;
```

2. 为了更加直观形象，我们允许用户指定一个图标显示在对话框上，以警示用户。根据消息的严重性，给出不同的图标，比如：

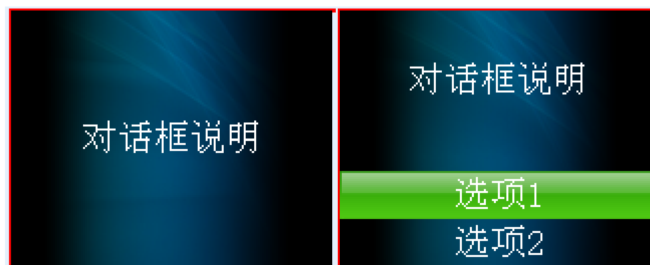
版 权 所 有 侵 权 必 究

控件场景 UI 模板

Common UI Editor 工程为对话框控件提供了 4 个控件模板，如下图所示：



DIALOG_MSG　　DIALOG_ASK



DIALOG_MSG_V　　DIALOG_ASK_V

如果以上控件模板没有合适的，用户可自己在 Common 工程创建自己的控件模板。关于控件模板的进一步介绍，请参考 可配置化 UI 一节。

控件接口设计

app_result_e **gui_dialog** (uint16 style_id, dialog_com_data_t *dialog_com)：全功能的对话框。

app_result_e **gui_dialog_msg** (uint16 style_id, dialog_type_e type, uint16 str_id)：用于无按钮的简单消息提示型对话框。

版 权 所 有 侵 权 必 究

```
/*!
 * \brief
 *  dialog_com_data_t 对话框控件初始化结构体
 */
typedef struct
{
    /*! 对话框类型，参见 dialog_type_e 定义 */
    uint8 dialog_type;
    /*! 对话框按键组合类型，参见 button_type_e 定义 */
    uint8 button_type;
    /*! 对话框标识图标ID，值为-1表示不显示 */
    uint16 icon_id;
    /*! 详细说明字符串联合体 */
    string_desc_t *string_desc;
    /*! 默认激活按键项 */
    uint8 active_default;
    /*! 对话框刷新模式，仅供应用层使用 */
    dialog_draw_mode_e draw_mode;
    /*! 保留字节 */
    uint8 reserve[2];
} dialog_com_data_t;
```

接口返回值如下表所示：

返回值	说明
RESULT_DIALOG_XXX	选择了有效按键后返回
RESULT_REDRAW	无效返回，AP 需要重绘 UI
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

5.5.3.5 USB 连接对话框

USB 连接对话框是对话框控件的一个特例，专用于 USB 线插入后进行连接模式选择。在 UI 设计上，就是在带按钮的对话框基础上添加自动选择计时提示。

控件场景 UI 模板

Common UI Editor 工程为 USB 连接对话框控件设计的控件模板，如下图所示：

版权所有 侵权必究

如果以上控件模板不合适，用户可自己在 Common 工程创建自己的控件模板。关于控件模板的进一步介绍，请参考 可配置化 UI 一节。

控件接口设计

app_result_e **gui_usbconnect** (void)

接口返回值如下表所示：

返回值	说明
RESULT_USB_TRANS	选择 USB 数据传输模式
RESULT_USB_PLAY	选择 USB 充电播放模式或者取消返回默认连接方式
RESULT_REDRAW	无效返回，AP 需要重绘 UI
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

5.5.3.6 参数设置框控件

参数设置框是用来设置参数最简洁的控件之一，一般用于线性参数的设置，并且支持多参数设置，这样就可以用来设置日期和时间等参数。

控件的特性及设计

参数设置框在风格上分为 3 类：

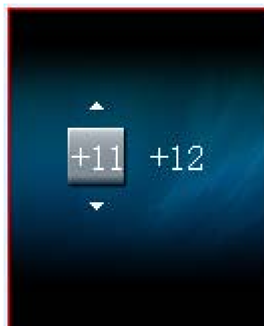
1. 数值型设置框：这种风格能够突出参数的值，用于年月日和时间之类的设置。
2. 滑动条设置框：这种风格能够突出参数的状况，让用户知道当前设置值相对于整个设置区间的位置，主要用于音量、亮度之类的设置。
3. 数值型和滑动条设置框：这种风格主要突出参数的状况，还附带给出具体的参数值，主要用于背光时间、定时关机之类的设置。

控件场景 UI 模板

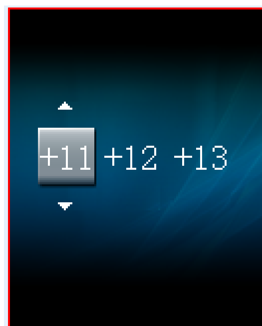
Common UI Editor 工程为对话框控件提供了 10 个控件模板，如下图所示：

版 权 所 有 侵 权 必 究

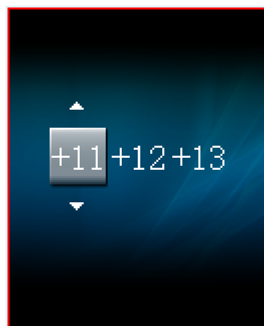
PARAM_SIMPLE



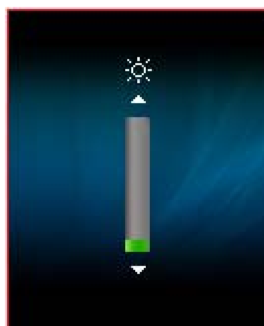
PARAM_TIME_24

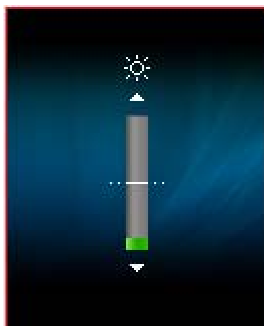


PARAM_TIME_12

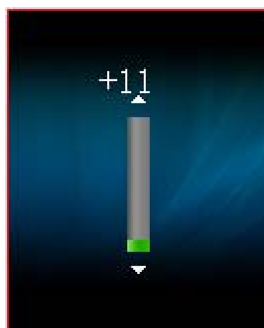


PARAM_DATE

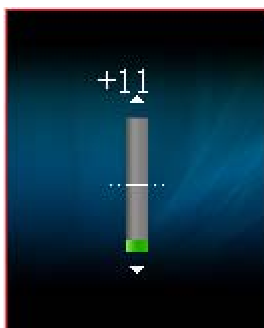

SLIDER_SIMPLE


SLIDER_SIMPLE_SYM

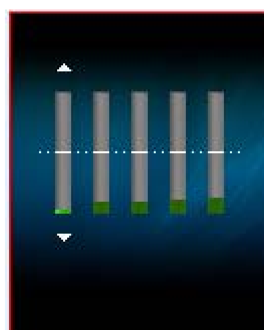
SLIDER_SIMPLE_V



SLIDER_NUM



SLIDER_NUM_SYM


SLIDER_SETEQ

如果以上控件模板没有合适的，用户可自己在 Common 工程创建自己的控件模板。关

于控件模板的进一步介绍，请参考 可配置化 UI 一节。

控件接口设计

app_result_e **gui_set_parameter** (uint16 style_id, param_com_data_t *param_com)

```
/*!
 * \brief
 *  param_com_data_t 参数设置控件初始化结构体
 */
typedef struct
{
    /*! ParamBox私有数据结构指针 */
    parambox_private_t *private_data;
    /*! ParamBox刷新模式, 仅供应用层使用 */
    parambox_draw_mode_e draw_mode;
} param_com_data_t;

/*!
 * \brief
 *      parambox_private_t parambox私有数据结构
 */
typedef struct
{
    /*! 背景图片ID, 允许用户更换模板背景图片 */
    uint16 back_id;
    /*! 标志图标ID */
    uint16 icon_id;
    /*! 设置标题资源字符串ID */
    uint16 title_id;
    /*! 设置参数个数 */
    uint8 param_cnt;
    /*! 当前激活项, 也作为多参数设置时默认激活项
    uint8 active;
    /*! 之前激活项 */
    uint8 old;
    /*! 是否显示参数值符号, 0表示不显示, 1表示显
    uint8 sign_flag;
    /*! 刻度像素点值, 仅对slider有效 */
    uint8 scale;
    uint8 reserve;
    /*! 设置参数列表 */
    parambox_one_t *items;
} parambox_private_t;

/*!
 * \brief
 *      parambox_one_t parambox参数描述符结构体
 */
typedef struct parambox_one_struct
{
    /*! 参数单位资源字符串ID, 其实也不限于单位, 可以是任意辅助说明字符
    uint16 unit_id;
    /*! 参数值最小值 */
    int16 min;
    /*! 参数值最大值 */
    int16 max;
    /*! 参数值步进 */
    int16 step;
    /*! 参数当前值 */
    int16 value;
    /*! 是否允许循环设置, 即最大递增变为最小, 最小递减变为最大 */
    uint8 cycle;
    /*! 参数值最大位数 */
    uint8 max_number;
    /*! 参数当前值字符串显示, 可以通过adjust_func转换 */
    uint8 *value_str;
    /*! 参数值检测适配回调函数, 比如用来限制日期设置等; 这里传递进来的
     *  参数数组, 我们假设在多参数设置中, 参数位置是固定的, 或者是可以
     *  过全局变量获得的;
     */
    adjust_result_e (*adjust_func)(struct parambox_one_struct *params,
    /*! 设置即时回调函数, 比如声音设置等 */
    bool (*callback)(int16 value);
} parambox_one_t;
```

接口返回值如下表所示：

返回值	说明
RESULT_CONFIRM	确定参数设置返回
RESULT_REDRAW	参数设置无效返回 AP，重绘 UI
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

控件使用说明

参数设置中，有时候参数值（或其范围）会根据某些条件而有略微变化，比如在设置日期中，“日”的范围会根据“年”“月”有所区别，而我们进入了参数设置控件后，又不能再干预参数值，那么应该如何解决呢？我们想到了一种方法，为参数增加一个回调函数 **adjust_func** ，实时的去对参数的数值进行检测，发现超出范围了，就会自动跳转到某个合法数值去，这样就解决问题了。

另外，某些参数设置中，需要实时的预览参数设置的效果，这样我们还需要提供另一个回调函数 **callback** 来完成效果预览。

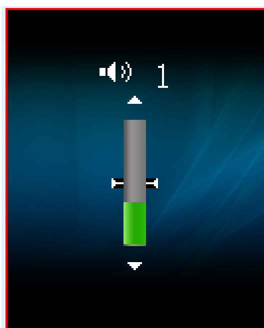
版 权 所 有　侵 权 必 究

5.5.3.7 音量条控件

音量条是参数设置框控件的一个特例，专用于当前音量值和其他音量相关参数设置，其中调节当前音量值设计为一种快捷键功能。在 UI 设计上，音量条就是在滑动条设置框的基础上添加额外 UI 元素。

控件场景 UI 模板

Common UI Editor 工程为音量条控件设计的控件模板，如下图所示：



如果以上控件模板不合适，用户可自己在 Common 工程创建自己的控件模板。关于控件模板的进一步介绍，请参考 可配置化 UI 一节。

控件接口设计

app_result_e **gui_volumebar** (uint8 *volume, uint8 *limit, uint8 mode)

接口返回值如下表所示：

返回值	说明
RESULT_CONFIRM	确定音量设置返回
RESULT_REDRAW	取消音量设置退出，对于调节音量限制将不保存结果（调节当前音量值时 4 秒钟没操作会自动返回 RESULT_REDRAW）
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

5.5.3.8 文本阅读框控件

文本阅读框是一个用来显示文本文件的控件，把文本文件分成若干页显示，通过 Next/Prev 上下切换。

控件的特性及设计

文本阅读框可支持的编码类型包括 ANSI（建议只有英文才使用）、UNICODE 16（小端）和 UTF-8。

在文本文件的解码算法上，我们只提供从当前位置解释一页的接口，并不提供从当前位置往前解释一页的接口。这样就要求上层应用必须把解码出来的结果，即每一页的偏移位置，缓冲起来，然后在向上翻页时再从缓冲中读出上一页偏移位置来解码。

另外，解码算法为了能够处理 SD 中的文本文件和用户文件系统中的文本文件，我们抽象了文本文件定位和读接口。我们只要把 SD / 文件系统 的定位和读接口封装下就可以轻松实现上述目标。

控件接口设计

1. 文本统一解码接口如下：

text_end_mode_e **text_decode_one_page**(text_decode_t *text_decode, text_file_t *text_file, uint16 *page_bytes)

```
/*!
 * \brief
 *  text_buf_t 文本阅读解码缓冲区结构体
 */
typedef struct
{
    /*! 一行字符串缓冲区 */
    uint8 text_prev[BUFF_ONE_ROW];
    /*! 一个扇区缓冲区, 512个字节 */
    uint8 text_buf[FILE_SECTOR];
    /*! 缓冲区中有效数据指针 */
    uint8 *text_buf_valid;
    /*! 缓冲区中有效数据长度 */
    uint16 text_buf_length;
    /*! 缓冲区中剩余字节数 */
    uint8 remain;
    /*! buffer 数据是否有效 */
    bool valid;
    /*! 文本显示参数结构体 */
    text_show_param_t param;
    /*! text 显示行回调函数 */
    void (*text_show_line)(string_desc_t *string_desc, uint8 index);
} text_decode_t;
```

```
/*!
 * \brief
 *  text_file_t 文本阅读虚拟文件结构体，为了兼容SD文件和文件系统文件
 */
typedef struct
{
    /*! 打开的文件句柄 */
    void* file_handle;
    /*! 文件总扇区数 */
    uint32 file_sectors;
    /*! 文件总字节数 */
    uint32 file_length;
    /*! 文件偏移地址，以字节为单位 */
    uint32 file_offset;
    /*! 文件定位接口函数指针，以扇区为单位 */
    bool (*fseek) (void* file_handle, uint8 where, int32 sector_offset);
    /*! 文件数据读接口函数指针，以扇区为单位 */
    bool (*fread) (void* file_handle, uint8 *text_buf, uint32 sector_count);
} text_file_t;
```

该接口返回参数为当前页字节数，解码失败返回-1。

该接口返回值如下：

```
/*!
 * \brief
 *  电子书断行结束类型
 */
typedef enum
{
    /*! 初始断行结束类型 */
    TEXT_END_INIT           = 0,
    /*! 缺页结束 */
    TEXT_END_PAGE_MISS      = 1,
    /*! 遇到 '\0' 字符结束 */
    TEXT_END_NUL            = 2,
    /*! 遇到换行符结束 */
    TEXT_END_LR             = 3,
    /*! 长度已经超长结束 */
    TEXT_END_OVER           = 4,
    /*! 读取文本数据失败 */
    TEXT_END_READ_ERR       = 5,
} text_end_mode_e;
```

2. SD 文本文件阅读接口设计如下：

app_result_e **gui_text_read** (uint16 style_id, const char *file_name)

其中 file_name 为 SD 文件命名方式，如 legal.txt 。

接口返回值如下表所示：

返回值	说明

RESULT_REDRAW	正常返回 AP，重绘 UI
RESULT_ERROR	无法打开文件等，返回错误（除非系统 bug，否则不可能返回）
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 结果返回

SD 文本文件阅读控件使用说明

1. SD 文本文件阅读可以用来阅读法律信息、用户使用指南等。

2. 由于 SD 中的文本文件一般比较短，我们限定文本最长不能超过 32 页，这样在翻页过程中，就可以把每页起始位置缓冲起来，以便往前翻页。

3. SD 中的文本文件是扇区对齐存放的，阅读时并不知道文件确切的结束点，所以必须自己在 SD 文本文件的结尾添加结束符 **'\0'**，UNICODE 16 必须添加 2 个 **'\0'**。

4. 在调用 gui_text_read 接口时要注意，一般不能直接文件名字填写在接口参数列表上，而要先把名字拷贝到栈中，以栈中的文件名字传递，使用时请参考 ap_setting 例子。这是因为调用 gui_text_read 时通常会发生 bank 切换，把当前 bank 代码冲掉，如果文件名字放在当前 bank 的.text 段或.rodata 段中，那就会被冲掉。

5.5.3.9 状态栏控件

状态栏控件显示当前系统状态，包括当前前台应用的类型，电池状态，插卡状态，电缆连接状态，系统时间等。状态栏除了显示以上状态图标或字符串外，还有剩余空间可以用来显示前台应用某些 UI 元素，这些必须在状态栏初始显示完之后才可显示。

控件的特性及设计

状态栏实现为周期更新，检测周期为 0.5S，一旦检测到有状态变化时才会去更新。当然，用户也可以主动调用状态栏显示接口强制更新。

控件接口设计

app_result_e **gui_headbar**(headbar_update_e update)：状态栏显示接口

版 权 所 有 侵 权 必 究

```
/*!
 * \brief
 *   headbar_update_e 状态条更新模式枚举类型
 */
typedef enum
{
    /*! 初始更新状态栏，除了会全部更新状态栏外，还会初始化状态栏相关全局变量 */
    HEADBAR_UPDATE_INIT      = 0,
    /*! 全部更新状态栏 */
    HEADBAR_UPDATE_ALL       = 1,
    /*! 更新变化了的状态栏 */
    HEADBAR_UPDATE_CHANGE    = 2,
} headbar_update_e;
```

void **gui_headbar_handle**(void)：检测状态栏是否发生变化，如果发生变化就调用 gui_headbar 进行更新。该接口在系统定时器 sys_status_timer_id 的 handle 函数 sys_status_handle 调用。

void **gui_set_headbar_mode**(headbar_mode_e mode, headbar_icon_e icon_id)：设置状态栏模式。

void **gui_get_headbar_mode**(headbar_mode_e *mode, headbar_icon_e *icon_id)：获取当前状态栏模式。

```
void gui_set_headbar_mode(headbar_mode_e mode, headbar_icon_e icon_id)
{
    this_headbar_mode = mode;
    this_headbar_icon_id = icon_id;

    switch(mode)
    {
    case HEADBAR_MODE_NORMAL:
        this_headbar_style.icon_sty = V_U16_INVALID;
        this_headbar_style.battery_sty = HEADBAR_BATTERY;
        this_headbar_style.bg_sty = V_U16_INVALID;
        this_headbar_style.time_sty = V_U16_INVALID;
        this_headbar_style.cable_sty = V_U16_INVALID;
        this_headbar_style.card_sty = HEADBAR_CARD;

        g_headbar_update = HEADBAR_UPDATE_INIT;
        restart_app_timer(sys_status_timer_id);
        break;

    default:
        stop_app_timer(sys_status_timer_id);
        break;
    }
}
```

状态栏要显示哪些元素，是通过 this_header_style 配置实现的，如果对应的控件有效就进行更新显示。

get_set_headbar_mode 接口还控制着系统定时器 sys_status_timer_id 的启动和停止。

5.5.3.10 动画显示控件

动画显示控件是多帧 PicBox 控件的连续显示控件，实现从头到尾或从尾到头动态显示。

控件的特性及设计

动画显示具有如下属性：

1. 连续 2 帧图片显示时间间隔。
2. 动画显示方向，正向动画显示和反向动画显示。
3. 能够在任意位置终止，比如按键锁 UI，在上锁动画中可以终止并继而显示解锁动画。
4. 能够在连续 2 帧图片显示中执行某些处理，比如显示完前一帧图片后在图片上面显示其他 UI 元素。

控件接口设计

app_result_e **gui_animation** (style_infor_t *style_infor, animation_com_data_t *anm_com)：使用应用级定时器定时显示图片，并且接收消息，可以由按键消息和应用私有消息终止动画显示。

app_result_e **gui_logo** (style_infor_t *style_infor, animation_com_data_t *anm_com)：直接使用 sys_os_time_dly 延时定时显示图片，不接收消息，不能由按键消息和应用私有消息终止动画，所以该接口一般只用于开机关机 logo。

```
/*!
 * \brief
 *  animation_com_data_t 动画显示初始化结构体
 */
typedef struct
{
    /*! 每帧图片显示时间间隔，单位是1ms */
    uint16 interval;
    /*! 动画方向，可以选择正向（0）和方向（1）  */
    uint8 direction;
    /*! 是否允许中止，TRUE表示不允许中止，FALSE表示允许中止 */
    uint8 forbid;
    /*! 终止动画显示的按键映射列表，为NULL表示允许所有快捷键终止动画 */
    const key_map_t *key_map_list;
    /*! 每帧动画中执行回调函数 */
    void (*callback)(void);
} animation_com_data_t;
```

接口返回值如下表所示：

返回值	说明

RESULT_NULL	动画显示正常结束后返回 AP
RESULT_REDRAW	应用消息分发处理返回结果 RESULT_REDRAW
其他非 RESULT_NULL 返回值	动画显示终止返回，或因用户 gui 输入而终止，依据 key_map_list 映射返回消息，或收到返回值不为 RESULT_NULL 返回结果返回

5.5.3.11 按键锁控件

按键锁控件用来提示界面上锁、界面解锁和锁住状态。

控件的特性及设计

系统按键锁状态定义如下：

1. 系统开机后，初始化为“没锁状态”；如果 HOLD 为拨动开关，则先读取按键锁状态，如果读取到为“锁住状态”，则要显示“锁住状态”图片。

2. 如果系统处于“没锁状态”，这时按下锁按键或者把拨动开关拨到锁住位置，那么必须显示“上锁”动画。

3. 如果系统处于“锁住状态”，这时按下锁按键或者把拨动开关拨到解锁位置，那么必须显示“解锁”动画。

4. 如果系统处于“锁住状态”，按下任何其他物理按键，除了关机键外，则要显示“锁住状态”图片；另外，在“锁住状态”提示中，不能重复提示。

5. 在显示动画或者锁住状态提示中，如果按下锁按键或者把拨动开关的状态切换了，那么必须终止当前动画或图片提示，显示“上锁”或“解锁”动画。

对于使用普通按键实现的按键锁，我们一般不会在按键驱动中处理按键锁逻辑，而把这部分逻辑放到应用层处理。

而按键锁的设计需要达到这样的目标：对应用程序来说完全透明，也就是说要在 Common 解决按键锁的所有细节问题。

所以我们把按键锁当作快捷键处理，上锁和解锁比较好实现，直接在快捷键映射表中把锁按键映射为锁事件，并根据当前按键锁的状态即可判断是上锁还是解锁。对于在“锁住状态”下按其他按键显示“锁住状态”图片的处理就需要使用一点技巧了。在 Common_func 模块我们讲过按键消息预处理，解决方法正好利用它。我们在 com_gui_msg_hook 函数中，判断到处于“锁住状态”，就把其他按键消息转换为虚拟按键 KEY_LOCKHOLD，同样该按键当作快捷键处理。

控件接口设计

app_result_e **gui_keylock** (bool lockkey)

接口返回值如下表所示：

返回值	说明
RESULT_REDRAW	正常返回 AP，重绘 UI
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 应用消息处理返回

5.5.3.12 屏幕保护

当系统闲置一段时间后，进入屏幕保护场景，可以显示数字时间，显示当前正在播放的音乐的 album art 图片，显示演示图片，或者关掉屏幕背光灯等。

控件的特性及设计

屏幕保护其实和屏幕变黑的处理有点类似，都没有进入新场景，仍然运行原来场景的消息循环，只是不再刷新原场景的 UI。

屏幕保护会启动其他的 UI 显示，并且通常需要定时器周期更新，所以我们为屏幕保护分配了一个应用级定时器场景 tag，这样保证屏幕保护与原场景不会混乱。

控件接口设计

app_result_e **gui_screen_save_enter** (screen_save_mode_e ss_mode)：进入屏幕保护。

app_result_e **gui_screen_save_exit** (void)：退出屏幕保护。

```
/*!
 * \brief
 *  screen_save_mode_e 屏幕保护类型枚举类型
 */
typedef enum
{
    /*! 关闭屏幕保护 */
    SCREEN_NULL                 = 0x00,
    /*! 数字时钟，显示日期和时间 */
    SCREEN_DIGIT_CLOCK          = 0x01,
    /*! 音乐专辑图片，如果不是前台应用不是音乐，切换到数字时钟 */
    SCREEN_ALBUM_ART            = 0x02,
    /*! 关掉lcd背光灯 */
    SCRREN_SCREEN_OFF           = 0x03,
    /*! 演示模式，显示各种应用Demo界面 */
    SCREEN_DEMO_MODE            = 0x04,
} screen_save_mode_e;
```

5.5.3.13 关机对话框

关机对话框是一个简单的快捷键功能，当用户按住关机键时，弹出关机对话框，显示按住关机键的时长，如果计时到 3 秒或其他，则表示用户确认要关机而非按错了关机键。

控件场景 UI 模板

Common UI Editor 工程为关机对话框控件设计的控件模板，如下图所示：

如果以上控件模板不合适，用户可自己在 Common 工程创建自己的控件模板。关于控件模板的进一步介绍，请参考 可配置化 UI 一节。

控件接口设计

app_result_e **gui_shut_off** (void)

接口返回值如下表所示：

返回值	说明
RESULT_APP_QUIT	按住关机键 3 秒确认关机，返回后应用应无条件退出
RESULT_REDRAW	放弃关机返回，返回 AP 重绘 UI
其他非 RESULT_NULL 返回值	收到返回值不为 RESULT_NULL 应用消息处理返回

5.5.4 Common_misc 设计与使用

应用公共子功能杂项接口库 common_misc，包括系统定时器，应用睡眠，默认消息处理，应用私有消息预处理，屏幕方向设置，背光亮度映射，电池电量映射，闹钟消息处理，天线检测处理，等等。

这部分接口是只能为前台应用使用，后台应用不能也不需要使用。

5.5.4.1 系统定时器

系统定时器指所有前台应用共同拥有的应用级定时器，用于提供所有前台应用某些行为一致的定时服务。

US212A 实现了 2 个系统定时器：

1. CONTROL 类定时器 **sys_counter_timer_id**，用于实现定时屏幕变暗/黑、屏幕保护、返回正在播放界面、省电关机、定时关机以及低电和充电满电检测等功能。该定时器的周期为 500 ms。

2. UI 类定时器 **sys_status_timer_id**，用于实现周期状态栏更新功能。该定时器的周期为 500 ms。

前面介绍应用级定时器时说过，应用级定时器带有场景 tag，系统定时器的场景 tag 为 APP_TIMER_TAG_SYS，这样就保证了系统级定时器在任何一个前台应用都是有效的，当然 sys_status_timer_id 在屏幕变黑的情况下是不会执行的。

接口设计如下：

void **sys_timer_init**(void)：系统定时器初始化，创建了以上 2 个定时器。前台应用（除了 ap_config 、ap_udisk 、ap_playlist 等特殊应用外）在应用级定时器初始化之后就应该调用该接口初始化。需要强调的一点是，sys_status_timer_id 创建成功之后，会调用 stop_app_timer(sys_status_timer_id) 停止使用，直到调用 gui_set_headbar_mode 设置状态栏并启动计时。

void **sys_timer_exit**(void)：系统定时器销毁，在应用退出时调用删除以上 2 个定时器。

5.5.4.2 应用睡眠

应用睡眠，即让前台应用等待一段时间。

在应用睡眠时，用户可以通过按键让应用提前退出睡眠。提前退出睡眠的按键定义有 2 种，一种是任意按键都会退出，并且会过滤后续按键动作，另一种是指定按键消息映射表，只有匹配了按键映射表的有效事件才会退出。一般应用情景选择第一种。

从上面需求知道，应用睡眠时只需要处理按键消息循环，所以睡眠定时可以简单使用 sys_os_time_dly 释放应用控制权的方式实现。

接口设计如下：

app_result_e **com_app_sleep**(uint32 sleep_timer, const key_map_t *key_map_list)

版 权 所 有 侵 权 必 究

接口返回值如下表所示：

返回值	说明
RESULT_NULL	睡眠时间到时返回
gui_event	系统消息返回处理，避免嵌套

5.5.4.3 默认消息处理

系统中有些公共消息/事件一般情况下采用一种默认方式处理，包括快捷键事件、系统消息和其他公共消息/事件等。当然，并不是说公共消息/事件就只能按照默认方式处理，如果用户对某个公共消息/事件感兴趣，可以在自己的消息/事件分发列表中优先处理掉。

US212A 方案处理的公共消息/事件如下：

1. 快捷键事件：按键锁事件、关机对话框事件、音量条事件、返回主界面事件、等等。
2. 系统消息：USB 插入消息、USB 拔出消息、充电器 ADAPTOR 插入消息、充电器 ADAPTOR 拔出消息、卡插入消息、卡拔出消息、U 盘插入消息、U 盘拔出消息、RTC 定时闹钟消息、耳机（天线）插入消息、耳机（天线）拔出消息、等等。
3. 其他公共消息/事件：低电消息，充电满电消息、屏幕保护消息、返回正在播放消息、等等。

接口设计如下：

app_result_e **com_message_box**(msg_apps_type_e msg_type)

接口返回值如下表所示：

返回值	说明
RESULT_NULL	返回空结果，AP 可不进行任何处理
RESULT_REDRAW	返回重绘 UI，AP 一般需要重新 redraw 所有 UI
其他非 RESULT_NULL 返回值	直接返回消息（事件）处理的结果

使用方法如下例子：

```
例子1：在按键消息（事件）循环中调用，处理快捷按键事件
input_gui_msg_t input_msg;
msg_apps_type_e gui_event;
bool ret;

ret = get_gui_msg(&input_msg);
if(ret == TRUE)
{
    if(com_key_mapping(&input_msg, &gui_event, key_map_list) == TRUE)
    {
        switch(gui_event)
        {
        case EVENT_DO_SOMETHING:
            break;

        default:
            result = com_message_box(gui_event);
            break;
        }
    }
}
else
{
    gui 消息（事件）处理完毕，接下来可以处理应用私有消息了
}

例子2：在应用私有消息处理中，处理默认系统消息
private_msg_t private_msg;
msg_apps_type_e msg_type
app_result_e result;

if(get_app_msg(&private_msg) == TRUE)
{
    msg_type = private_msg.msg.type;
    switch(msg_type)
    {
    case XXX://处理应用私有消息，或者一些应用需要特殊处理的系统消息
        break;

    default:
        result = com_message_box(msg_type);
        break;
    }
}
```

5.5.4.4 应用私有消息预处理

在上面一节我们说到用户可以在自己的消息/事件分发列表中优先处理掉公共消息/事

件，那么也就意味着有一些系统消息可能不会执行默认处理，而如果我们在默认处理中放置一些重要处理，那么这些重要处理也就丢失了，从而导致系统某些方面失控了。

为此，我们参考了按键消息预处理的机制，为应用私有消息也增加了预处理，也就是说，只要收到应用私有消息，那么就一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。

当然，预处理是一种对应用私有消息干预的机制，允许修改消息等。

US212A 方案中，预处理实现的功能包括：

1. 接收到重要消息时，恢复屏幕亮度、退出屏幕保护、清零空闲计时器以从新开始计时背光变暗定时等、等等。这里说的重要消息包括拔插卡、拔插 U 盘、USB 拔插（包括充电线拔插）、关机和低电、等等。

2. 系统把 USB 拔插和充电线拔插 混在一起处理，所以需要在预处理中区分是 USB 拔插 还是充电线拔插。

5.5.5 Common 的空间分配

5.5.5.1 数据空间

前台应用的数据空间

0x9fc1d200-0x9fc1d9ff 共 2KB，该全局数据空间是 AP、Common 模块以及 enhanced 模块共用的。

后台应用的数据空间

0x9fc1da00-0x9fc1dfff 共1.5KB，该全局数据空间是 AP、Common模块以及enhanced模块共用的。

Case全局数据空间

0x9fc19f80-0x9fc19fff 共128B，该全局数据空间是所有AP共享的，只会在 ap_manager 中初始化一次，主要用来存放 Common 模块的一些 case 全局数据。

5.5.5.2 代码空间

前台应用的代码空间

前台应用常驻代码段：0xbfc1ee00-0xbfc1f5ff 共2KB，用来存放 AP、Common模块、enhanced模块的常驻接口及相关const data、bank data。

Front Control bank 组：(0x40**0000+0x1fe00)-(0x40**0000+0x205ff) 共2KB，主要用

来存放AP 非常驻接口、Common 控件场景等。其中 Common 模块占用 control bank 组中 bank 48 ~ bank 63这16 bank号。

Front UI bank 组：(0x48**0000+0x1f600)-(0x48**0000+0x1fdff) 共2KB，主要用来存放 Common 模块其他接口等。另外需要特别说明的一点是，由于 UI bank 组基本上是 Common 模块独占的，所以 Common 模块可以很好的进行分配，为了实现 AP bank 间大数据的传递/共享，我们把 (0x48**0000+0x1f600)-(0x48**0000+0x1f7ff) 共0.5KB用来存放 Common 模块的一些 bank data，所以UI bank段在很多时候实际上可用的空间只有 1.5KB。

另外，有些AP在某些特殊情境下，需要把部分接口放在 UI bank组，以避免因放在 Control bank 组中频繁切bank的情况，所以我们预留 UI bank 组中bank 0 ~ bank 15 这16 bank号给AP，所以实际上Common 模块只能使用bank 16 ~ bank 63这 48 bank号。

后台应用的代码空间

后台应用常驻代码段：0xbfc1e800-0xbfc1edff 共1.5KB，主要用来存放AP、Common模块、enhanced模块的常驻接口及相关const data、bank data。

Back Control bank 组：(0x60**0000+0x1e000)-(0x60**0000+0x1e7ff) 共2KB，主要用来存放AP 非常驻接口、Common 非常驻接口等。其中 Common模块占用bank 40 ~ bank 63 这 24 bank号。

说明：地址中的**高 6bit 表示 bank 组内的 bank 号，比如0x10则表示 bank 4。

5.5.5.3 空间分配说明

1. 充分考虑 bank 机制的特点：
   1) bank数据在生命周期中不能被切换出去，除了一些可重新读回的纯数据，在这种情况下，切换回来后需要重新加载 bank data。
   2) 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，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。
2. 满足 bank 切换目标：
   1) 对于后台应用，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，原则上要做到在屏幕变黑后并且没有按键动作，也没有收到应用私有消息的情况下不会发生 bank 切换。
3. 特别说明：
   1) 应用级定时器的 ISR 的执行从某种程度上对用户是透明的，其调用时机并不

版 权 所 有 侵 权 必 究

是直接由用户控制的，所以有时用户并没有特别在意由于应用级定时器 ISR 引起的bank 切换。所以在这里特别提出来说明一下。要想减少因此而引起的 bank 切换，那么要遵循规定：尽量避免将应用级定时器的 ISR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应用级定时器的 ISR 集中存放在尽量少 bank 上。

5.5.5.4 链接脚本模板

Common 模块提供了 3 个不同的链接脚本模板，这些模板都是可以直接使用的，应用只需要在 makefile 中把适合的 xn 文件添加到 LD_SCRIPT 参数值列表中即可：

1. common_front.xn：完全功能的前台应用 Common 模块链接脚本。
2. common_front_no_selector.xn：没有文件选择器的前台应用 Common 模块链接脚本。
3. common_engine.xn：后台应用 Common 模块链接脚本。

前台应用 Common 模块的链接脚本之所以区分有无文件选择器，是因为有文件选择器的前台应用，必须链接 enhanced 模块，其链接工程会复杂很多。

使用相同的 Common 模块链接脚本有一个好处，就是不管是哪个应用程序，相同 Common 接口链接地址是一致的，方便程序调试。但是这样有些应用程序会把本来可以不用打包进去的 Common 接口也一同打包进去了，浪费了固件空间。对于 nandflash 方案来说不成问题，但对于 nor flash 方案，如果浪费的空间较多的话，那就应该考虑从模板中抽取出最小链接脚本。

5. 6 前台应用设计与开发

前台应用，也叫 UI 应用，是指需要进行 UI 显示并通过 GUI 与用户交互的应用。

5.6.1 前台应用的组成结构

5.6.1.1 应用组成部分

前台应用主要由以下几个部分组成：

- 应用主体，即应用业务相关部分，以场景为业务单元，整个应用可以理解为一个场景调度循环，而每个场景又是一个消息处理循环

版 权 所 有 侵 权 必 究

- Common 模块，即应用基础功能接口库
- 中间件和函数库，包括文件浏览器，文件选择器，歌词解释器，ID3 解释器，USB 函数库，等等
- 运行时库，ctor.o，应用被调度时，会先运行该库中的 __start 函数，由该函数调用应用的 main 函数
- API 接口库，api.a，kernel 和各驱动的访问入口
- make 脚本，*.makefile
- xn 链接脚本，*.xn ，一般来说，前台应用包括 3 个 xn 脚本，分别是 AP 自身链接脚本、enhanced 部分链接脚本和 common 部分链接脚本；链接脚本也充当 AP 打包的配置文件
- 配置文件，包括可配置化 UI *.sty、可配置化菜单 *.mcg 和 固件配置化 config.bin

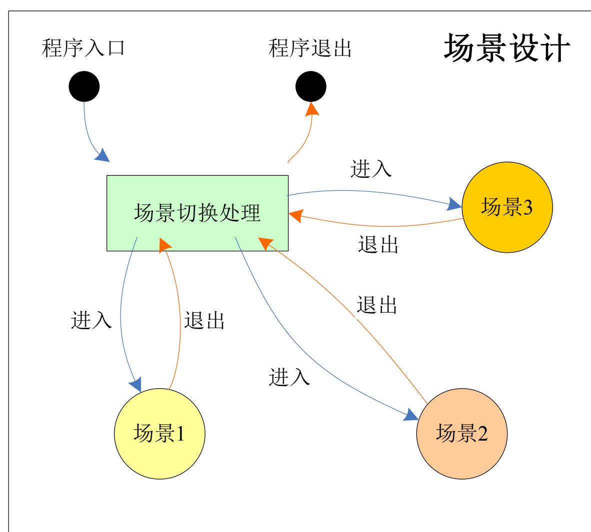
说明：前台应用并没有包含解码/编码中间件和库，为了更内聚的模块管理，我们把解码/编码中间件和库打包为 *.al 文件，并且以子线程的方式加载并运行。

更多关于 *.al 文件的说明请参考 一节。

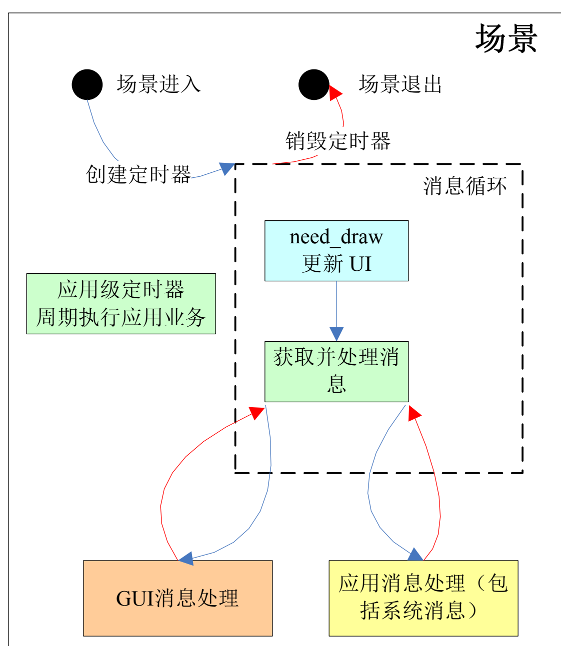
5.6.1.2 应用基本架构

前台应用以场景调度循环->消息处理循环为基本架构。所谓场景，就是在应用设计时，为了使工作可拆分，往往会将某一个主要界面（一般是全屏界面）以及它周边的一些次要界面（一般是非全屏界面）在同一个控制循环中实现，我们称这个控制循环和这些界面的集合体为场景。

前台应用的场景调度如下图所示：



场景的基本架构如下图所示：



说明：

❖ 场景以消息处理循环为架构，在消息处理循环中接收并处理 gui 消息以及应用私有消息，并且根据需要更新 UI。

❖ 另外，场景中往往有一些定时处理事件，比如定时更新 UI ，由 应用级定时器完成。

场景可以分为两类：AP 自定义场景和 Common 控件场景。Common 控件场景可以实现为与其他 AP 自定义场景并行，也可以作为 AP 自定义场景的子场景嵌套进入。

其中，菜单列表控件场景是一种特殊的场景，这种场景通过菜单回调函数，可以在菜单列表控件场景中运行应用业务功能。

5.6.1.3　应用主体介绍

前台应用的应用主体是存放在应用子目录下的源代码，从代码的作用上可以分成应用基本架构、应用 UI 显示及应用业务功能：

❖ 应用基本架构
  ✧ 初始化，包括 applib 注册，消息管理器初始化，应用级定时器注册，系统定时器初始化，读取应用 vm 环境变量，打开 mcg，sty 文件等。
  ✧ 退出，包括关闭 mcg，sty 文件，保存应用 vm 环境变量，系统定时器关闭，applib 注销等。

版 权 所 有　侵 权 必 究

- ✧ 场景循环，按场景组织应用业务。
- ✧ 消息循环，场景的主体，与用户和系统交互。

❖ 应用业务功能

- ✧ 以菜单回调函数形式封装，由菜单解释器执行。
- ✧ 利用应用级定时器周期执行。
- ✧ 场景消息循环的 gui 事件或应用私有消息的处理。

❖ 应用 UI 显示

- ✧ 一般以在场景的消息循环内，如果应用业务功能需要更新 UI 显示，就会执行，一般每个 UI 元素对应 UI 更新标志变量的一个位，需要更新某个 UI 元素，就置上对应标志位。
- ✧ 应用级定时器的 ISR 内可以直接运行应用 UI 显示，以一种更加简单的方式实现周期更新 UI 显示。

5.6.1.4 如何使用 Common

在前面章节 Common 模块的设计与使用 中已经对 Common 做过详细的介绍，这里主要说明如何使用 Common。

- ❖ AP 与 COMMON 间系统配置数据交互，通过 com_set_sys_comval 设置后，COMMON 直接通过 sys_comval 访问 AP 的系统配置数据。
- ❖ COMMON 实现了应用级定时器，可以完美支持定时器 handle 为 bank 函数或者调用 bank 函数；在 handle 中发生某件事需要通知 AP 的，除了通过全局变量传递外，还可以给自己发送异步消息。
- ❖ COMMON 实现 2 个系统定时器。其中，1 个用来处理背光、屏保、返回正在播放界面、省电关机、定时关机等；1 个用来更新 headbar 状态。
- ❖ COMMON 中响应应用私有消息，调用通过 applib_message_init 接口传递下来的消息分发函数进行分发处理。另外还要注意一点，COMMON 调用 AP 的消息分发函数，一般会切换到 Control bank，如果调用者是 Control bank 且有 bank data，那么 bank data 就会被冲走，除非该 bank data 空间为此两 bank 共同预留空间。
- ❖ 为了 UI bank 组中不同 bank 能够方便实现较大数据交互，我们把 UI bank 组中前 512 字节预留作为 bank data 空间。
- ❖ COMMON 中有一块大小为 128 字节的系统全局变量区 applib_globe_data.o(.bss)，存放系统管理变量，这部分变量只会在系统启动时初始化一次；这部分变量管理着系统所有应用的基本信息以及系统状态信息，可以很方便的辅助管理系统。

5.6.1.5 应用映像文件结构

版 权 所 有 侵 权 必 究

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，这种映像文件结构如下：

Name	Offset (byte)	Length (bytes)	Descriptor
file_type	0	1	FILE_AP 'P' 文件扩展名：" .AP "
ap_type	1	1	AP Type:AP_SYSTEM 0X00 AP_USER 0X01 All others reserved.
Major_version	2	1	主版本号
minor_version	3	1	次版本号
magic	4	4	Magic 标志 "bx29"
text_offset	8	4	代码段文件内偏移
text_length	12	4	代码段长度
text_addr	16	4	代码段在内存中的地址
data_offset	20	4	初始化数据段文件内偏移
data_length	24	4	初始化数据段长度
data_addr	28	4	初始化数据段在内存中的地址
bss_length	32	4	未初始化数据段长度
bss_addr	36	4	未初始化数据段在内存中的地址，未初始化数据段在应用程序被加载时自动初始化为 0
entry	32	4	AP 入口地址
researe	36	XXX	填充字节
AP_Bank_head	XXX	12*X	AP bank 头，每 12 个字节对应 1 个 bank
Text_content	2K 对齐	text_length	常驻代码内容
data_content	2K 对齐	data_length	常驻数据内容
AP BANK CODE	2K 对齐	XXX	Bank 代码

说明：

❖ 从应用程序文件结构可以看出，一个应用程序只能包含一个常驻代码段和一个初始化常驻数据段，以及一个未初始化常驻数据段，所以，如果应用程序上述 3 种段的空间分散为 2 个或以上空间分段，必须采用特殊的手段，如果需要则请咨询 FAE 工程师。

版 权 所 有 侵 权 必 究

- Entry 一般是运行时库的 __start 函数，在 xn 文件中用 ENTRY(__start) 声明。
- 补充知识：段 segment

段 segment	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
.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）
.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
指定名字输出代码段	使用 __section__(“.section”) 定义函数，然后在链接脚本 XN 文件中用 filename.o(.section) 表示该输出段（.号可有可无）

5.6.2 前台应用的内存空间

这里说的内存空间，并不是 SRAM 物理内存空间，而是指应用程序中所用到的所有地址空间的总和。

这些空间包括以下几个部分：

- 常驻代码空间
- 常驻数据空间
- BANK 代码及 BANK 数据空间
- 运行栈空间
- 共享堆空间
- VRAM 空间

我们在前面介绍过 BANK 机制，把 32bit 的地址空间划分为 2 部分，前面 14bit 表示 page No，后面 18bit 表示物理内存地址。

其中，常驻代码空间的 page No 为 0xbfc，常驻数据空间的 page No 为 0x9fc，BANK 代码和 BANK 数据的 page No 较多，在下面详细说明。

版 权 所 有 侵 权 必 究

另外，Common 作为应用程序的重要组成部分，其内存分配请参考 Common 的空间分配 一节。

5.6.2.1 常驻代码空间

前台应用常驻代码段：0xbfc1ee00-0xbfc1f5ff 共 2KB，用来存放 AP、Common 模块、enhanced 模块的常驻接口及相关 const data、bank data。

5.6.2.2 常驻数据空间

前台应用的数据空间

0x9fc1d200-0x9fc1d9ff 共 2KB，该全局数据空间是 AP、Common 模块以及 enhanced 模块共用的。

Case全局数据空间

0x9fc19f80-0x9fc19fff 共 128B，该全局数据空间是所有 AP 共享的，只会在 ap_manager 中初始化一次，主要用来存放 Common 模块的一些 case 全局数据。

5.6.2.3 BANK 代码及 BANK 数据空间

Front Control bank 组：(0x40**0000+0x1fe00)-(0x40**0000+0x205ff) 共2KB，主要用来存放AP 非常驻接口、Common 控件场景等。其中 Common 模块占用 control bank 组中 bank 48 ~ bank 63这16 bank号。

Front UI bank 组：(0x48**0000+0x1f600)-(0x48**0000+0x1fdff) 共2KB，主要用来存放 Common 模块其他接口等。另外需要特别说明的一点是，由于 UI bank 组基本上是 Common 模块独占的，所以 Common 模块可以很好的进行分配，为了实现 AP bank 间大数据的传递/共享，我们把 (0x48**0000+0x1f600)-(0x48**0000+0x1f7ff) 共0.5KB用来存放 Common 模块的一些 bank data，所以UI bank段在很多时候实际上可用的空间只有 1.5KB。

另外，有些 AP 在某些特殊情境下，需要把部分接口放在 UI bank 组，以避免因放在 Control bank 组中频繁切 bank 的情况，所以我们预留 UI bank 组中 bank 0 ~ bank 15 这 16 bank 号给 AP，所以实际上 Common 模块只能使用 bank 16 ~ bank 63 这 48 bank 号。

Front Enhanced BANK 1 组：(0x78**0000+0x27000) – (0x78**0000+0x277ff) = 2KB ，用于存放中间件模块接口，如果不需要中间件支持，可放任何代码和数据。

Front Enhanced BANK 2 组：(0x71**0000+0x27800) – (0x71**0000+0x27fff) = 2KB ，用于存放中间件模块其他接口，如果不需要中间件支持，可放任何代码和数据。

5.6.2.4 运行栈空间

前台应用可能会创建子线程，而子线程是需要独立的运行栈的，所以系统分配给前台应用的运行栈空间应该根据有无子线程分别看待：

- 无子线程时：0x9fc26eb0 ~ 0x9fc265c0 = 0x8f0 B
- 有子线程时：
  - 主线程：0x9fc26eb0 ~ 0x9fc26ac0 = 0x3f0 B
  - 子线程：0x9fc26ac0 ~ 0x9fc265c0 = 0x500 B

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。

另外，以上栈空间的划分，并不是限制死了，用户可以根据自己应用程序的实际情况灵活调整，即主线程可以在定义 OS_STK *ptos 时灵活赋值，而子线程则在创建线程时灵活填写 pthread_param_t->ptos 成员的值。

5.6.2.5 堆空间

系统提供了 512 字节的堆空间，为整个系统共用，空间相对来说还是很局限的。应用程序可以临时申请少量内存。

申请堆：sys_malloc(&menu_history_addr, path_size);

释放堆：sys_free(&menu_history_addr);

5.6.2.6 VRAM 空间

VRAM 是一块从当前主盘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。

系统为 Case 分配 512KB 的 VRAM 空间，每个应用分配 1KB，用于保存应用环境变量，其余空间用作应用的临时数据缓冲。

分配使用 VRAM 空间要注意不能覆盖其他模块的 VM 空间，并且要注意 0x80000 开始的空间分配给 PSP 使用，Case 原则上不能使用，如果真的需要使用 0x80000 后的空间，请与我们的工程师联系取得支持。

5.6.3 应用程序的启动

- 同步创建 AP，用于创建后台音频，比如音乐引擎
  1) 前台应用发送 MSG_CREAT_APP_SYNC 给 manager.app，让出控制权。
  2) manager.app 收到同步创建 AP 消息，调用 sys_exece_ap(ap_name, 1,

版 权 所 有 侵 权 必 究

(int32)ap_param) 创建后台引擎。

3) 等待后台引擎创建完成回到 manager.app 后 reply_sync_msg，让前台应用继续执行。

4) 至此，同步创建 AP 完成。

❖ 异步创建 AP，用于创建前台应用，需等待当前前台应用退出后才能创建，这是因为内存空间资源的限制导致的。

1) 前台应用发送 MSG_CREAT_APP 给 manager.app。

2) manager.app 收到异步创建 AP 消息，先调用 libc_waitpid 等待当前前台 AP 退出，然后再调用 sys_exece_ap(ap_name, 0, (int32)ap_param) 创建新前台应用。

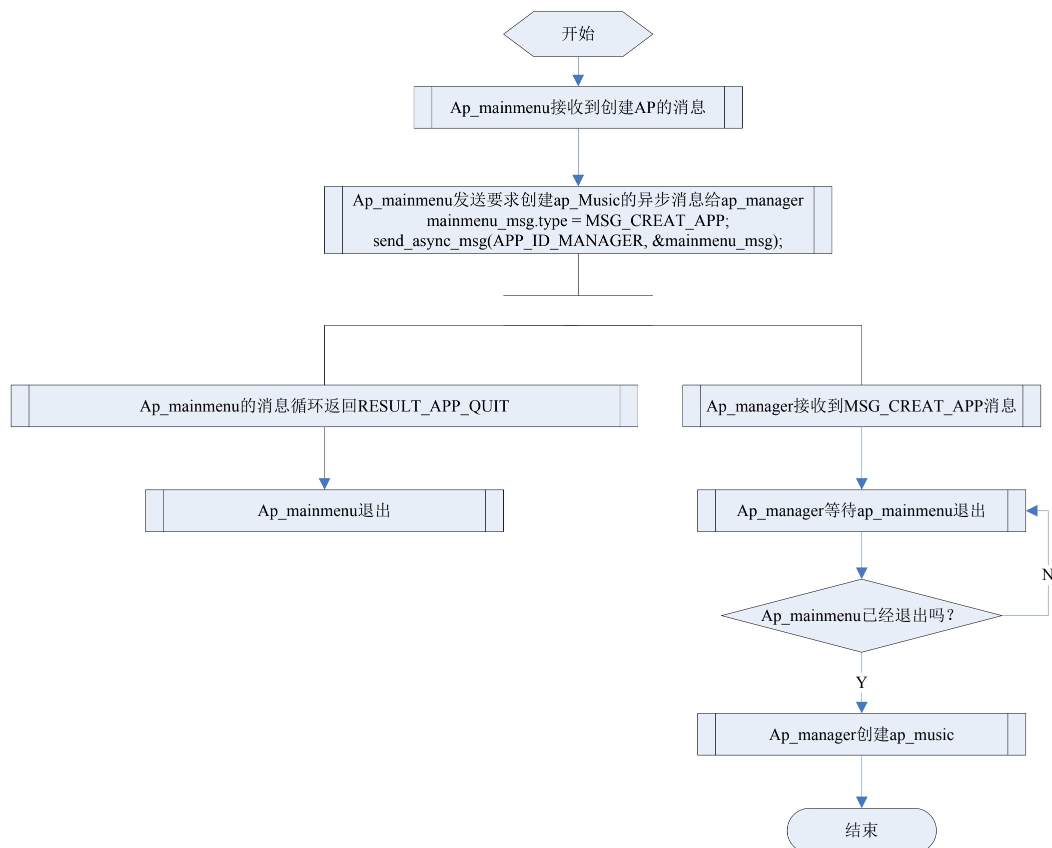
3) 至此，manager.app 创建 AP 动作完成，至于是否能够成功创建 AP 则不管。

以 ap_music 应用的启动为例进行说明。Ap_Music 应用的启动模式分成 2 种：从主界面启动和从 ap_browser 应用启动。

✧ 从主界面启动（主动模式）：用户在 ap_mainmenu 里头选中了 ap_Music 应用运行。Ap_mainmenu 应用的消息循环里头，收到 EVENT_MAINMENU_CREATE_APP 消息，并在 mainmenu_msg_handle.c 的消息处理函数 mainmenu_gui_msg_handle()里头处理这个消息，这个函数里头根据不同的 index，创建不同的应用，而不同的 index 对应哪个 AP，则从配置文件 usdk212a\case\fwpkg\config.txt 的字段 MAINMENU_AP_ID_ARRAY 里头获取，每个 AP ID 对应什么应用，则定义在了 usdk212a\case\inc\case_type.h 文件里头。当 index 对应的 AP ID 值为 0x00 时，则创建 ap_music。

启动从 ap_mainmenu 启动 ap_music 流程如下：

版 权 所 有 侵 权 必 究



从 ap_mainmenu 启动 ap_music 的流程图解释如下：

第一步：ap_maimenu 收到按键消息；

第二步：按键消息被映射成了事件代号 EVENT_MAINMENU_CREATE_APP；（调用 com_key_mapping 接口，对应的代码在 mainmenu_msg_loop.c 里头）

第三步：在事件处理代码里头响应 EVENT_MAINMENU_CREATE_APP 事件：向任务管理器发送异步消息：send_async_msg(APP_ID_MANAGER, &mainmenu_msg)；（对应的代码在 mainmenu_msg_handle.c 里头，消息类型定义为 MSG_CREAT_APP）之后退出 AP。

第四步：ap_manager 响应消息 MSG_CREAT_APP，并调用 sys_exece_ap(ap_name, 0, (int32) ap_param)接口运行应用（对应的代码在 ap_manager/manager_msg_handle.c 里头）

第五步：请注意，第三步中，发送的是“异步”消息，ap_mainmenu 不等消息响应就返回 RESULT_APP_QUIT 并退出了 ap_mainmenu。

版 权 所 有 侵 权 必 究

send_async_msg () 接口是一个异步消息发送接口，定义在了 usdk212a\case\ap\common\applib\Message_Bank_Send.c 和头文件 usdk212a\case\inc \applib.h 里头。其第一个参数，是接收消息的 AP 的 ID，第二个参数是消息结构体 private_msg_t，此结构体每个成员的说明如下：

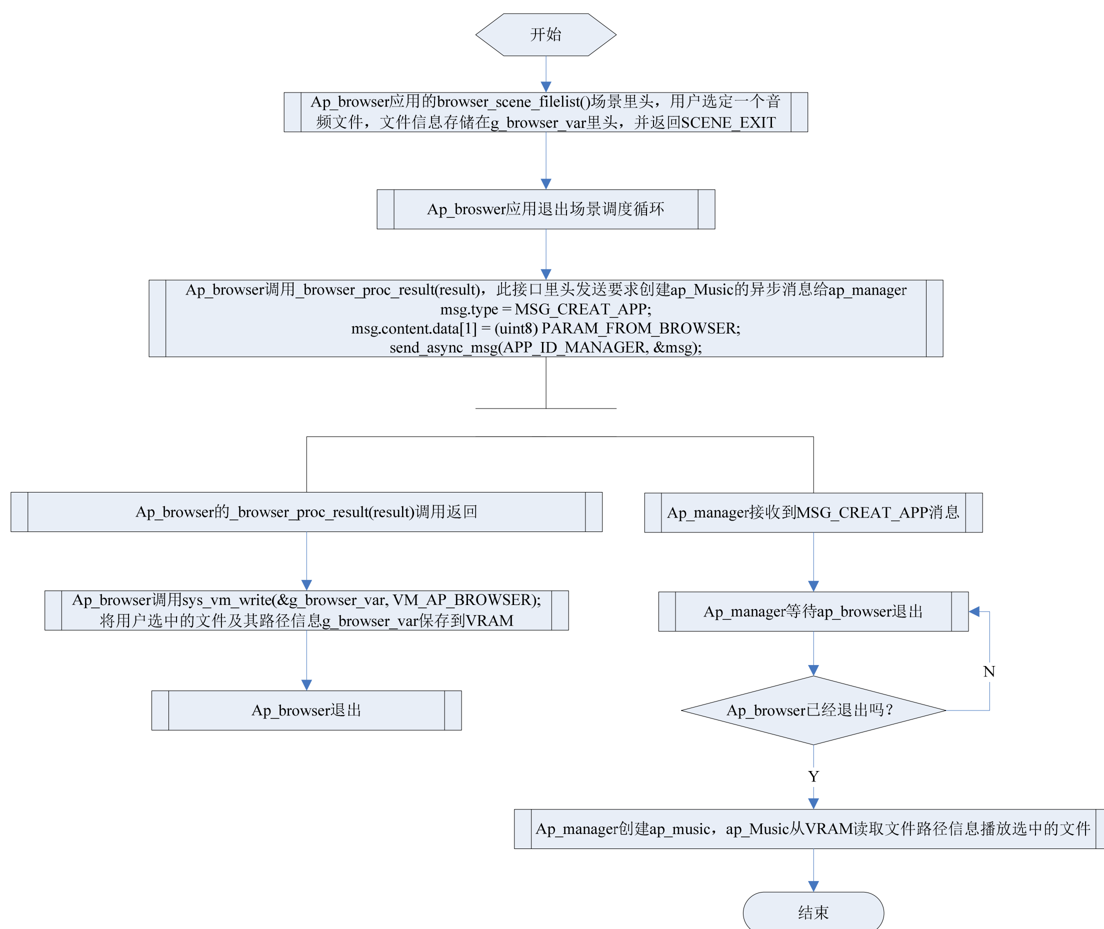
```
typedef struct
{
    /*! 应用消息类型，参见 msg_apps_type_e 定义 */
    uint32 type;
    /*! 应用消息内容 */
    union
    {
        /*! 消息内容真实数据 */
        uint8 data[4];
        /*! 消息内容缓冲区指针，指向消息发送方的地址空间 */
        void *addr;
    }content;
} msg_apps_t;
```

此数据结构中，data[0]存储了需要创建的 AP 的 ID（根据 case_type.h 里头的定义确定），data[1]定义创建 AP 时需要引用的参数，不同的参数，AP 创建之后，可能会根据这个参数进行不同的业务流程的处理。

- 从 browser 启动（被动模式）：用户在 ap_browser 选中一个音乐文件打开的时候，将会启动 ap_music，并播放相应的音乐文件。

从 ap_browser 启动 ap_music 的流程图如下：

版 权 所 有 侵 权 必 究



从 ap_browser 启动 ap_music 的流程图解释如下：

第一步：Ap_browser 应用的 browser_scene_filelist()场景里头（此场景定义在文件 browser_scene_browsing.c 里头），通过调用 common 里头的 gui_directory()接口，实现文件的浏览（此接口定义在文件 case\ap\common\common_ui\ui_directory.c 和头文件 case\inc\common_ui.h 里头）。用户在浏览文件的时候，选定一个音频文件，文件信息存储在此 AP 的全局变量 g_browser_var 里头，然后退出了场景调度。

第二步：Ap_browser 调用_browser_proc_result(result)，此接口里头发送要求创建 ap_Music 的异步消息给 ap_manager（此接口定义在 usdk212a\case\ap\ap_browser\browser_main.c 里头）：send_async_msg(APP_ID_MANAGER, &msg);，并保存全局变量 g_browser_var 里头的文件及其路径信息到 VRAM，由于是发送异步消息，因此，无需等待接收方的回应，就执行后续的代码退出了 ap_browser。

send_async_msg()定义在 usdk212a\case\ap\common\applib\Message_Bank_Send.c 和头文件 usdk212a\case\inc \applib.h 里头，

第三步：ap_manager 响应消息 MSG_CREAT_APP，并调用 sys_exece_ap(ap_name, 0, (int32) ap_param)接口运行应用（对应的代码在 ap_manager/manager_msg_handle.c 里头）

第四步：ap_music 应用运行的时候，会从 VRAM 中读取保存的文件和路径信息，播放对应的音频文件。

5.6.4 应用程序的退出

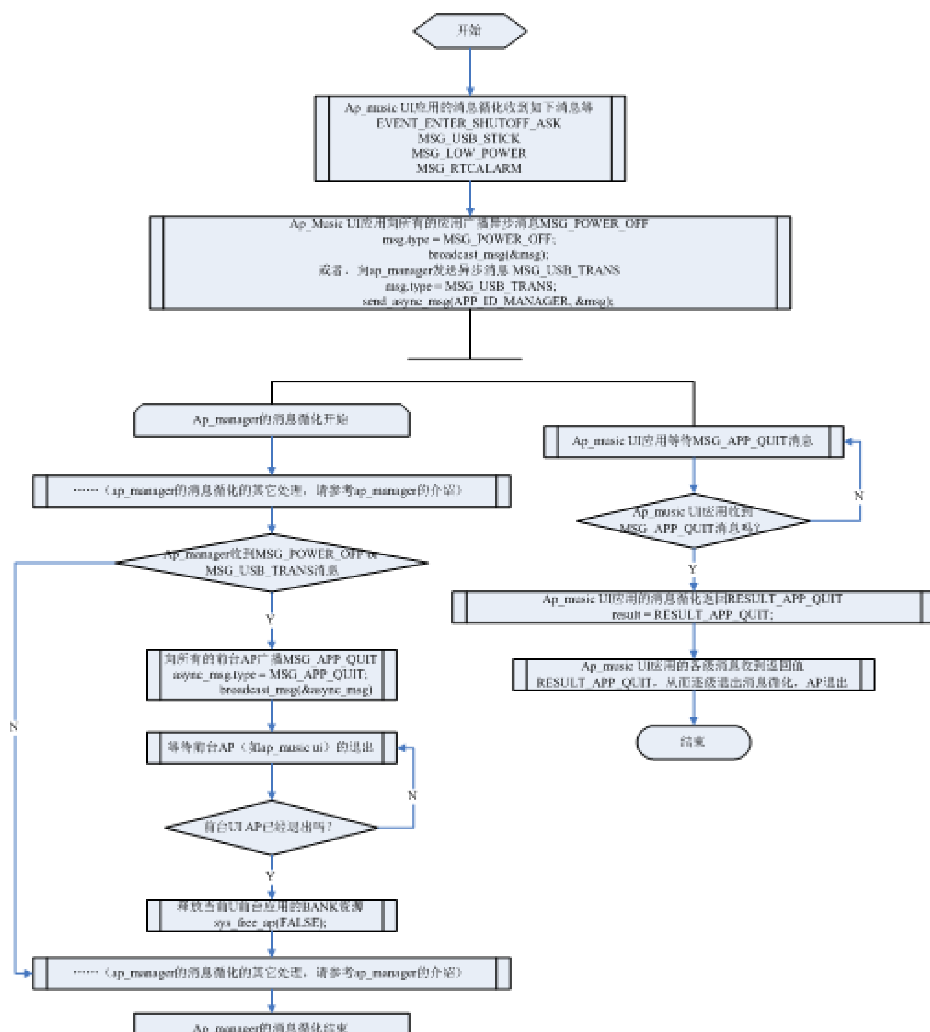
AP 退出（AP 退出会由 manager AP 调用的 libc_waitpid 捕捉到），退出事件包括以下几个：

- 前台应用在发送异步创建新前台应用后，主动退出。
- manager.app 在关机或者进入 U 盘时会杀死所有 AP，即发送 MSG_APP_QUIT 异步消息给 AP；AP 收到消息后，主动退出。
- 前台 AP 需要时可以同步杀死后台引擎，比如 video.app 进入播放时，需要关掉音乐引擎。
    1) 前台应用发送 MSG_KILL_APP_SYNC 给 manager.app，让出控制权。
    2) manager.app 收到同步杀死应用消息，同样是发送 MSG_APP_QUIT 给后台引擎。
    3) 后台引擎收到消息后，主动退出；退出后 manager.app 再调用 reply_sync_msg 让前台应用继续执行。
    4) 至此，同步杀死 AP 完成。

以 ap_music 应用为例进行说明。Music UI 应用作为前台应用，其退出应用的流程，有如下几种情况：

- 接收到 MSG_APP_QUIT 消息退出
- 用户通过按键操作选择了退出 Music UI

因为接收到 MSG_APP_QUIT 消息而退出 Music UI 应用的场景，包括：用户主动关机，USB 连接，低电关机，定时闹钟等。其代码处理流程如下：



对于以上流程，解释如下：

第一步：会导致ap_Music UI应用退出的原因有如下几个：USB线插入时，需要退出ap_Music UI应用，运行USB应用；闹铃时间到了，需要ap_Music UI应用，运行ap_tool应用；检测到低电，需要关机了，必须正常退出ap_music UI应用；用户长按关机键要求关机。以上的的任何原因，ap_music任何场景的消息循环里头，都会收到相应的消息，包括EVENT_ENTER_SHUTOFF_ASK，MSG_USB_STICK，MSG_LOW_POWER，MSG_RTCALARM等消息。因此，这也意味这任何消息循化里头，都必须处理以上消息。而在US212A的实现中，在usdk212a\case\ap\common\common_misc\Message_box.c文件里头定义了所有这些每个消息循化都必须处理的消息的处理函数com_message_box()。

第二步：ap_music 的任何消息循环收到以上消息，都会调用相应的处理程序进行处理。比如，收到长按按键产生的 EVENT_ENTER_SHUTOFF_ASK，则会调用 usdk212a\case\ap\common\common_ui\ui_shutoff.c 里头的 gui_shut_off()接口，该接口异步广播了 MSG_POWER_OFF 消息后，就开始等待 MSG_APP_QUIT 消息（注意，如果无法等到 MSG_APP_QUIT 消息，则 gui_shut_off()接口会在等待循化里无法退出）。

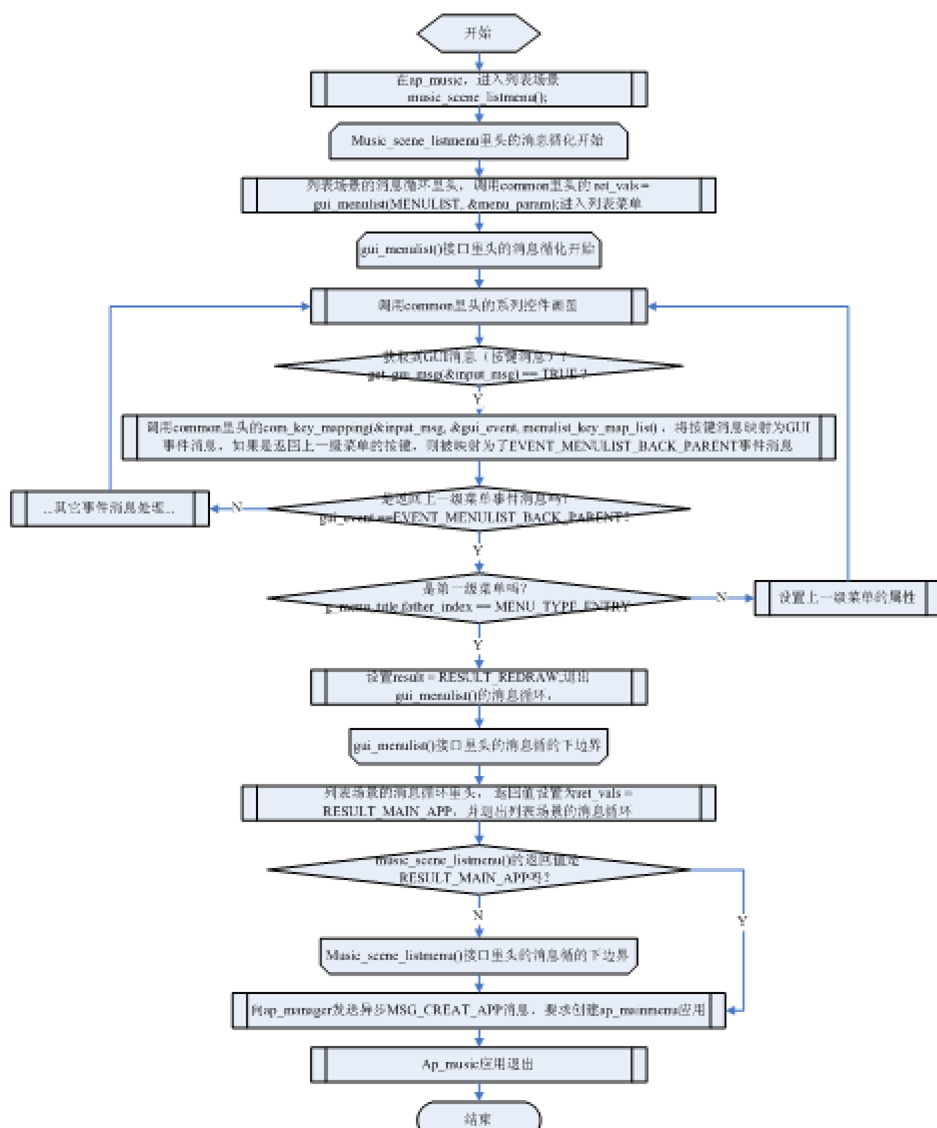
第三步：ap_manager 收到 MSG_POWER_OFF/MSG_USB_TRANS 消息后，向所有的前台应用异步广播 MSG_APP_QUIT 消息，并等待前台应用的退出。

第四步：ap_music 等到 MSG_APP_QUIT 消息后，消息循环返回 RESULT_APP_QUIT，从而导致 ap_music 的退出。

第五步：ap_manager 等待 ap_music 退出后，释放 AP 的相应资源。

根据以上过程，请思考：如果在 Music UI 应用里头，遇到连接 USB 线，发现没有 USB 连接，而且 Music UI 不退出的状况时，该如何 Debug？

而用户通过按键操作，选择退出 ap_music 的流程，在此，以从 ap_music 的列表场景退出为例，去说明 ap_music 是如何在用户的按键操作下退出应用的。



就以上流程图，解释如下：

第一步：在列表场景里头，有两个嵌套的消息循环，分别是定义在 Music_scene_listmenu.c 文件里头的 music_scene_listmenu() 函数的 while(1) 循环和此循环内部调用 usdk212a\case\ap\common\common_ui\ui_menulist.c 里头的 gui_menulist()接口内，也有个 while(1)消息循环。当调用 music_scene_listmenu()接口时，初始化好菜单参数 menu_param （menu_com_data_t 类型的结构体）之后，进入一个 while(1)消息循环；而在该消息循环里头，调用了接口 gui_menulist()，该接口根据之前初始化的菜单参数，从 mcg 文件里头读取

对应的菜单信息显示，并进入菜单处理的消息循环里头。

第二步：在 gui_menulist()接口的菜单处理消息循环里头，调用 get_gui_msg(&input_msg) 接收 GUI 消息（包括按键消息），一旦接收到按键消息，会通过调用定义在 usdk212a\case\ap\common\common_func\common_msgmap.c 里头的接口 com_key_mapping() 将按键键值映射为一个事件消息值，其中返回上一级菜单的按键，映射的事件消息值是 EVENT_MENULIST_BACK_PARENT。对于该事件消息值，gui_menulist()接口菜单的消息循环的处理方式是：如果判断当前菜单的父菜单已经是 MENU_TYPE_ENTRY（入口菜单）了，则退出 gui_menulist()的消息循环，而且返回值是 RESULT_REDRAW，从而回到了 music_scene_listmenu()里头的消息循环。

第三步：在 music_scene_listmenu()里头的消息循环里头，判断到 gui_menulist()的返回值是 RESULT_REDRAW，则会退出列表场景，music_scene_listmenu()的调用返回 RESULT_MAIN_APP。

第四步：ap_music 退出场景调度，并向 ap_manager 发送要求创建 ap_mainmenu 应用的异步消息后，退出 ap_music。

请思考：如果想要改变返回上一级菜单的按键定义，该如何实现？？——只需修改 usdk212a\case\ap\common\common_func\common_msgmap_data.c 里头定义的常量数组 menulist_key_map_list[]对应的按键和事件消息值即可。

5.6.5 前台应用的开发指南

5.6.5.1 前台 AP 开发流程

1 编写好应用基本架构。

2 先在 UI 模拟器上调试 AP UI。

  2.1 编写场景基本代码。

  2.2 准备好 AP 图片资源和多国语言字符串资源（可先提供简体中文，但要在 config.app 中把语言设置为简体中文）。

  2.3 在 UI Editor 工具上设计 UI，生成 *.sty, *_res.h, *_sty.h 文件。

版 权 所 有 侵 权 必 究

2.4 编写场景 UI 显示代码，调试 UI。

2.5 调试非硬件相关业务功能。

2.6 注：前台应用需要在 mainmenu AP 中添加入口 UI 项，也可以把该 AP 编译链接为标案中应用，比如 music.ap 。

3 如果有菜单，设计菜单。

3.1 编写入口菜单列表和菜单项列表，当然还要编写菜单回调函数，如：test_cfg_menu_data.c。

3.2 再次编译链接生成 *.ap 文件

3.3 在 fw modify 工具中的菜单编辑选项页上，选择该 AP 文件，双击进入菜单编辑窗口，编辑 OK 后生成 *.mcg 文件，再导出到 fwpkg/mcg 目录下。（详见后面辅助工具介绍）

3.4 在 UI 模拟器调试菜单，一些涉及硬件功能的回调函数不可调试。

4 完善开发，上板调试

4.1 编写剩余业务功能模块。

4.2 编译链接工程，生成 *.ap 文件。

4.2.1 在 makefile 模板上修改为该 AP 使用的 makefile 。

4.2.2 在 xn 模板上修改为该 AP 使用的 xn 。

4.2.3 编译链接，清除错误和某些警告，生成 *.ap 文件。

4.3 在 fwimage*.cfg 中增加打包项*.ap，*.sty, *.mcg，打包生成固件 *.fw 文件。

4.4 上板调试。

前面说到，前台应用的应用主体包括基本架构，UI 显示和业务功能，其中基本架构比较简单，在前面几个小节已经详细介绍过了，接下来，我们将重点介绍 UI 显示和业务功能。

5.6.5.2 UI 显示开发详解

UI 显示开发步骤如下：

1 首先是创建前台应用的 UI Editor 工程，编辑并生成 *.sty 、*_sty.h 、*_res.h 等结果文件。详细请参考 界面设计与开发 一章。

2 在前台应用的初始化流程中打开 *.sty 和 common.sty 文件，并在应用退出时关闭。

3 参考《us212a_ui_driver 接口说明书.chm》，根据具体的使用情景，编写逐个控件的显示处理代码。

4 如果要在小机进行调试，需要把 *.sty 文件也一并打包进去；如果是在 UI Simulator 工具上进行调试，就无须理会。强烈建议前台应用的开发和调试尽量在 UI Simulator 上进行。

在界面设计与开发一章中我们会说到，控件由图片、字符串和数字这三种基本元素组成。基本元素组成控件时，关键在于控件属性的选择：

版 权 所 有 侵 权 必 究

- 第 1 类　哪些属性可以在控件中默认设置
- 第 2 类　哪些属性必须开放给用户，可以由用户根据具体需要进行自定义
- 第 3 类　哪些即可使用默认设置，又允许用户优先自定义

下面以 PicBox 控件为例进行讲解。

```
/*!
 * \brief
 *  picbox_t 描述picbox的数据结构
 */
typedef struct
{
    /*! 如果id=-1，表示多帧pic,帧的列表在frame中，其他值表示res id */
    uint16 id;
    /*! picbox x坐标*/
    uint16 x;
    /*! picbox y坐标*/
    uint16 y;
    /*!
     * \li attrib<0>表示是否显示picbox，0为不显示，1为显示；
     * \li attrib<1>表示frame 的存储类型，0表示多帧（12帧以内）图片均存储在frame;
     *     1表示存储在frame[0],frame[1]指定的地方；
     * \li attrib<8-15>表示图片的帧数。
     */
    uint16 attrib;
    /*! 如果attrib<1>=0, 多帧时，帧对应id列表；
     *   如果attrib<1>=1, frame[0],frame[1]为frame 的开始地址
     */
    uint16 frame[12];
} picbox_t;
```

用户指定属性
或用户优先指定属性

默认属性

说明：

1. 在这个数据结构中，如果是单帧 PicBox 控件，那么 id 属于第 3 类属性，即默认使用设置的图片 ID，但如果用户需要自己指定其他图片，可以优先指定；如果是多帧 PicBox 控件，用户需要指定第几帧，这个属于第 2 类属性，必须由用户指定。

2. 其他属性，都属于第 1 类属性，直接使用在 UI Editor 上设置的默认值。

与 ui 驱动交互：

- 控件的显示由 ui 驱动提供 *_private_t 结构体给应用，用来传递控件的第 2, 3 类属性，ui 驱动再根据控件显示机制，综合所有属性值进行解析显示。
- 为了更有效的进行控件显示，我们会给控件显示提供多个刷新模式，比如 ListBox 控件，会有刷全部，刷列表，刷激活项等。
- 所以，控件显示接口形式大致为：

  ui_show_xxx(style_infor_t *, *_private_t *, uint8 mode);

以 PicBox 为例：

1. ui 驱动提供 picbox_private_t 结构体给应用，用来传递控件的第 2, 3 类属性，该结构体如下：

```
/*!
 * \brief
 *      picbox_private_t picbox私有数据结构
 * \note
 *      pic_id!= -1, 优先显示pic_id所指向的图片;
 *      pic_id = -1, frame_id! =-1, 则显示StyleID所指向的Picbox的frame_id的图片
 */
typedef struct
{
    /*! 如果pic_id不为-1, 那么就是优先用pic_id显示 */
    uint16 pic_id;
    /*! 指示显示哪一帧图片 */
    uint8 frame_id;
    /*! 显示模式, 分为普通模式, 透明图标, 垂直滚屏 */
    uint8 reserve;
} picbox_private_t;
```

2. 图片显示只有一种显示模式，无需 mode 参数。

所以，PicBox 显示接口形式为：

ui_show_picbox(style_infor_t *, picbox_private_t *);

5.6.5.3 消息通信开发详解

US212A 是多任务方案，除了要有像 AS211A 的按键输入消息和系统消息外，还必须具备任务间通信的消息。

在前面的内容中我们说到，AP 分为 manager AP、前台 AP 和后台引擎 AP，那么相对应的，为了能实现任务间通信的需求，系统必须提供 3 个应用私有消息队列。

因此，US212A 包含 5 个消息队列：

- GUI 消息队列，只有前台 AP 才会访问到，用来获取用户的按键输入等输入消息。
- 系统消息队列，系统用来发送一些系统消息，比如 USB 插拔，卡插拔，ALARM 定时已到，低电，等。
- 3 个应用私有消息队列，专用来存放其他应用（应用可以给自己发送异步消息，用于不同模块之间通信）发送给自己的消息，用于应用间通信。

消息通信开发要点：

- 按键消息
  - ✧ GUI 消息队列专用于按键驱动发送按键消息，或者触摸屏驱动发送触摸消息；其中按键消息包括 DOWN，LONG，HOLD，SHORT UP。
  - ✧ 按键驱动在发送 SHORT UP 消息时，如果判断到之前已经发送了超过 8 个（GUI 消息队列深度）HOLD 消息，会先接收 1 个消息，以确保消息队列至

版 权 所 有 侵 权 必 究

少有 1 个空位，而保证 SHORT UP 消息能发送到消息队列中。

- ✧ 前台 AP 在初始化阶段，调用 applib_message_init 进行初始化，该接口会把 GUI 消息队列清空。
- ✧ 前台 AP 收到 GUI 消息时，会先调用 com_gui_msg_hook 进行预处理，包括按键过滤、按键音、按键锁、背光控制、屏保控制，等等处理。

- 系统消息
  - ✧ 系统消息队列专用来存放系统消息，并由前台 AP 在接收应用私有消息前接收并广播到前台 AP 和后台引擎 AP 的消息队列中。
  - ✧ 系统消息队列不会在 AP 初始化阶段调用的 applib_message_init 初始化时被清空，这样就保证已经发送到系统消息队列中的消息不会丢失；这点特点正好可以用来缓存一些在应用切换阶段产生的消息。
- 应用私有消息
  - ✧ 应用私有消息队列主要用于任务间通信，任务间通信主要包括以下几类：
    - ◆ 与 manager AP 交互的创建和杀死应用的消息。
    - ◆ 前台 AP 控制对应的后台引擎 AP 的消息。
  - ✧ 前台 AP 和后台引擎 AP 在初始化阶段，调用 applib_message_init 进行初始化，该接口会把应用私有消息队列清空。
  - ✧ AP 在收到 APP_MSG_QUIT 消息时，必须无条件立即退出。
  - ✧ 前台 AP 收到消息时，会先调用 com_app_msg_hook 进行预处理，包括屏保处理、卡状态切换等。

消息处理循环使用要点：

为何引进 gui 事件驱动开发：us212a 把控制流和显示流分开，事实上就已经引进了 gui 模块的概念，只是限于内存资源无法把 gui 模块单独出来。但是我们还是保留了 gui 事件的概念，这样也是为了可以更好支持不同物理输入设备（按键、触摸屏、g-sensor 等）而设计的。

gui 事件驱动的关键在于按键消息映射，把按键消息按照用户在某个场景下定义的按键映射表中映射，得到 gui event。

按键映射表分为两部分，一部分为用户自定义映射表，一部分为方案定义的快捷键映射表，前者优先级高于后者。

比如：USB 连接对话框的按键映射表屏蔽了快捷键映射表部分映射项

```
/*!
 * \brief
 *  usbconnect_key_map_list：设备连接对话框按键映射表
 */
const key_map_t usbconnect_key_map_list[] =
{
    /*! PREV 按键转换为 PREV_BUTTON 事件 */
    {{KEY_PREV, 0, KEY_TYPE_DOWN | KEY_TYPE_LONG | KEY_TYPE_HOLD}, EVENT_DIALOG_PREV_BUTTON},
    /*! NEXT 按键转换为 NEXT_BUTTON 事件 */
    {{KEY_NEXT, 0, KEY_TYPE_DOWN | KEY_TYPE_LONG | KEY_TYPE_HOLD}, EVENT_DIALOG_NEXT_BUTTON},
    /*! 短按KEY_PLAY 按键转换为 DIALOG_CONFIRM 事件 */
    {{KEY_PLAY, 0, KEY_TYPE_SHORT_UP}, EVENT_DIALOG_CONFIRM},
    /*! 短按KEY_VOL 按键转换为 DIALOG_CANCEL 事件 */
    {{KEY_VOL, 0, KEY_TYPE_SHORT_UP}, EVENT_DIALOG_CANCEL},
    /*! 短按KEY_MODE 按键转换为 DIALOG_CANCEL 事件 */
    {{KEY_MODE, 0, KEY_TYPE_SHORT_UP}, EVENT_DIALOG_CANCEL},

    /*! 不响应以下快捷键 */
    {{KEY_PLAY, 0, KEY_TYPE_LONG}, MSG_NULL},
    {{KEY_LOCKHOLD, 0, KEY_TYPE_ALL}, MSG_NULL},
    {{KEY_VOL, 0, KEY_TYPE_LONG | KEY_TYPE_HOLD}, MSG_NULL},

    /*! 结束标志 */
    {{KEY_NULL, 0, KEY_TYPE_NULL}, MSG_NULL},
};

/*!
 * \brief
 *  key_shortcut_list：快捷键映射表
 */
const key_map_t key_shortcut_list[] =
{
    /*! 长按 PLAY 按键关机 */
    {{KEY_PLAY, 0, KEY_TYPE_LONG}, EVENT_ENTER_SHUTOFF_ASK},
    /*! 锁键拨到常关状态，键盘上锁 */
    {{KEY_LOCK, 0, KEY_TYPE_DOWN}, EVENT_ENTER_KEYLOCK},
    /*! 锁键拨到常开状态，键盘解锁 */
    {{KEY_UNLOCK, 0, KEY_TYPE_DOWN}, EVENT_ENTER_KEYLOCK_UN},
    /*! 虚拟按键，当键盘锁住时按下任何其他按键，会转换为 KEY_LOCKHOLD，显示键盘锁住 */
    {{KEY_LOCKHOLD, 0, KEY_TYPE_ALL}, EVENT_ENTER_KEYLOCK_HOLD},
    /*! 长按 VOL 键弹出音量调节框 */
    {{KEY_VOL, 0, KEY_TYPE_LONG | KEY_TYPE_HOLD}, EVENT_ENTER_VOLUMEBAR},

    /*! 结束标志 */
    {{KEY_NULL, 0, KEY_TYPE_NULL}, MSG_NULL},
};
```

消息处理循环的典型结构如下：

```
while (1)//消息处理循环
{
    ...//其他处理，比如UI刷新
    //获取到 gui消息
    if(get_gui_msg(&input_msg) == TRUE)
    {
        //进行 gui消息映射
        if(com_key_mapping(&input_msg, &gui_event, dialog_key_map_list) == TRUE)
        {
            switch(gui_event)
            {
            case EVENT_DIALOG_NEXT_BUTTON://选择下一个按钮选项
                ...
                break;
            case EVENT_DIALOG_PREV_BUTTON://选择上一个按钮选项
                ...
                break;
            ...//其他dialog事件处理
            default://其他默认事件处理
                result = com_message_box(gui_event);
                break;
            }
        }
    }
    //到此，gui消息队列已经处理完毕，获取ap私有消息和系统消息（会先转换为私有消息再返回）
    else
    {
        if(get_app_msg(&private_msg) == TRUE)
        {//应用私有消息分发函数
            result = g_this_app_msg_dispatch(&private_msg);
        }
    }

    sys_os_time_dly(1);//挂起10ms，多任务调度
}
```

消息处理循环有几个点要注意：

1. gui 消息的接收和处理优先于系统消息和应用私有消息，只有在接收不到 gui 消息时才会接收并处理系统消息和应用私有消息。

2. 对 gui 消息映射出来的事件，一般会在循环中显式处理场景所关心的事件，而其他事件调用 com_message_box 接口进行处理。如果不进行这一步处理，可能会漏掉一些重要的快捷键事件。

3. 在应用私有消息分发函数中，应用会显式处理自己关心的消息，而其他消息也是要求调用 com_message_box 接口进行分发处理。如果不进行这一步处理，可能会漏掉一些重要的系统消息等。

4. 消息处理循环中一定要适时调用 sys_os_time_dly 函数释放控制权，以便让优先级较低的任务调度运行。

最后再补充两点：

1. 应用的消息处理，应该封装为消息分发函数，传递到 COMMON 中，在 COMMON 中使用该消息分发函数对应用私有消息进行处理。

2. 消息处理完成后，需要对返回结果进行处理，应用开发人员应阅读好各 COMMON 接口的说明，准确处理 result，比如收到 COMMON 返回 RESULT_APP_QUIT 后，应用

版 权 所 有 侵 权 必 究

应该无条件立即退出。

5.6.5.4 应用级定时器开发详解

使用硬件定时器，一般要求 ISR 放在常驻代码中，这样才不会中断太久；并且硬件定时器需要系统管理，使用的是系统的资源。所以我们引进了应用级定时器，把定时器的管理挪到应用来，并且使用的资源也是应用的。

应用级定时器有以下特点：

- 应用级定时器 ISR 可以放在 bank 代码段中，我们不用担心 ISR 执行时间的长短。
- 应用级定时器是场景相关的，也就是说，在某个场景下创建的定时器，其 ISR 只会在该场景下运行。
- 应用级定时器的功能远比硬件定时器强，可以创建，暂停，重启，删除，可以单发，也可以周期发送。
- 应用级定时器只会消耗 AP 自身的常驻数据空间，定时器数目仅仅受限于数据空间。
- 另外，应用级定时器的 handle 也是不能传递参数，只能通过全局变量传参。
- 应用级定时器是在获取应用私有消息时执行的，故会延迟获取应用私有消息的时间。

5.6.5.5 可配置化菜单开发详解

请参见 可配置化菜单 一节。

5.6.6 应用程序 makefile 与 xn 脚本编写

5.6.6.1 应用程序 makefile 脚本

什么是 makefile ?

make 命令执行时，需要一个 Makefile 文件，以告诉 make 命令需要怎样去编译和链接程序。Makefile 就是指定编译链接规则的脚本文件。

在应用和驱动开发时，系统会提供应用和驱动的 makefile 模板，指定应用和驱动编译链接时应遵守的共同基本规则。

应用程序在编写自己模块的 makefile 时，只需要在系统提供的 makefile 模板上，修改对应模块的源文件路径信息、生成的目标文件名、目标文件的打包路径、链接脚本文件名等信息。

Makefile 模板及其修改

以 Music 应用为例，makefile 需要修改的地方如下：

```
# Name of application 开发人员需修改处(1)
IMAGENAME = music
#所要编译的源文件的存放位置,开发人员需修改处(2)
SRC = $(CASE)/ap
SRCDIR_16 = $(SRC)/ap_music
SRCDIR_16_O2 = $(SRC)/ap_music/O2
SRCDIR_32 =
#目标文件送往的文件夹路径,开发人员需修改处(3)
OBJECT_BIN_PATH = $(CASE)/fwpkg/ap
#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
VPATH     =     $(SRCDIR_16) $(SRCDIR_16_O2) $(SRCDIR_32) $(OBJ_DIR)
 #指定自定义链接脚本的名称
LD_SCRIPT = ap_music.xn $(ENHANCED_XN)/music_link.xn
                ./../common/common_front.xn
#获得.c 后缀源码
SRC_C_16_O2    = $(foreach dir,   $(SRCDIR_16_O2), $(wildcard $(dir)/*.c))
#转换为.o 格式文件名称，不带路径信息
OBJ_C_16_O2    = $(notdir $(patsubst %.c, %.o, $(SRC_C_16_O2)))

 #获得所有的.o 文件名称
OBJ = $(OBJ_C_16) $(OBJ_C_16_O2) $(OBJ_S_16) $(OBJ_C_32) $(OBJ_S_32)

$(OBJ_C_16_O2) : %.o : %.c
        $(CC) $(CC_OPTS_O2_16) -o $(OBJ_DIR)/$@ $<
        @echo
```

说明：

1. IMAGENAME 表示应用程序名字，上面的例子应用程序全名为：music.ap
2. 默认所有源文件都放在当前应用程序目录下，采用 MIPS 16e 指令集、O0 优化编译，如果有特殊的去修，比如上面的例子，当前目录下有个 O2 的子目录，里面的源文件用 O2 优化编译。所以需要添加额外的源文件路径、编译规则等。
3. LD_SCRIPT 指定链接脚本，指示如何链接驱动中的各个段，该脚本也是打包为*.ap 文件格式的脚本文件。应用程序的映像文件的链接可以分为 3 部分，应用主体、enhanced 模块和 Common 模块，这 3 个模块分别用 1 个链接脚本进行链接，这样的做法能让 enhanced 模块和 Common 模块的编译链接更加透明、更好维护。

5.6.6.2 应用程序 xn 脚本

什么是 linker 脚本文件 ？

描述如何把输入文件中的节（sections）映射到输出文件中，并控制输出文件的存储布局的脚本文件。

在应用和驱动开发时，系统会根据 bank 分组，提供应用和驱动的各组 bank 段的 linker 模板，指定应用和驱动编译链接时应遵守的链接模板。

应用程序在链接脚本模板基础上，根据应用程序代码和数据空间的分配情况，编写各自模板的 linker 脚本文件。

以 Music 应用程序为例，XN 文件的格式如下：

```
*****************************************************************/
/*----------------------- memory map --------------------------------------------------------*
*   ap_code            address                                          length   *
*   rcode              0xbfc1ee00-0xbfc1f5ff                            0x800(2k) *
* front control     (0x40**0000+0x1fe00)-(0x40**0000+0x205ff)    0x800(2k) *
* front UI           (0x48**0000+0x1f600)-(0x48**0000+0x1fdff)    0x800(2k) *
* front enhance1   (0x78**0000+0x27000)-(0x78**0000+0x277ff)    0x800(2k) *
* front enhance2   (0x71**0000+0x27800)-(0x71**0000+0x27fff)    0x800(2k) *
*-------------------------------------------------------------------------------------------*
*   ap_data            address                                          length   *
*   rdata              0x9fc1d200-0x9fc1d9ff                            0x800(2k) *
* lcd buffer data    0x9fc18000-0x9fc183ff                            0x400(1k) *
* applib(global data) 0x9fc19f80-0x9fc19fff                            0x80     *
*------------------------------------------------------------------------------------------*/
/*定义 GROUP BANK INDEX  和其他地址变量*/
INPUT(link_base.xn)

/*1.AP INDEX*/
AP_INDEX = FRONT_AP;   → 表示是前台应用，如果是后台应用，必须为 BACK_AP

/*2.从地址 map 图中获取 card 模块的常驻代码段物理地址，只需写低 14 位，开发人员需填写，集成开始后只由集成人员修改，驱动人员不允许修改*/
SRAM_TEXT_ADDR = SRAM_AP_FRONT_RCODE_ADDR;     → 前台应用 rcode  空间
SRAM_DATA_ADDR = SRAM_AP_FRONT_DATA_ADDR;      → 前台应用 rdata  空间
/*3.转换为链接地址,不能修改*/
RCODE_TEXT_ADDR = RCODE_ADDR_BASE + SRAM_TEXT_ADDR;
RDATA_DATA_ADDR = RDATA_ADDR_BASE + SRAM_DATA_ADDR;
```

版 权 所 有　侵 权 必 究

/*只是建议起始地址，也可以往后偏移，如 control_2 一般不是从这个地址，如果要占全部的空间，则使用 control_1*/ → BANK 地址空间说明，指定 BANK 组号和物理地址

```
BANK_CONTROL_1_ADDR_BASE = (AP_BANK_FRONT_CONTROL_1 << 24) + SRAM_AP_BANK_FRONT_CONTROL_ADDR;
BANK_UI_1_ADDR_BASE = (AP_BANK_FRONT_UI_1 << 24) + SRAM_AP_BANK_FRONT_UI_ADDR;
/*固定复用 basal 或者 codec 的空间，不能跨空间使用，跟 control1 和 control2 不一样*/
BANK_ENHANCED_1_ADDR_BASE = (AP_BANK_FRONT_ENHANCED_1 << 24) + SRAM_AP_BANK_FRONT_ENHANCED_1_ADDR;
BANK_ENHANCED_2_ADDR_BASE = (AP_BANK_FRONT_ENHANCED_2 << 24) + SRAM_AP_BANK_FRONT_ENHANCED_1_ADDR;

/*bank 的实际空间大小,不能修改*/   → 指定 BANK 段大小
BANK_CONTROL_SIZE = SRAM_AP_BANK_FRONT_CONTROL_SIZE;
BANK_UI_SIZE = SRAM_AP_BANK_FRONT_UI_SIZE;
BANK_ENHANCED1_SIZE = SRAM_AP_BANK_FRONT_ENHANCED_1_SIZE;
BANK_ENHANCED2_SIZE = SRAM_AP_BANK_FRONT_ENHANCED_2_SIZE;
/*系统允许的 bank 的最大空间大小，不能修改*/
AP_BANK_SPACE = BANK_SPACE;

OUTPUT_ARCH(mips)

ENTRY(__start)   → 应用程序加载后起始函数
EXTERN(base_op_entry)   →因库的链接是有顺序的，可能会导致某些库没链接进去，比如某些 api 库，这时可以在 xn 脚本中添加 ENTERN 声明该库中的符号确保会被链接进去
SECTIONS
{
    /* case  全局数据，各 AP 共享  */
    . = 0x9fc19f80;
    APP_GLOBE_DATA :
    {
        /*common globe  数据*/
        applib_globe_data.o(.bss)
        . = 0x80;
    }

     . = 0x9fc18000;
    LCD_BUFFER_DATA     :
    {
```

```
        music_playing_show_lyric.o(.lcdbuffer)
        . = 0x400;
    }
    /*常驻代码*/
    .text RCODE_TEXT_ADDR   :
    {
        /*4.填写 rcode 输入节*/
        api.a(.text)
        music_control.o(.text .rodata)
        app_timer_rcode.o(.text)
        message_rcode.o(.text)
    }
    /* 初始化了的全局数据段*/
    .data RDATA_DATA_ADDR     :    AT(ADDR(.text) + SIZEOF(.text))
    {
        /*5.填写 rdata 输入节，所以文件产生的.data 都被链接到这里*/
        music_main.o(.data)

        eh_rdata.o(.data)
        eh_fsel_rdata.o(.data)
        eh_bs_rdata.o(.data)
        eh_id3_rdata.o(.data)
        eh_lrc_get_rdata.o(.data)
    }
    /*未初始化的全局数据段，系统不会进行清零*/
    .bss :
    {
        music_main.o(.bss)
        music_setmenu_soundset_param.o(.bss)
        music_playing_show_lyric.o(.bss)
        music_setmenu_playmode_abset.o(.bss)
        music_setmenu_show_bookmark.o(.bss)
        applib_app_data.o(.bss)
        common_ui_data.o(.bss)
        common_func_data.o(.bss)

        eh_rdata.o(.bss)
        eh_fsel_rdata.o(.bss)
        eh_bs_rdata.o(.bss)
```

版 权 所 有 侵 权 必 究

```
        eh_id3_rdata.o(.bss)
        eh_lrc_get_rdata.o(.data)
        *(.sbss)
        *(.common)
        *(common)
    }

    /*6.链接 bank control 0 代码*/
    . = BANK_CONTROL_1_ADDR_BASE;
    → CONTROL BANK 组开头，把地址设置为第一个 bank 段开头
    OFFSET = . & 0x3ffff;

    BANK_CONTROL_1_0      :        →  ctor.o 要求链接在 CONTROL 组 bank 1 上
                                   → BANK 段名字必须以 BANK 开头
    {
        ctor.o(.text .rodata)
    }

    /*bank control 1*/
    . = ((. + AP_BANK_SPACE) & (~(AP_BANK_SPACE - 1)))   + OFFSET;
    → 把地址设置为下一个 bank 段开头
     BANK_CONTROL_1_1     :
    {
        music_main.o(.text .rodata)
        . = BANK_CONTROL_SIZE;
    }

    。。。
}
```

5.6.7 如何增加一个前台应用

下面以系统预留的 user1 应用为例，说明如何增加一个前台应用。

5.6.7.1 user1 演示目录说明

前台应用：user1

源代码路径：./ap_sample/ap_user1

User1 目录说明

源文件/数据文件	说明
User1_main.c	应用主程序，包括应用初始化，应用退出，应用主函数（场景循环），应用私有消息处理，以及全局变量定义等
User1_scene_play.c	播放场景主程序
User1_scene_menu.c	菜单场景主程序
User1_menu_cfg.c	可配置化菜单配置数据
User1.h	应用头文件，包含 user1_res.h 和 user1_sty.h
User1_res.h	应用图片和字符串资源头文件，由 UI Editor 工具自动生成
User1_sty.h	应用可配置化 UI 头文件，由 UI Editor 工具自动生成
User1.mcg	可配置化菜单配置文件
User1.ui 和 user1.sty	可配置化 UI 配置脚本和配置文件
User1.xls 和 ./pic	应用图片和字符串资源文件
makefile	应用工程 make 配置脚本
User1.xn	应用工程链接配置脚本，也是打包为*.ap 文件配置脚本

表 5.4.4-1 ./ap_sample/ap_user1 目录说明

5.6.7.2 user1 应用概要设计

在前面的 5.4.1 节前台应用的组成结构介绍中，我们知道前台应用是由场景调度循环构成的，而每个场景（除了菜单场景和目录浏览场景等控件场景外）则是由消息处理循环（gui 消息和应用私有消息处理循环，主要是 gui 消息处理循环）构成的。所以，在应用设计阶段，第一步是要确定应用由哪几个场景构成以及场景调度循环，每个场景的 ui 如何设计（对于菜单场景就是如何配置菜单树）以及如何与用户交互。

假设 user1 设计为 2 个场景，播放场景和菜单场景，场景调度循环描述如下：

1. 播放场景：

a) 进入播放场景的事件：

i. 应用初始进入，处于暂停状态。

ii. 从菜单场景返回播放场景。

b) 退出播放场景的事件：

i. 按下菜单键，进入菜单场景。

ii. 按下退出键，退出场景循环，进而退出应用。

版 权 所 有 侵 权 必 究

2. 菜单场景：

a) 进入菜单场景的事件：

i. 在播放场景下按下菜单键。

b) 退出菜单场景的事件：

i. 在最顶层菜单按下退出键。

ii. 选择叶子菜单项，执行菜单功能，返回播放场景。

3. 公共处理部分：

a) 收到应用退出消息，退出场景循环，进而退出应用。

假设**播放场景的 ui** 需求及设计描述如下：（当然，项目开发中，最终会由 UI 设计师制作资源图片和效果图）

1. 一个应用大背景。
2. 一个播放区域，0～9 循环播放，播放周期为 1 秒。
3. 一个静态标签，显示标语“演示前台应用”；一个状态标签，显示“正在播放”或“暂停播放”。

假设**菜单场景的菜单树配置**如下：

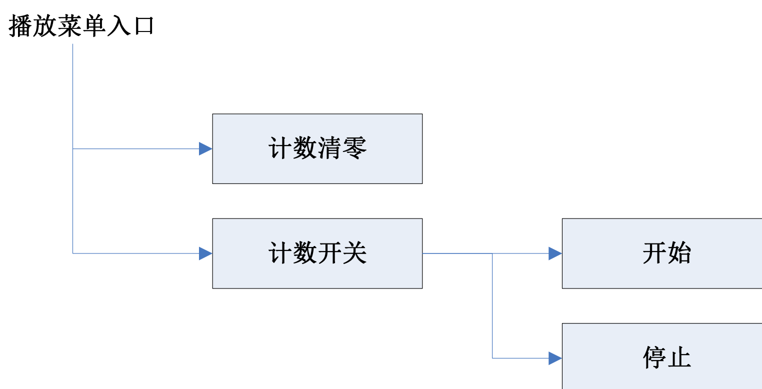


图 5.4.4-1 菜单场景的菜单树

明确以上设计要点以及完成详细设计（本演示前台应用比较简单，此阶段忽略）后，我们可以开始实现前台应用了。

5.6.7.3 步骤 1：搭建应用基本架构

选择合适的模板，即标案中的前台应用，或者演示前台应用，修改模板快速搭建前台应

版 权 所 有 侵 权 必 究

用基本架构。本演示前台应用选择的模板应用是 ap_picture。

步骤 1.1：编写 user1_main.c

在 case/ap 目录下新建 ap_user1 子目录，拷贝 pic_main.c 到该目录，改名为 user1_main.c。

User1.c 修改，具体步骤如下：

1. 修改文件描述。
2. 修改头文件包含。
3. 修改全局变量定义，仅保留基本全局变量，详细见下表说明。

全局变量	说明
g_user1_var	应用环境变量
g_comval	Case 环境变量
Ptos 和 prio	应用主线程（任务）栈地址和优先级
g_user1_scene_next	应用场景调度循环控制变量，指向接下来调度目标场景
User1_app_timer_vector	应用级定时器数组，应用级定时器的全局数据空间消耗由应用本身提供

表 5.4.4-2 应用基本全局变量说明

4. 修改接口及其实现，仅保留基本接口，详细见下表说明。（各个接口的详细实现参见源代码。）
5. 从 pic_comfunc.c 拷贝 pic_msg_callback 接口实现到 user1_main.c，并改名为 user1_msg_callback，仅保留 **MSG_APP_QUIT 和 default** 分支。

全局变量	说明
_user1_read_var 和 _user1_save_var	前者加载应用环境变量和 case 环境变量，以恢复现场；后者在应用退出时保存应用环境变量和 case 环境变量
_user1_app_init	应用装载初始化，必须为应用分配 ID，即在 case/inc/case_type.h 中添加，APP_ID_USER1 为系统预留前台应用，无需添加
_user1_app_deinit	应用注销处理，是应用装载初始化的逆过程
_user1_select_next_scene	场景退出后，决定接下来应该调度哪个场景
_deal_user1_result	应用退出前，决定接下来应该创建哪个前台应用
user1_msg_callback	应用私有消息分发处理接口，common 及各场景消息处理循环都统一调用该接口
main	应用主函数

表 5.4.4-3 应用基本接口说明

版 权 所 有 侵 权 必 究

步骤 1.2：编写 user1.h

拷贝 picture.h 到该目录，改名为 user1.h 。

User1.h 修改，包括文件描述，头文件包含，宏定义，类型定义，全局变量声明，接口声明等。

5.6.7.4 步骤 2：开发播放场景

步骤 2.1：编写 User1_scene_play.c 消息处理循环

拷贝 pic_play.c 到 ap_user1 目录，改名为 user1_scnen_play.c 。

User1_scene_play.c 修改，先实现基本的消息处理循环，具体步骤如下：

1. 修改文件描述。
2. 修改头文件包含。
3. 修改 gui 消息映射数组，即 user1_play_keymap_list ，该列表决定了播放场景响应哪些 gui 消息。另外，映射事件要添加在 case/inc/app_msg.h 的枚举类型 msg_apps_type_e 中。比如，本演示应用添加了 2 个映射事件，EVENT_USER1_PLAYING_PAUSE 和 EVENT_USER1_MENU。
4. 修改接口实现，仅保留必要的消息处理循环接口 user1_play_msg_deal 、user1_play 等。

步骤 2.2：可配置化 UI 设计

在 case/resource 目录下新建 user1 子目录，把 user1.xls 和 pic 目录拷贝到该子目录下，并从 case/resource/picture 下拷贝 copyfile.bat 批处理到 user1 子目录，用文本编辑器打开批处理文件，把 ap_picture 修改为 ap_user1。

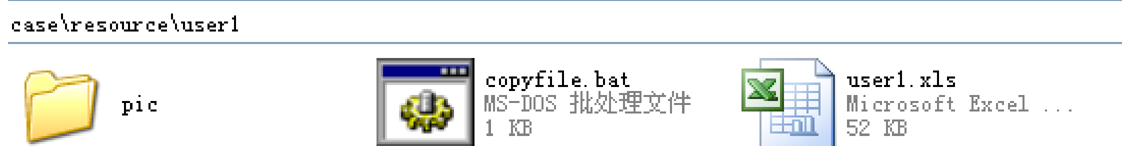


图 5.4.4-2 可配置化 UI 工程目录

打开 UI Editor 工具，选择工作目录 case/resource ，进入工具主界面，选择 AP 工程视图，再选择 user1 工程 。在弹出的对话框中（或者通过选择菜单“编辑->设置资源...”弹出该对话框），导入图片资源和字符串资源，接着编辑工程及其控件。

版 权 所 有 侵 权 必 究

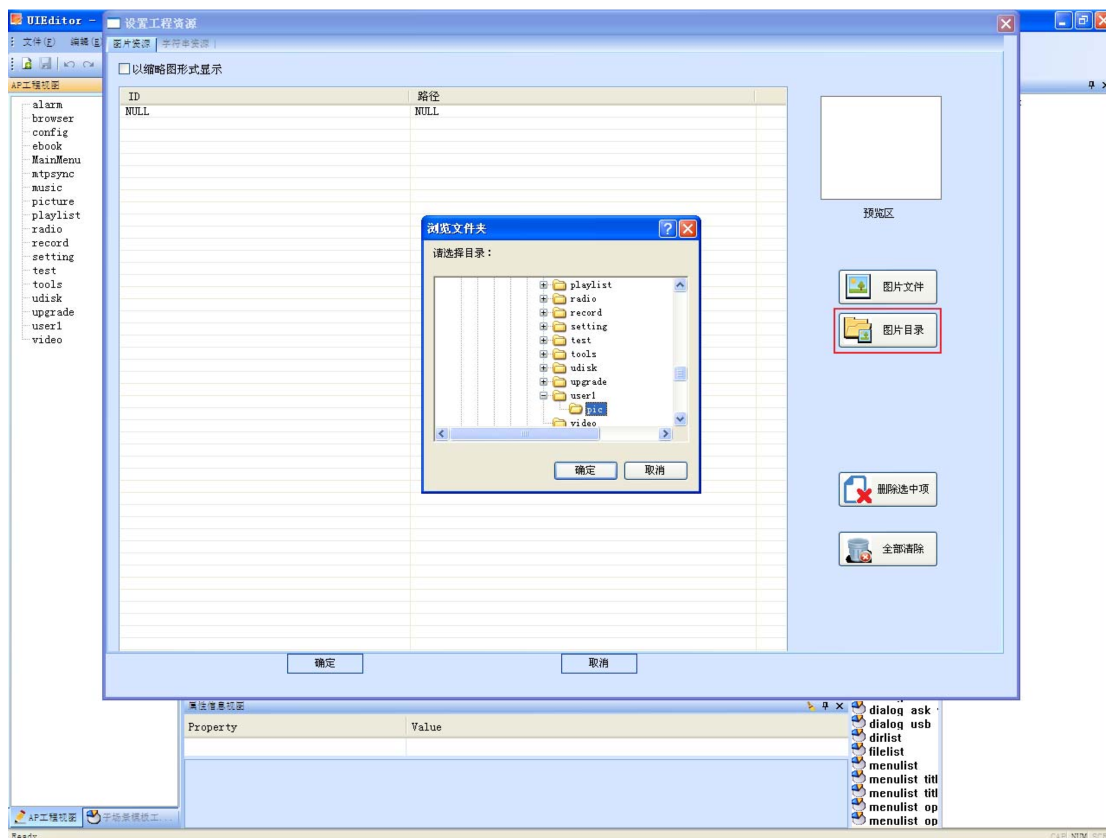


图 5.4.4-3 user1 设置工程资源-添加图片

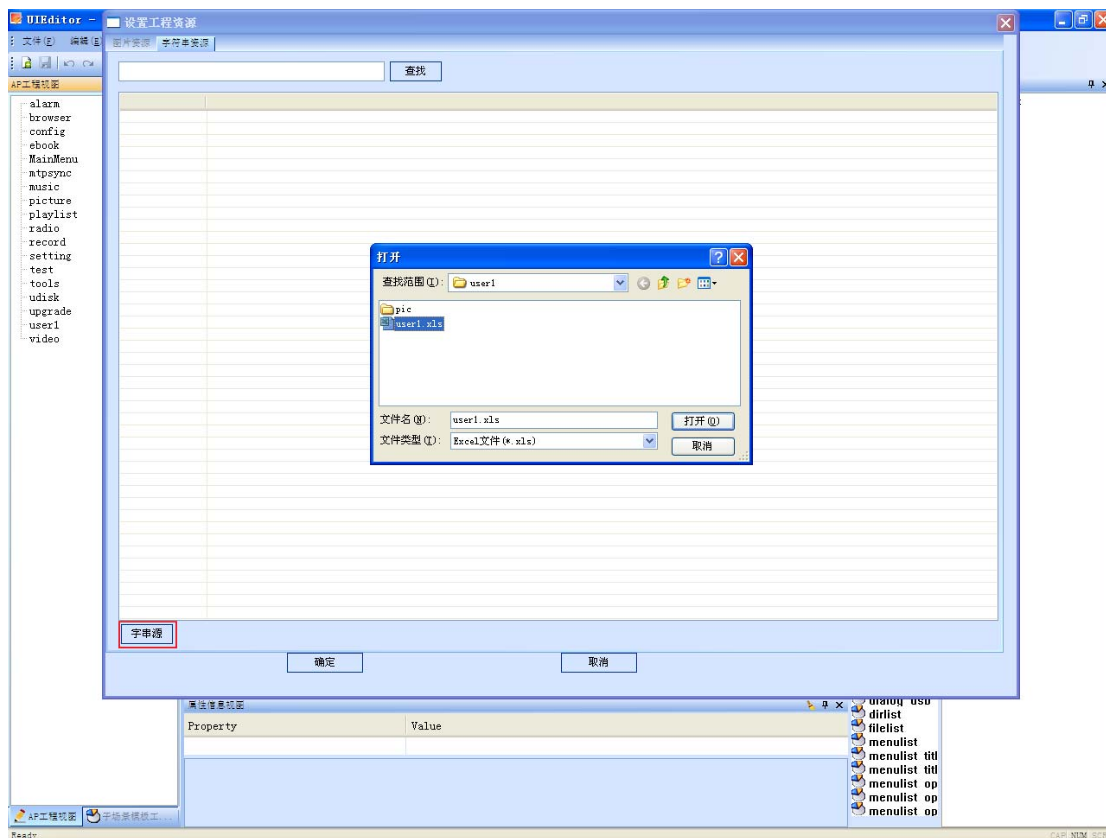


图 5.4.4-4 user1 设置工程资源-添加字符串

工程要选择其语言类型为简体中文（其他语言环境可选择对应语言类型）。

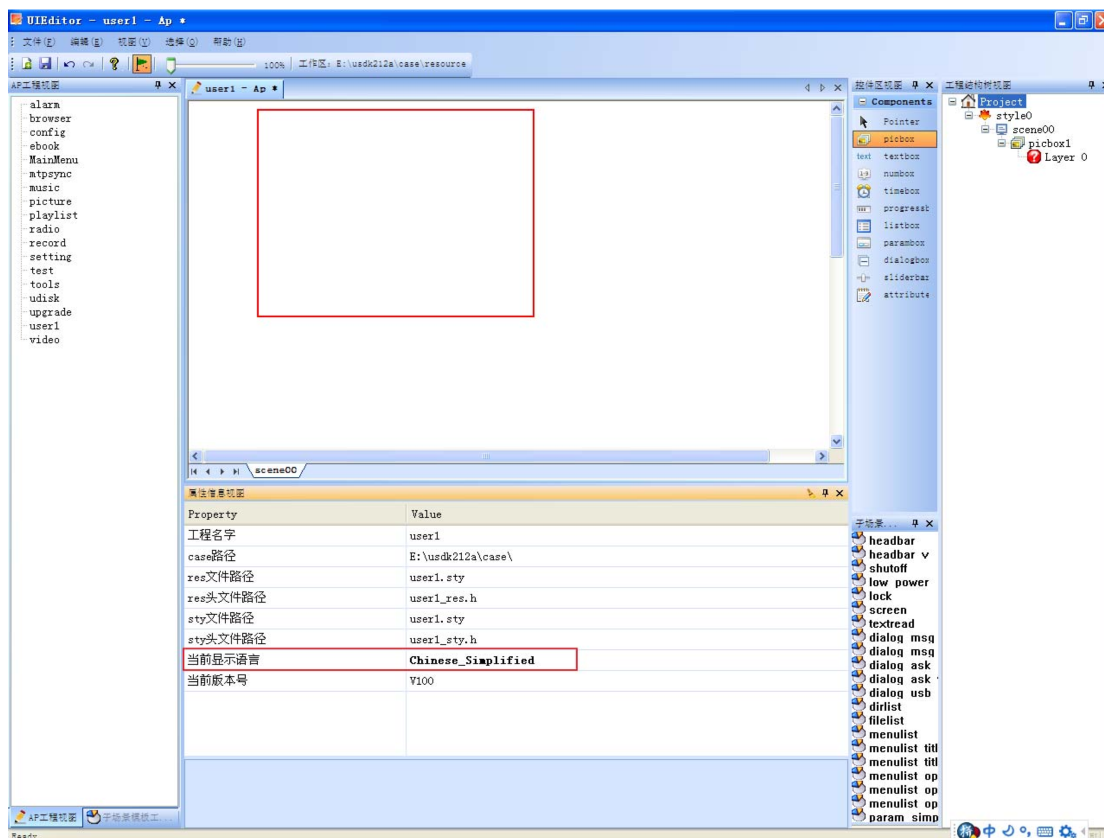


图 5.4.4-5 user1 工程设置当前显示语言为简体中文

为播放场景的 UI 编辑如下控件：

1. 一个大背景 picbox，名为 picbox_bg。
2. 一个用于计数的多帧 picbox，名为 picbox_playcount。
3. 一个静态标签 textbox，名为 textbox_string；一个状态标签 textbox，名为 textbox_state。

版权所有 侵权必究

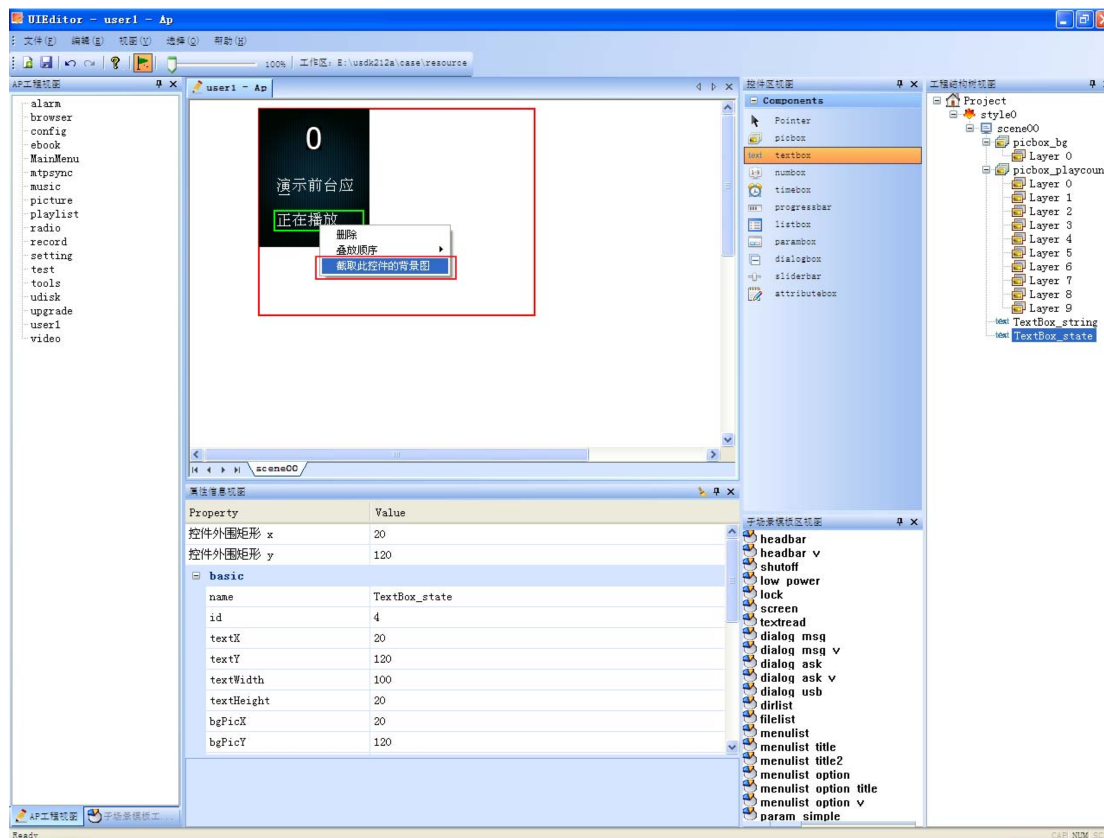


图 5.4.4-6 user1 编辑播放场景 UI

说明： textbox_state 控件的背景图通过右击弹出菜单“截取此控件的背景图”来自动截取，存放为 case/resource/user1/ BgPicDoc/ TextBox_state/ TextBox_state_bg.bmp 。

编辑完成后，选择菜单“文件->生成资源文件...”生成 user1.sty 数据文件，以及头文件 user1_res.h 和 user1_sty.h ，执行批处理文件 copyfile ，把这些结果文件拷贝到对应目标目录下。

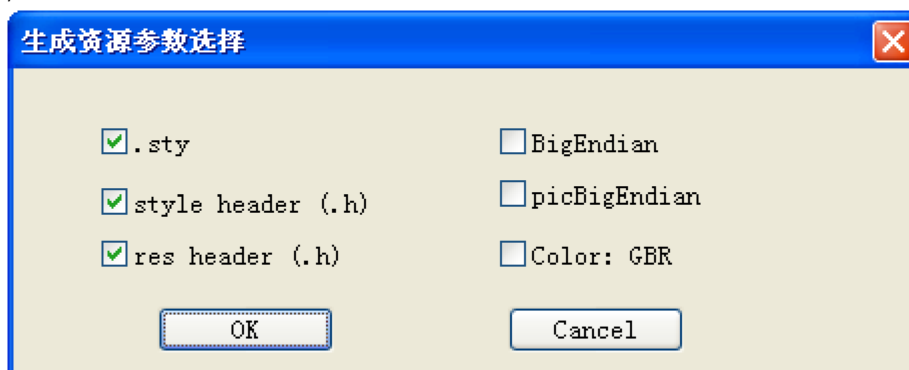


图 5.4.4-7 user1 生成资源文件

版 权 所 有 侵 权 必 究

接下来就可以编写播放场景的 UI 处理代码了。

步骤 2.3：继续完善 User1_scene_play.c UI 处理部分

打开 《us212a_lcd_driver 接口说明书.chm》 ，参考控件示例代码，编写控件显示处理代码。

步骤 2.5：继续完善 User1_scene_play.c 定时播放部分

打开 《us212a_common 接口说明书.chm》 ，参考定时器示例代码，编写定时计数处理代码。

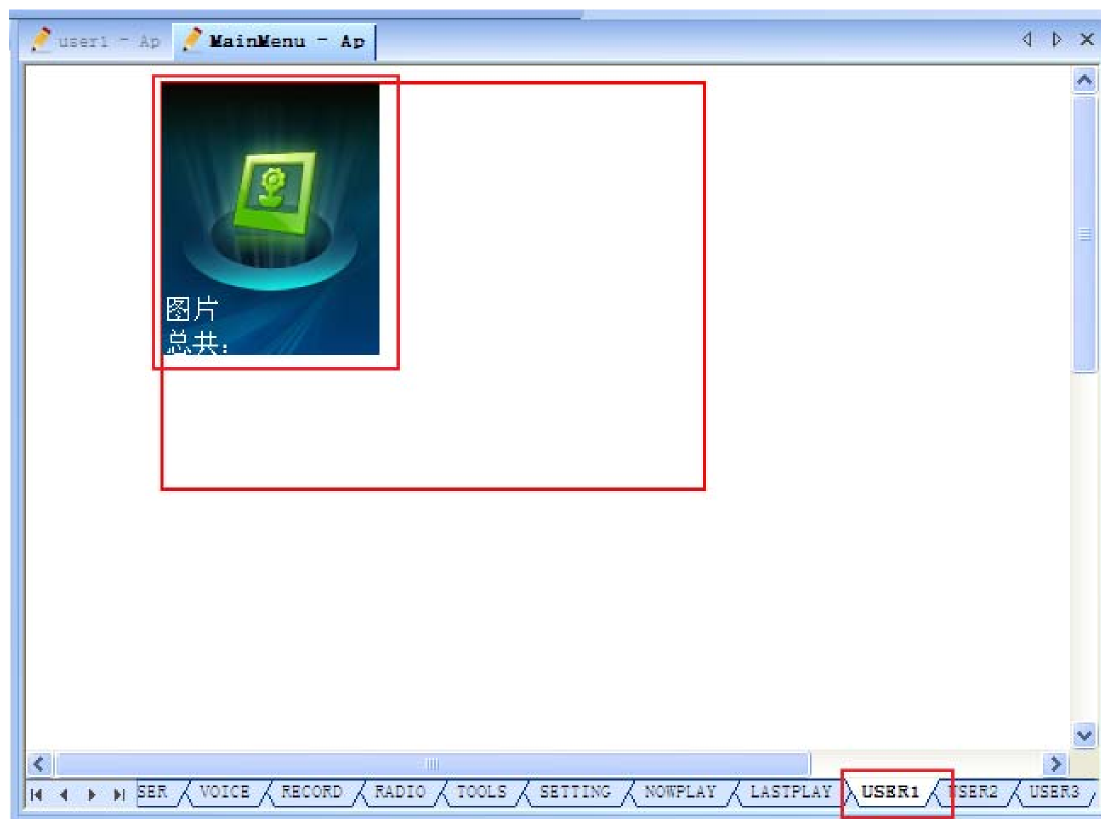
5.6.7.5 步骤 3：添加新 AP 到 case 中

步骤 3.1：修改 ap_manager

在 ap_manager 中 manager_get_name.c 的 app_name_ram 增加 AP 名称，由于 user1.ap 是预留的扩展 AP 名称，已经添加，此步骤跳过。

步骤 3.2：修改 ap_mainmenu

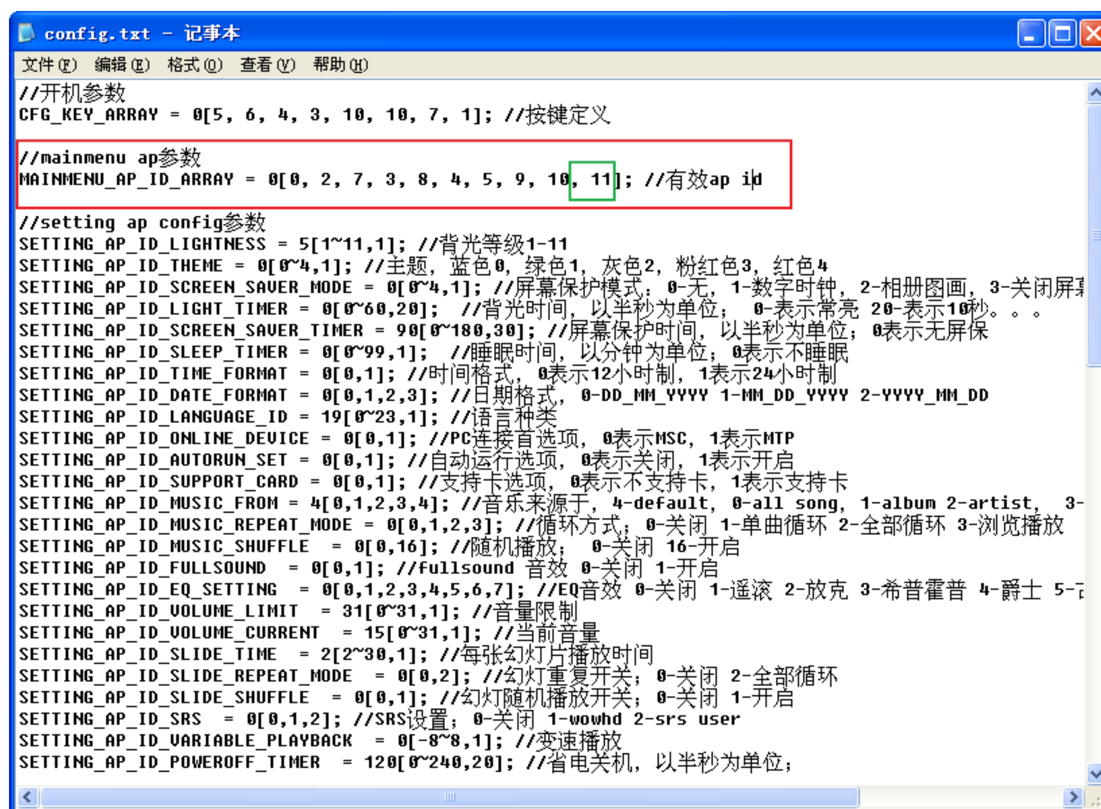
在 ap_mainmenu 中添加 user1 AP 的图标入口，需要编辑其可配置化 UI 工程，修改 mainmenu_display_init.c 中资源数组。详细见 ap_mainmenu 应用相应章节说明。



步骤 3.3：修改 config.txt

修改 case/fwpkg 下的 config.txt 脚本 MAINMENU_AP_ID_ARRAY 条目，把新增 AP 的编号添加到指定序号位置中，并执行批处理 case/tools/Gen_config/genconfig.bat 更新配置数据文件 config.bin。

注意：刚修改好 config.txt 更新 config.bin 后，在 UI simulator 上调试，必须先删除 case_simulator/bin/debug/VMdata.bin 文件，这样 ap_mainmenu 应用才会重新加载有效应用 ID 列表。另外，删除 VMdata.bin 后第一次运行可能会崩溃，重新运行以后就正常了。



```
config.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
//开机参数
CFG_KEY_ARRAY = 0[5, 6, 4, 3, 10, 10, 7, 1]; //按键定义

//mainmenu ap参数
MAINMENU_AP_ID_ARRAY = 0[0, 2, 7, 3, 8, 4, 5, 9, 10, 11]; //有效ap id

//setting ap config参数
SETTING_AP_ID_LIGHTNESS = 5[1~11,1]; //背光等级1-11
SETTING_AP_ID_THEME = 0[0~4,1]; //主题, 蓝色0, 绿色1, 灰色2, 粉红色3, 红色4
SETTING_AP_ID_SCREEN_SAVER_MODE = 0[0~4,1]; //屏幕保护模式: 0-无, 1-数字时钟, 2-相册图画, 3-关闭屏
SETTING_AP_ID_LIGHT_TIMER = 0[0~60,20];  //背光时间, 以半秒为单位;  0-表示常亮 20-表示10秒。。。
SETTING_AP_ID_SCREEN_SAVER_TIMER = 90[0~180,30]; //屏幕保护时间, 以半秒为单位; 0表示无屏保
SETTING_AP_ID_SLEEP_TIMER = 0[0~99,1];  //睡眠时间, 以分钟为单位; 0表示不睡眠
SETTING_AP_ID_TIME_FORMAT = 0[0,1]; //时间格式, 0表示12小时制, 1表示24小时制
SETTING_AP_ID_DATE_FORMAT = 0[0,1,2,3]; //日期格式, 0-DD_MM_YYYY 1-MM_DD_YYYY 2-YYYY_MM_DD
SETTING_AP_ID_LANGUAGE_ID = 19[0~23,1]; //语言种类
SETTING_AP_ID_ONLINE_DEVICE = 0[0,1]; //PC连接首选项, 0表示MSC, 1表示MTP
SETTING_AP_ID_AUTORUN_SET = 0[0,1]; //自动运行选项, 0表示关闭, 1表示开启
SETTING_AP_ID_SUPPORT_CARD = 0[0,1]; //支持卡选项, 0表示不支持卡, 1表示支持卡
SETTING_AP_ID_MUSIC_FROM = 4[0,1,2,3,4]; //音乐来源于, 4-default, 0-all song, 1-album 2-artist,  3-
SETTING_AP_ID_MUSIC_REPEAT_MODE = 0[0,1,2,3]; //循环方式; 0-关闭 1-单曲循环 2-全部循环 3-浏览播放
SETTING_AP_ID_MUSIC_SHUFFLE  = 0[0,16]; //随机播放;  0-关闭 16-开启
SETTING_AP_ID_FULLSOUND  = 0[0,1]; //fullsound 音效 0-关闭 1-开启
SETTING_AP_ID_EQ_SETTING  = 0[0,1,2,3,4,5,6,7]; //EQ音效 0-关闭 1-遥滚 2-放克 3-希普霍普 4-爵士 5-
SETTING_AP_ID_VOLUME_LIMIT  = 31[0~31,1]; //音量限制
SETTING_AP_ID_VOLUME_CURRENT  = 15[0~31,1]; //当前音量
SETTING_AP_ID_SLIDE_TIME  = 2[2~30,1]; //每张幻灯片播放时间
SETTING_AP_ID_SLIDE_REPEAT_MODE  = 0[0,2]; //幻灯重复开关; 0-关闭 2-全部循环
SETTING_AP_ID_SLIDE_SHUFFLE  = 0[0,1]; //幻灯随机播放开关; 0-关闭 1-开启
SETTING_AP_ID_SRS  = 0[0,1,2]; //SRS设置; 0-关闭 1-wowhd 2-srs user
SETTING_AP_ID_VARIABLE_PLAYBACK  = 0[-8~8,1]; //变速播放
SETTING_AP_ID_POWEROFF_TIMER  = 120[0~240,20]; //省电关机, 以半秒为单位;
```

图 5.4.4-8 config.txt 增加前台 AP

5.6.7.6 步骤 4：在 UI simulator 上调试播放场景

步骤 4.1：创建 UI simulator 工程

在 case_simulator/Apps 目录下新建 ap_user1 工程目录，拷贝 ap_picture.dsp、ap_picture.dsw、SMcommval.c 三个文件，前两者改名为 ap_user1.dsp 和 ap_user1.dsw，用文本编辑器如 UE 打开这两个文件，把所有 ap_picture 替换为 ap_user1。接着打开 case_simulator/Apps/ap_mainmenu 下的 ap_mainmenu.dsw 工程文件，插入 ap_user1 工程，把工程中的源文件替换为 case/ap/ap_user1 目录下的源文件。再打开工程设置对话框，设置

好目标名称为 user1.ap 。

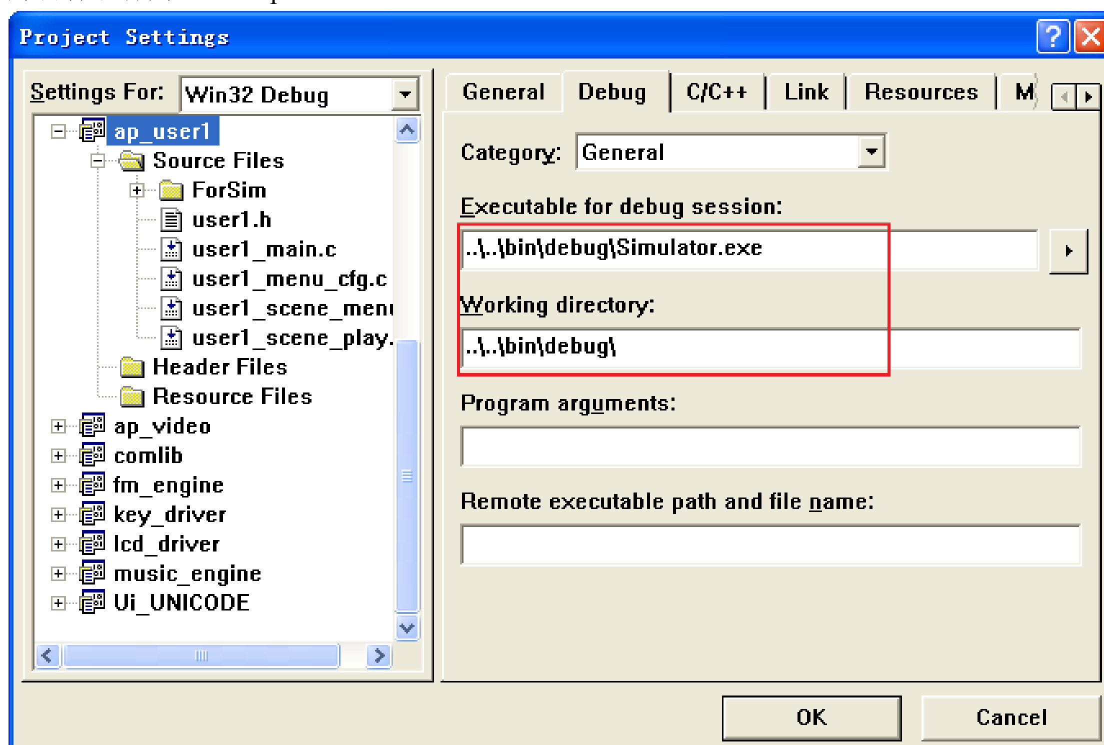


图 5.4.4-8 user1 工程调试选项设置

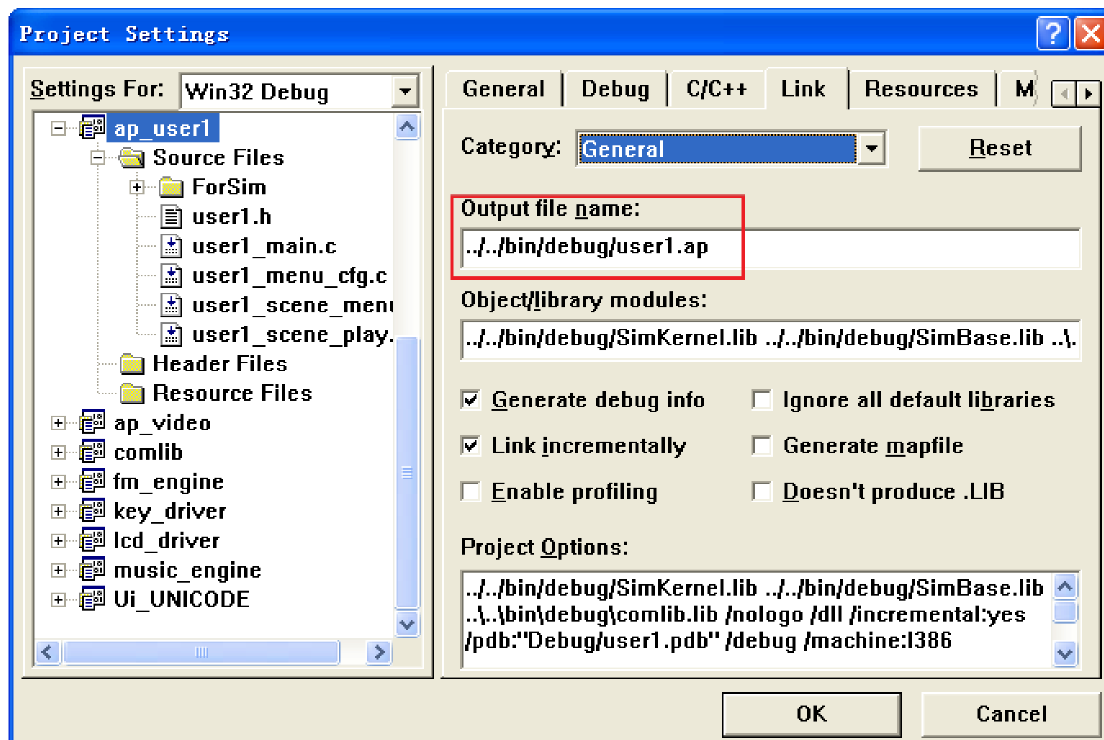


图 5.4.4-8 user1 工程输出选项设置

步骤 4.2：build、调试

重新 build 工程 ap_manager 、ap_mainmenu 、ap_user1 （先屏蔽掉 _user1_app_init 和 _user1_app_deinit 接口中菜单相关的代码行），清除 error 和 warnning。build 成功之后调试运行，在 ap_mainmenu 应用主界面选择 user1 AP 图标，即可进入 ap_user1 开始调试。

图 5.4.4-9 user1 工程运行截图

版 权 所 有 侵 权 必 究

5.6.7.7 步骤 5：开发菜单场景

步骤 5.1：编写 user1_scene_menu.c

拷贝 pic_menu.c 到 ap_user1 目录，改名为 user1_scnen_menu.c 。

修改菜单场景入口函数 user1_menu，编写各个菜单项响应函数 menu_func_count_clear 、 menu_func_count_switch 、 menu_func_count_start 、 menu_func_count_stop。

步骤 5.2：编写菜单配置文件 user1_menu_conf.c

拷贝 pic_menu_cfg.c 到 ap_user1 目录，改名为 user1_menu_cfg.c 。

修改 user1_entrymenu 、user1_entry 、item_head 、item 等。

步骤 5.3：编辑可配置化菜单

将以上两个源文件添加到 ui simulator 工程中，重新编译。

将生成的 case_simulator/bin/debug/user1.ap 文件拷贝到 case/fwpkg/ap 目录下，并确保该目录下存在 user1.sty 文件（在 UI Editor 工程生成资源文件后运行 copyfile.bat 自动拷贝到该目录下），然后打开 case/tools 目录下 TreeLayer.exe 工具，点击 select app 按钮，选择 user1.ap 文件，工具列出入口菜单和菜单项列表，编辑好菜单树，点击 Gen mcg 按钮，生成 user1.mcg 文件。

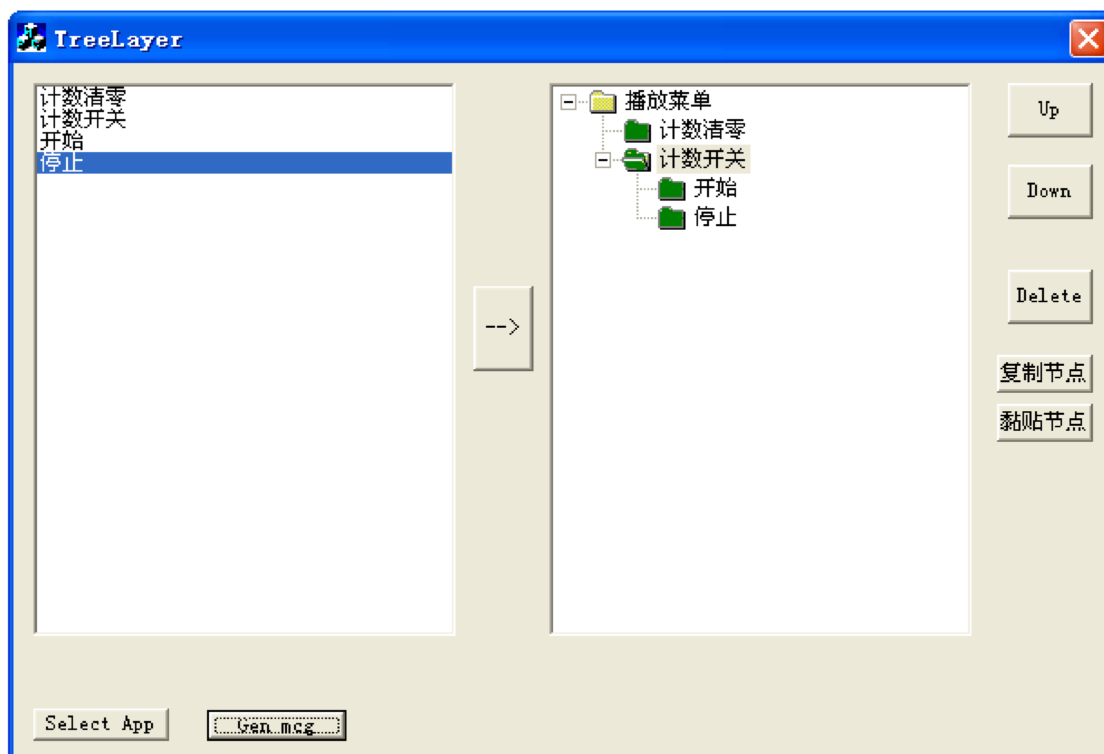


图 5.4.4-10 user1 可配置化菜单树编辑

5.6.7.8 步骤 6：在 UI simulator 上调试菜单场景

打开之前屏蔽掉的 _user1_app_init 和 _user1_app_deinit 接口中菜单相关的代码行，重新 build ap_user1 工程，清除 error 和 warning，build 成功之后调试运行接着调试菜单场景。

图 5.4.4-11 user1 菜单场景运行效果图

5.6.7.9 步骤 7：上板调试 user1 AP

拷贝 makefile 和 ap_picture.xn 到 ap_user1 目录下，后者改名为 user1.xn 。

步骤 7.1：修改 makefile

详细参见节 makefile 模版修改指南。

步骤 7.2：修改 user1.xn

详细参见节 xn 模版修改指南。

步骤 7.3：make

在 cygwin 上 make ap_user1 工程，清除编译链接错误，生成 user1.ap 文件。

步骤 7.4：固件打包

在 case/fwpkg 目录下的 fwimage.cfg 固件打包脚本中添加 ap_user1.ap 、user1.mcg 、user1.sty 等打包项，重新 make case 所有工程，运行批处理文件 case/build_sty.bat 更新 mcg 文件，运行批处理文件 buildfw.bat ，生成固件 US212A_EVB.fw 和 US212A_DEMO.fw。

步骤 7.5：下载固件到小机，即可验证结果是否正确。

5.7 后台应用设计与开发

后台 AP：又称引擎 AP，是在后台的实际功能应用，现在方案中有两个引擎，即音乐引

擎和 FM 引擎，引擎 AP 需要由配对的前台 AP 控制，处于被动位置。

后台 AP 对 psp 来说，都属于 AP，其映像结构和前台 AP 是一样的。当然，后台使用的 Common 模块 applib 和 common_func 中的公共接口，这些都已经在前台应用章节做过介绍，这里就不再重复。

5.7.1 后台应用的组成结构

后台应用以消息处理循环为基本架构，应用程序总体构成是一个消息循环，在消息循环中接收并处理应用消息。另外，后台应用同样会有一些定时处理事件，比如定时获取解码器状态，这些就由应用级定时器完成。

5.7.1.1 应用组成部分

后台应用主要由以下几个部分组成：

- 应用主体，即业务部分
- Common 模块，即应用基础功能接口库
- 中间件和函数库，包括文件浏览器，文件选择器，歌词解释器，ID3 解释器，USB 函数库，等等
- 运行时库，ctor.o，应用被调度时，会先运行该库中的 __start 函数，由该函数调用应用的 main 函数
- API 接口库，api.a，kernel 和各驱动的访问入口
- make 脚本，*.makefile
- xn 链接脚本，*.xn ，一般来说，后台应用包括 3 个 xn 脚本，分别是 AP 自身链接脚本、enhanced 部分链接脚本和 common 部分链接脚本；链接脚本也充当 AP 打包的配置文件
- 固件配置化脚本 config.bin

5.7.1.2 应用主体介绍

后台应用的应用主体是存放在应用子目录下的源代码，从代码的作用上可以分成应用基本架构、应用业务功能：

- AP 基本架构
  - 初始化，包括 applib 注册，消息管理器初始化，应用级定时器注册，读取应用 vm 环境变量等。
  - 退出，包括保存应用 vm 环境变量，applib 注销等。
  - 消息处理循环，（被动）接受前台应用控制及与系统交互。

版 权 所 有 侵 权 必 究

- AP 业务功能
  - 利用应用级定时器周期执行。
  - 消息循环的应用私有消息的处理。

5.7.2 后台应用的内存空间

5.7.2.1 常驻代码空间

后台应用常驻代码段：0xbfc1e800-0xbfc1edff 共 1.5KB，用来存放 AP、Common 模块、enhanced 模块的常驻接口及相关 const data、bank data。

5.7.2.2 常驻数据空间

后台应用的数据空间

0x9fc1da00-0x9fc1dfff 共 1.5KB，该全局数据空间是 AP、Common 模块以及 enhanced 模块共用的。

Case全局数据空间

0x9fc19f80-0x9fc19fff 共 128B，该全局数据空间是所有 AP 共享的，只会在 ap_manager 中初始化一次，主要用来存放 Common 模块的一些 case 全局数据。

5.7.2.3 BANK 代码及 BANK 数据空间

Back Control bank 组：(0x60**0000+0x1e000)-(0x60**0000+0x1e7ff) 共2KB，主要用来存放AP 非常驻接口、Common 非常驻接口等。其中 Common 模块占用 control bank 组中bank 40 ~ bank 63这24 bank号。

Back Enhanced BANK 1 组：(0x7a**0000+0x28800)-(0x7a**0000+0x28fff) = 2KB ，用于存放中间件模块接口，如果不需要中间件支持，可放任何代码和数据。

Back Enhanced BANK 2 组：(0x7b**0000+0x2a000)-(0x7b**0000+0x2afff) = 4KB ，用于存放中间件模块其他接口，如果不需要中间件支持，可放任何代码和数据。

5.7.2.4 运行栈空间

后台应用可能会创建子线程，而子线程是需要独立的运行栈的，所以系统分配给后台应用的运行栈空间应该根据有无子线程分别看待：

- 无子线程时：0x9fc265c0 ~ 0x9fc25b00 = 0xac0 B
- 有子线程时：

版 权 所 有 侵 权 必 究

- ✧ 主线程：0x9fc265c0 ~ 0x9fc26000 = 0x5c0 B
- ✧ 子线程：0x9fc26000 ~ 0x9fc25b00 = 0x500 B

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。

另外，以上栈空间的划分，并不是限制死了，用户可以根据自己应用程序的实际情况灵活调整，即主线程可以在定义 OS_STK *ptos 时灵活赋值，而子线程则在创建线程时灵活填写 pthread_param_t->ptos 成员的值。

5.7.2.5 堆空间

系统提供了 512 字节的堆空间，为整个系统共用，空间相对来说还是很局限的。应用程序可以临时申请少量内存。

申请堆：sys_malloc(&menu_history_addr, path_size);

释放堆：sys_free(&menu_history_addr);

5.7.2.6 VRAM 空间

VRAM 是一块从当前主盘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。

系统为 Case 分配 512KB 的 VRAM 空间，每个应用分配 1KB，用于保存应用环境变量，其余空间用作应用的临时数据缓冲。

分配使用 VRAM 空间要注意不能覆盖其他模块的 VM 空间，并且要注意 0x80000 开始的空间分配给 PSP 使用，Case 原则上不能使用，如果真的需要使用 0x80000 后的空间，请与我们的工程师联系取得支持。

5.7.3 引擎的互斥处理流程

引擎作为功能的实际执行模块，一般会占用平台的硬件资源等排他性资源，比如 music 引擎需要占用音频解码 DSP 硬件模块，而视频应用也一样需要该硬件资源来解码，所以 music 引擎与视频应用就产生了冲突。在后台正在播放音乐时，进入视频应用，在视频解码前必须主动将 music 引擎杀死。

当然，还有另一种内存资源冲突的情况，比如 music 引擎和 fm 引擎，所以后台正在播放音乐时，进入 radio 应用，在启动 fm 引擎前，必须将 music 引擎杀死。对于这种类型的

版 权 所 有 侵 权 必 究

冲突，我们还要注意前台应用因为“非法”的使用引擎应用的内存资源的情形，也需要前台应用主动将引擎应用杀死。

我们规定：引擎冲突由前台应用负责检查并执行同步杀死引擎的任务，我们提供了 engine_type_e get_engine_type(void) 获取引擎类型和 engine_state_e get_engine_state(void) 获取引擎状态等接口，用户可以根据引擎类型和状态方便判断是否与当前前台应用有冲突。

5.7.4 如何增加一个后台应用

5.7.4.1 user1_engine 演示目录说明

后台应用：user1_engine

源代码路径：./ap_sample/user1_engine

user1_engine 目录说明

源文件/数据文件	说明
User1engine_main.c	应用主程序，包括应用初始化，应用退出，应用主函数，以及全局变量定义等
User1engine_control.c	应用控制模块，包括应用私有消息循环等
User1engine.h	应用头文件
makefile	应用工程 make 配置脚本
User1.xn	应用工程链接配置脚本，也是打包为*.ap 文件配置脚本

5.7.4.2 user1_engine 应用概要设计

后台应用相比前台应用会简单很多，没有了场景循环，没有了可配置化 UI 和可配置化菜单，也没有了 gui 消息循环，仅仅剩下应用私有消息循环和应用级定时器。

本后台应用以 music_engine 为模版。

5.7.4.3 步骤 1：搭建应用基本架构

在 case/ap 目录下新建 user1_engine 子目录，拷贝 mengine_main.c 、mengine_control.c 和 app_mengine.h 到该目录，改名为 user1engine_main.c 、user1engine_control.c 和 user1engine.h 。

User1engine_main.c 修改，具体步骤如下：

1. 修改文件描述。

2. 修改头文件包含。
3. 修改全局变量定义，仅保留必要的 g_u1engine_var 、ptos 、prio 、u1engine_app_timer_vector 等。
4. 修改接口实现，仅保留必要的 _user1_read_var 、_app_init 、_app_deinit 、main 等，各个接口的详细修改参见源代码。

User1engine_control.c 修改，具体步骤如下：

- 修改文件描述。
- 修改头文件包含。
- 修改接口实现，仅保留必要的 u1engine_control_block 、u1engine_reply_msg 等，各个接口的详细修改参见源代码。

User1engine.h 修改，包括文件描述，头文件包含，宏定义，类型定义，全局变量声明，接口声明等。

5.7.4.4 步骤 2：在 case 添加 user1_engine AP

在 ap_manager 中 manager_get_name.c 的 app_name_ram 增加 AP 名称 u1engine.ap。

5.7.4.5 步骤 3：在 UI simulator 上调试消息通信等

在 case_simulator/Apps 目录下新建 user1_engine 工程目录，拷贝 music_engine.dsp 、music_engine.dsw 、SMcommval.c 三个文件，前两者改名为 user1_engine.dsp 和 user1_engine.dsw，用文本编辑器如 UE 打开这两个文件，把所有 music_engine 替换为 user1_engine。接着打开 case_simulator/Apps/ap_mainmenu 下的 ap_mainmenu.dsw 工程文件，插入 user1_engine 工程，把工程中的源文件替换为 case/ap/user1_engine 目录下的源文件。再打开工程设置对话框，设置好目标名称为 u1engine.ap 。

重新 build 工程 ap_manager 、user1_engine，设置好断点，调试运行，选择 user1 AP 图标，即可进入 ap_user1 开始调试。

5.7.4.6 步骤 4：上板调试

拷贝 makefile 和 music_engine.xn 到 user1_engine 目录下，后者改名为 user1_engine.xn 。

步骤 4.1：修改 makefile，详细参见节 makefile 模版修改指南。

步骤 4.2：修改 user1_engine.xn，详细参见节 xn 模版修改指南。

步骤 4.3：在 cygwin 上 make user1_engine 工程，清除编译链接错误，生成 u1engine.ap 文件。

步骤 4.4：在 case/fwpkg 目录下的 fwimage.cfg 固件打包脚本中添加 u1engine.ap 等打

版 权 所 有 侵 权 必 究

包项，重新 make case 所有工程，运行批处理文件 buildfw.bat ，生成固件 US212A_EVB.fw 和 US212A_DEMO.fw。

步骤 4.5：下载固件到小机，即可验证结果是否正确。

5. 8 多线程设计与开发

5.8.1 多线程架构

并行运行有两种方法，一种是进程（应用）级并行，一种是线程级并行。进程需要很多资源，并且独立性很强。而（子）线程只需要一个独立的运行栈，也能表现出同样的并行性。

所以，若某个功能需要独立于应用主体，或为了并行运行（比如电子书应用中计算总页数），或为了独立可控（比如图片应用长时间 bmp 解码，为了用户可随时中止解码），都可以创建一个子线程来运行。

当然，子线程需要一个独立的运行栈，由于空间有限，我们只允许创建一个前台应用子线程。

5.8.1.1 创建子线程

```
1)  thread_para.start_rtn = thread_loop;
2)  thread_para.arg = (void *)arg_pointer;
3)  thread_para.ptos = (void *) AP_FRONT_HIGH_STK_POS;
4)  libc_pthread_create(&thread_para, AP_FRONT_HIGH_PRIO,
    CREATE_NOT_MAIN_THREAD );
```

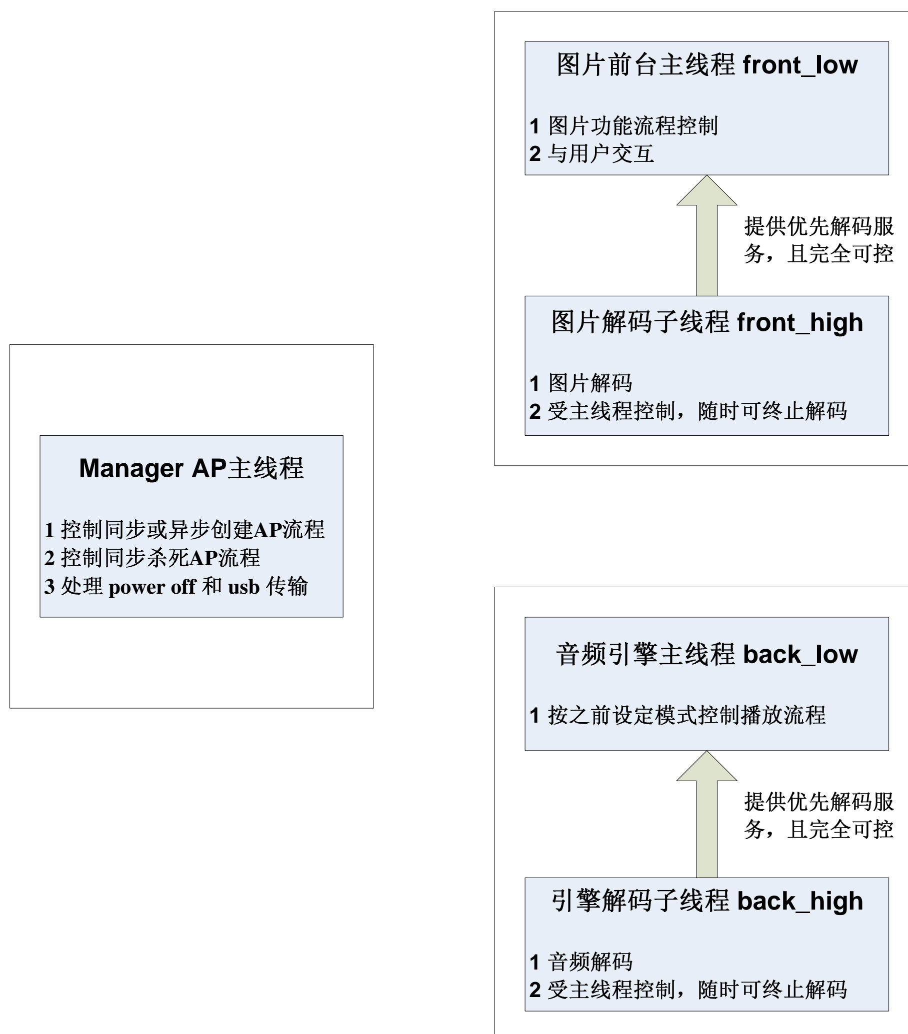
5.8.1.2 销毁子线程

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 start_rtn 函数的最后调用 libc_pthread_exit(); 接口主动退出子线程。

子线程的退出有时是由应用程序控制的，也可以自己确定执行完子线程的所有任务而自然退出的。

5.8.1.3 典型多线程场景

在典型的双应用场景下，即后台播放音乐，前台观看图片，其多线程框架如下所示：



5.8.2 多线程开发

5.9 驱动程序设计与开发

US212A 的驱动程序是自定义的一种设备驱动，其后缀为 DRV。

US212A 方案中，驱动的架构与接口都跟以前的 9X 方案有较大改动：

- 每个驱动提供一个统一的接口，节省接口 API 占用的常驻空间。上层调用不同接口

函数时用命令区分，类似 linux 驱动中的 iotcl 的形式。

- 装载驱动时可以获得驱动总接口的地址，系统只提供这个总接口的 API 给 AP 。也就是说，驱动需定义接口表，提供接口表的首地址给系统。
- 调用具体接口的分发跳转过程代码由系统驱动管理部分统一实现。同样可以节省各驱动的常驻空间。
- 为了使 AP 应用调用时看到的接口调用与原来 9X 平台的接口调用类似，故定义各接口调用的宏。AP 调用接口时，直接使用该接口对应的宏定义即可。

5.9.1 驱动程序的组成结构

5.9.1.1 驱动程序工程

US212A 的驱动程序工程包括以下组成部分：

- 源代码
  - ✧ 初始化接口和退出接口，并使用 module_init 和 module_exit 定义
  - ✧ 驱动接口表声明及定义，与外部接口命令号及宏定义一一对应
  - ✧ 其他源代码
- 外部接口头文件：外部接口命令号定义、外部接口宏定义、其他数据结构定义
- Makefile：驱动 make 脚本，make 包括编译和链接等整个过程
- xn：驱动链接和打包脚本

驱动接口详细说明请参考 驱动程序接口详解 一节。
makefile 和 xn 脚本的编写请参考 驱动 makefile 与 xn 脚本编写 一节。

5.9.1.2 驱动程序映像文件

US212A 的驱动程序映像文件 .drv 的文件头如下所示：

Name	Offset (byte)	Length (bytes)	Descriptor
DRV_FileType	0	1	FILE_DRV 'D' 文件扩展名： **“. DRV”**
DRV_DriverType	1	1	Driver Type:
Majo_version	2	1	主版本号

版 权 所 有 侵 权 必 究

Minor_version	3	1	次版本号
Magic	4	4	Magic 标志
Text_offset	8	4	常驻代码段在文件内偏移
Text_length	12	4	常驻代码段长度，0 表示没有常驻代码段
Text_addr	16	4	常驻代码段在内存中的地址
Data_offset	20	4	常驻数据段在文件内偏移
Data_length	24	4	常驻数据段长度，0 表示没有常驻数据段
Data_addr	28	4	常驻数据段在内存中的地址
Bss_length	32	4	Bss 段长度
Bss_addr	36	4	Bss 段起始地址
DRV_init_func_addr	40	4	初始化函数地址
DRV_exit_func_addr	44	4	退出函数地址
DRV_BankAFileAddr	48	4	Bank A banks 在驱动程序文件中的起始地址
DRV_BankBFileAddr	52	4	Bank B banks 在驱动程序文件中的起始地址
DRV_BankCFileAddr	56	4	Bank C banks 在驱动程序文件中的起始地址
Drv_op_entry	60	4	驱动接口表地址，该接口表必须放在常驻数据段中

从上面表格可以知道，驱动程序映像文件由以下部分组成：

- 常驻代码段：有且只有一个常驻代码段，如果驱动需要包含多段不连续的常驻代码段，则实现方法会稍微复杂点。
- 常驻数据段：包括 data 段和 bss 段，前者是已经初始化了的全局变量，后者是未初始化的全局变量，在驱动加载时，由 OS 清 0。
- Bank a/b/c 代码段：每个驱动模块可以包含 3 个 bank 组，通过驱动 API entry 号区分，每个 bank 组最多支持 64 个 bank，其中 bank c 组目前仅给 ui 驱动使用。驱动 API entry 号定义如下：

```
typedef enum
{
 DRV_GROUP_STG_BASE = 0,
 DRV_GROUP_STG_CARD,
```

```
DRV_GROUP_STG_UHOST,
DRV_GROUP_FAT32,
DRV_GROUP_FAT16,
DRV_GROUP_EXFAT,
DRV_GROUP_UI,
DRV_GROUP_LCD,
DRV_GROUP_FM,
DRV_GROUP_KEY,
DRV_GROUP_I2C,
DRV_GROUP_AUDIO_DEVICE,
DRV_GROUP_SYS = 15,
} drv_type_t;
```

- 3 个重要地址：初始化函数地址、退出函数地址和驱动接口表地址。
  - ✧ 初始化函数地址：驱动安装时调用，通过 module_init(DRV_init_func) 定义。
  - ✧ 退出函数地址：驱动卸载时调用，通过 module_exit(DRV_exit_func) 定义。
  - ✧ 驱动接口表地址：在驱动链接脚本 xn 中，通过 ENTRY(Drv_op_entry) 定义。

5.9.2 驱动程序的内存空间

驱动程序从内存占用的角度上看，可以分成两种：一种是常驻内存的，比如 KEY 驱动、LCD 驱动、UI 驱动等，另一种是按需加载的，比如文件系统、卡驱动等。

对于第一种驱动，系统务必要为其分配独占的常驻代码和常驻数据空间，而对于第二种，则可以按使用场景分场景复用。

驱动的常驻代码空间和常驻数据空间我们就不再说了，具体查看各个驱动的内存使用说明，我们这里会说明下各个驱动公用的 BANK 空间。

在前面的内容我们也已经介绍过，驱动有 3 组 BANK ，即 BANK A，BANK B，BANK C，其中 BANK 专用于 UI 驱动，以提升 UI 驱动的执行效率。

- BANK A：(0x1***0000+0x24c00)-(0x1***0000+0x24fff)，共 1KB
- BANK B：(0x2***0000+0x25000)-(0x2***0000+0x257ff)，共 2KB
- BANK C：(0x3***0000+0x21a00)-(0x3***0000+0x21dff)，共 1KB

其中红色*号为上面驱动程序映像文件中说的驱动 ID，用来区分是哪个驱动的 BANK，

后面**号的高 6bit 表示 BANK 组内的 BANK 号。

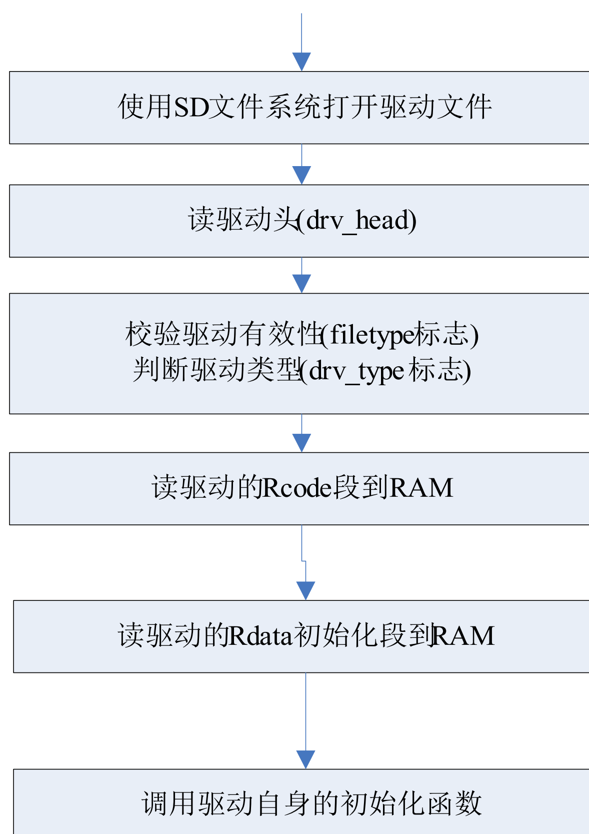
5.9.3 驱动程序的启动和退出

AP 调用驱动接口之前，必须先正确进行驱动的装载和卸载。

装载驱动接口：
int sys_drv_install(uint8 drv_type, uint8 work_mode, const char* drv_name);
卸载驱动接口：
int sys_drv_uninstall(uint8 drv_type);

例：装载 UI 驱动
sys_drv_install(DRV_GROUP_UI, 0, "drv_ui.drv");

OS 装载驱动的流程：



❖ 驱动装卸载函数的编写格式

为了符合系统装卸载驱动的要求，驱动的装卸载函数的定义有一定的要求，以 UI 驱动为例：

```
int ui_init(void *null0, void*null1, void*null2);
int ui_exit(void *null0, void *null1, void *null2);

module_init(ui_init)
module_exit(ui_exit)
```

驱动的初始化/退出函数是系统在安装/卸载驱动时主动隐式调用，不需要应用去调用。

5.9.4 驱动程序接口详解

US212A 驱动程序 API 机制和 9X 以前的方案有较大的区别。因为 UA212A 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，包括 ap bank 和 bank a/b/c 。所以，US212A 的驱动程序 API，不再需要考虑 bank 机制，bank 这个概念对 API 来说完全透明。

US212A 驱动程序 API 由 4 部分组成：

- 由 kernel 管理的驱动统一入口
- 驱动接口表
- 内部接口声明和定义
- 外部接口命令号及宏定义

下面以 UI 驱动为例进行说明。

5.9.4.1 驱动统一入口

UI 驱动统一接口定义如下：

void * ui_op_entry(void *param1,void *param2,void *param3, ui_cmd_e cmd);

- ui_op_entry ：包含在 api.a 中，4 个字节，需要打包到常驻代码段中。

li v1,API_UI_ENTRY

break 0 → 进入 API 调用流程

- 输入参数说明：

版 权 所 有 侵 权 必 究

- ✧ Kernel 提供的 ui_op_entry 包括 4 个参数，其中一个预留传递命令号，所以 ui 驱动只能使用 3 个有效输入参数。
- ✧ param1：输入参数 1，没有该参数时，可以传入 0 或 NULL。
- ✧ param2：输入参数 2，没有该参数时，可以传入 0 或 NULL。
- ✧ param3：输入参数 3，没有该参数时，可以传入 0 或 NULL。
- ✧ cmd：传递的驱动命令，取值为 ui_cmd_e 中的值。根据不同命令，调用不同的驱动接口函数。

- 输出参数说明：void 类型指针，即 32bit 返回值，具体定义由接口指定。

5.9.4.2 驱动接口表

UI 驱动接口表声明如下：

```
typedef void* (*ui_op_func)(void*, void*, void*);    → 只能使用 3 个有效输入参数
typedef struct
{
    ui_op_func res_open;
    ui_op_func res_close;
    ui_op_func show_picbox;
    ui_op_func show_textbox;
    ui_op_func show_timebox;
    ...
    ui_op_func show_listbox;
    ui_op_func show_dialogbox;
    ui_op_func show_parambox;
    ……
    ui_op_func scroll_string_ext;
    ui_op_func set_language;
    …
} ui_driver_operations;
```

UI 驱动接口表定义如下：

```
ui_driver_operations ui_driver_op =
{
    (ui_op_func) res_open,
    (ui_op_func) res_close,
    (ui_op_func) show_picbox,
    (ui_op_func) show_textbox,
```

```
        (ui_op_func) show_timebox,
        ...
        (ui_op_func) show_listbox,
        (ui_op_func) show_dialogbox,
        (ui_op_func) show_parambox,
        ...
        (ui_op_func) scroll_string_ext,
        (ui_op_func) set_language,
        …
};
```

UI 驱动接口表必须在 XN 文件中声明为 Drv_op_entry ，即：

ENTRY(ui_driver_op)

Kernel 在装载 UI 驱动会把 Drv_op_entry 即 ui_driver_op 装载到 kernel 管理的统一入口表中。

5.9.4.3 内部接口声明和定义

需要添加到驱动接口表，作为外部接口的内部接口必须声明为 typedef void* (*ui_op_func)(void*, void*, void*); 这样的形式，即必须包含 3 个输入参数，不足 3 个参数的接口，后面必须填充 null 参数。

```
void clear_screen(region_t *clrregion, void *null2, void *null3);
bool set_language(uint8 lang_id, void *null2, void *null3);
ui_result_e show_listbox(style_infor_t *listbox_style, listbox_private_t *listbox_data,
list_draw_mode_e mode);
```

5.9.4.4 外部接口命令号和宏定义

注意：命令号和宏定义必须与接口表中的接口一一对应。

UI 驱动接口命令号定义如下：

```
typedef enum
{
    UI_RES_OPEN = 0,
    UI_RES_CLOSE,
    UI_SHOW_PICBOX,
    UI_SHOW_TEXTBOX,
    UI_SHOW_TIMEBOX,
```

版 权 所 有 侵 权 必 究

```
        …
        UI_SHOW_LISTBOX,
        UI_SHOW_DIALOG,
        UI_SHOW_PARAMBOX,
        …
        UI_SCROLLSTRING_EXT,
        UI_SET_LANGUAGE,
        …
} ui_cmd_e;
```

UI 驱动接口宏定义如下：

```
#define ui_res_open(filename,type)
ui_op_entry((void*)(filename), (void*)(type), (void*)(0), UI_RES_OPEN)
#define ui_res_close(type)
ui_op_entry((void*)(type), (void*)(0), (void*)(0), UI_RES_CLOSE)
#define ui_show_picbox(style,data)
ui_op_entry((void*)(style), (void*)(data), (void*)(0), UI_SHOW_PICBOX)
#define ui_show_textbox(style,data,mode)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_TEXTBOX)
 #define ui_show_timebox(style,data,mode)
 ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_TIMEBOX)
 …
 #define ui_show_listbox(style,data,mode)
 ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_LISTBOX)
 #define ui_show_dialogbox(style,data,mode)
 ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_DIALOG)
 #define ui_show_parambox(style,data,mode)
 ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_PARAMBOX)
 …
 #define ui_scroll_string_ext(infor,param,style_infor)
 ui_op_entry((void*)(infor), (void*)(param), (void*)(style_infor),
 UI_SCROLLSTRING_EXT)
 #define ui_set_language(lang_id)
 ui_op_entry((void*)(uint32)(lang_id), (void*)(0), (void*)(0), UI_SET_LANGUAGE)
 …
```

5.9.4.5 外部接口调用流程

版 权 所 有 侵 权 必 究

5.9.4.6 修改驱动外部接口

驱动内部增加或删除某个作为外部接口的函数后，还需要做以下 2 件事情：

- 修改驱动接口表的声明和定义，必须确保驱动全局数据空间足够
- 修改外部接口命令号和宏定义，确保同驱动接口表中的接口一一对应

5.9.5 驱动 makefile 与 xn 脚本编写

5.9.5.1 驱动 makefile 脚本

什么是 makefile ?

make 命令执行时，需要一个 Makefile 文件，以告诉 make 命令需要怎样去编译和链接程序。Makefile 就是指定编译链接规则的脚本文件。

在应用和驱动开发时，系统会提供应用和驱动的 makefile 模板，指定应用和驱动编译链接时应遵守的共同基本规则。

驱动在编写自己模块的 makefile 时，只需要在系统提供的 makefile 模板上，修改对应模块的源文件路径信息、生成的目标文件名、目标文件的打包路径、链接脚本文件名等信息。

Makefile 模板及其修改

以 UI 驱动为例，makefile 需要修改的地方如下：

```
#Name of application        (modify-1)
IMAGENAME = drv_ui

#所要编译的源文件的存放位置 (modify-2)
SRC = $(CASE)/drv
SRCDIR_16 = $(SRC)/ui
SRCDIR_16_ROM = $(SRC)/ui/ui_drv_rom

#目标文件送往的文件夹路径 (modify-3)
OBJECT_BIN_PATH = $(CASE)/fwpkg/drv

#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
VPATH   =     $(SRCDIR_16) $(SRCDIR_16_ROM) $(SRCDIR_32) .
```

```
#指定自定义链接脚本的名称 (modify-4)
LD_SCRIPT = ui_driver.xn

#获得.c 后缀源码
SRC_C_16_ROM    = $(foreach dir,    $(SRCDIR_16_ROM), $(wildcard $(dir)/*.c))
#转换为.o 格式文件名称，不带路径信息
OBJ_C_16_ROM    = $(notdir $(patsubst %.c, %.o, $(SRC_C_16_ROM)))

#获得所有的.o 文件名称
OBJ = $(OBJ_C_16) $(OBJ_C_16_ROM) $(OBJ_S_16) $(OBJ_C_32) $(OBJ_S_32)

$(OBJ_C_16_ROM) : %.o : %.c
    $(CC) $(CC_OPTS_O2_16) -o$@ $<
    @echo
```

说明：

1. IMAGENAME 表示驱动名字，上面的例子驱动全名为：drv_ui.drv
2. 默认所有源文件都放在当前驱动目录下，采用 MIPS 16e 指令集、O0 优化编译，如果有特殊的去修，比如上面的例子，当前目录下有个 ui_drv_rom 的子目录，里面的源文件用 O2 优化编译，并且以固化代码方式打包为.bin 格式。所以需要添加额外的源文件路径、编译规则等。
3. LD_SCRIPT 指定链接脚本，指示如何链接驱动中的各个段，该脚本也是打包为*.drv 文件格式的脚本文件。

5.9.5.2 驱动 xn 脚本

什么是 linker 脚本文件 ？

描述如何把输入文件中的节（sections）映射到输出文件中，并控制输出文件的存储布局的脚本文件。

在应用和驱动开发时，系统会根据 bank 分组，提供应用和驱动的各组 bank 段的 linker 模板，指定应用和驱动编译链接时应遵守的链接模板。

驱动在链接脚本模板基础上，根据驱动代码和数据空间的分配情况，编写各自模板的 linker 脚本文件。

以 UI 驱动为例，XN 文件的格式如下：

```
定义 GROUP BANK INDEX 和其他地址变量
INPUT(link_base.xn)
INPUT(sdk_link_base.xn)
```

获取 UI 驱动模块的 group index，驱动开发人员需选择
BANK_GROUP_INDEX = DRV_GROUP_UI；

从地址 map 图中获取 UI 模块的常驻代码段和常驻数据段物理地址
SRAM_TEXT_ADDR = SRAM_UI_RCODE_ADDR；
SRAM_DATA_ADDR = SRAM_UI_DATA_ADDR；

转换为链接地址,不能修改
RCODE_TEXT_ADDR = RCODE_ADDR_BASE + SRAM_TEXT_ADDR；
ROM_TEXT_ADDR = RCODE_ADDR_BASE + SROM_UI_ADDR；
RDATA_DATA_ADDR = RDATA_ADDR_BASE + SRAM_DATA_ADDR；

KERNEL_BANK_A_ADDR_BASE = (KERNEL_BANK_A << 28) + (BANK_GROUP_INDEX << 24) + SRAM_BANK_A_ADDR；
KERNEL_BANK_B_ADDR_BASE = (KERNEL_BANK_B << 28) + (BANK_GROUP_INDEX << 24) + SRAM_BANK_B_ADDR；
KERNEL_BANK_C_ADDR_BASE = (KERNEL_BANK_C << 28) + (BANK_GROUP_INDEX << 24) + SRAM_BANK_C_ADDR；

bank 的实际空间大小,不能修改
KERNEL_BANK_A_SIZE = SRAM_BANK_A_SIZE；
KERNEL_BANK_B_SIZE = SRAM_BANK_B_SIZE；
KERNEL_BANK_C_SIZE = SRAM_BANK_C_SIZE；
系统允许的 bank 的最大空间大小，不能修改
KERNEL_BANK_SPACE = BANK_SPACE；
OUTPUT_ARCH(mips)

驱动接口表声明
ENTRY(ui_driver_op)

输出段定义
```
SECTIONS
{
    .text RCODE_TEXT_ADDR   :
    {
        api.a(.text .rodata)
        rcode_ui_op_entry.o(.text .rodata)
        rcode_ui_functions.o(.text .rodata)
```

```
}
.data RDATA_DATA_ADDR    :  AT(ADDR(.text) + SIZEOF(.text))
{
        rcode_ui_op_entry.o(.data)
}
.bss :
{
        *(.bss)
        *(.sbss)
        *(.common)
        *(common)
 }

Bank a 代码段   → 确保 bank a 的输出段名字以 BANK_A 开头
. = KERNEL_BANK_A_ADDR_BASE;
BANK_A_0:
{
        *.o(.text .rodata)
        . = KERNEL_BANK_A_SIZE;
}
/*假设还有一个 bank a，文件名为 bank_a_ui_xx.c*/
. += (KERNEL_BANK_SPACE - KERNEL_BANK_A_SIZE); →确保地址低 18bit 全 0
BANK_A_1    :
{
        bank_a_ui_xx.o(.text .rodata)
        . = KERNEL_BANK_A_SIZE;
 }

Bank b 代码段   → 确保 bank b 的输出段名字以 BANK_B 开头
. = KERNEL_BANK_B_ADDR_BASE;
BANK_B_0     :
{
       *.o(. text .rodata)
       . = KERNEL_BANK_B_SIZE;
}

Bank c 代码段   → 确保 bank c 的输出段名字以 BANK_C 开头
. = KERNEL_BANK_C_ADDR_BASE;
BANK_C_0     :
```

```
        {
            *.o(. text .rodata)
            . = KERNEL_BANK_C_SIZE;
        }
    }
```

5.9.6 如何增加一个驱动程序

因为增加一个新的驱动需要额外的常驻数据空间和常驻代码空间，并且一个空闲的驱动 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，而是建议把驱动功能合并到已有的 case 驱动中，比如把红外遥控器驱动功能合并到 KEY 驱动中。

5. 10 内存空间分配调整指南

5.10.1 Case 内存分配图

以下是 US212A 方案的内存分配情况图，图中只是详细指出 ap_manager 、前台应用 、后台应用 、 KEY 驱动 、 LCD 驱动 、 UI 驱动等模块的内存分配，其他一律由系统分配，Case 不能干涉。

颜色	含义	颜色	含义
紫色	Ap_manager	黄色	KEY驱动
蓝色	前台应用	青色	LCD驱动
深红色	后台应用	绿色	UI驱动

地址区间	说明
[0x20600 -- 0x20eff] KEY驱动常驻代码段(2.25K)	
[0x1fe00 -- 0x205ff] 前台应用CONTROL BANK段 (2K)	
[0x1f600 -- 0x1fdff] 前台应用UI BANK段 (2K)	
[0x1ee00 -- 0x1f5ff] 前台应用常驻代码区(2K)	
[0x1e800 -- 0x1edff] 后台应用常驻代码区 (1.5K)	Record 应用该空间被系统占用 (3.5K)
[0x1e000 -- 0x1e7ff] 后台应用CONTRL BANK段 (2K)	
[0x1da00 -- 0x1dfff] 后台应用数据(AP+COMMON+ENHANCED) 1.5K	record 和 video 应用该空间被系统占用 1.5K
[0x1d200--- 0x1d9ff] 前台应用数据区(AP+COMMON+ENHANCED) (2K)	
[0x1ce00 -- 0x1d1ff] UI驱动数据区 (1K)	
[0x1b000 -- 0x1cdff] 系统占用	
[0x1ae00 -- 0x1afff] manager BANK段 (0.5K)	
[0x1ad00 - 0x1adff] manager常驻代码&数据段 0.25k	
[0x1ab80 - 0x1acff] KEY驱动数据区(0.375 K)	
[0x1a000 - 0x1ab7f] 系统占用	
[0x19f80 - 0x19fff] 应用全局数据区(0.125K)	
[0x19f00 - 0x19f7f] 系统占用	
[0x19e80 - 0x19eff] LCD驱动数据区 (0.125 K)	
[0x18400 - 0x19e7f] 系统占用	
[0x18000 - 0x183ff] UI驱动缓冲区 (1 K)	
[0x00000 - 0x15fff] 系统占用	

地址区间	说明
[0x34000 －∞] 系统占用	
[0x32000 -- 0x33fff] UI驱动缓冲区(8k), record编码、U盘数据传输、图片解码时被系统占用	
[0x2b000 － 0x31fff] 系统占用	
[0x2a000 -- 0x2afff] 后台应用ENHANCED2 BANK段(4k)	音频解码时被系统占用
[0x29000 -- 0x29fff] 系统占用	
[0x28800 -- 0x28fff] 后台应用ENHANCED1 BANK段(2k)	音频解码时被系统占用
[0x28000 -- 0x287ff] 系统占用	
[0x27800 -- 0x27fff] 前台应用ENHANCED2 BANK段(2k)	图片解码时被系统占用
[0x27000 -- 0x277ff] 前台应用ENHANCE1 BANK段(2k)	
[0x26eb0 -- 0x26fff] 系统占用	
[0x26ac0 -- 0x26eaf] 前台应用栈空间(1K)	
[0x265c0 -- 0x26abf] 前台子线程栈空间(1.25K)	
[0x26000 -- 0x265bf] 后台应用栈空间(1.5K)	
[0x25b00 -- 0x25fff] 后台子线程栈空间(1.25k)	
[0x25800 -- 0x25aff] ap_manager栈空间 (0.75k)	
[0x25000 -- 0x257ff] 驱动程序BANK B (2K)	
[0x24c00 -- 0x24fff] 驱动程序BANK A (1K)	
[0x21e00 -- 0x24bff] 系统占用	
[0x21a00 -- 0x21dff] 驱动程序BANK C (1k),建议仅由UI驱动使用,以加速UI显示速度	
[0x21880 -- 0x219ff] LCD驱动常驻代码段2 0.375K	
[0x21580 -- 0x2187f] UI驱动常驻代码段 (0.75K)	
[0x20f00 -- 0x2157f] LCD驱动常驻代码段1 1.625K	

说明：

- Enhanced 部分空间和系统复用的情况比较普遍，所以建议不要调整 enhanced 的段大小和边界。如果在某个场景下确认 enhanced 无需使用，可以直接拿来做其他用途，但是必须告诉系统对应的 enhanced 段已被占用，即清除对应 enhanced 段的 TLB 项，具体请咨询我们的工程师。

版 权 所 有 侵 权 必 究

5.10.2 分配说明

US212A 作为一个整体方案，对有限内存空间的分配是在以系统功能场景为分配依据，系统各个部分严谨配合下的一种优化分配方案，所以对 case 层来说，能够对内存空间分配进行调整的灵活度是比较低的，为了系统的整体稳定性，我们建议用户使用下一节所述调整方法进行调整，尽量避免做更大的调整，如果有需要请咨询我们的工程师。

在前面内容中我们已经说明了应用程序和驱动程序的内存空间分配情况，以及与内存空间分配相关的脚本 *.xn 和 makefile 的使用细节。这里我们在补充一些内存空间分配的一些内容。

- Makefile 中指定了连接器生成 *.map 文件，该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。
- 物理内存空间在空间上是互斥的，即相同时刻，同一个物理内存空间不能分配给不同模块；物理内存空间在同一个模块内也是互斥的；而在不同时间不同场景上，物理空间是可以复用的。
- 常驻代码段和常驻数据段在应用程序或驱动程序加载时搬到对应的物理内存中，目前应用程序和驱动程序的常驻代码段和常驻数据段都只能有一个，这就可能带来了一个问题，如果应用程序或驱动程序可用的常驻代码段或常驻数据段空间分散在多个不连续物理内存空间上，就无法在加载时把所有常驻代码段或常驻数据段都搬到物理内存空间了。解决该问题的方法比较复杂，具体实现请咨询我们的工程师。

5.10.3 调整指南

本文档的调整指南，只限于在应用程序和驱动程序内部，以及使用空闲内存空间进行调整，其他方面的内存空间调整请咨询我们的工程师。

内存空间调整方法主要是在相邻内存块之间腾挪、寻找其他可用空间等等。内存空间的调整需要通过修改 *.xn 文件来实现，主要是修改内存块的起始地址和大小，比如要调整前台应用的 CONTROL BANK 段，从 UI BANK 段腾挪出 0x100 B 过来，可以这样修改：

BANK_CONTROL_1_ADDR_BASE = (AP_BANK_FRONT_CONTROL_1 << 24) +

SRAM_AP_BANK_FRONT_CONTROL_ADDR; → 起始地址不用动，大小加 0x100 即可

BANK_UI_1_ADDR_BASE = (AP_BANK_FRONT_UI_1 << 24) + SRAM_AP_BANK_FRONT_UI_ADDR + 0x100; → 起始地址往后挪 0x100

BANK_CONTROL_SIZE = SRAM_AP_BANK_FRONT_CONTROL_SIZE + 0x100; → 大小加 0x100

BANK_UI_SIZE = SRAM_AP_BANK_FRONT_UI_SIZE – 0x100; → 大小减 0x100

注意修改时要修改到位，比如修改前台应用，可能需要同步修改到 3 个 *.xn 文件，包括 common 和 enhanced 这两个 xn 文件，并且只是修改单个应用程序，需要把共用的 common 和 enhanced 这两个 xn 文件拷贝到当前应用目录下，修改 makefile 的 xn 文件列表，指向这两个 xn 文件。

另外，调整内存空间时还应该注意以下几点：

- 系统在同一时刻只会存在并运行一个前台应用和后台应用，所以应用程序的内存空间调整只需要处理好自己的空间分配，而无须关心其他应用。但是应用程序如果涉及 common 的调整会稍微麻烦一些，因为 common 模块的代码是为所有应用程序共用的，对其代码进行调整需要谨慎顾及其他所有应用。所以也可以对 common 进行个性化调整，即拷贝一份代码和 xn 脚本，独立为该应用程序进行个性化调整。
- 系统中会同时存在并运行多个驱动程序，所以驱动程序的 BANK A、BANK B 的修改会影响到多个驱动程序，不能调整。而对于 BANK C，因为是 UI 驱动独占的，所以在某些条件下可以调整。
- 寻找其他可用空间：这种方法是最安全的，但具体方案有哪些空闲空间，则视乎客户最终案子的具体情况而定。
- 空间复用：这种方法往往会很有效果，但是需要对系统比较了解的情况下才能够安全使用，它需要清楚把握某种使用情景下系统所有内存分配情况，并且需要处理好该情景的切入和切出细节。这种方法比较复杂，具体实现请咨询我们的工程师。

6 case 驱动程序详解

Case 驱动程序包括 KEY 驱动、LCD 驱动、UI 驱动、Welcome 驱动、FM 驱动等，是 Case 相关的驱动，具体的 Case 的 KEY、LCD 模组、FM 模组有可能有所区别，并且即使使用相同的硬件，其软件功能规格也可能会有所差异，所以我们把这几个驱动程序放在 Case

中来设计和实现。

其中的 KEY 驱动、LCD 驱动、UI 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 Case 起到至关重要的作用。而 FM 驱动仅仅是收音机应用的底层模块，对整个 Case 来说并没有什么特别的作用。所以下面我们只介绍前面 3 个驱动，而把 FM 驱动留在应用程序那一章与收音机应用一起介绍。

另外，Welcome 驱动由于与 LCD 驱动关系比较密切，并且它也是 Case 中必不可少的组成部分，所以也会在下面一并介绍。

6. 1 KEY 驱动设计

我们一般支持 GPIO 按键、线控 LRADC 按键这 2 种按键类型，虽然硬件原理有所差别，但是按键消息处理基本上是一致的。

另外，我们在 KEY 驱动中增加了红外遥控 IR 按键，IR 按键额外的红外接收头支持，红外接收头接收到的信号通过 IC 中 IR 模块的分析处理，直接得出物理按键类型。

6.1.1 需求概述及设计原则

KEY 驱动主要提供 KEY 扫描并发送按键消息，另外，电池充电、卡检测、耳机检测等独立于应用程序的功能模块也在 KEY 驱动中实现。

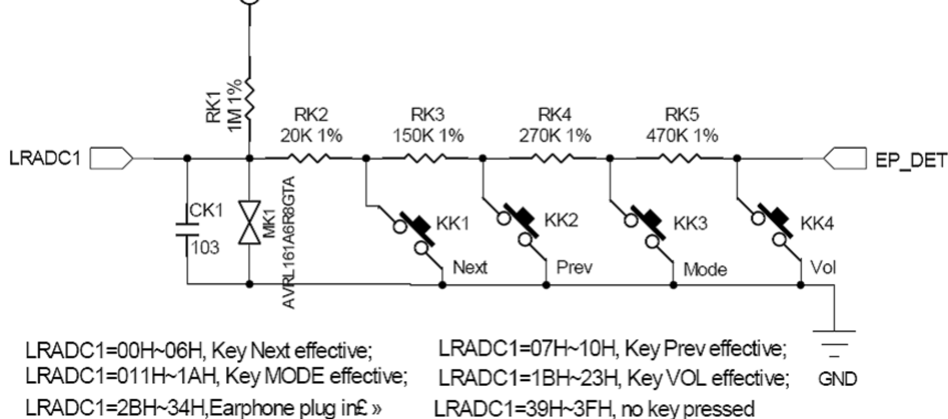
很多时候，我们可以把 KEY 驱动看成是一个简单的驱动架构，挂载了一个周期为 20ms 的硬件定时器中断。所以 KEY 驱动非常适合完成一些定时扫描的任务。

作为一个驱动程序，可以使用 BANK A、BANK B，所以任何一些独立于应用程序的功能接口都可以封装为 KEY 驱动的 BANK A、BANK B 接口。

6.1.2 KEYBOARD 原理

6.1.2.1 ADC 按键的原理

ADC 按键，其原理是，当按键按下的时候，不同的按键按下，从 AVCC 到 GND 的电



.RADC1 的

应的 ADC

1. **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：

$$\frac{V_{lradc1}}{AVCC}\times 2^7$$

2. **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。

请注意：居于以上原理，通过 ADC 按键，是无法实现两个按键同时按下这样的功能的。

6.1.2.2 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：

请注意：居于以上原理，可以通过 GPIO 按键，去实现多个按键同时按下的检测。

6.1.2.3 IR 按键的设计原理

在 Ucos 系统环境下，主要实现按键设置的初始化、键值的获取保存等功能，为上层的应用提供需要的信息。

6.1.3 KEY 驱动功能模块

KEY 驱动功能模块如下所示：

函数模块划分	模块简述
key_rcode_scan.c key_rcode_scan_02.c	按键扫描及消息发送、耳机检测
key_rcode_charge.c key_banka0_charge.c	充电控制模块

版 权 所 有 侵 权 必 究

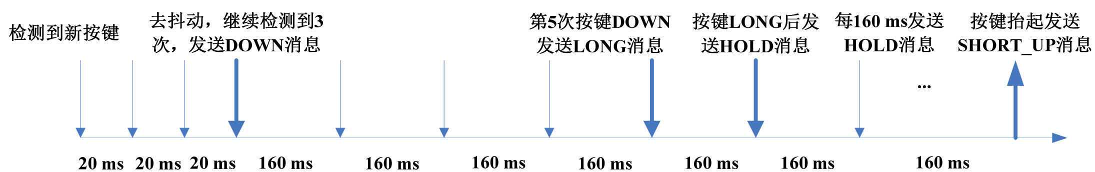
key_bankb0_init.c	按键驱动初始化
key_bankb1_exit.c	按键驱动卸载
key_banka1_state.c	获取按键映射表地址和 Hold 状态处理
key_rcode_op_entry.c	应用 API 接口函数处理

接下来将详细介绍各个功能模块：

6.1.3.1 按键扫描及消息发送

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：



按键消息的处理通过 PutSysMsg 接口实现，该接口会按照上述的按键生命周期处理按键消息。按键消息发送通过 post_key_msg 接口实现，按键消息在发送时会被压缩为 4 字节的字节流以节省宝贵的系统数据空间。这样就需要对应在 get_gui_msg 中对按键消息进行解压缩。压缩与解压缩的格式如下：

Bit 0	**Bit 8**	**Bit 16** ... **Bit 31**
input_msg_type_e 1B	**key_value_e 1B**	**key_type_e 2B**

注意：play 按键是一个特殊的按键，独立于 LRADC 处理。所以 play 按键不能同其他按键交换，它固定在按键映射表的第 1 个。

6.1.3.2 充电控制

开始充电

调用 key_chargeset(CHARGE_START, ChargeCurrent250mA, BATFULL_LEVEL1); 接口开始充电，充电时要配置充电电流和满电电压参考值。

版 权 所 有 侵 权 必 究

停止充电

一般用户无需自己停止充电，系统会在检测到充电已满后自动停止充电。如果用户需要自己停止充电，调用 key_chargeset(CHARGE_STOP, 0, 0); 接口停止充电。

充电流程

开始充电时，会先用小电流充电几十 ms，然后才开始用指定的充电电流充电。充电过程中，每隔 1 分钟进行一次充电自检，如果检测到充电已满，则停止充电。

快速充电

为了能让用户充电 5 分钟就可以播放音乐 2 小时，我们引进了快速充电技术。这种快速充电技术就是在检测到开始充电时电压参考值偏低时，我们临时把充电电流提升到 350ma 充电 5 分钟，5 分钟后将充电电流设置为用户指定的充电电流。

充电检测

按键驱动并不会把充电状态主动上报给应用程序，所以需要应用程序周期调用 key_chargeget(0) 接口获取当前充电状态，以及直接读取 BATADC_DATA 寄存器获取当前电池电压参考值，并根据这些状态和电池电压参考值判断低电或充电已满。

6.1.3.3 KEY 驱动初始化及卸载

KEY 驱动初始化完成的事情包括：

- 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。

6.1.3.4 获取按键映射表地址

我们的方案支持用户自配置按键，也就是通过配置按键映射表实现的。

我们的实现方法是，应用程序通过 key_getkeytabaddress() 接口获取 KEY 驱动中的按键映射表 key_map 的地址，然后由应用程序将用户自定义的按键映射表拷贝到 key_map 中。

版 权 所 有 侵 权 必 究

6.1.4 KEY 驱动的外部接口设计

KEY 驱动的主要功能是捕捉用户的按键输入，但是除此之外，我们通常还会把其他一些硬件相关的功能放在 KEY 驱动中，比如充电、卡检测、USB 检测、耳机检测、等等，其中有些需要挂载在 20ms 扫描按键函数中，有些则通过外部接口给 AP 调用。

这里只是简单介绍下 KEY 驱动模块外部接口设计要点，对于具体接口如何使用，请参考《us212a_key_driver 接口说明书.chm》。

注意：《us212a_key_driver 接口说明书.chm》中的接口是 key 驱动内部接口，接口命名和应用程序中使用的宏名字是不一样的。请查找 key_rcode_op_entry.c 中的 blk_op 接口表对应命令 ID 号的接口。

KEY 驱动的统一接口定义如下：

```
void * key_op_entry(void *param1,void *param2,void *param3, key_cmd_e cmd);
```

KEY 驱动外部接口命令号定义如下：

```
typedef enum
{
    KEY_CHARGEGET = 0,
    KEY_CHARGESET ,
    KEY_KEYTABADDR,
    KEY_HOLDSTATE,
    KEY_SPEAKCHECK,
} key_cmd_e;
```

KEY 驱动外部接口宏定义如下：

```
#define key_chargeget(a)
key_op_entry((void*)(a),0,0,KEY_CHARGEGET)
#define key_chargeset(a,b,c)
key_op_entry((void*)(a),(void*)(b),(void*)(c),KEY_CHARGESET)
#define key_getkeytabaddress()
key_op_entry((void*)0,(void*)0,(void*)0,KEY_KEYTABADDR)
#define key_holdcheck()
key_op_entry((void*)0,(void*)0,(void*)0,KEY_HOLDSTATE)
```

```
#define key_speakcheck(a)
key_op_entry((void*)(a),0,0,KEY_SPEAKCHECK)
```

6.1.5 KEY 驱动的内存分配说明

KEY驱动模块的代码空间分配，具体分配如下：

- 常驻代码空间包括：0xbfc20600 ~ 0xbfc20eff = 0x900 字节。
- bank a/b 代码空间：
  - bank a：(0x19**0000+0x24c00) ~ (0x19**0000+0x24fff) = 0x400 字节。
  - bank b：(0x29**0000+0x25000) ~ (0x29**0000+0x257ff) = 0x800 字节。
  - 说明：**高 6bit 为 bank 号，比如 0x19064c00 中 0x06 高 6bit 为 0x01，故为 KEY 驱动的 BANK A 组 BANK A1。

KEY驱动模块的常驻数据空间为 0x9fc1ab80-0x9fc1acff = 0x180字节。

6.1.6 KEY 驱动修改指南

6.1.6.1 修改物理按键及按键映射表

这里仅说明 LRADC 按键类型，GPIO 按键类型也是大同小异。

在物理上增加了一个按键后，还需要在软件上做以下 2 件事情：

- 修改 ADC_KEY_NUM 和 ADC 参考值列表 Adc_data，这样就可以多区分一个物理按键。
- 修改按键映射表 key_map ，在新增加的物理按键对应位置上增加新的逻辑按键值，当然，如果整个逻辑按键值要重新调整分配的话，那么几乎整个按键映射表都需要更新。

<gui_msg.h>

在 key_value_e 枚举类型中添加逻辑按键值。

<config.txt>

修改 CFG_KEY_ARRAY 配置项，按照设计的按键映射表，把 key_value_e 中的逻辑按键值逐一对应填入配置项值中。

如果仅仅要修改按键映射，则只需要修改 <config.txt> 配置文件即可。

6.1.6.2 修改按键生命周期

通过配置下面的几个宏即可修改：

```
#define DOWN_KEY_TIMER    3//按键按下消息时间为 60ms
#define LONG_KEY_TIMER    5//按键长按消息时间为 60ms +
                            (5-1)*HOLD_KEY_TIMER = 700ms
#define HOLD_KEY_TIMER    8//按键连续 hold 消息间隔为 160ms
```

6.2 LCD 驱动设计

US212A 方案体系中，LCD 驱动仅仅是显示系统的物理驱动，而显示系统的逻辑驱动则由 UI 驱动实现。这样做的目的是为了更明确把显示系统的物理和逻辑分开处理，让各自更加独立，更好维护。

6.2.1 需求概述及设计原则

LCD 驱动主要提供 LCD 物理层的接口函数，包含屏的硬件初始化、设窗、设置显示模式、设置对比度、屏的打开与 standby、屏的更新、屏的背光控制、写屏、读屏、反白等接口给 UI 驱动、AP 及图片视频解码的 CODEC 调用。

LCD 驱动在设计时，主要考虑以下几点：

- 对 LCD 物理硬件层的操作进行封装，使用户尽量不需要关心 LCD 内部的细节，而直接调用 UI 驱动层的接口函数，即能实现显示相关的各种功能。
- 将 LCD 物理层与 LCD 逻辑层分离，便于二次开发。做到不需要修改 UI 驱动，只需要修改 LCD 物理驱动的个别文件，即可实现 LCD 屏的替换。
- 将 Welcome 驱动的 LCD 物理层接口合并到 LCD 驱动中，两者使用同一套代码，通过 __WELCOME__ 宏区别编译，这样更方便 LCD 屏的开发和维护。

而对于 LCD 驱动接口的代码分布，则需要考虑以下几点：

版权所有 侵权必究

- 根据函数调用关系及频度，对代码空间进行合理的 bank 分组，尽可能地减少 bank 切换次数。
- 由于 LCD 驱动是物理驱动，其中的显示相关接口对每个显示任务来说几乎都可能会调用到，所以，为了最大限度的提升显示效率，要把所有这些接口都放到常住代码段中。这样设计，UI 驱动才能独占 bank c，才能更自由的划分代码段，LCD 驱动和 UI 驱动的配合，才能显现出最大的 UI 显示效率。
- 一些调用频度很低的接口，比如屏幕亮屏和黑屏，屏幕 standby，屏幕对比度设置等接口，可以放在 bank a/b。

6.2.2 LCD 驱动功能模块

LCD 驱动按照功能模块进行划分，尽量减少各模块之间的耦。本着最小改动的原则，对各个模块进行划分，便于更换 LCD 后的二次开发。驱动所包含的 C 文件说明如下表：

函数模块划分	模块简述
bank_a_lcd_functions.c	LCD 模组功能，包括背光控制、对比度设置、standby
bank_a_lcd_init.c	LCD 驱动的装载、卸载
lcd_hardware_init.c	LCD 硬件初始化
rcode_lcd_functions.c	外扩总线方式写数据、写命令、读屏，CE 的切换恢复，主控 IC 的 LCD 控制器初始化
rcode_lcd_functions_1.c	LCD 相关 MFP、CLK 切换控制
rcode_lcd_functions_2.c	LCD 的 DMA 传输控制
rcode_lcd_functions_3.c	LCD 基本功能，包括设置显示窗口、设置刷屏模式等
rcode_lcd_op_entry.c	LCD 接口函数表，总入口定义

接下来将详细介绍各个功能模块：

6.2.2.1 LCD 模组功能

LCD 模组功能包括背光控制、对比度设置、standby：

背光控制

void backlight_on_off(bool blctl, void *null2, void *null3)

背光由一个带 PWM 功能的 GPIO 口控制。打开背光就是配置好 PWM 模块，然后将该

GPIO 口设置为 PWM 模式；关掉背光就是关掉 PWM 模块，然后将该 GPIO 口设置为 GPIO_OUT 模式，并输出 off 电平。注：off 电平是高电平还是低电平取决于硬件电路设计。

对比度设置

void set_contrast(uint8 contrast_value, void *null2, void *null3)

设置对比度就是设置 PWM 的占空比。注：占空比与对比度的关系取决于硬件电路设计。

Standby

void standby_screen(bool standby_flag, void *null2, void *null3)

LCD 模组进入 standby 的方式由 LCD 模组 IC 决定，这里不对这种方式进行说明。LCD 模组 IC 进入 standby 前，需要先将对比度设置为 0，然后关掉背光；LCD 模组从 standby 退出后，需要打开背光，然后恢复对比度。

6.2.2.2 LCD 硬件初始化

void lcd_hardware_init(void)

LCD 硬件初始化，包括设置 LCD clock、reset LCD 模组 IC、发送初始化命令系列、清屏。这样就为显示图片和字符串做好准备。

注：该接口虽然放在 LCD 驱动中定义，但是一般会在 Welcome 中复用并调用，放在 LCD 驱动中只是为了管理和维护更加方便。

6.2.2.3 LCD 基本硬件功能

LCD 基本硬件功能包括写数据、写命令、读屏、外扩总线方式送屏、DMA 方式送屏及相关 DMA 接口、设置显示窗口、设置刷屏模式、以及一些主控 IC 的硬件配置接口，比如主控 IC 的 LCD 控制器初始化、MFP 和 CLK 切换控制等。

由于主控 IC 的 LCD 控制器接管了很多与 LCD 模组通信的细节，所以只要做好主控 IC 的硬件配置接口，其他功能的实现都是很简单的。

设置显示窗口和设置刷屏模式是显示系统的两个逻辑接口，我们定义显示系统的几种应用场景，LCD 驱动需要根据应用场景的需求实现这两个接口，下面分别说明这几种应用场景：

- DRAW_MODE_H_DEF：水平方向优先，即竖屏左上到右下，用于竖屏显示字符串和资源图片，清屏，welcome 图片，JPEG，GIF

版 权 所 有 侵 权 必 究

- DRAW_MODE_V_DEF：垂直方向优先，即横屏左上到右下，用于横屏显示字符串和资源图片，清屏，JPEG，GIF，AVI
- DRAW_MODE_H_PIC_DEF：水平方向优先，即竖屏左下到右上，用于竖屏显示BMP
- DRAW_MODE_V_PIC_DEF：垂直方向优先，即横屏左下到右上，用于横屏显示BMP，AMV
- DRAW_MODE_H_SCROLL_DEF：垂直方向优先，即竖屏左上到右下，用于竖屏显示字符串滚屏
- DRAW_MODE_V_SCROLL_DEF：水平方向优先，即横屏左上到右下，用于横屏显示字符串滚屏

注：这里说的横屏和竖屏是以样机为参照物的，样机正摆看为竖屏，样机旋转（可以是逆时钟，也可以是顺时钟，具体以样机设计为准）90 度看为横屏，是一个简单的逻辑概念，与 LCD 屏物理宽度和高度没有关系。

6.2.3 LCD 驱动的外部接口设计

这里只是简单介绍下 LCD 驱动模块外部接口设计要点，对于具体接口如何使用，请参考 《us212a_lcd_driver 接口说明书.chm》。

注意：《us212a_lcd_driver 接口说明书.chm》中的接口是 lcd 驱动内部接口，接口命名和应用程序中使用的宏名字是不一样的。但是一般宏名字是在内部接口名字的前面添加 lcd_ 前缀构成的，如果不是，请查找 rcode_lcd_op_entry.c 中的 lcd_driver_op 接口表对应命令 ID 号的接口。

LCD 驱动的统一接口定义如下：

```
void * lcd_op_entry(void *param1,void *param2,void *param3, lcd_cmd_e cmd);
```

LCD 驱动外部接口命令号定义如下：

```
typedef enum
{
    /*!设窗*/
   LCD_SET_WINDOW= 0,
    /*!设置对比度*/
   LCD_SET_CONTRAST,
```

```
    /*!设置显示模式*/
    LCD_SET_DRAWMODE,
    /*!屏幕 standby 控制*/
    LCD_STANDBY_SCREEN,
    /*!开关背光控制*/
    LCD_BACKLIGHT_ONOFF,
    /*!更新指定的屏幕区域*/
    LCD_UPDATE_SCREEN,
    /*!将 buffer 中的数据送屏*/
    LCD_TRANS_BUFFDATA,
    /*!获取 LCD 屏上的数据*/
    LCD_GET_BUFFDATA,
    /*!将显示 BUFFER 中的数据反色*/
    LCD_INVERT_BUFFDATA,
    /*!LCD  控制器初始化*/
    LCD_CONTROLLER_INIT,
    /*!设置 DMA 传输的数据宽度*/
    LCD_DMA_SET_COUNTER,
    /*!控制 DMA 开始传输*/
    LCD_DMA_START_TRANS,
    /*!设置 DMA 源地址*/
    LCD_DMA_SET_SRC_ADDR,
    /*!设置 RAM8 的 CLK 到 MCU*/
    LCD_SET_JRAM_CLK,
    /*!恢复原来的 CLK*/
    LCD_RESTORE_JRAM_CLK
}lcd_cmd_e;
```

LCD 驱动外部接口宏定义如下：

```
/*!设窗*/
#define lcd_set_window(rgn)
lcd_op_entry((void *)(rgn), (void *)(0), (void *)(0), LCD_SET_WINDOW)

/*!设置对比度*/
```

版 权 所 有 侵 权 必 究

```
#define lcd_set_contrast(value)
lcd_op_entry((void *)(uint32)(value), (void *)(0), (void *)(0), LCD_SET_CONTRAST)

/*!设置显示模式*/
#define lcd_set_draw_mode(mode)
lcd_op_entry((void *)(uint32)(mode), (void *)(0), (void *)(0), LCD_SET_DRAWMODE)

/*!屏幕 standby 控制*/
#define lcd_standby_screen(flag)
lcd_op_entry((void *)(uint32)(flag), (void *)(0), (void *)(0), LCD_STANDBY_SCREEN)

/*!开关背光控制*/
#define lcd_backlight_on_off(on_off)
lcd_op_entry((void *)(uint32)(on_off), (void *)(0), (void *)(0), LCD_BACKLIGHT_ONOFF)

/*!更新指定的屏幕区域*/
#define lcd_update_screen(rgn)
lcd_op_entry((void *)(rgn), (void *)(0), (void *)(0), LCD_UPDATE_SCREEN)

/*!将 buffer 中的数据送屏*/
#define lcd_buff_data_trans(buff,pix_cnt)
lcd_op_entry((void *)(buff), (void *)(uint32)(pix_cnt), (void *)(0),
LCD_TRANS_BUFFDATA)

/*!获取 LCD 屏上的数据*/
#define lcd_get_buff_data(buff,pix_cnt)
lcd_op_entry((void *)(buff), (void *)(uint32)(pix_cnt), (void *)(0), LCD_GET_BUFFDATA)

/*!将显示 BUFFER 中的数据反色*/
#define lcd_invert_buff_data_trans(buff,pix_cnt)
lcd_op_entry((void *)(buff), (void *)(uint32)(pix_cnt), (void *)(0),
LCD_INVERT_BUFFDATA)

/*!LCD 控制器初始化*/
```

```
#define lcd_controller_init(mode)
lcd_op_entry((void *)(uint32)(mode), (void *)(0), (void *)(0), LCD_CONTROLLER_INIT)

/*!设置 DMA 传输的数据宽度*/
#define lcd_dma_set_counter(pix_cnt,data_width)
lcd_op_entry((void *)(uint32)(pix_cnt), (void *)(uint32)(data_width), (void *)(0),
LCD_DMA_SET_COUNTER)

/*!控制 DMA 开始传输*/
#define lcd_dma_start_trans(dma_ram)
lcd_op_entry((void *)(uint32)(dma_ram), (void *)(0), (void *)(0),
LCD_DMA_START_TRANS)

/*!设置 DMA 源地址*/
#define lcd_dma_set_src_addr(addr0,addr1,addr2)
lcd_op_entry((void *)(uint32)(addr0), (void *)(uint32)(addr1), (void *)(uint32)(addr2),
LCD_DMA_SET_SRC_ADDR)

/*!设置 RAM8 的 CLK 到 MCU*/
#define lcd_mcu_set_JRAM_clk()
lcd_op_entry((void *)(0), (void *)(0), (void *)(0), LCD_SET_JRAM_CLK)

/*!恢复原来的 CLK*/
#define lcd_restore_JRAM_clk()
lcd_op_entry((void *)(0), (void *)(0), (void *)(0), LCD_RESTORE_JRAM_CLK)
```

6.2.4 LCD 驱动的内存分配说明

LCD 驱动模块的代码空间分配，具体分配如下：

- 常驻代码空间包括：0xbfc20f00 ~ 0xbfc2157f = 0x680 字节，0xbfc21880 ~ 0xbfc219ff = 0x180 字节，总共 0x800 字节，即 2K 字节。
- bank a/b 代码空间：
  - ✧ bank a：(0x17**0000+0x24c00) ~ (0x17**0000+0x24fff) = 0x400 字节。

✧ bank b：(0x27**0000+0x25000) ~ (0x27**0000+0x257ff) = 0x800 字节。

✧ 说明：**高 6bit 为 bank 号，比如 0x17064c00 中 0x06 高 6bit 为 0x01，故为 LCD 驱动的 BANK A 组 BANK A1。

LCD驱动模块的常驻数据空间为 0x9fc19e80 ~ 0x9fc19eff = 0x80字节。

6.2.5 LCD 驱动的修改指南

6.2.5.1 替换新的 LCD 屏

在当前的 LCD 驱动架构的基础上，如何更换一个新的 LCD 驱动 IC ？ 如何能有效的在修改最少代码的基础上，快速有效的使得新的 LCD 驱动 IC 正常工作 ？ 当前驱动的分层架构为这种应用提供了很大的方便。

更换 LCD 驱动 IC 只涉及物理层的变动，不需要修改 UI 驱动。

需要修改的文件有 lcd_hardware_init.c 、 rcode_lcd_functions.c 、 rcode_lcd_functionsf_3.c 三个文件。需根据所选 LCD 屏的SPEC，重新编写 LCD 屏的硬件初始化代码以及设窗、设模式、读屏等函数。

另外，不同 LCD 屏的各个命令的定义可能也不一样，因此，还需修改 lcd_driver.h 中LCD相关命令的定义。

6.2.5.2 LCD 相关 GPIO 修改

LCD 驱动用到的 GPIO 口有 2 个： reset（复位）和 backlight（背光）。

在 LCD 驱动中，已经将 GPIO 做成宏定义域的形式，如果要替换其中一个或则几个，只需要修改相关的宏定义即可。

相关的宏定义在 lcd_driver.h 中，如下：

```
#define LCMRST_GIO_EN_REG          GPIO_AOUTEN      //GPIO_A5 (Output)
#define LCMRST_GIO_DATA_REG        GPIO_ADAT
#define LCMRST_GIO_EN_BIT          (0x00000001<<5)
#define LCMRST_SET_BIT             (0x00000001<<5)
#define LCMRST_CLR_BIT             ~(0x00000001<<5)

#define LCMBL_GIO_EN_REG           GPIO_AOUTEN          //GPIO_A6 (Output)
#define LCMBL_GIO_DATA_REG         GPIO_ADAT
```

```
#define LCMBL_GIO_EN_BIT            (0x00000001<<6)
#define LCMBL_SET_BIT               (0x00000001<<6)
#define LCMBL_CLR_BIT               ~(0x00000001<<6)
```

例如：要将屏的背光 GPIO 由 GPIO_A6 替换为 GPIO_B1

则以上对背光控制 GPIO 的定义修改如下：

```
#define LCMBL_GIO_EN_REG            GPIO_BOUTEN         //GPIO_B1 (Output)
#define LCMBL_GIO_DATA_REG          GPIO_BDAT
#define LCMBL_GIO_EN_BIT            (0x00000001<<1)
#define LCMBL_SET_BIT               (0x00000001<<1)
#define LCMBL_CLR_BIT               ~(0x00000001<<1)
```

6.2.5.3 如何修改屏尺寸

LCD 驱动中显示字符或者显示图片都需要判断屏的尺寸，以免内容显示不完整。有如下的应用情况存在：128*160 的屏驱动，如何适应 128*64 的应用；128*160的屏驱动，如何适应160*128的应用？在相同的LCD驱动IC的情况下，如何根据模具不同修改屏的尺寸呢？

涉及屏尺寸的函数同样很多，对屏尺寸的操作同样也做成宏的形式，如果要修改应用尺寸，同样只需要修改相应的宏定义即可。

lcd_driver.h中对屏幕尺寸的定义如下：

```
#define DISPLAY_LENGTH          128
#define DISPLAY_HEIGHT          160

#define LCD_WIDTH 132      /*LCD  模组实际的分辨率的实际长度*/
#define LCD_HEIGHT 162     /*LCD 模组分辨率的实际宽度*/

#define LCD_WIDTH_OFFSET    2       /*132X162  的 LCD 居中显示时，相对于 LCD 模组长度的起始偏移*/

#define LCD_HEIGHT_OFFSET 1     /*132X162  的 LCD 居中显示时，相对于 LCD 模组宽度的起始偏移*/
```

例如： 将 128*160 应用为 160*128：

则需要修改相关的宏定义：

#define DISPLAY_LENGTH 160

#define DISPLAY_HEIGHT 128

#define LCD_WIDTH 162 /*LCD 模组实际的分辨率的实际长度*/

#define LCD_HEIGHT 132 /*LCD 模组分辨率的实际宽度*/

#define LCD_WIDTH_OFFSET 1 /*132X162 的 LCD 居中显示时，相对于 LCD 模组长度的起始偏移*/

#define LCD_HEIGHT_OFFSET 2 /*132X162 的 LCD 居中显示时，相对于 LCD 模组宽度的起始偏移*/

6.2.6 LCD 驱动的功能配置

需要配置小彩屏是否支持读屏功能，是否支持读屏 UI 驱动将编译不同代码。

<lcd_driver.h>

```
/*! 是否支持读屏功能 */
#define SUPPORT_READ_CMD
```

在某些特殊场景下，LCD/UI 驱动只分配了 1KB 的显示缓冲区，如果字库单个字模的显示缓冲区大于 1KB，并且在该特殊场景下需要显示该字体的字符串，那么就必须另外指定足够大的空间了。

特殊场景包括：udisk AP 建立起连接后处于忙状态时，record AP正在录音时，等等。

显示缓冲区大小计算公式（565小彩屏）：字模实际高度 * 字模实际宽度 * 2 B

所以，如果是 24*24 的实际字模，缓冲区大小为 24*24*2 = 1152 B，就超出了 1KB。

当缓冲区超出 1KB 时，必须修改以下2个宏，并且修改 UI 驱动的 set_display_buf 函数：

<lcd_driver.h>

```
#define LCD_BUF_ADDR 0x18000//如果字模显示缓冲区超出 1KB，请另外指定足够大的空间做缓冲区，并修改 set_display_buf 函数
#define LCD_BUF_LEN 0x400
```

6.3 UI 驱动设计

US212A 方案体系中，UI 驱动仅仅是显示系统的逻辑驱动，而显示系统的物理驱动则由 LCD 驱动实现。这样做的目的是为了更明确把显示系统的物理和逻辑分开处理，让各自更加独立，更好维护。

6.3.1 需求概述及设计原则

US212A 的 UI 驱动需要实现以下功能：

- 提供各种控件显示的接口，包括图片控件显示
- 提供资源打开、关闭的接口
- 提供设置屏幕模式的接口(横屏或竖屏)
- 提供字符串显示、滚屏接口
- 提供获取字符串坐标、长度等信息的接口
- 支持 unicode 字库
- 支持 unicode 和内码、utf8 转 unicode 的转换
- 支持 28 种语言的显示（包括 arabic，hebrew，Thai 三种特殊语言的处理）

UI 驱动的设计原则如下：

- 根据函数调用关系及频度，对代码空间进行合理的 bank 分组，尽可能地减少 bank 切换次数。
- 为了达到较好的控件刷新效果，需要在控件显示接口中添加一个显示模式的参数，用来区分第一次全部显示和根据需要局部刷新，以达到最佳效果。
- 根据 API 接口函数的功能来划分，每一个逻辑功能函数作为一个小模块，每个小模块对应一个源文件。
- 将 UI 驱动中部分调用频度较高，且纯逻辑性的代码进行固化，提高代码执行的效率。

6.3.2 显示系统的优化

显示系统的核心是 UI 驱动，还包括配套工具等。显示系统的优化包括功能增强、性能提升、及架构优化等。

版 权 所 有 侵 权 必 究

以前的方案中，UI 驱动的功能主要包括以下几点：

- 支持图片和字符串显示
- 图片：支持彩色 BMP 图片，一般是 24bit 深度和黑白图
- 字符串：支持 ANSI 和 小端 Unicode 编码，支持多种语言
- 支持 resource builder 工具打包
- 支持一些简单图形绘制接口，比如矩形区域填充、3D 边框绘制（绘制线段）、反色显示（黑白屏）

显然，这样的 UI 驱动有以下几个缺点：

- 功能过于简单，没有提供字符串滚屏，控件显示等接口
- 用户必须在代码中指定 UI 细节进行显示处理，修改产品 UI 设计是件麻烦事
- AP 需要关心字符串处理的某些细节，比如语言类型，如果是阿拉伯和希伯莱语，用户需要特别处理
- 透明显示处理得不好， UI 设计限制性较大，很难设计出绚丽的 UI

所以，在 US212A 方案中，我们优化了 UI 驱动，在以下几个方面进行了优化：

- 把字符串滚屏、控件显示收进 UI 驱动，完全把显示任务都扔给 UI 驱动，尽量让 AP 能做到控制流与显示流分开
- 引进更加强大的 PC 端辅助工具 UI Editor 工具，把以前需要在代码中指定的 UI 细节通过工具配置化实现；并且实现所见即所得，大大提高用户的 UI 设计和开发效率
- 消除 AP 中所有字符串处理细节，让用户只需要简单调用某几个字符串显示和处理的接口，即可轻松实现所有字符串相关功能
- 优化透明显示处理，最大限度减少 UI 设计的限制条件，让用户有条件设计出绚丽的 UI
- 考虑把 UI 驱动部分代码固化，以最大限度提升代码执行效率

US212A 显示系统优化包括以下几点：

- 设计时要考虑到与 LCD 物理层、应用层之间的交互
  - ✧ 为了更好的支持不同的 LCD 驱动，把之前的 UI 驱动分为物理的 LCD 驱动以及逻辑的新 UI 驱动， LCD 驱动所有常用接口都常驻，其他接口可以放在 bank a/b 中
  - ✧ 为了做到控制流与显示流分开，把控件的显示部分代码、滚屏的显示部分代码

版 权 所 有 侵 权 必 究

都收到 UI 驱动

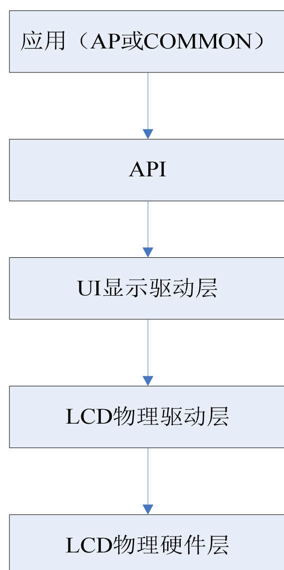
- 以"字符"为中心重新架构 UI 驱动的字符串显示处理，更好支持 UNICODE 码，ANSI 码，UTF-8 码
  - ✧ 集中在一个函数处理这三种编码
  - ✧ 泰语和阿拉伯语（希伯莱语）特殊处理集中化
  - ✧ 把任一字符串的显示看成：前面不完整字符 + 中间一段字符串 + 尾部不完整字符，这样 UI 架构将更加简洁
- 透明显示机制
  - ✧ 伪透明色图片，显示时与背景图片透明叠加
  - ✧ 要能够准确获取背景图进行叠加，如果背景图是横向的必须按列优先存储
  - ✧ 透明显示支持透明图标和透明字符串滚屏
- 设计时充分利用缓冲机制，可以减少对 SD 资源的读取，也可以更好使用 DMA 在更短时间更新 UI ，减少 UI 显示延时
  - ✧ 显示字符串时，缓冲一段字符串点阵及背景
  - ✧ 显示字符串时，缓冲最近访问的字符点阵，获取字符点阵时会先在缓冲区搜索，如果能搜索到就直接从缓冲区读取，否则就从 SD 区获取，并将获取到的字符点阵缓冲起来
  - ✧ 如果尚未剩余数据空间，还可以考虑缓冲内码对应的 UNICODE 码，获取内码对应的 UNICODE 码时会先在缓冲区搜索，如果能搜索到就直接从缓冲区读取，否则就从 SD 区读取，并将获取到的 UNICODE 码缓冲起来
- 接口固化设计原则
  - ✧ 调用频度较高接口基本功能接口，如果逻辑清晰稳定，即可考虑固化
  - ✧ 固化时多留一些 hook 接口，以便适应不同的业务需求，增强固化代码的可用性

6.3.3 UI 驱动功能模块

6.3.3.1 总体架构

UI 显示驱动向上与 AP、COMMON 有关，向下与 LCD 物理驱动有关，调用关系如下所示：

版 权 所 有 侵 权 必 究



6.3.3.2 功能模块

UI 驱动按照功能模块进行划分，尽量减少各模块之间的耦。本着最小改动的原则，对各个模块进行划分，便于客户进行二次开发。驱动所包含的 C 文件说明如下表：

函数模块划分	模块简述
bank_a_ui_res_open.c	打开 style 文件并初始化；关闭 style 文件
bank_b_ui_show_picbox.c	显示 PicBox 控件
bank_b_ui_show_textbox.c	显示 TextBox 控件
bank_a_ui_get_listbox_attrb.c bank_a_ui_get_picbox_attrb.c bank_a_ui_get_textbox_attrb.c	获取控件属性
bank_a_ui_load_attribute.c	获取 AttributeBox 属性值
bank_a_ui_show_listbox.c bank_a_ui_show_listbox_2.c bank_b_ui_show_listbox_1.c	显示 ListBox 控件
bank_a_ui_show_parambox.c bank_a_ui_show_parambox_2.c bank_b_ui_show_parambox_1.c	显示 ParamBox 控件
bank_a_ui_show_sliderbar.c bank_a_ui_show_sliderbar_3.c	显示 SliderBar 控件

版权所有 侵权必究

bank_b_ui_show_sliderbar_1.c bank_b_ui_show_sliderbar_2.c	
bank_b_ui_show_dialogbox.c	显示 DialogBox 控件
bank_b_ui_show_progressbar.c	显示 ProgressBar 控件
bank_a_ui_show_numbox_1.c bank_b_ui_show_numbox.c	显示 NumberBox 控件
bank_a_ui_show_timebox_3.c bank_b_ui_show_timebox.c bank_b_ui_show_timebox_1.c bank_b_ui_show_timebox_2.c	显示 TimeBox 控件
bank_c_ui_init.c	UI 驱动初始化；UI 驱动退出
bank_c_ui_set_language.c	设置语言类型，并重新初始化字符串处理环境
bank_c_ui_char_to_unicode.c bank_c_ui_unicode_to_char.c bank_c_ui_utf8_to_unicode.c	字符编码类型转换，包括 ANSI 转 UNICODE、UNICODE 转 ANSI、UTF-8 转 UNICODE
bank_a_ui_draw_3Drect.c bank_a_ui_draw_rect_frame.c bank_a_ui_fill_rect_dump.c	图形处理接口，包括区域填充、绘制 3D 边框等
rom_ui_image.c bank_c_ui_show_transparent.c bank_b_ui_show_picbox.c	显示图片功能函数，前者是普通图片显示以及区域纯色填充，后者是透明图片显示；bank_b_ui_show_picbox.c 中的 show_picbox_id3 接口用于 ID3 垂直滚屏中的图片垂直滚屏
bank_a_fix_ui_string.c bank_a_fix_ui_string_2.c bank_a_ui_putstring_sub_2.c bank_c_ui_putstring_sub.c rom_ui_string.c bank_b_ui_string_sdimage.c bank_b_ui_putstring.c bank_b_ui_scroll_string.c bank_b_ui_scroll_string_ext.c bank_c_ui_ellipsis_to_longstr.c bank_b_ui_get_text_line.c	显示字符串功能函数，包括普通字符串显示、透明字符串显示、字符串水平滚屏、字符串垂直滚屏、字符串添加省略号、字符串分行处理、及各种字符串处理子函数

版权所有 侵权必究

bank_a_ui_thai_point.c	泰语叠加处理
bank_b_ui_arabic_ligature.c bank_b_ui_arabic_nsm.c bank_b_ui_copy_arabic.c	阿拉伯和希伯来语特殊处理，包括阿拉伯连写、左右倒置处理等
rom_data_and_common.c bank_c_misc_functions.c bank_a_ui_itoa.c rcode_ui_functions.c	其他函数
rcode_ui_op_entry.c	UI 驱动外部接口表定义

接下来将详细介绍各个功能模块：

6.3.3.3 Style 文件打开与关闭

在 US212A 的显示系统架构中，Style 文件是非常重要的一个数据单元，它涉及到图片显示、资源字符串显示、各种控件显示参数等。所以 UI 显示处理中会频繁读取 style 文件中的数据。

每个前台应用在初始化阶段都需要打开 style 文件并初始化 style 文件读环境，在退出时关闭 style 文件。

打开 Style 文件

bool res_open(char *filename, res_type_e type, void *null3)

打开 style 文件，检查其合法性，然后读出资源图片索引表 offset、当前语言类型下资源字符串索引表 offset、控件显示参数区 offset。

关闭 Style 文件

void res_close(res_type_e type, void *null2, void *null3)

6.3.3.4 控件显示及获取属性

每个控件都提供一个显示接口，并且如果用户需要根据控件属性做一些控制处理时，控件还需要提供一个获取控件属性的接口。

控件显示接口形式大致为：

show_xxx(style_infor_t *, *_private_t *, uint8 mode);

其中 mode 并非必须的，如果控件只有一种显示模式，那么该参数可以去掉，比如图

片显示只有一种显示模式，无需 mode 参数。所以，PicBox 显示接口形式为：ui_show_picbox(style_infor_t *, picbox_private_t *);

UI 控件显示模块主要是根据 ui_editor 中编辑的各个控件的属性以及应用给每个控件指定的参数，解析 style 文件中相应控件对应的数据结构，调用逻辑功能实现模块的函数，来绘制和解析各个控件（各个控件的数据结构定义可详见 ui_driver 工程中的 ui_driver.h 头文件）。UI 控件显示模块包含的内容有：

6.3.3.5 设置语言类型

bool set_language(uint8 lang_id, void *null2, void *null3)

设置语言类型会重新读出当前语言类型下资源字符串索引表 offset、关闭先前语言类型 ANSI 转换为 UNICODE 的 codepage，打开当前语言类型 ANSI 转换为 UNICODE 的 codepage。

因为 US212A 的字库是 UNICODE 字库，所以在字符串显示过程中 ANSI 转换 UNICODE 的频率会比较高，所以在设置语言类型时预先打开了对应的 codepage 文件。

6.3.3.6 字符编码转换

US212A 支持 ANSI、UNCODE、UTF-8 三种编码类型，在不同的情境下需要使用不同的编码类型。

ANSI：一些对空间要求较高的情景下，使用 ANSI 存储字符串并进行处理。比如创建 playlist，因为支持的歌曲条目数量较大，US212A 为 4000 首，所以使用 ANSI 编码存储歌曲 ID3 key 信息并进行排序。

UNICODE：显示时必须以 UNICODE 编码来获取字符点阵并显示，可以在单个字符显示时进行转换；US212A 的文件名要求为 UNICODE 编码；等等。

ANSI 转 UNICODE

bool char_to_unicode(uint8 *dest, uint8 *src, uint16 len)

UNICODE 转 ANSI

bool unicode_to_char(uint8 *str, uint16 len, uint8 lang_id)

UTF-8 转 UNICODE

void utf8_to_unicode(uint8* src, uint16*dest, int16* size)

版权所有 侵权必究

6.3.3.7 图形处理

US212A 的显示系统主要以图片和字符串主，只有很少一些图形处理接口，包括区域填充、绘制区域边框、绘制 3D 边框等。

区域填充：分为纯色填充和单位图片填充

纯色填充，可以用来绘制线段，包括单像素线段

bool romf_fill_rect(region_t *fill_region, uint16 color)

单位图片填充，这种填充方式可以实现填充方向的垂直方向渐变色效果

bool fill_rect_dump(region_t *fill_region, uint16 pic_id, fill_dump_direction_e direction)

绘制区域边框

void draw_rect_frame(region_t * frame_region)

绘制 3D 边框，包括选中框效果、突出效果、凹进效果等

void draw_3D_rect(region_t * D3region, uint8 type)

6.3.3.8 图片显示

图片显示分为普通图片显示和特殊透明效果图片显示，后者可以用来实现数字图片等在大背景图上的任意位置透明显示、图片垂直滚动效果等。

普通图片显示

bool romf_show_pic(uint16 id, uint16 x, uint16 y)

透明效果图片显示，是 US212A 的显示系统优化的一个重要部分，用于解决数字显示、时间显示、进度条显示等不能以任意图片为背景的缺陷，也增加了图片垂直滚动效果的能力。

解决数字显示、时间显示、进度条显示等不能以任意图片为背景的缺陷

void show_pic_transparent(transparent_pic_t *p_trs_pic)

版 权 所 有 侵 权 必 究

```
/*!
 * \brief
 *  transparent_pic_t 描述透明图标/清除图标的数据结构
 */
typedef struct
{
    /*! 透明图标ID */
    uint16 pic_id;
    /*! 透明图标X坐标 */
    uint16 pic_x;
    /*! 透明图标Y坐标 */
    uint16 pic_y;
    /*! 透明图标背景颜色 */
    uint16 back_color;
    /*! （实时）背景图ID */
    uint16 back_id;
    /*! （实时）背景图X坐标，如果是横向图标，表示X起始值 */
    uint16 back_x;
    /*! （实时）背景图Y坐标，如果是竖向图标，表示Y起始值（最底部值） */
    uint16 back_y;
    /*! 是否仅仅清除痕迹 */
    bool clear;
    /*! 背景图方向，0为横向，1为竖向 */
    uint8 direct;
} transparent_pic_t;
```

算法要点：这是一种伪透明色图片显示，显示时与背景图片透明叠加，所以要求能够准确获取要先伪透明色图片区域的部分背景图进行叠加，也就要求如果背景图是横向的必须按列优先存储。伪透明色图片显示需要 UI Editor 工具的配合，下面分别对这 3 种控件进行说明。

NumberBox 和 TimeBox：要求所有的数字图片和分隔符必须设计为伪透明色图片（请见可配置化 UI 一节解释），并将背景图片按列优先存储起来，这只要使用 UI Editor 工具的“截取此控件的背景图”截取背景图即可。

ProgressBar：如果是横向 ProgressBar，要求将背景图片按列优先存储起来，这只要使用 UI Editor 工具的“截取此控件的背景图”截取背景图即可。

图片垂直滚动效果

ui_result_e show_picbox_id3(style_infor_t *picbox_style, picbox_private_t *picbox_data, uint16 start_row)

该接口专用于 ID3 垂直滚屏的图片垂直滚动，共 4 张 ID3 图标，其中 1 张是背景图，放在第 0 帧，设置为非嵌入式子背景图，其他 3 张为伪透明色图标，设置为普通图标。

6.3.3.9 字符串显示

版 权 所 有 侵 权 必 究

字符串显示作为 UI 驱动的重点和难点，我们将在下面单独用一节重点介绍。

6.3.4 UI 驱动的外部接口设计

这里只是简单介绍下 UI 驱动模块外部接口设计要点，对于具体接口如何使用，请参考《us212a_ui_driver 接口说明书.chm》。

注意：《us212a_ui_driver 接口说明书.chm》中的接口是 ui 驱动内部接口，接口命名和应用程序中使用的宏名字是不一样的。但是一般宏名字是在内部接口名字的前面添加 ui_ 前缀构成的，如果不是，请查找 rcode_ui_op_entry.c 中的 ui_driver_op 接口表对应命令 ID 号的接口。

UI 驱动的统一接口定义如下：

```
void * ui_op_entry(void *param1,void *param2,void *param3, ui_cmd_e cmd);
```

UI 驱动外部接口命令号定义如下：

```
typedef enum
{
    /*! 打开资源文件*/
    UI_RES_OPEN = 0,
    /*! 关闭资源文件*/
    UI_RES_CLOSE,
    /*! 显示 PicBox*/
    UI_SHOW_PICBOX,
    /*! 显示 TextBox*/
    UI_SHOW_TEXTBOX,
    /*! 显示 TimeBox*/
    UI_SHOW_TIMEBOX,
    /*! 显示 NumBox*/
    UI_SHOW_NUMBOX,
    /*! 显示 ProgressBar*/
    UI_SHOW_PROGRESSBAR,
    /*! load AttributeBox 的数据*/
    UI_LOAD_ATTRIBUTEBOX,
```

版权所有 侵权必究

```
/*! 显示 ListBox*/
UI_SHOW_LISTBOX,
/*! 显示 DialogBox*/
UI_SHOW_DIALOG,
/*! 显示 ParamBox*/
UI_SHOW_PARAMBOX,
/*! 获取 textbox 属性*/
UI_GET_TEXTBOX_ATTRB,
/*! 获取 picbox 属性*/
UI_GET_PICBOX_ATTRB,
/*! 获取 listbox 属性 */
UI_GET_LISTBOX_ATTRB,
/*! ID3 滚屏专用函数*/
UI_SCROLLSTRING_EXT,
/*! 设置语言种类*/
UI_SET_LANGUAGE,
/*! 获取 DC 状态*/
UI_GET_DC_STATUS,
/*! 清屏*/
UI_CLEARSCREEN,
/*! 设置画笔颜色*/
UI_SET_PEN_COLOR,
/*! 设置背景颜色*/
UI_SET_BACKGD_COLOR,
/*! 显示字符串*/
UI_SHOW_STRING,
/*! 获取字符串的长度*/
UI_GET_STRING_LENGTH,
/*! 文本解码断行，包括过滤掉空白字符和分词显示等 */
UI_GET_TEXT_LINE,
/*! 设置显示 BUFFER 的地址及大小*/
UI_SET_DISPLAYBUF,
/*! Unicode 转换成内码*/
UI_UNICODE_TO_CHAR,
```

```
    /*! 内码转换成 Unicode*/
    UI_CHAR_TO_UNICODE,
    /*! 获取多国语言字符串的 UNICODE 编码*/
    UI_GET_MULTI_STRING_UNICODE,
    /*! UTF8 编码转换成 UNICODE*/
    UI_UTF8_TO_UNICODE,
    /*! 设置横屏或竖屏模式*/
    UI_SET_SCREEN_DIRECTION
    /*! 显示资源图片*/
    UI_SHOW_PIC,
    /*! 将资源图片读到 buffer 中*/
    UI_READ_PIC_TO_BUFFER
} ui_cmd_e;
```

UI 驱动的外部接口宏定义如下：

```
/*! 打开资源文件*/
#define ui_res_open(filename,type)
ui_op_entry((void*)(filename), (void*)(type), (void*)(0), UI_RES_OPEN)

/*! 关闭资源文件*/
#define ui_res_close(type)
ui_op_entry((void*)(type), (void*)(0), (void*)(0), UI_RES_CLOSE)

/*! ID3 滚屏专用函数*/
#define ui_scroll_string_ext(infor,param,region)
ui_op_entry((void*)(infor), (void*)(param), (void*)(region), UI_SCROLLSTRING_EXT)

/*! 显示 PicBox*/
#define ui_show_picbox(style,data)
ui_op_entry((void*)(style), (void*)(data), (void*)(0), UI_SHOW_PICBOX)

/*! 显示 TextBox*/
#define ui_show_textbox(style,data,mode)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_TEXTBOX)
```

版 权 所 有 侵 权 必 究

```
/*! 显示 TimeBox*/
#define ui_show_timebox(style,data,mode)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_TIMEBOX)

/*! 显示 NumBox*/
#define ui_show_numbox(style,data,mode)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_NUMBOX)

/*! 显示 ProgressBar*/
#define ui_show_progressbar(style,data, mode)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_PROGRESSBAR)

/*! load AttributeBox 的数据*/
#define ui_load_attributebox(style,data, count)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(count), UI_LOAD_ATTRIBUTEBOX)

/*! 显示 ListBox*/
#define ui_show_listbox(style,data,mode)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_LISTBOX)

/*! 显示 DialogBox*/
#define ui_show_dialogbox(style,data,mode)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_DIALOG)

/*! 显示 ParamBox*/
#define ui_show_parambox(style,data,mode)
ui_op_entry((void*)(style), (void*)(data), (void*)(uint32)(mode), UI_SHOW_PARAMBOX)

/*! 获取 textbox 的属性*/
#define ui_get_textbox_attrb(style,attrb,type)
ui_op_entry((void*)(style), (void*)(attrb), (void*)(type), UI_GET_TEXTBOX_ATTRB)

/*! 获取 picbox 的属性*/
```

版权所有 侵权必究

```
#define ui_get_picbox_attrb(style,attrb,type)
ui_op_entry((void*)(style), (void*)(attrb), (void*)(type), UI_GET_PICBOX_ATTRB)

/*! 获取 listbox 的属性*/
#define ui_get_listbox_attrb(style,attrb,type)
ui_op_entry((void*)(style), (void*)(attrb), (void*)(type), UI_GET_LISTBOX_ATTRB)

/*! 设置语言种类*/
#defineui_set_language(lang_id)
ui_op_entry((void*)(uint32)(lang_id), (void*)(0), (void*)(0), UI_SET_LANGUAGE)

/*! 获取 DC 状态*/
#define ui_get_DC_status(pdc)
ui_op_entry((void*)(pdc), (void*)(0), (void*)(0), UI_GET_DC_STATUS)

/*! 清屏*/
#define ui_clear_screen(clrregion)
ui_op_entry((void*)(clrregion), (void*)(0), (void*)(0), UI_CLEARSCREEN)

/*! 设置画笔颜色*/
#define ui_set_pen_color(color)
ui_op_entry((void*)(color), (void*)(0), (void*)(0), UI_SET_PEN_COLOR)

/*! 设置背景颜色*/
#define ui_set_backgd_color(color)
ui_op_entry((void*)(color), (void*)(0), (void*)(0), UI_SET_BACKGD_COLOR)

/*! 显示字符串*/
#define ui_show_string(infor, region,mode)
ui_op_entry((void*)(infor), (void*)(region), (void*)(uint32)(mode), UI_SHOW_STRING)

/*! 获取字符串的长度*/
#define ui_get_string_length(str,length,type)
ui_op_entry((void*)(str), (void*)(length), (void*)(type), UI_GET_STRING_LENGTH)
```

版 权 所 有 侵 权 必 究

```
/*! 文本解码断行，包括过滤掉空白字符和分词显示等 */
#define ui_get_text_line(desc, mode)
ui_op_entry((void*)(desc), (void*)(mode), (void*)(0), UI_GET_TEXT_LINE)

/*! 设置显示 BUFFER 的地址及大小*/
#define ui_set_display_buf(mode)
ui_op_entry((void*)(mode), (void*)(0), (void*)(0), UI_SET_DISPLAYBUF)

/*! Unicode 转换成内码*/
#define ui_unicode_to_char(str,len,lang_id)
ui_op_entry((void*)(str), (void*)(len), (void*)(lang_id), UI_UNICODE_TO_CHAR)

/*! 内码转换成 Unicode*/
#define ui_char_to_unicode(dest,src,len)
ui_op_entry((void*)(dest), (void*)(src), (void*)(len), UI_CHAR_TO_UNICODE)

/*! 获取多国语言字符串的 UNICODE 编码*/
#define ui_get_multi_string_unicode(id,type,infor)
ui_op_entry((void*)(id), (void*)(type), (void*)(infor), UI_GET_MULTI_STRING_UNICODE)

/*! UTF8 编码转换成 UNICODE*/
#define ui_utf8_to_unicode(src,dest,size)
ui_op_entry((void*)(src), (void*)(dest), (void*)(size), UI_UTF8_TO_UNICODE)

/*! 设置横屏或竖屏模式*/
#define ui_set_screen_direction(screen_mode)
ui_op_entry((void*)(uint32)(screen_mode), (void*)(0), (void*)(0),
UI_SET_SCREEN_DIRECTION)

/*! 显示图片资源*/
#define ui_show_pic(id,x,y)
ui_op_entry((void*)(uint32)(id), (void*)(uint32)(x), (void*)(uint32)(y), UI_SHOW_PIC)
```

版 权 所 有 侵 权 必 究

```
/*! 将图片资源的数据读到 buffer 中*/
#define ui_read_pic_to_buffer(id,buffer,res_size)
ui_op_entry((void*)(uint32)(id), (void*)(buffer), (void*)(res_size),
UI_READ_PIC_TO_BUFFER)
```

6.3.5 关键的数据结构

```
//资源的类型，属于 AP 还是 COMMON
typedef enum
{
   /*! 资源类型为 AP 资源 */
     UI_AP = 0,
     /*! 资源类型为 COMMON 资源 */
     UI_COM = 1
}res_type_e;

typedef struct
{
     uint16 style_id;     要显示的控件在 Style 文件中的 ID;
     res_type_e type;   //资源类型为 AP or COMMON
} style_infor_t;

/*!
 * \brief
 *    ui_error_e：ui 驱动处理错误类型
 */
typedef enum
{
     /*! 不显示控件 */
     UI_NO_DISPLAY                    = 0x00,
     /*! 正常显示 */
     UI_NO_ERR                        = 0x01,
     /*! 字符串显示需要滚屏 */
     UI_NEED_SCROLL                   = 0x02,
     /*! 参数非法错误 */
     UI_PARAM_ERR                     = 0x03,
```

```
    /*! 字符串显示 X 轴越界错误 */
    UI_PUTCHAR_X_ERR            = 0x04,
    /*! 字符串显示 Y 轴越界错误 */
    UI_PUTCHAR_Y_ERR            = 0x05,
    /*! 字符串显示其他错误 */
    UI_PUTCHAR_OTHER_ERR        = 0x06,
    /*! 显示最后一帧图片 */
    UI_PICBOX_LAST_FRAME        = 0x07
} ui_result_e;
```

AP 和 common 中需要调用某种控件时，需要传递给控件显示 API 函数传递参数，各控件私有数据结构定义如下：

6.3.5.1 显示 PicBox 的私有数据

```
typedef struct
{
    /*! 如果 PicID 不为-1，那么就是优先用 PicID 显示 */
    uint16 pic_id;
    /*! 指示显示哪一帧图片 */
    uint8 frame_id;
    /*! 显示方向（顺序），可以选择正向（0）和反向（1） */
    uint8 direction;
} picbox_private_t;

//picbox 显示顺序
#define DIRECTION_NORMAL  0     //正序显示
#define DIRECTION_REVERSE  1    //逆序显示
```

说明：显示的优先顺序：

（1） pic_id!= -1，优先显示 pic_id 所指向的图片；

（2） pic_id = -1，frame_id！=-1，则显示 StyleID 所指向的 Picbox 的 frame_id 的图片（多帧图片中的一帧），方向由 direction 指定，此时，必须给 direction 赋值；

（3） 若直接显示 picbox.id 指向的图片，则该结构体指针参数可以直接赋值为 NULL

6.3.5.2 显示 TextBox 的私有数据

```
/*!
```

```
 * \brief
 *         textbox_draw_mode_e：列表控件绘制模式枚举类型
 */
typedef enum
{
    /*! 无需绘制 */
    TEXTBOX_DRAW_NULL          = 0,
    /*! 以正常方式显示 TEXTBOX */
    TEXTBOX_DRAW_NORMAL          = 1,
    /*! 滚屏显示 */
    TEXTBOX_DRAW_SCROLL          = 2,
    /*! 滚屏显示 */
    TEXTBOX_DRAW_SCROLL_ID3      = 3,
    /*! 标题+内容 */
    TEXTBOX_DRAW_ITEM          = 4,
    /*! 滚屏显示：标题+内容，只滚内容 */
    TEXTBOX_DRAW_ITEM_SCROLL   = 5,
} textbox_draw_mode_e;
    typedef struct
    {
    /*! 字符串资源 ID */
        uint16 str_id;
        /*! str_value 字符串的语言类型 */
        int8 lang_id;
        /*! 保留字节 */
        uint8 reserve;
        /*! 字符串指针，如果非 NULL，用 str_value 显示 */
        uint8 *str_value;
  } textbox_private_t;
```

说明：显示的优先顺序：

（1）若 str_value ！=NULL，则优先显示 str_value 指向的内容；

（2）str_value =NULL

a）str_id！=-1，则优先显示 str_id 所指向的字符串；

b）str_id = -1,则显示 TextBox 中指定的字符串

（3）若直接显示 textbox.id 指向的图片，则该结构体指针参数可以直接赋值为 NULL

6.3.5.3 显示 TimeBox 的私有数据

版 权 所 有 侵 权 必 究

TimeBox 的私有数据结构定义

```
/*!
 * \brief
 *   time_disp_mode_e：时间显示模式枚举类型
 */
typedef enum
{
    /*! 日期显示：YYMMDD */
    DATE_DISP_YYMMDD     = 0,
    /*! 日期显示：MMDDYY */
    DATE_DISP_MMDDYY     = 1,
    /*! 日期显示：YYMM */
    DATE_DISP_YYMM       = 2,
    /*! 日期显示：MMDD */
    DATE_DISP_MMDD       = 3,
    /*! 时间显示：HHMMSS */
    TIME_DISP_HHMMSS= 4,
    /*! 时间显示：HHMM */
    TIME_DISP_HHMM       = 5,
    /*! 时间显示：MMSS */
    TIME_DISP_MMSS       = 6
} time_disp_mode_e;

/*!
 * \brief
 *       timebox_private_t：timebox 私有数据结构
 */
typedef struct
{
    /*! 自定义 time 结构，时分秒或年月日的数值 */
    struct
    {
        /*! 时间联合体 */
        union
```

```
        {
            date_t date;
            time_t time;
        } union_time;
        /*! time 类型：'D'表示日期，'T'表示时间，其他强制为时间 */
        char time_type;
        /*! 字符串显示模式下的分隔符 */
        char partition;
        /*! 时间显示模式 */
        time_disp_mode_e mode;
    } times[2];
    ///字符串模式下，日期时间分隔符默认为空格，时间时间分隔符默认为/
}timebox_private_t;
```

TimeBox 的显示模式定义

```
/*!
 * \brief
 *      timebox_draw_mode_e：时间控件绘制模式枚举类型
 */
typedef enum
{
    /*! 无需绘制 */
    TIMEBOX_DRAW_NULL       = 0,
    /*! 重绘时间或日期 */
    TIMEBOX_DRAW_TIME       = 1,
    /*! 重绘全部 */
    TIMEBOX_DRAW_ALL        = 2,
} timebox_draw_mode_e;
```

6.3.5.4 显示 NumBox 的私有数据

NumBox 的私有数据结构定义

```
/*!
 * \brief
 *      numbox_private_t：numbox 私有数据结构
 */
```

```
typedef struct
{
    /*! 要显示的数值，其小数点位置由配置信息决定 */
    uint16 value;
    /*! 总数，以查看当前数值所处的位置 */
    uint16 total;
} numbox_private_t;
```

NumBox 的显示模式定义

```
/*!
 * \brief
 *        numbox_draw_mode_e：数字控件绘制模式枚举类型
 */
typedef enum
{
    /*! 无需绘制 */
    NUMBOX_DRAW_NULL          = 0,
    /*! 重绘数字 */
    NUMBOX_DRAW_NUMBER          = 1,
    /*! 重绘全部 */
    NUMBOX_DRAW_ALL             = 2,
} numbox_draw_mode_e;
```

6.3.5.5 显示 ProgressBar 的私有数据

Progressbar 的私有数据结构定义

```
/*!
 * \brief
 *        progressbar_private_t：progressbar 私有数据结构
 */
typedef struct
{
    /*! 进度条当前进度（数值，由 progressbar 内部转换为步数） */
    uint16 value;
    /*! 进度条总进度 */
    uint16 total;
} progressbar_private_t;
```

版 权 所 有 侵 权 必 究

Progressbar 的显示模式定义

```
/*!
 * \brief
 *          progress_draw_mode_e：进度条控件绘制模式枚举类型
 */
typedef enum
{
    /*! 无需绘制 */
    PROGRESS_DRAW_NULL        = 0,
    /*! 重绘进度 */
    PROGRESS_DRAW_PROGRESS        = 1,
    /*! 重绘全部 */
    PROGRESS_DRAW_ALL         = 2,
} progress_draw_mode_e;
```

6.3.5.6 显示 ListBox 的私有数据

ListBox 的私有数据结构定义

```
/*!
 * \brief
 *          listbox_private_t：listbox 私有数据结构
 */
typedef struct
{
    /*! 标题字符串 */
    textbox_private_t title;
    /*! 列表项字符串数组 */
    textbox_private_t items[LIST_NUM_ONE_PAGE_MAX];
    /*! 列表项 items 中有效前几项 */
    uint8 item_valid;
    /*! 当前激活项 */
    uint16 active;
    /*! 之前激活项 */
    uint16 old;

    /*! 应用中所有项总数，用于滑动杆和 ratio */
    uint16 list_total;
    /*! 当前激活项在应用中所有项的位置，用于滑动杆和 ratio */
    uint16 list_no;
```

```
} listbox_private_t;
```

ListBox 的显示模式定义

```
*!
 * \brief
 *         list_draw_mode_e：列表控件绘制模式枚举类型
 */
typedef enum
{
    /*! 无需绘制 */
    LIST_DRAW_NULL         = 0,
    /*! 重绘激活项 */
    LIST_DRAW_ACTIVE = 1,
    /*! 重绘整个列表 */
    LIST_DRAW_LIST         = 2,
    /*! 重绘全部 */
    LIST_DRAW_ALL          = 3,
} list_draw_mode_e;
```

6.3.5.7 显示 DialogBox 的私有数据

DialogBox 的显示模式定义

```
/*!
 * \brief
 *         dialog_private_t：dialog 私有数据结构
 */
typedef struct
{
    /*! 对话框 Icon ID */
    uint16 icon_id;
    /*! desc_info 语言类型 */
    int8     language;
    uint8    reserve;
    /*! 对话框描述字符串 */
    uint8    *desc_info;
    /*! 按钮数组，最多支持 3 个按钮 */
    uint16 buttons[3];
    /*! 按钮数目，最多 3 个 */
    uint8 button_cnt;
    /*! 当前激活按钮序号 */
```

版 权 所 有 侵 权 必 究

```
    uint8 active;
    /*! 之前激活按钮序号 */
    uint8 old;
} dialog_private_t;
```

DialogBox 的显示模式定义

```
/*!
 * \brief
 *        dialog_draw_mode_e：对话框控件绘制模式枚举类型
 */
typedef enum
{
    /*! 无需绘制 */
    DIALOG_DRAW_NULL          = 0,
    /*! 重绘按钮 */
    DIALOG_DRAW_BUTTON        = 1,
    /*! 重绘对话框 */
    DIALOG_DRAW_ALL           = 2,
} dialog_draw_mode_e;
```

6.3.5.8 显示 ParamBox 的私有数据

ParamBox 的私有数据结构定义

```
/*!
 * \brief
 *        parambox_one_t：parambox 参数描述符结构体
 */
typedef struct parambox_one_struct
{
    /*! 参数单位资源字符串 ID，其实也不限于单位，可以是任意辅助说明字符串 */
    uint16 unit_id;
    /*! 参数值最小值 */
    uint16 min;
    /*! 参数值最大值 */
    uint16 max;
    /*! 参数值步进 */
    uint16 step;
    /*! 参数当前值 */
    uint16 value;
    /*! 是否允许循环设置，即最大递增变为最小，最小递减变为最大 */
```

版 权 所 有 侵 权 必 究

```
    uint8 cycle;
    /*! 参数值最大位数 */
    uint8 max_number;
    /*! 参数当前值字符串显示，可以通过 adjust_func 转换 */
    uint8 *value_str;
    /*! 参数值检测适配回调函数，比如用来限制日期设置等 */
    bool (*adjust_func)(struct parambox_one_struct *one);
    /*! 设置即时回调函数，比如声音设置等 */
    bool (*callback)(uint16 value);
} parambox_one_t;

/*!
 * \brief
 *       parambox_private_t：parambox 私有数据结构
 */
typedef struct
{
    /*! 标志图标 ID */
    uint16 icon_id;
    /*! 设置标题资源字符串 ID */
    uint16 title_id;
    /*! 设置参数个数 */
    uint8 param_cnt;
    /*! 当前激活项，也作为多参数设置时默认激活项 */
    uint8 active;
    /*! 之前激活项 */
    uint8 old;
    /*! 设置参数列表 */
    parambox_one_t *items;
} parambox_private_t;
```

ParamBox 的显示模式定义

```
/*!
 * \brief
 *       parambox_draw_mode_e：参数设置控件绘制模式枚举类型
 */
typedef enum
{
    /*! 无需绘制 */
```

```
    PARAMBOX_DRAW_NULL          = 0,
    /*! 重绘参数值 */
    PARAMBOX_DRAW_VALUE         = 1,
    /*! 重绘参数（更换参数） */
    PARAMBOX_DRAW_PARAM         = 2,
    /*! 重绘参数设置框 */
    PARAMBOX_DRAW_ALL       = 3,
} parambox_draw_mode_e;
```

6.3.5.9 显示字符串数据结构

```
/*!
 * \brief
 *      string_desc_t：字符串描述符结构体
 */
typedef struct
{
    /*! 字符串数据，或多国语言字符串 ID，或 code（内码或 Unicode） */
    union
    {
        /*! code 缓冲区指针 */
        uint8 *str;
        /*! 字符串 ID */
        uint16 id;
    } data;
    /*! 字符串数据类型
     *  分类 1：
     *  UNICODELANGUAGE（0x7f） 表示 Unicode 资源；
     *  内码语言 ID（如 ENGLISH） 表示 Ansi code data；
     *  分类 2：
     *  UNICODEID（0x7e） 表示 Unicode ID；
     *  UNICODEDATA（0x7d） 表示 Unicode code data；
     *  UTF_8DATA（0x7c） 表示 utf-8 code data；
     *  ANSIDATAAUTO（0x7b） 表示 Ansi code data（自动处理为 ui 驱动当前语言）；
     *  内码语言 ID（如 ENGLISH） 表示 Ansi code data；
     *  说明：分类 1 和分类 2 可以自由根据使用场景选择，比如在已明确 data 是 str，并且只有 Unicode
     *      和内码两种，那么可以选择第一种，并且程序在解释时也需要采用这种分类处理；
     */
```

```
int8 language;
uint8 reserve;
/*! OUT：有效字节；IN：缓冲区大小 */
uint16 length;
/*! 任意参数，由具体场合决定其意义；如在断行中可以指定行最大像素点宽度； */
uint16 argv;
/*! 用于存放结果，由具体场合决定其意义；如在断行中可以表示显示行字节数； */
uint16 result;
    } string_desc_t;
```

6.3.6 控件显示流程

6.3.6.1 PicBox 的显示流程

6.3.6.2 ListBox 的显示流程

6.3.6.3 TextBox 的显示流程

6.3.6.4 SliderBar 的显示流程

6.3.6.5 ProgressBar 的显示流程

6.3.6.6 TimeBox 的显示流程

6.3.6.7 DialogBox 的显示流程

6.3.6.8 NumBox 的显示流程

6.3.7 字符串显示

6.3.7.1 设计概述

版 权 所 有 侵 权 必 究

UI 显示驱动逻辑功能实现模块归根结底分成两部分：字符串和图片的显示处理。US212A 方案对 UI 显示的效果及性能有较高要求，故 UI 显示驱动中需要采取一些优化设计来提升 UI 显示的性能，包括使用 DMA 方式进行刷屏、分析应用的内存使用情况，充分利用不同场景下，可用的内存空间来作为显示 buffer，提升图片和字符串显示的性能等。另外，在 US212A 方案中，字符串的显示处理流程较为复杂，需要考虑各种模式下的显示处理，例如：透明、非透明、多行显示、居左、居中、居右、左右滚屏、上下滚屏、是否需要循环滚屏、是否丢弃不完整字符等。因此，显示 BUFFER 在不同应用场景下的选择设置、图片的显示、字符串在各种显示模式下的处理、特殊语言字符串显示处理等部分的设计实现，是 UI 显示驱动的核心关键部分。

6.3.7.2 显示 BUFFER 的选择

根据 US212A 方案的内存空间规划，RAM6(0x2e000 – 0x32000)、RAM7(0x32000 – 0x34000)、RAM8(0x34000 – 0x35800)、JRAM5（0x358000 – 0x36000）这四片区域是用作 JPEG 解码的 Buffer。根据不同的应用场景，UI 显示驱动可以根据考虑使用 RAM8 和 JRAM5 这两片空间（8KB）作为显示 BUFFER，由用户指定显示模式来决定显示接口的运行流程。

1、JRAM5 和 RAM8 不可用的情景： Udisk、Record 应用、生成 PLAYLIST。在这几个场景下， 这四块空间已经被占用，不可以用作 UI 显示的 buffer。对于这几个场景，能用作显示 BUFFER 的空间为：

（1）Udisk 、Record、MTP、playlist 应用：UI_LCD_BUF（1KB）；

此时，UI 显示驱动只能选择 UI_LCD_BUF 这 1KB 的数据空间作为 buffer。在以上这些场景下，UI 显示驱动会使用 IK buffer 显示图片和字符串。具体方法如下所示：

进入场景时：

ui_set_display_buf (LCD_RAM_INDEX);　　// 选择 LCD Data Buffer 作为 LCD 的缓冲区

退出场景时：

ui_set_display_buf (JPEG_RAM_INDEX);　　// 选择 JRAM5、RAM8 作为 LCD 的缓冲区

3、JRAM5、RAM8 可用的情景：除了以上几种场景外，其他情景下都是可用的。在这些场景下，UI 显示驱动会使用快速模式显示图片，并且如果用户指定字符串显示模式为读屏模式，UI 显示驱动也会用以上这些空间作为显示 buffer，加速字符串的显示。

在 US212A 方案中，绝大多数场景都可以使用 JRAM5、RAM8 作为显示 BUFFER，所以 UI 显示时默认 JRAM5、RAM8 作为显示 BUFFER，也就是说不用任何设置就可以使用快速模式显示图片，并且在设置字符串显示模式为读屏模式下，快速显示字符串。

版 权 所 有　侵 权 必 究

6.3.7.3 字符串显示处理的设计

a）对于字符串的显示处理，程序中支持直接打点和背景与字符点阵数据叠加缓存输出两种方式，默认是采用背景叠加的方式。先将需要显示字符串的区域的背景与每个字符进行像素点的叠加，叠加完成后，再用 DMA 方式送到 LCM 上，这样便可以在很短时间内完成字符串的刷新，避免字符串刷新的延迟感。这种方式对于纯色背景（非透明）字符串的显示速度有较高的提升。对于透明字符串的显示，该方式的效率也比打点方式快。

b）对于透明字符串，由于其背景不是纯色的，每次刷新字符串之前，由于显示 buffer（6K）有限，需要将背景图片先刷新到 LCM 上，然后再通过读屏的方式将背景数据从 LCD 的 GRAM 读回来，然后与字符像素点叠加之后，再输出到 LCM 上。由于读屏的速度较慢，这种方式对于需要频繁刷新的字符串（例如滚屏），会存在屏幕闪烁的现象。为了解决闪屏的问题，程序中对于背景图片尺寸小于 6K 的字符串的显示处理作了优化设计，即先将背景图片数据从 SD 区读出存入显示 BUFFER 中，然后与字符像素点进行叠加，最后才通过 DMA 方式输出到 LCM 上。这样，就可以达到与非透明字符串同样的显示效果。当然，对于尺寸大于 6K 的图片，就无法采用该方式。

c）字符串显示抽象为以字符为核心，把字符串理解为字符序列，对字符序列进行处理，把内码和 unicode 统一起来处理，并且对特殊语言类型如阿拉伯、希伯来、泰文等进行特殊处理，对实现细节进行封装，对上层应用提供统一的透明接口。上层应用在调用字符串显示接口时，不需要区分是否内码和 UNICODE，也不需要对特殊语言进行处理，只需要将这些信息传递给 UI 驱动，驱动内部会根据这些信息进行不同的处理。

d) 字符串处理添加字符字模点阵缓冲机制，利用 JRAM5 2K 的内存空间，可以将常用的字模点阵缓存起来，减少了从 flash 读字模的次数，也可以提高字符串显示的速度。

6.3.7.4 字符串向上滚屏的实现

US212A 方案中，需要实现字符串向上滚屏的功能，而且要求显示效果流畅，不能有停顿感。因此，对于向上滚屏的显示处理也是 UI 显示驱动需要实现的一个重要功能。

字符串向上滚屏时，相当于有两个字符串在同时滚屏显示。当上面一个字符串向上滚动的同时，下面一个字符串也要相应地向上滚动。应用需要传入两个字符串指针给 UI 显示驱动以实现滚屏。而且，同一个字符串中的内容是同时向上滚屏，即应将每个字符串中的字符进行整理处理后，才统一输出到 LCM 上。否则，如果采取单个字符处理的方式，会产生字符刷新的延迟感，显示效果不流畅。因此，如何充分利用可用的内存空间作为显示 BUFFER，对需要滚屏的字符串进行整理处理输出到 LCM 上，是向上滚屏需要重点考虑的问题。

版 权 所 有 侵 权 必 究

a）向上滚屏时，将需要滚屏的两个字符串的所有字符的点阵字模取出，根据当前滚屏位置，将两个字符串中相应部分的像素点进行前景/背景转换，直接保存成需要输出的像素点信息，存到缓冲区中，然后用 DMA 方式输出到 LCM 上。

b）特殊的语言类型处理：对于阿拉伯和希伯来语，会先进行转连写和左右反序映射处理之后，在进行滚屏处理。

c）向上滚屏所需要的 buffer 空间：对于 128×160 的彩屏，缓存一行像素点信息所需要的空间为（默认字符高度为 16）：16×128×2=4096（4KB），因此，向上滚屏需要 4K 的 buffer 空间。

6.3.7.5 泰语特殊处理

泰语的书写具有叠加的特点，它分为主字符和上下标，它有特定的几个字符是上下标，需要把它们叠加到主字符的上端或者下端。

例：“น”、 “ปั”、 “สู” 、“คุ้”这四个字符

“น”仅有一个主字符（0xE19）

“ปั”是一个主字符加一个上标组合成的（0xE1B、0xE31）

“สู”是一个主字符加一个下标组合成的（0xE2A、0xE39）

“คุ้”是一个主字符加一个下标和一个上标组成（0xE04、0xE38、0xE49）

uint8 thai_point(uint8 * str, uint8 uni_flag, uint16 source_code, uint8* char_buf)

该函数用于对泰文字符进行分析，如果当前字符有上标下标，则将其叠加，正确输出。该函数被 fix_get_next_char_point_hook 调用。

规则：泰文的字符是以主子符开头，上标、下标（如果存在）结束，当前字符是主子符，直到找到下一个主子符，当前文字的编码才结束。

6.3.7.6 阿拉伯特殊处理

阿拉伯与希伯来属于闪含语系，它们的书写、查看习惯是从右往左的，字符串滚动方向是从左往右，跟一般的语言习惯完全相反。而这两种语言在查看英文和阿拉伯数字时又是正常显示。

阿拉伯的书写还具有连写的特征：一个字符的书写形式是跟它前面一个字符，后面两个字符相关的，就是需要根据前后的字符进行相应的变形，这样的处理使得它整体看起来是连在一起的，而这种连写字模需要使用 unicode 字库来显示，暂时没有内码的连写字库。

阿拉伯和希伯来语字符串的来源有 3 种：xls 文件中的资源字符串、非资源 UNICODE

字符串、ANSI 字符串。注：US212A 没有对阿拉伯语的资源字符串连写处理。

由于我们把字符串统一按照从左到右显示，于是阿拉伯和希伯来语字符串最终显示时还是需要临时将字符串内容左右反序处理。

为了更方便对阿拉伯和希伯来语字符串的跟踪处理，我们为阿拉伯和希伯来语字符串处理设计了一个状态机，如下所示：

```
/*!
 * \brief
 *      arabic_status_e 阿拉伯语和希伯来语编码流状态枚举类型
 */
typedef enum
{
    /*! 初始状态， 由 string_desc_t 描述 */
    ARABIC_INIT        = 0,
    /*! UNICODE 连写状态 */
    ARABIC_UNI_JOIN    = 1,
    /*! 西文翻转状态 */
    ARABIC_ASCII_TURN  = 2,
    /*! 中间完整单行翻转状态 */
    ARABIC_LINE_TURN   = 3
} arabic_status_e;
```

阿拉伯和希伯来字符串处理和状态转变如下所示：

ARABIC_INIT：调用 **arabic_uni_join** 接口进行 UNICODE 转换和阿拉伯连写处理，arabic_uni_join 接口实现如下：

```
void arabic_uni_join(string_desc_t *desc)
{
    //内码转换为 Unicode 码，以便进行阿拉伯连写处理
    if (desc->language != UNICODEDATA)
    {
        arabic_char_to_unicode(desc);
        desc->language = UNICODEDATA;
    }

    //阿拉伯连写处理
    if (language_id == ARABIC)
    {
        arabic_do_join(desc);
```

```
    }
}
```

处理完成之后，状态转变为 ARABIC_UNI_JOIN。

ARABIC_UNI_JOIN：调用 arabic_ascii_turn 接口进行西文翻转处理，这样西文最终显示方向为从左到右。处理完成之后，状态转变为 ARABIC_ASCII_TURN。

ARABIC_ASCII_TURN：这种状态下的字符串是用来从右到左显示的，而我们单行字符串显示处理为：前面不完整字符 + 中间一段字符串 + 尾部不完整字符，所以对每行字符串，中间一段字符串需要临时左右反序，用统一的从左到右的处理逻辑进行显示。

6.3.8 UI 驱动的内存分配说明

UI 驱动模块的代码空间分配，具体分配如下：

- 常驻代码空间包括：0xbfc21580 ~ 0xbfc2187f = 0x300 字节，即 0.75K 字节。
- bank a/b/c 代码空间：
  - ✧ bank a：(0x16**0000+0x24c00) ~ (0x16**0000+0x24fff) = 0x400 字节。
  - ✧ bank b：(0x26**0000+0x25000) ~ (0x26**0000+0x257ff) = 0x800 字节。
  - ✧ bank c：(0x36**0000+0x21a00) ~ (0x36**0000+0x21dff) = 0x400 字节。
  - ✧ 说明：**高 6bit 为 bank 号，比如 0x16064c00 中 0x06 高 6bit 为 0x01，故为 ui 驱动的 BANK A 组 BANK A1。

UI 驱动模块的常驻数据空间为 0x9fc1ce00-0x9fc1d1ff = 0x400字节。另外，系统还专门为UI 驱动分配UI_LCD_BUF 1KB的缓冲区空间。

6.3.9 多种语言支持

US212A 方案支持 28 种语言，覆盖全世界主要语种。

Language	语言	所属语系
Czech	捷克语	印欧语系（西斯拉夫语族）
Greek	希腊语	印欧语系
Engllish	英语	印欧语系（日尔曼语族）

版 权 所 有 侵 权 必 究

Danish	丹麦语	印欧语系（日尔曼语族北语支）
German	德语	印欧语系（日尔曼）
Spanish	西班牙语	印欧语系
French	法语	印欧语系
Italian	意大利语	印欧语系
Hungarian	匈牙利语	印欧语系（西斯拉夫语族）
Dutch	荷兰语	印欧语系（日尔曼）
Norwegian	挪威语	印欧语系
Polish	波兰语	印欧语系（西斯拉夫语族）
Portuguese_Epropean	葡萄牙语	印欧语系
Portuguese_Brazilian	巴西葡萄牙语	印欧语系
Russian	俄语	斯拉夫语系
Slovak	斯洛伐克语	印欧语系（西斯拉夫语族）
Finnish	芬兰语	印欧语系
Swedish	瑞典语	印欧语系（日尔曼语族）
Turkish	土尔其语	阿尔泰语系（突厥语族）
Schinese	简体中文	汉藏语系
Tchinese	繁体中文	汉藏语系
Japanese	日语	阿尔泰语系
Korean	韩语	阿尔泰语系
Hebrew	希伯莱语	闪含语系闪含语族
Arabic	阿拉伯语	闪含语系
Thai	泰语	（泰语语族）
Romanian	罗马尼亚语	印欧语系（）
Indonesian	印尼语	印欧语系

6.3.9.1 字符编码与字库

为了使用计算机处理字符，就需要将字符和二进制编码对应起来，这种对应关系就是字符编码。

制定编码首先要确定字符集，并将字符集内的字符排序，然后和二进制数字对应起来。

版 权 所 有 侵 权 必 究

根据字符集内字符的多少，会确定用几个字节来编码。

每种编码都限定了一个明确的字符集合，叫做被编码过的字符集(Coded Character Set)，这是字符集的另外一个含义。而每一个字符集都可以有多种编码。

Unicode 码

Unicode 码，也叫统一码，是一种在计算机上使用的字符编码。它为每种语言中的每个字符设定了统一并且唯一的二进制编码。unicode(或 UCS-4)中，根据最高字节，分成 128 个 group。每个 group 根据次高字节分成 256 个 plane。目前可以使用的是 0 group 中 0~16 共 17 个 plane，编码范围为 0~10FFFF ，约 110 多万个码位。

其中最重要的是 plane0(0~FFFF)，称为基本多语言平面，即 BMP(Basic Multilingual Plane)。BMP 包含目前世界上使用的所有书写系统中的全部常用字符，以及一些历史上的不常用字符。BMP 去掉两个高字节的 0，就是 UCS-2。

通常所说的 unicode 码指的就是 2 字节编码的 BMP。

内码

字符集常和一种具体的语言文字相对应。如英文字符集，繁体汉字字符集，日文字符集等。不同的字符集的本地编码称为内码。

常用的内码编码：

- ASCII
- 中文编码
  - GB2312
  - GBK
  - Big5
  - GB18030

Codepage

Codepage 就是各国的文字编码和 Unicode 之间的映射表。比如简体中文和 Unicode 的映射表就是 CP936。以下是几个常用的 codepage。

- codepage=932 日文
- codepage=936 简体中文 GBK
- codepage=949 韩文
- codepage=950 繁体中文 BIG5

字库

版 权 所 有 侵 权 必 究

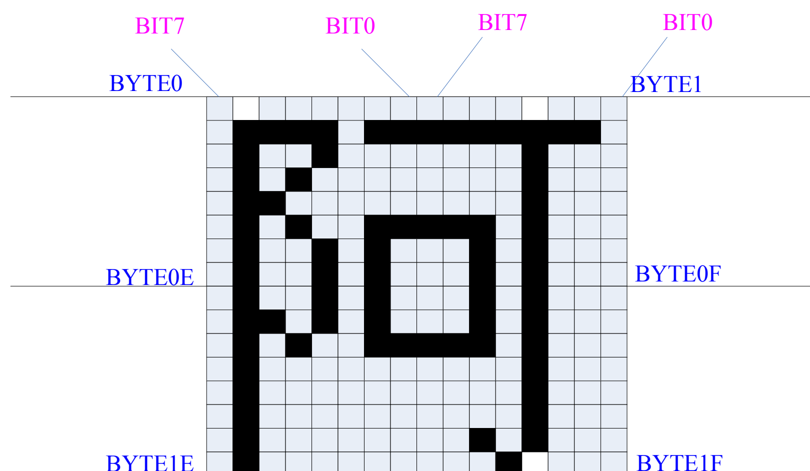
字库按照数据存储内容分为：

- 点阵字库
- 矢量字库

字库按照编码格式又可分为：

- 内码字库
- Unicode 字库

点阵字库里存储的数据为：



十六进制数据列表如下，其中第 3 行的 10 表示字符宽度为 16 像素点：

00 00 7D FE 44 08 48 08 49 E8 51 28 49 28 49 28

45 28 45 28 45 E8 69 28 50 08 40 08 40 28 40 10

10

6.3.9.2 需求与设计要点

多种语言支持需求

- 资源字符串，以 excel 表格形式输入，其编码是 UNICODE 16，通过 UI Editor 工具打包为 sty 文件
- 文件系统，包括文件名，ID3 信息等，其编码可以是 UNICODE 16、ANSI 和 UTF-8

设计要点

- US212A 使用 Unicode 点阵字库，要来显示的字符/字符串最终转换为 Unicode 16 编码索引字库获取点阵并显示
- US212A 支持 UNICODE 16 编码、ANSI 编码和 UTF-8 编码，其中 UTF-8 编码通

版 权 所 有 侵 权 必 究

过转换规则很容易转换为 Unicode 16 编码，而 ANSI 编码必须通过语言索引表 codepage 进行转换，所以 US212A 必须提供所有支持的语言对应的 codepage 表

-

6.3.9.3 资源字符串

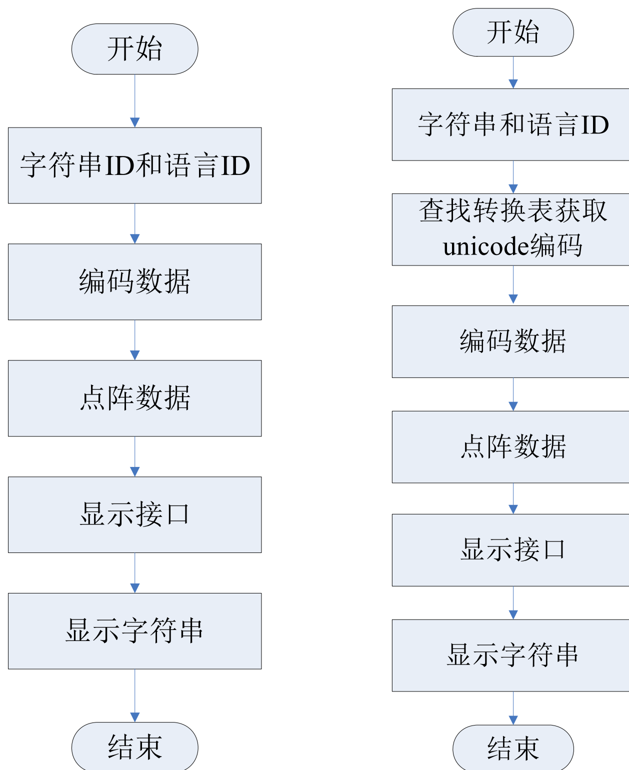
US212A 为了让各个 AP 的 UI 显示彼此独立，每个 AP 的图片和资源字符串都独立处理，所以每个 AP 都有一个资源字符串 excel 文件，COMMON 也有一个资源字符串 excel 文件，我们要求多种语言的排序必须以 COMMON 为准，并且 UI Editor 的 COMMON 工程会生成一个 lang_id.h 文件，其中就定义了各种语言的 ID。

	A	B	C	D	E	F
1	ResID	Czech	Greek	English	Danish	German
2	s_button_ok	OK	OK	OK	OK	OK
3	s_button_cancel	Zrušit	Ακύρωση	Cancel	Annuller	Abbrechen
4	s_button_yes	Ano	Ναι	Yes	Ja	Ja
5	s_button_no	Ne	Όχι	No	Nej	Nein
6	s_button_abort	Ukončení	Ματαίωση	Abort	Aborter	aufhoeren

UI Editor 工程会为 Common 和 各个 AP 生成其资源字符串 ID 头文件 *_res.h ，使用头文件中的 ID 宏就可以在对应的 sty 文件中索引资源字符串，获取该字符串的 Unicode 编码流，这些编码流以 ‘\0’’\0’ 结束符结尾。

6.3.9.4 显示流程

资源字符串显示流程　　ANSI 编码字符串显示流程



6.3.9.5 添加语言

修改驱动：

- 在 UI 显示驱动多国语言转换表数组 MBtoUnicTabName 中加入日文转换表；
- 在 UI 显示驱动设置语言函数中，加入相应的语言类型及转换表处理 set_language(uint8 lang_id);
- 添加 unicode 转换表文件

  例如：新添加日语，则需要在 fwimage.cfg 文件中添加

  FWIM=“V932JIS.TBL”； //添加日文 unicode 转换表

在菜单编辑工具中增加新增语言的菜单选项

- 修改 ap_setting
- 修改 ap_setting.h 中 LANGUAGE_TOTAL 宏定义
- 在 ap_setting 的 menu_callback_language.c 中 language_text_table 中增加新增语言的字符串资源 ID
- 在 ap_setting 的 menu_callback_language.c 中增加新增语言的菜单响应回调函数。

版权所有 侵权必究

修改 Case 配置文件 Config.txt

SETTING_AP_ID_LANGUAGE_ID = 19[0~23,1]; //语言种类

注意最大语言类型 ID 和默认语言类型 ID 的变化。

Xls 文件中添加新增语言列

语言类型 ID 表是以 Common 的 xls 中的语言排列顺序为准的，所以新增的语言类型的位置必须保证 AP 和 Common 一致。

6.3.10 多字库/字体支持

US212A 的 UI 驱动支持多字库方案，最多可支持 3 种字库/字体，这 3 种字库命名为大字库/字体 UNICODEL.FON、中字库/字体 UNICODE.FON 和小字库/字体 UNICODES.FON，并且对应于 UI Editor 工具中小字体、中字体和大字体。

注意：字库/字体文件名写在 UI 驱动源代码中，所以不能随意修改名字，如果要修改，请确保与 UI 驱动源代码同步修改。

6.3.10.1 字库格式

字库格式定义如下：

1、字模保存必须保存在 Height*Width = H*8|H*16|H*24|H*32 矩阵中，H >= 字模实际高度，并且 <= 矩阵宽度。

2、当 H >= 字模实际高度时，字模点阵垂直方向上居中存放，如果实际高度为偶数，那么上下边各填充一半；如果为奇数，下边比上边多 1 个点。

3、字模点阵水平方向上从做到右存放，从字节最高 bit 开始存放；如果实际宽度大于矩阵宽度 W 的一半，那么每行（W bits）存放 1 行字模点阵；否则，每行存放 2 行字模点阵，即第 2N 和 2N + 1 行（从 0 开始编号）字模点阵紧连着存放；其中字模点阵的行数等于矩阵高度，即包括填充行。

注意：8*8 矩阵字库不支持每行存放 2 行字模点阵，也就是说，即使实际宽度少于等于 4，也是每行存放 1 行字模点阵。

4、字模点阵数据后首字节为字模点阵的实际宽度，比如，16*16 矩阵字库，第 32 个字节（从 0 开始编号）即为实际宽度。

6.3.10.2 字库配置说明

多字库/字体配置

通过宏 USE_MULTI_FONTLIB 进行配置，该宏定义如下：

1、 定义为一个 8 bit 整数，其中 bit0 表示小字体， bit1 表示中字体， bit2 表示大字体

2、 如果定义为 0，表示使用默认标准字库，即字模实际宽高为 16*16 的字库，这种情况下可以不用理会“各种字体字库配置”和“字模点阵缓冲区大小”。

```
< display.h >
//字库定义，对应 UI Editor 工具中的小字体、中字体和大字体
//bit0 表示小字体，其名称必须为 UNICODES.FON
//bit1 表示中字体，其名称必须为 UNICODE.FON
//bit2 表示大字体，其名称必须为 UNICODEL.FON
//如果为 0，表示使用默认标准字库，即字模实际宽高为 16*16
//注：如果 lcd 屏不支持读屏，即使不需要使用多字库，也需要把 USE_MULTI_FONTLIB 定义为 0x02
#define    USE_MULTI_FONTLIB              0x06
```

同步配置固件打包脚本

多字库/字体配置后，必须同步配置固件打包配置脚本 fwimage.cfg 中的配置项：

```
<fwimage.cfg>
//小字体
FWIM="UNICODES.FON";
//中字体
FWIM="UNICODE.FON";
//大字体
FWIM="UNICODEL.FON";
```

配置具体字库/字体规格

每种字库都必须配置这些信息：字库矩阵高度、字库字模实际最大高度、字库矩阵宽度、字模点阵缓冲区大小，以字库 2 为例定义如下：

```
< display.h >
//字库 2，对应 UI Editor 工具中的中字体，该字体字库必须有
#define    FONTLIB_2_HEIGHT               16    //生成字模点阵最大高度
#define    FONTLIB_2_HEIGHT_ACTUAL        16    //生成字模点阵实际最大高度
#define    FONTLIB_2_WIDTH                16    //生成字模点阵最大宽度，要求为 8 倍数
```

版 权 所 有 侵 权 必 究

```
#define FONTLIB_2_SIZE (FONTLIB_2_HEIGHT * FONTLIB_2_WIDTH / 8 + 1)
```

配置字库字模缓冲区大小

< display.h >

```
//字库字模缓冲区大小，同大字体字库，用于定义字模缓冲区；如果没有大字体字库，则选用中字体字库；
#define MAX_CHARPOINTE_SIZE FONTLIB_3_SIZE
```

6. 4 黑白屏 UI 驱动设计

本章节用于指导 US212A 黑白屏驱动的开发，提供彩屏 LCD 驱动修改改成黑白屏显示的修改方法。

6.4.1 需求概述及设计原则

彩屏驱动与黑白屏有如下不同：

- 硬件接口不同：彩屏一般为 8bit 接口；黑白屏分串行与并行接口。
- 像素数据格式不同：黑白屏 1 bit 表示 1 个 pixel；彩屏则 2 bytes 表示 1 个 pixel。
- 刷屏方式不同。
- 更新方式不同。
- 其他

黑白屏驱动的设计原则如下：

- 保持.sty 资源的数据格式不变，保持 ui-editor 工具不变。
- 使用彩屏图片格式（24 位位图 bmp，彩色属性）。

这样设计的原因是：既要满足工具及数据格式的兼容性，使设计最简单化，又要满足黑白屏的显示需要。

方法也很简单：因为彩色图片为 RGB565 格式，即 2Byte 表示 1 个 pixel，而黑白屏显示 1 个 pixel 为 1bit。所以在刷图接口需要特别处理，读数据依然为彩图方式，但 2 byte 仅取 1bit，以此类推，组织够 1 个 page 后统一刷屏。

6.4.2 修改指南

一、硬件修改。根据使用黑白屏的接口方式，修改响应的 GPIO。

修改点为 display.h 中的宏定义，如下：

```
//lcd reset control
#define LCMRST_GIO_EN_REG          GPIOCOUTEN      //GPIO_C0 (Output)
#define LCMRST_GIO_DATA_REG        GPIOCDAT
#define LCMRST_GIO_EN_BIT          0x01
#define LCMRST_SET_BIT             0x01
#define LCMRST_CLR_BIT             0xFE

//lcd backlight control
#define LCMBL_GIO_EN_REG           GPIOCOUTEN          //GPIO_C1 (Output)
#define LCMBL_GIO_DATA_REG         GPIOCDAT
#define LCMBL_GIO_EN_BIT           0x02
#define LCMBL_SET_BIT              0x02
#define LCMBL_CLR_BIT              0xFD

//lcd read and write control
#define LCMRS_GIO_EN_REG           GPIOBOUTEN          //GPIO_B4 (Output)
#define LCMRS_GIO_DATA_REG         GPIOBDAT
#define LCMRS_GIO_EN_BIT           0x10
#define LCMRS_GIO_DN_BIT           0xEF
#define LCMRS_SET_BIT              0x10
#define LCMRS_CLR_BIT              0xEF

//lcd chip select control
#define CE_GIO_EN_REG              GPIOAOUTEN          //GPIO_A7 CS pin
#define CE_GIO_DATA_REG            GPIOADAT
#define CE_EN_BIT                  0x80      //ce enable
#define CE_DN_BIT                  0x7F
#define CE_SET_BIT                 0x80
#define CE_CLR_BIT                 0x7F
```

二、修改 LCD 分辨率。

```
#define Display_Length        128
#define Display_Height        160
```

版 权 所 有 侵 权 必 究

三、修改图片资源。

如果已有黑白屏图片资源，需将黑白图片重新编辑，另存为 24 位位图 bmp。

如果使用彩屏图片资源，不必转换格式，只需确保图片分辨率满足 LCD 分辨率即可。

再通过 UI-EDITOR 进行资源编辑，生成最终的.sty 文件及.h 文件。(可参考 7.2 UI Editor 使用说明。)

四、修改 welcome。

修改点：case\drv\welcome

函数模块划分	模块简述
welcome.c	修改刷屏调用方式等
lcd_hardware_init.c	修改 LCD 硬件初始化等
rcode_lcd_functions.c	修改 write_data、write_cmd、data_trans 等 LCD 操作接口
welcome.xn	编译新增加的文件

如果 welcome 正常显示，则可证明硬件 OK，可进入下一步。

五、修改 LCD 驱动：

修改点：case\drv\lcd_ZD932

函数模块划分	模块简述
rcode_lcd_functions_3.c	修改或删除“设窗”、“刷屏模式”等 LCD 接口
bank_a_lcd_functions.c	修改或删除“设对比度”“屏休眠”“开关背光”等 LCD 接口
rcode_lcd_functions.c	修改 write_data、write_cmd、data_trans、读屏等 LCD 操作接口
rcode_lcd_functions_2.c	删除 DMA 操作，改为 MCU 方式
rcode_lcd_op_entry.c	其他定义

LCD 驱动中的接口在 welcome 中都出现过，不过是供 UI 驱动及 ap 层调用。LCD 硬件初始化没有必要重新再做。

六、修改 UI 驱动：

修改点：case\drv\ui\，Ui 驱动的修改总结只有 2 点：

1. 增加黑白屏的显示接口 u_put_sd_image()(彩屏接口为 rom_put_sd_image)，参考文件 bank_b_ui_show_picbox_2.c。

版 权 所 有 侵 权 必 究

2. 重新封装黑白屏刷图接口 u_show_pic()(彩屏为 romf_show_pic)，参考文件 rcode_ui_functions.c。搜索所有刷图接口的调用 romf_show_pic，改为 u_show_pic。
3. 修改 rcode_ui_functions.c 文件中的 fix_dma_buffer_data_hook 将显示 buffer 中的数据通过 DMA 送屏，改成通过 MCU 方式。

七、Ap 应用的修改：

有了上面的基础之后，应用的修改就很简单。因为 us212a 方案的设计初衷就是实现显示流与控制流完全分开，绝大部分的显示在 ui 驱动中已实现，应用中很少直接调用刷图接口。以 mainmenu_ap 为例，编辑生成 sty 文件，根据新的 h 文件重新编译 mainmenu_ap 即可。

八、其他未尽之处

6.5 Welcome 驱动设计

6.5.1 Welcome 的定义和作用

Welcome 驱动，顾名思义，就是一个在机器启动时用来显示“欢迎”界面的模块；更准确的定义是，Welcome 驱动是一个在机器启动一开始就运行的用户代码模块，该代码模块以驱动的形式进行封装。这里说的用户代码模块是指区别于系统平台即 PSP 的，具体 case 的一段代码。

所以，Welcome 驱动的功能几乎都与启动有关，最典型的包含以下几个：

- 显示“欢迎”界面，当然，必须先初始化 LCD。
- 启动选项，比如按住某个按键启动进入 ADFU，或者按住某个按键组合进入启动菜单等。
- 系统硬件环境检测，比如低电检测，如果硬件环境检测失败，可以选择进入睡眠模式等异常处理。
- 系统硬件环境初始化，比如禁用喇叭。

6.5.2 Welcome 的硬件接口

Welcome 由于需要显示“欢迎”界面，所以需要包含的 LCD 驱动部分功能，即硬件初始化和显示图片。为了方便方案维护，我们让 Welcome 驱动共享 LCD 驱动的部分代码，通

版 权 所 有 侵 权 必 究

过 makefile 编译选项 -D __WELCOME__ 选择 Welcome 驱动代码，和链接选项 -gc-section 在链接时自动去掉没有调用到的接口以节省代码空间，这样做到 Welcome 驱动和 LCD 驱动共享同一份 LCD 驱动代码，方便后续方案维护。

void disp_starting(void(*adfu_launch)(void), void(*read_res)(uint8*, uint16, uint16))

read_res(buffer_addr, 0, 40)：读取 Welcome 资源图片数据，3 个参数分别为目标缓冲区；Welcome 资源图片偏移地址，以扇区为单位；读取资源数据大小，以扇区为单位。

void lcd_hardware_init(void)：LCD 模组硬件初始化。

由于 Welcome 模块被打包为 brec 模块的一部分，在启动的 mbrec 阶段被加载到从 0xbfc1e000 开始的 8KB 内存中，并且在 brec 阶段以绝对地址直接调用的，所以 Welcome 并不需要定义 init 和 exit 接口。（在系统加载起来后，驱动加载需要调用 install 接口，该接口会把驱动的常住内存的.text 段等加载到内存，然后调用该驱动的 init 函数进行初始化。）

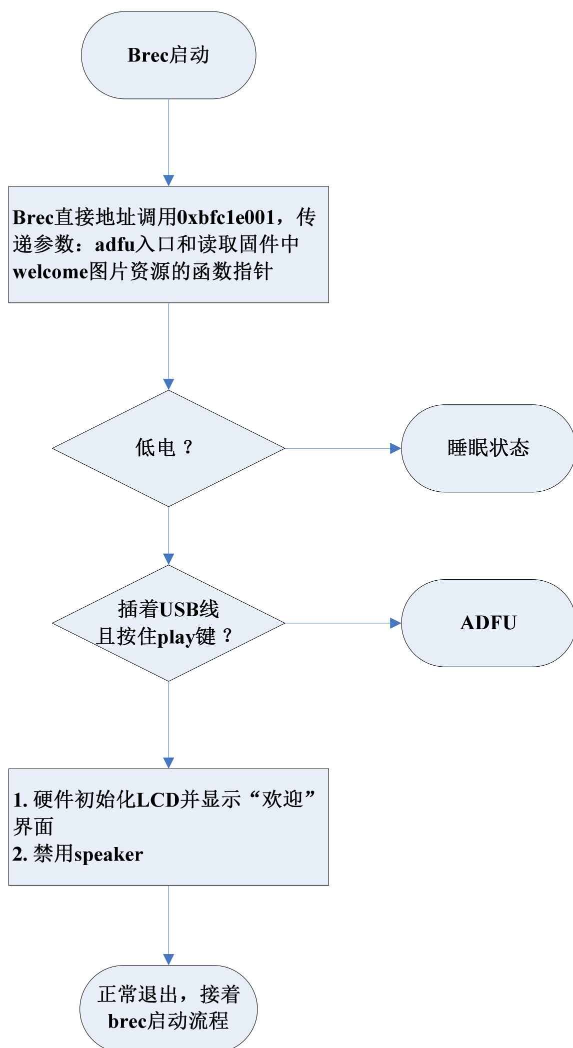
注意：Welcome 模块的大小不能超过 8KB。

6.5.3 Welcome 的启动和业务流程

Welcome 模块被打包为 brec 模块的一部分，在启动的 mbrec 阶段被加载到从 0xbfc1e000 开始的 8KB 内存中，并且在 brec 阶段以绝对地址直接调用的。（所以要求 Welcome 的入口必须链接到约定的绝对地址 0xbfc1e001，即在 xn 文件中，把 Welcome.o 链接到.text 段的开头。）此时显示“欢迎”界面；如果系统硬件环境检测通过，brec 才会接着执行后续启动流程。

下面简单说明 Welcome 模块业务流程，如下图所示：

版 权 所 有 侵 权 必 究

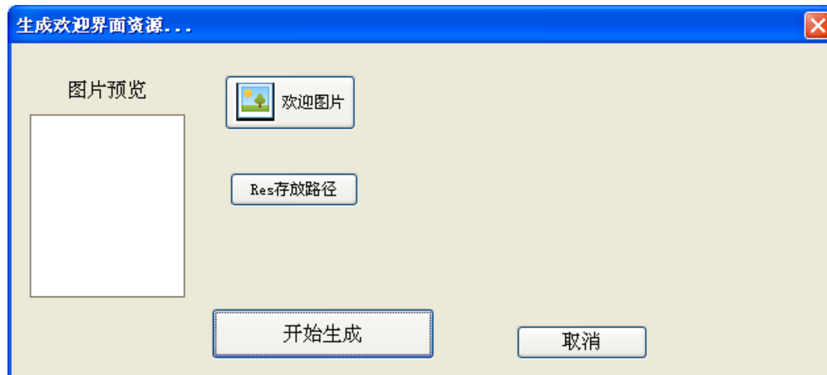


6.5.4 如何修改 Welcome 的界面

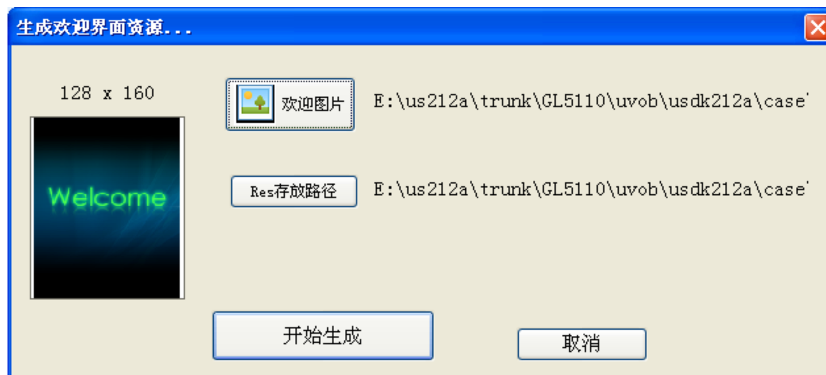
Welcome 图片的制作是通过 UI Editor 工具完成的。操作流程如下：

1. 打开 UI Editor 工具，选择工作路径，然后进入。

2. 选择**文件->生成欢迎图片资源...**，弹出制作 Welcome 图片的窗口。



3. 选择欢迎图片，图片规格必须和 LCD 屏尺寸相符；选择 res 存放路径，请选择到 case/fwpkg 目录下，生成 welcome.res。



4. 点击开始生成按钮，即可生成 Welcome 资源文件。
5. 重新打包固件，即可看到更新了的“欢迎”界面。

7 界面设计与开发

界面包括 UI 和菜单 2 部分。

7.1 引进可配置化方法

US212A 引进了可配置化 UI 和可配置化菜单技术，把 UI 和菜单的许多细节从代码剥离出来，放到数据文件去，通过可视化工具修改数据文件达到配置 UI 和菜单的目的。这些细节主要就是那些在方案确定后，在使用过程中不再变化的属性，比如 UI 的坐标位置、背景、颜色、对齐方式、字体大小等，菜单的菜单树结构等。

很明显，相比以前的处理方法，使用可配置化的方法设计和开发 UI 和菜单不再需要去关心那些本来就不应该关心的细节，大大减轻了编写代码的负担，并且，在开发后期及项目维护期间，对 UI 和菜单的细节修改并不需要修改到源码，而只是在可视化工具进行调整即可，大大减轻了这些工作的难度。

7.2 可配置化 UI

可配置化 UI 使用 UI Editor 工具，实现“所见即所得”的人性化 UI 设计手段，操作简单，功能强大。

7.2.1 UI Editor 概述

7.2.1.1 基本概念

UI Editor 以场景为 UI 组织单元，以控件为组成元素；一个场景由一个或多个控件有机组合而成。

UI 显示是以控件为单位的，场景仅仅是一个逻辑上的概念，在显示时需要逐个有序的显示场景中的全部或部分控件。

UI Editor 实现了以下基本控件：

控件	说明
PicBox	图片显示框，用来描述一个图片区域，可以显示一帧或者多帧图片，每一个图帧都具有相同的 PicBox 的属性。
TextBox	文本显示框，用于显示一个字符串，字符串包括指定的字符串和变化的字符串，用户可以设置显示的区域大小，显示的背景，以及输出模式。
NumberBox	数字显示框，提供一个特殊数字显示控件，支持显示数字间可以添加一个分隔符，可以用来显示小数点和 ratio。
TimeBox	日期/时间显示框，提供一个显示时间/日期的控件，用户只需要设置显示时间或日期，而不必关心其他显示的细节，就可以完成时间或日期的显示。
ProgressBar	进度条，提供一个进度条控件给用户，用于显示进度。
ListBox	列表显示框，以列表方式显示活动字符串资源，如菜单列表，目录、文件列表。
ParamBox 和 SliderBar	参数设置框，提供一个参数显示控件给用户，用于查看和设置参数，包括连续参数和离散参数。
DialogBox	对话框，提供一个对话框控件给用户，用于消息提示，警告，错误提示，或者询问等。
AttributeBox	属性框，用来描述某一个控件的属性，或者记录一些常量。

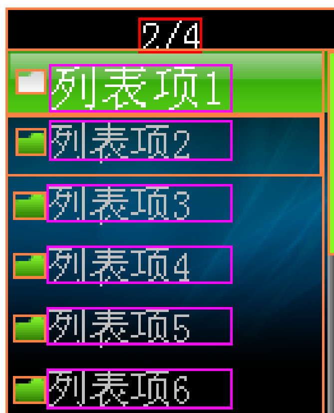
7.2.1.2 控件基本组成

控件由以下基本元素组成：（以列表控件为例）

- ❖ 图片（橙色框）
- ❖ 字符串（紫色框）

版 权 所 有 侵 权 必 究

❖ 数字，可以是字符串，也可以是数字图片（红色框）



图片显示属性：

❖ 图片内容：资源图片 ID，可以在代码中指定，支持多帧图片。
❖ 图片显示区域，只需指定图片显示左上角坐标即可，宽度和高度由资源图片决定。

字符串显示属性：

❖ 字符串内容：资源字符串 ID，字符串编码流。
❖ 字符串显示区域。
❖ 字符串显示颜色。
❖ 字符串背景，分背景图片，背景颜色填充，透明三种。
❖ 若背景为背景图片，需要指定背景图片显示区域。
❖ 字符串显示模式如下：16bit，见下页

```
16bit 字符串显示模式格式：
0-1      | 2-3      | 4-7          | 8-9      | 10-11 | 12       | 13       | 14           | 15
填充模式 | 对齐方式 | 多行显示模式 | 滚屏属性 | 字号  | 自动分词 | 取消显示 | 语言列表菜单 | 取消阿拉伯属性
```

✧ 填充模式：分为普通模式，非透明模式，直接读图模式，直接打点模式。
✧ 对齐方式：分为左对齐，居中，右对齐。
✧ 多行显示模式：分为显示不完整字符，丢弃不完整字符，多行显示，加省略号显示，滚屏显示。
✧ 滚屏属性：分为水平滚屏，向上滚屏，循环滚屏。
✧ 字号：分为大字号，中字号，小字号，需要多套字库支持。
✧ 语言列表菜单：表明当前字符串是语言列表菜单中的一项，需要做特殊处理。
✧ 取消阿拉伯右对齐。

数字显示概述：

❖ 数字是一种特殊的字符，即可以用字符串来显示，也可以用数字图片来显示。

- 在某些场合下，我们需要一些尺寸比较小的数字及符号，而字库是无法提供小尺寸的数字及符号的，这时就需要制作这样的数字及符号图片。另外，数字及符号图片可以设计得很艺术感，对 UI 美化有很大帮助。
- 一般要求数字图片宽度一致，符号图片可以略小。
- 一般要求数字图片名称要按字典排序，这样生成的 res_id 才会连续递增，方便访问。
- 数字用数字图片进行显示，最好该数字具有固定格式，可以在固定位置显示，这样更新显示非常简单。

数字显示属性：

- 数字，由代码指定。
- 数字显示区域。
- 数字显示背景，分背景图片，背景颜色填充，透明三种。
- 若背景为背景图片，需要指定背景图片显示区域。
- 数字显示模式，分字符串模式和数字图片两种。
- 数字显示位数及是否显示前置 0（012）。
- 是否显示符号，即正数是否显示+号。
- 字符串模式专用属性
  - 字符串显示颜色
- 数字图片模式专用属性
  - 数字图片 0 ID 及其宽度
  - 分隔符图片 ID 及其宽度

基本元素组成控件时，关键在于控件属性的选择：

- 第 1 类　哪些属性可以在控件中默认设置
- 第 2 类　哪些属性必须开放给用户，可以由用户根据具体需要进行自定义
- 第 3 类　哪些即可使用默认设置，又允许用户优先自定义

下面以 PicBox 控件为例进行讲解。

版 权 所 有 侵 权 必 究

```
/*!
 * \brief
 *  picbox_t 描述picbox的数据结构
 */
typedef struct
{
    /*! 如果id=-1，表示多帧pic,帧的列表在frame中，其他值表示res id */
    uint16 id;
    /*! picbox x坐标*/
    uint16 x;
    /*! picbox y坐标*/
    uint16 y;
    /*!
     * \li attrib<0>表示是否显示picbox，0为不显示，1为显示；
     * \li attrib<1>表示frame 的存储类型，0表示多帧（12帧以内）图片均存储在frame;
     *     1表示存储在frame[0],frame[1]指定的地方；
     * \li attrib<8-15>表示图片的帧数。
     */
    uint16 attrib;
    /*! 如果attrib<1>=0，多帧时，帧对应id列表；
     *   如果attrib<1>=1，frame[0],frame[1]为frame 的开始地址
     */
    uint16 frame[12];
} picbox_t;
```

用户指定属性
或用户优先指定属性

默认属性

说明：

1. 在这个数据结构中，如果是单帧 PicBox 控件，那么 id 属于第 3 类属性，即默认使用设置的图片 ID，但如果用户需要自己指定其他图片，可以优先指定；如果是多帧 PicBox 控件，用户需要指定第几帧，这个属于第 2 类属性，必须由用户指定。

2. 其他属性，都属于第 1 类属性，直接使用在 UI Editor 上设置的默认值。

与 ui 驱动交互：

- 控件的显示由 ui 驱动提供 *_private_t 结构体给应用，用来传递控件的第 2, 3 类属性，ui 驱动再根据控件显示机制，综合所有属性值进行解析显示。
- 为了更有效的进行控件显示，我们会给控件显示提供多个刷新模式，比如 ListBox 控件，会有刷全部，刷列表，刷激活项等。
- 所以，控件显示接口形式大致为：

  ui_show_xxx(style_infor_t *, *_private_t *, uint8 mode);

以 PicBox 为例：

1. ui 驱动提供 picbox_private_t 结构体给应用，用来传递控件的第 2, 3 类属性，该结构体如下：

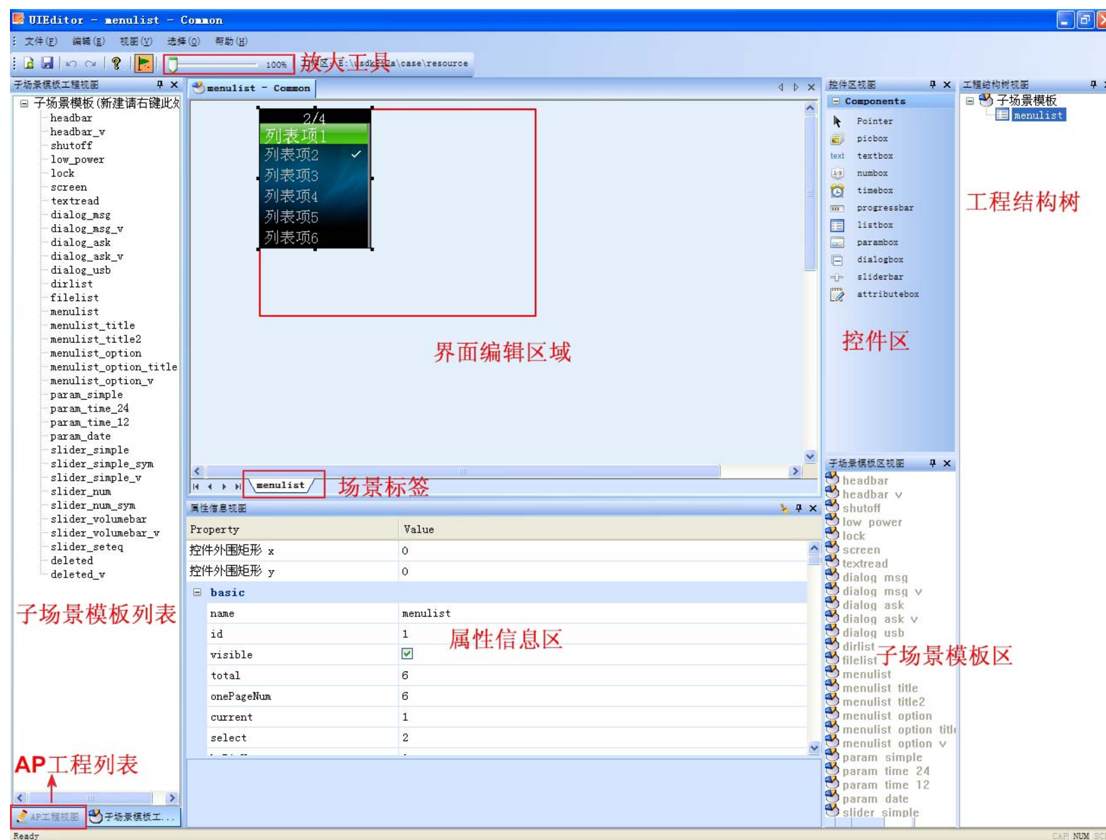
```
/*!
 * \brief
 *      picbox_private_t picbox私有数据结构
 * \note
 *      pic_id!= -1，优先显示pic_id所指向的图片；
 *      pic_id = -1，frame_id! =-1，则显示StyleID所指向的Picbox的frame_id的图片
 */
typedef struct
{
    /*! 如果pic_id不为-1，那么就是优先用pic_id显示 */
    uint16 pic_id;
    /*! 指示显示哪一帧图片 */
    uint8 frame_id;
    /*! 显示模式，分为普通模式，透明图标，垂直滚屏 */
    uint8 reserve;
} picbox_private_t;
```

2. 图片显示只有一种显示模式，无需 mode 参数。

所以，PicBox 显示接口形式为：

ui_show_picbox(style_infor_t *, picbox_private_t *);

7.2.1.3 UI Editor 工具布局



7.2.1.4 工具菜单说明

菜单		说明
文件	新建 AP 工程	新建空白 AP 工程
	关闭	关闭当前编辑的 AP 工程或子场景模板
	保存	保存当前编辑的 AP 工程或子场景模板
	另存为	只对 Common 工程有效，另存为子场景模板
	生成资源文件...	生成结果，包括*.sty 文件、*_res.h、*_sty.h 等
	生成欢迎图片资源...	生成 Welcome 欢迎图片资源文件
	图片转 24 位位图	用来把黑白 BMP 转为 24bit 深度彩色 BMP 图
	切换工作区	切换 resouce 工作目录
编辑	撤销	撤销界面编辑区域的操作
	重做	恢复界面编辑区域的操作
	剪贴	
	拷贝	
	粘贴	
	设置资源...	设置图片资源和字符串资源
	截取所有控件的背景图	更新背景图后选择该菜单，自动截取依赖于该背景图的所有背景图，从而达到快速更换背景风格的目的
视图	忽略	选择显示或隐藏视图部件
选择	自动保存设置...	设置工程自动保存的时间，默认 3 分钟自动保存
帮助	帮助手册	WORD 版本帮助手册
	关于 UI Editor...	工具关于信息

7.2.1.5 基本操作

1. 添加控件：单击控件图标，指针移到界面编辑区域就会出现控件初始效果图，移动到某个位置再单击鼠标即可；或者用鼠标拖曳控件图标到界面编辑区域也会出现控件初始效果图，移动到某个位置弹起鼠标，再单击鼠标即可。

2. 属性设置：单击工程结构树的子场景模板或 Project，属性信息区切换为工程属性列表，可以查看和设置结果文件名字和当前显示语言等；类似的单击选择 style、scene、控件，属性信息区切换为对应属性列表。

版 权 所 有 侵 权 必 究

3. 右键菜单：

右键菜单		说明
AP 工程列表中的工程	打开此工程	与双击工程一样打开
	查看 UI 文件	查看 xml 格式的 UI 脚本
	删除	删除工程，会把工程文件夹都删除掉
子场景模板列表中的子场景模板	编辑	与双击子场景模板一样打开
	新建子场景模板	直接在子场景模板列表中另存为子场景模板
	删除子场景	删除子场景模板
界面编辑区域的控件	删除	删除控件
	叠放顺序	可以选择置于顶层、置于底层、上移一层、下移一层
	截取此控件的背景图	自动截取更新控件的某些背景图
工程结构树的 style	新建场景	新建空白场景
工程结构树的 scene	删除场景	删除场景
工程结构树的控件	新建图层	只对 PicBox 有效，增加一个图层
	删除控件	删除控件
	叠放顺序	可以选择置于顶层、置于底层、上移一层、下移一层

其他操作：按 delete 按键可直接删除场景、控件、Pixbox 图层等。

7.2.1.6 工程与风格

UI Editor 上的工程分为 Common 工程和 AP 工程，其中 Common 工程用于设计子场景模板，是整个方案 UI 的基础。

UI Editor 的所有工程都放在目录 resource，每个工程放在一个子文件夹。子文件夹下存放自己工程用到的资源图片和资源字符串*.xls ，工程会生成 *.ui(*.com_ui)，*.sty，*_sty.h，*_res.h，lang_id.h 等文件。

多风格支持需要使用同样的名字而风格各异的多套资源图片，每套资源图片放在集中在一个子文件夹下；生成多风格的 *.sty 文件时，只需要切换子文件夹即可。

命名规范

图片资源文件和字符串资源 ID 的命名，建议图片资源文件加上前缀 p_，字符串资源 ID 加上前缀 s_，这样资源 ID 宏才更容易避开重名，也更容易辨认。

每个 AP 工程都会生成存放于自己源代码目录下的 *_res.h 和 *_sty.h 头文件，而每个 AP 工程还需要包含 Common 工程的 common_res.h 和 common_sty.h 头文件，所以 AP 工程的场景和控件名称，以及图片资源文件和字符串资源 ID 都不应该与 Common 工程重名。因此，建议 AP 工程的符号命名规范如下：

图片资源文件：加上 p_apname_前缀。

字符串资源 ID：加上 s_apname_前缀。

场景：加上 apname_前缀。

控件：加上 apname_scenename_前缀。

7.2.1.7 子场景模板

为了更高效开发 UI，我们会把各应用的公共场景设计为子场景模板，各应用可以直接使用。有些子场景模板由一个控件根据使用场合个性模板化而成，所以我们会把控件成为原生控件。

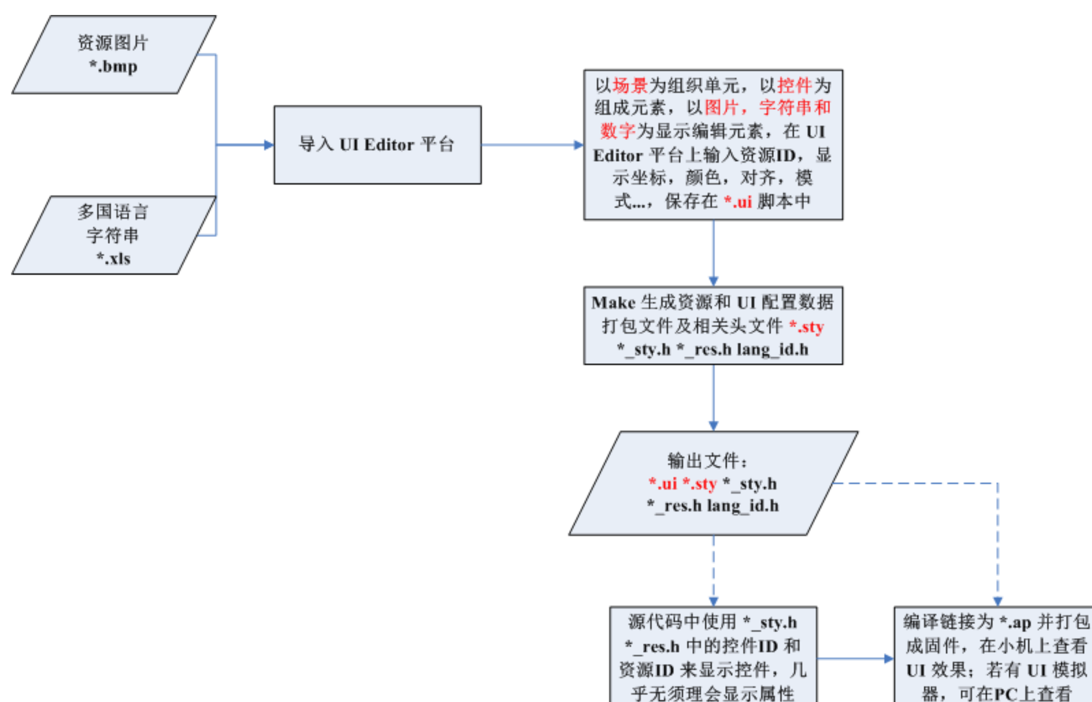
原生控件分类：

- 基本原子控件：不可再分的控件，包括 PicBox，TextBox，NumberBox，TmeBox，ProgressBar，这类控件无需控制流，直接显示即可，是 gui 的基本绘制单元。
- 综合类控件：可再细分为子部的控件，包括 ListBox，ParamBox，DialogBox，这类控件需要控制流，需要与用户交互；其子部是基本原子控件及其变体；这类控件显示时将占据整个屏幕。
- 其他：AttributeBox，该控件不能用来显示，只是用来向用户传递某些属性值（任意，可以不属于上述其他控件的属性）。

7.2.2 UI Editor 工作步骤

UI Editor 工程设计和开发的步骤如下图所示：

版 权 所 有 侵 权 必 究



- *.ui 脚本文件符合标准的 xml 文件规格，用来保存用户在 UI Editor 工具中导入的资源图片列表，资源字符串列表，以及设置的 UI 参数，其中 UI 参数按场景-控件进行组织。每个工程对应一个 *.ui 文件。
- *.sty 文件是一种自定义的二进制格式文件，是以前 *.res 文件的扩展文件，除了把工程中的资源图片和资源字符串打包进去外，还包含 UI Editor 工具上设置的 UI 参数。每个工程对应一个 *.sty 文件。

具体步骤详细描述如下：

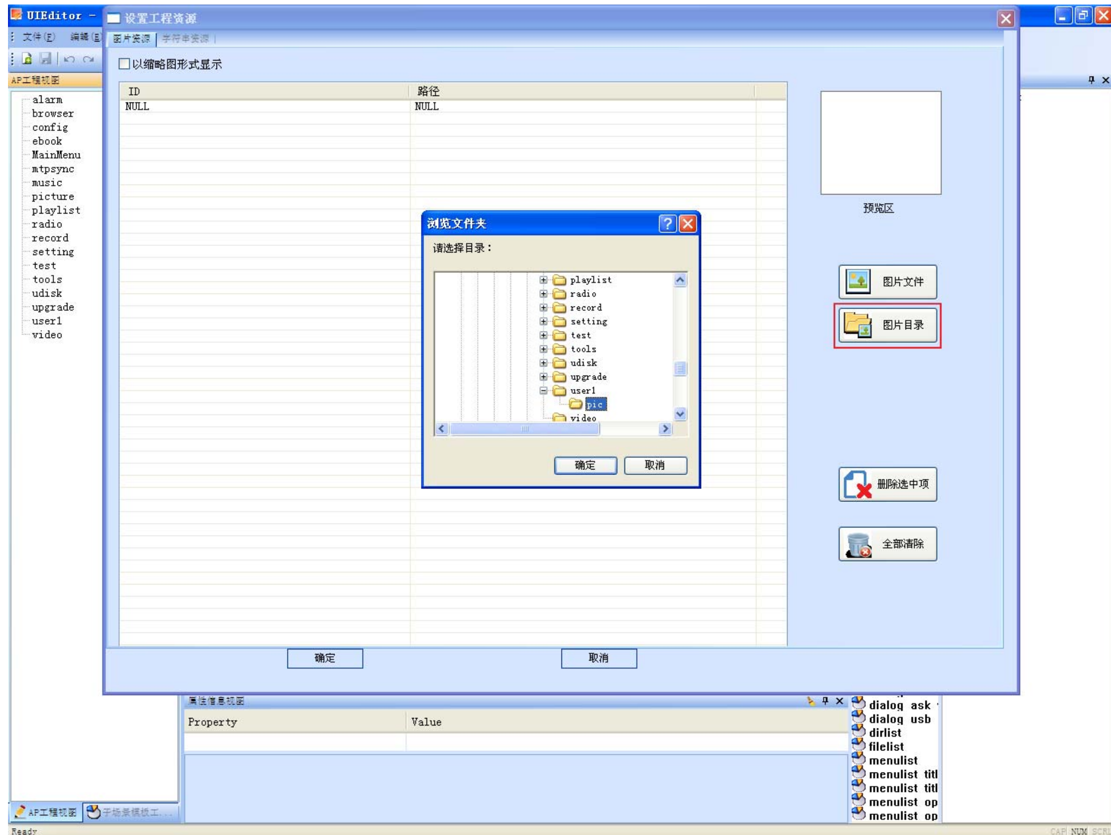
7.2.2.1 步骤 1：新建 AP 工程

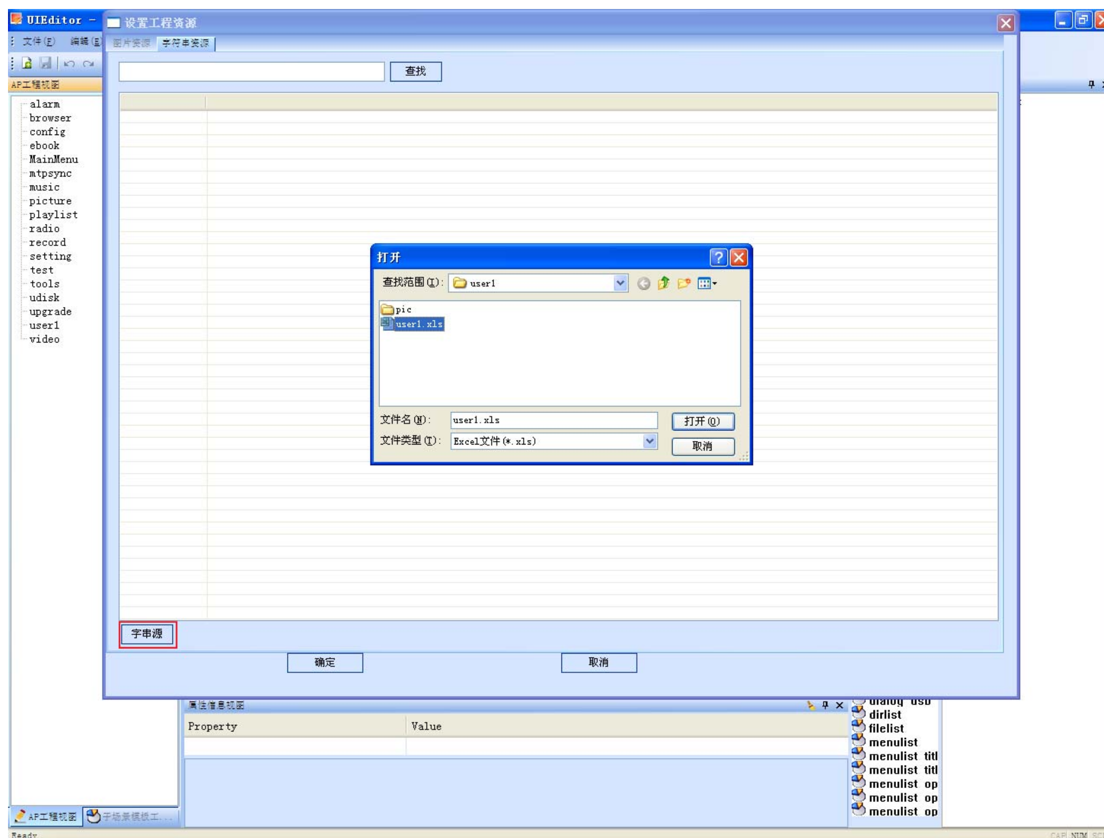
直接在 case/resource 目录下创建同工程名字一致的文件夹，或者打开 UI Editor 工具后，选择文件->新建 AP 工程创建空白 AP 工程。

7.2.2.2 步骤 2：设置资源

确定新建 AP 工程后，会自动弹出“设置工程资源”窗口；或者双击工程，如果是新工程，也会自动弹出上述窗口；又或者自己选择“编辑->设置资源...”，进入上述窗口。

然后选择图片资源文件夹和字符串资源文件 xls，完成资源设置。



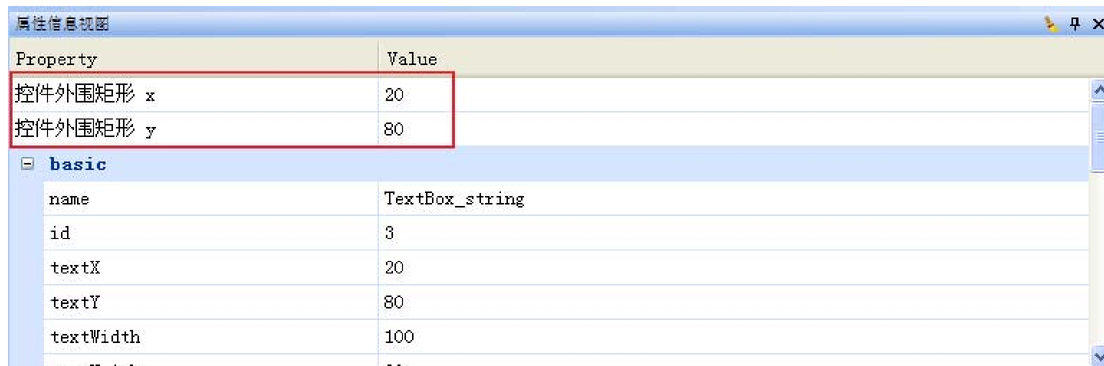


7.2.2.3 步骤 3：编辑场景 UI

按照 UI 设计添加逐个控件，配置好控件属性，即可完成场景 UI 的开发。

编辑场景 UI 有以下细节要注意：

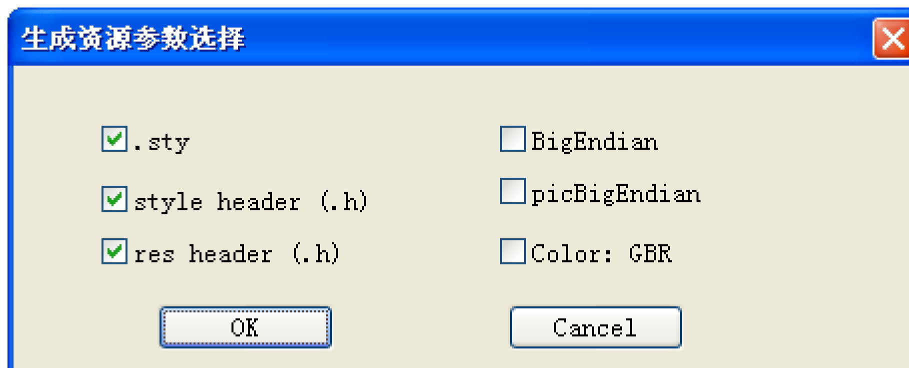
1. 每个控件添加到场景中，应该首先设置控件外围矩形 x / y 以精确定位。



2. 场景和控件的命名应该规范，易于辨认。

7.2.2.4 步骤 4：生成结果

编辑完成后，选择“文件->生成资源文件...”，生成结果。



生成结果后，执行 copyfile.bat 批处理，或者手动拷贝，把 *.sty 文件拷贝到 case/fwpkg/sty 和 case/fwpkg/ap 目录中，把 *_res.h 和 *_sty.h 拷贝到 case/ap/ 下对应源代码文件夹下。

7.2.2.5 步骤 5：调试 UI

UI 显示以及 UI 对按键输入和应用间消息的响应，是可以很轻松的在 UI Simulator 上进行调试的。

具体如何使用 UI Simulator 工具请参考 UI 模拟器 一节。

7.2.3 UI Editor 控件配置

这里并不介绍控件接口的使用方法，这部分可以查看 《us212a_ui_driver 接口说明书.chm》中对应接口说明。

注意：《us212a_ui_driver 接口说明书.chm》中的接口是 ui 驱动内部接口，接口命名和应用程序中使用的宏名字是不一样的。但是一般宏名字是在内部接口名字的前面添加 ui_ 前缀构成的，如果不是，请查找 rcode_ui_op_entry.c 中的 ui_driver_op 接口表对应命令 ID 号的接口。

7.2.3.1 PicBox 控件

控件定义

用于在指定显示区域显示资源图片的控件。

控件属性

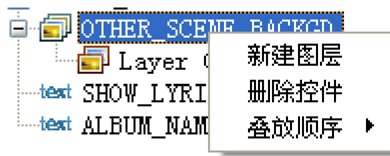
- 显示区域，只需指定 x, y 值，宽和高由资源图片宽和高决定。
- 资源图片 ID，支持多帧图片。
- 每帧图片的路径。
- 每帧图片的类型：普通图标、非嵌入式背景图、嵌入式背景图。
  - ✧ 普通图标：由用户通过 path 属性指定的图标。
  - ✧ 非嵌入式背景图：在通过普通图标确定 PicBox 的区域后，可以添加或把类型改为非嵌入式背景图；非嵌入式背景图是指按照区域直接从大背景自动截取下来的背景图，这样修改了 PicBox 的坐标后，重新选择“截取此控件的背景图”即可实现图标的更新。
  - ✧ 嵌入式背景图：在非嵌入式背景图上叠加 ForegrPic 指定的伪透明色图标，重组成一个透明效果的图标，同样，修改了 PicBox 的坐标后，重新选择“截取此控件的背景图”即可实现图标的更新。
  - ✧ 伪透明色图标：即背景部分为 ForegrbkColor 指定的颜色的图标，在图标叠加时抽取不等于 ForegrbkColor 的像素点叠加到自动截取的背景图上面。

控件使用

步骤 1：添加 PicBox 控件，修改控件名字，设置控件坐标，如果该控件显示在某个“背景”上面，那么还要选择 screenshotBgPic，作为非嵌入式背景图和嵌入式背景图的大背景图。

步骤 2：在工程结构树中，展开该控件，选择 Layer 编辑图层；如果要添加图层，右击该控件，选择添加图层；如果要删除图层，选择某个图层，按 delete 按键即可删除。

版 权 所 有 侵 权 必 究



步骤 3：编辑图层，分 3 种类型分别介绍：

普通图标：选择图标类型为普通图标，再选择图片 path

属性信息视图

Property	Value
id	0
path	.\pic\Music\musicbg_p.bmp
picType	普通图标
ForegrdPic	
ForegrdbkColor	000000

选择图片path
选择图标类型

非嵌入式背景图：选择图标类型为非嵌入式背景图，然后右击界面编辑区域中该控件，选择“截取此控件的背景图”，即自动从大背景图截取出一个背景图标，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg0.bmp”之类。

属性信息视图

Property	Value
id	1
path	.\BgPicDoc\test_play_count\test_play_count_bg1.bmp
picType	非嵌入式背景图
ForegrdPic	
ForegrdbkColor	000000

自动截取的背景图路径
选择图层类型

嵌入式背景图：选择图标类型为嵌入式背景图，选择前景图标，即伪透明色图标的路径，再指定伪透明色，然后右击界面编辑区域中该控件，选择“截取此控件的背景图”，即自动从大背景图截取出一个背景图标并与所选择的伪透明色图标叠加，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg0.bmp”之类。

属性信息视图

Property	Value
id	2
path	.\BgPicDoc\test_play_count\test_play_count_bg2.bmp
picType	嵌入式背景图
ForegrdPic	.\pic\test_count_2.bmp
ForegrdbkColor	000000

自动截取的背景图路径
选择图层类型
选择伪透明色图标
选择伪透明色

版 权 所 有 侵 权 必 究

步骤 4：修改 PicBox 的宽度和高度，暂时只能通过手动修改 *.ui （Common 工程为 common.com_ui）文件完成修改宽度和高度。即打开 *.ui 脚本，搜索控件名字，找到后手动修改 width 和 height 成员的值。

```
<element class="PictureBox">
    <property name="name" value="headbar_icon" />
    <property name="id" value="2" />
    <property name="x" value="80" />
    <property name="y" value="10" />
    <property name="width" value="20" />
    <property name="height" value="16" />    手动修改
    <property name="visible" value="0" />
    <property name="screenshotBgPic" value="" />
    <element class="layer">
```

7.2.3.2　TextBox 控件

控件定义

用于在指定显示区域显示字符串的控件。

控件属性

- 显示区域，需指定 x, y, width, height 值。
- 资源字符串 ID，可为 null。
- 字符串显示颜色。
- 字符串背景，可选背景图片或背景颜色。如果是背景图片，还要指定背景图片 ID 和显示区域。
- 字符串显示模式，包括对齐方式、长字符串显示模式、字体、阿拉伯右对齐属性等

控件使用

步骤 1：添加 TextBox 控件，修改控件名字，设置控件区域，设置背景图坐标（一般与控件坐标一致）。

属性信息视图

Property	Value
控件外围矩形 x	24
控件外围矩形 y	120
⊟ basic	
name	test_play_state
id	4
textX	24
textY	120
textWidth	80
textHeight	16
bgPicX	24
bgPicY	**120**
text	s_test_play

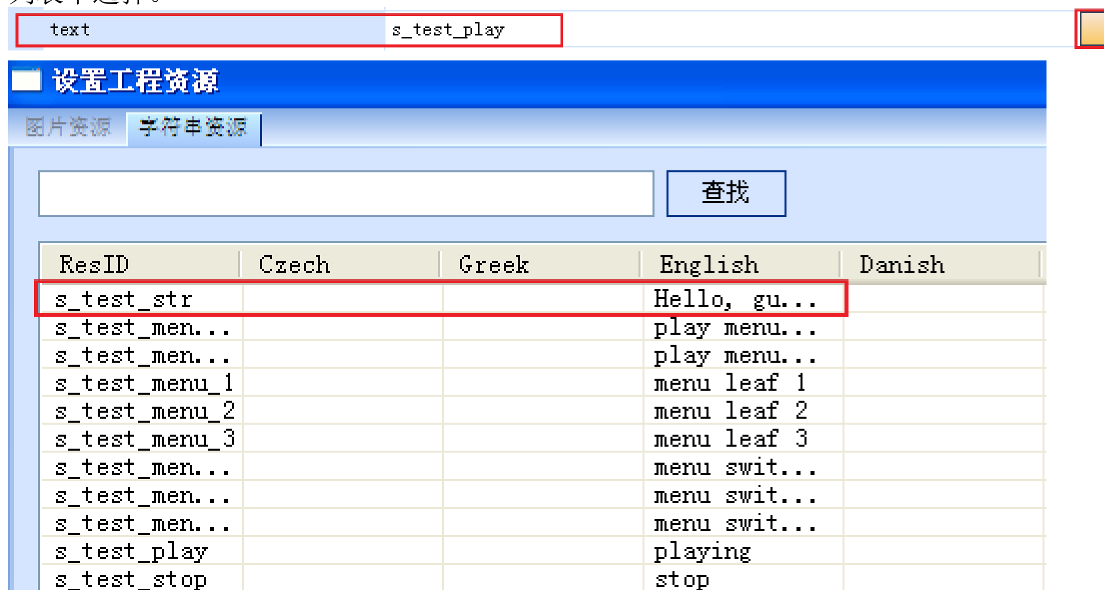
设置控件坐标

修改控件名字

设置控件区域

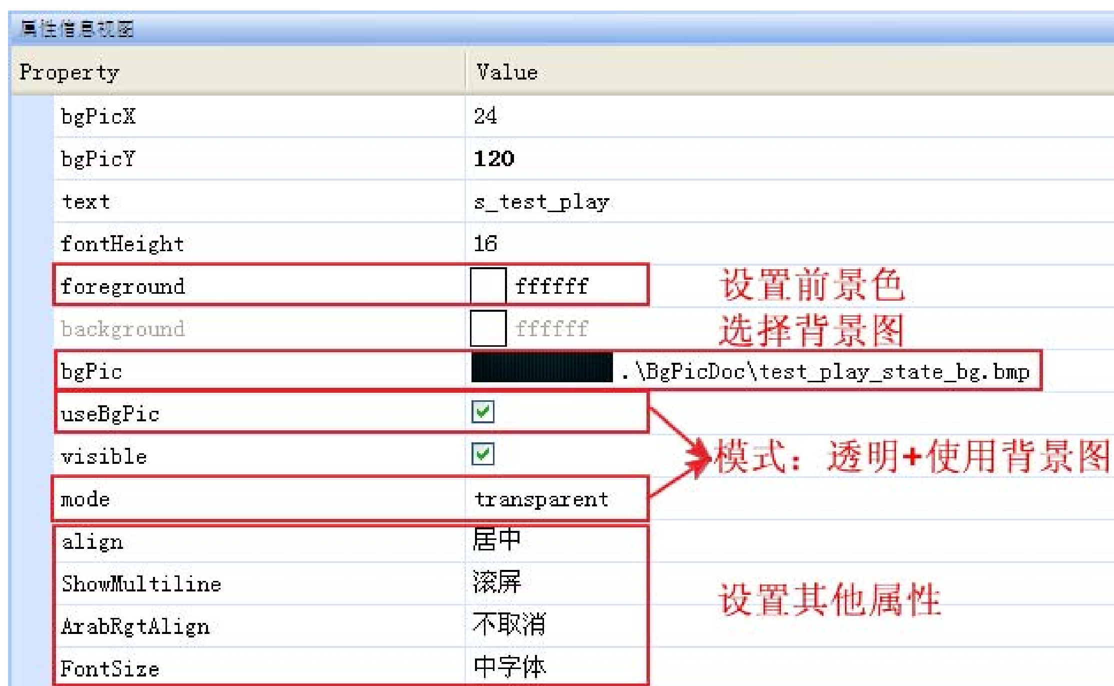
设置背景图坐标

步骤 2：选择资源字符串，即点击 text 属性框右边的选择按钮，在弹出来的资源字符串列表中选择。



步骤 3：设置 TextBox 模式，影响 TextBox 模式的属性有 2 个，即背景模式 mode 和是否使用背景图 UseBgPic ，这 2 个属性组合为 4 种情况，分别介绍如下：

背景模式 mode 选择 transparent，即透明模式，UseBgPic 选择是：选择模式为透明+使用背景图、选择背景图、设置前景色、设置其他属性。



背景模式 mode 选择 transparent，即透明模式，UseBgPic 选择否：选择模式为透明+不使用背景图、设置前景色、设置其他属性。

属性信息视图

Property	Value
bgPicX	24
bgPicY	**120**
text	s_test_play
fontHeight	16
foreground	ffffff
background	ffffff
bgPic	
useBgPic	☐
visible	☑
mode	transparent
align	居中
ShowMultiline	滚屏
ArabRgtAlign	不取消
FontSize	中字体

设置前景色

模式：透明+不使用背景图

设置其他属性

背景模式 mode 选择 normal，即纯色背景模式，UseBgPic 不可选：选择模式为纯色背景

版 权 所 有 侵 权 必 究

模式，设置前景色、设置其他属性。

属性信息视图

Property	Value
bgPicX	24
bgPicY	120
text	s_test_play
fontHeight	16
foreground	ffffff
background	ffffff
bgPic	
useBgPic	☐
visible	☑
mode	normal
align	居中
ShowMultiline	滚屏
ArabRgtAlign	不取消
FontSize	中字体

设置前景色

模式：纯色背景模式

设置其他属性

说明：

- 对于透明+使用背景图模式，可以右击界面编辑区域中该控件，选择“截取此控件的背景图”，即自动从大背景图截取出一个背景图标，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg.bmp”。
- 字符串内容预览，显示所用的语言类型是在工程属性中设置的。

属性信息视图

Property	Value
工程名字	test
case路径	E:\us212a\trunk\GL5110\uvob\usdk212a\case\
res文件路径	user1.sty
res头文件路径	test_res.h
sty文件路径	user1.sty
sty头文件路径	test_sty.h
当前显示语言	Chinese_Simplified
当前版本号	V100

7.2.3.3 NumBox 控件

控件定义

用于在指定显示区域显示指定格式数字的控件。

控件属性

- 显示区域，需指定 x, y, width, height 值。
- 数字显示模式，可选择字符串或数字图片。
- 对于数字图片模式，可以实现透明显示，即背景可以是任意图片，当然这种特性需要特殊条件支持。
- 对于数字图片模式，需要选择数字图片及其分隔符图片、符号图片；要求数字图片、背景小图标、正好图片、负号图片名字必须按字典升序排列，以保证其 ID 递增，并且图片宽度一致。
- 显示对齐属性。
- 其他见：数字显示属性。

控件使用

步骤 1：添加 NumberBox 控件，修改控件名字，设置控件区域，背景图坐标（一般和控件坐标一致）。一般来说，numWidth 为数字最大长度，这样设置对齐属性效果最佳。

属性信息视图

Property	Value	
控件外围矩形 x	38	设置控件坐标
控件外围矩形 y	4	
⊟ basic		
name	TRACK_NUMBER_BOX	修改控件名字
id	25	
numX	38	设置控件区域
numY	4	
numWidth	52	
numHeight	8	
visible	☑	
useBgPic	☑	
bgPic	.\BgPicDoc\TRACK_NUMBER_BOX_bg.bmp	
bgPicX	38	设置背景图坐标
bgPicY	4	

步骤 2：设置 NumberBox 模式，影响 NumberBox 模式的属性有 3 个，即显示模式 DisplayMode 、组成方式 Composition 和 是否使用背景图 UseBgPic，下面分别就几种组合模式进行说明：

显示模式为数字图片

这种显示模式需要设置数字图片及其分隔符图片、符号图片属性，下面就各种事情情景分别说明：

- 数字显示的大背景是纯色的
  - ✧ 如果数字总长度始终一致，可以将数字图片及其分隔符图片、符号图片设计为背景色同大背景一致，不用使用背景图 UseBgPic，不要使用“截取此控件的背景图”右键菜单。
  - ✧ 如果数字总长度会变化，除了以上说明外，还需要提供 1 张用来当总长度减少时清除痕迹的纯背景色图片，宽高同数字图片一致。
- 数字显示的大背景是透明的
  - ✧ 要求数字图片及其分隔符图片、符号图片设计为伪透明色图标，使用背景图 UseBgPic，且要使用“截取此控件的背景图”自动截取，即自动从大背景图截取出一个背景图标，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg.bmp”。
- 组成方式为由 2 个数字组成，就需要选择分隔符图片，如果是显示 1 个数字，只有小数才需要选择分隔符图片。
- 显示符号属性 ShowSign 为是，则需要选择正号图片和负号图片，如果为否，并且数字可能为负值，则需要选择负号图片。

属性信息视图

Property	Value	
⊟ **picture**		
partitionPic	.\pic\Music\slash_p.bmp	分隔符图片
pic0	.\pic\Music\num0_p.bmp	数字图片
pic1	.\pic\Music\num1_p.bmp	
pic2	.\pic\Music\num2_p.bmp	
pic3	.\pic\Music\num3_p.bmp	
pic4	.\pic\Music\num4_p.bmp	
pic5	.\pic\Music\num5_p.bmp	
pic6	.\pic\Music\num6_p.bmp	
pic7	.\pic\Music\num7_p.bmp	
pic8	.\pic\Music\num8_p.bmp	
pic9	.\pic\Music\num9_p.bmp	
picPositive		符号图片
picNegative		
picCover		数字背景图片
incX	6	数字图片宽度
partitionIncX	4	分隔符图片宽度
numBkColor	000000	伪透明色

属性信息视图

Property	Value	
numHeight	8	设置为数字图片高度
visible	☑	
useBgPic	☑	背景图
bgPic	.\BgPicDoc\TRACK_NUMBER_BOX_bg.bmp	
bgPicX	38	
bgPicY	4	
DisplayMode	数字图片	模式：数字图片
Composition	由2个数字组成	组成方式：由2个数字组成
number1	1234	
digits1	4	2个数字的位数
number2	5678	
digits2	4	
stringColor	000000	
stringBkColor	ffffff	
FontSize	中字体	
align	左对齐	对齐属性
showPartition	☑	是否显示分隔符
DecimalDigits	2	小数的小数点后位数
showSign	☐	是否显示符号

显示模式为字符串

这种情况相对简单点，不用设置数字图片等属性，简单设置高度、字符串前景色和背景色、选择字体等。

- 如果使用背景图，则可以使用“截取此控件的背景图”自动截取，即自动从大背景图截取出一个背景图标，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg.bmp”；也可以自己制定背景图。

属性信息视图

Property	Value	
numHeight	16	设置为字体高度
visible	☑	
useBgPic	☐	不使用背景图
bgPic		
bgPicX	38	
bgPicY	4	
DisplayMode	字符串	模式：字符串
Composition	由2个数字组成	
number1	1234	
digits1	4	
number2	5678	
digits2	4	
stringColor	ffffff	字符串前景色
stringBkColor	000000	字符串背景色
FontSize	中字体	选择字体
align	左对齐	
showPartition	☑	
DecimalDigits	2	
showSign	☐	

7.2.3.4 TimeBox 控件

控件定义

用于在指定显示区域显示指定格式日期或时间的控件。

控件属性

- 显示区域，需指定 x, y, width, height 值。
- 数字显示方式，可选择字符串或数字图片。
- 对于数字图片模式，可以实现透明显示，即背景可以是任意图片，当然这种特性需要特殊条件支持。
- 对于数字图片模式，需要选择数字图片及其分隔符图片；要求数字图片、背景小图标名字必须按字典升序排列，以保证其 ID 递增，并且图片宽度一致。
- 选择显示一个时间还是两个时间，这里的时间指日期或时间，由用户在调用 ui_show_timebox 时指定是日期还是时间。
- 小时或分钟高位是否自动隐藏高位 0。
- 显示对齐属性。

● 其他见：数字显示属性。

控件使用

步骤 1：添加 TimeBox 控件，修改控件名字，设置控件区域。一般来说，timeWidth 为数字最大长度，这样设置对齐属性效果最佳。

属性信息视图

Property	Value
控件外围矩形 x	21
控件外围矩形 y	123
basic	
name	MUSIC_CURTIME_TIMERBOX
id	22
timeX	21
timeY	123
timeWidth	64
timeHeight	8
visible	☑
useBgPic	☑
stringBkColor	ffffff
bgPic	.\BgPicDoc\MUSIC_CURTIME_TIMERBOX_bg.bmp
FontSize	中字体
DisplayMode	数字图片
Composition	显示2组时间

设置控件坐标

修改控件名字

设置控件区域

步骤 2：设置 TimeBox 模式，影响 TimeBox 模式的属性有 3 个，即显示模式 DisplayMode 、组成方式 Composition 和 是否使用背景图 UseBgPic，下面分别就几种组合模式进行说明：

显示模式为数字图片

这种显示模式需要设置数字图片及其分隔符图片属性，下面就各种事情情景分别说明：

- 数字显示的大背景是纯色的
  - ✧ 如果数字总长度始终一致，可以将数字图片及其分隔符图片设计为背景色同大背景一致，不用使用背景图 UseBgPic，不要使用“截取此控件的背景图”右键菜单。
  - ✧ 如果数字总长度会变化，除了以上说明外，还需要提供 1 张用来当总长度减少时清除痕迹的纯背景色图片，宽高同数字图片一致。
- 数字显示的大背景是透明的

✧ 要求数字图片及其分隔符图片、符号图片设计为伪透明色图标，使用背景图 UseBgPic，且要使用“截取此控件的背景图”自动截取，即自动从大背景图截取出一个背景图标，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg.bmp”。

- 组成方式为显示 2 组时间，就需要设置 3 个分隔符；如果为显示 1 组时间，就只需要设置 1 个分隔符；分隔符的设置分两步，先在分隔符图片区选择分隔符图片，再在分隔符类型中进行选择。

属性信息视图

Property	Value	
pic0	.\pic\Music\num0_p.bmp	数字图片
pic1	.\pic\Music\num1_p.bmp	
pic2	.\pic\Music\num2_p.bmp	
pic3	.\pic\Music\num3_p.bmp	
pic4	.\pic\Music\num4_p.bmp	
pic5	.\pic\Music\num5_p.bmp	
pic6	.\pic\Music\num6_p.bmp	
pic7	.\pic\Music\num7_p.bmp	
pic8	.\pic\Music\num8_p.bmp	
pic9	.\pic\Music\num9_p.bmp	
picNULL	.\pic\Music\num_bg.bmp	数字背景图片
picColon	.\pic\Music\doubledot_p.bmp	分隔符
picSlash	.\pic\Music\slash_p.bmp	
picDash		
picSpace		
numBkColor	000000	伪透明色

属性信息视图

Property	Value	
⊟ first		第1个时间
number1	00	
number2	00	
number3	00	
digits1	2	第一个数值数位
showHigh1	☐	第一个数值高位补0
showFirstPartition	☑	
FirstPartitionStr	冒号分隔符 :	分割符类型
⊟ second		第2个时间
number4	11	
number5	11	
number6	11	
digits2	2	
showHigh2	☐	
showSecondPartition	☑	
SecondPartitionStr	冒号分隔符 :	
⊟ middle		
showMiddlePartition	☑	
MiddlePartitionStr	斜杠分隔符 /	

其余属性同 NumberBox 基本一致，这里不再赘述。

显示模式为字符串

这种情况同 NumberBox 基本一致，这里也不再赘述。

7.2.3.5 ProgressBar 控件

控件定义

用于显示文件播放当前播放进度（或可视化比例，比如磁盘已用空间占全部空间的比例）的控件。

控件属性

- 进度条显示风格，可选择填充式，滑杆式，或两者皆有。
- 进度条背景图片 ID 及坐标。
- 进度条填充高亮图片 ID 及坐标。
- 进度条填充清除图片 ID。

版权所有 侵权必究

- 游标高亮图片 ID、坐标。（游标清除图片由自动截取的背景图提供）

控件使用

步骤 1：添加 ProgressBar 控件，修改控件名字，设置控件区域，选择进度条方向，以及选择进度条背景图片（根据进度条方向选择横向背景图片或竖向背景图片）。ProgressBar 控件的宽度和高度是由进度条背景图片和游标最小矩形包络决定的，具体细节在 步骤 2 详细说明。

属性信息视图

Property	Value
控件外围矩形 x	16
控件外围矩形 y	153
⊟ basic	
name	MUSIC_PLAY_PROGRESSBAR
id	20
visible	☑
total	100
current	50
length	112
ProgressBarStyle	填充式
horizontal	☑
bgPicX	16
bgPicY	153
bgPic	.\pic\Music\progress_bg_p.bmp

设置控件坐标

修改控件名称

选择进度条方向

选择进度条背景图片

步骤 2：设置 ProgressBar 模式，影响 ProgressBar 模式的属性有 2 个，即进度条方向 horizontal 和 进度条类型 ProgressBarStyle，下面分别就几种组合模式进行说明：

进度条方向为水平，即从左到右

- 填充式：这种模式需要选择填充高亮图片和填充清除图片，设置填充起始 X 坐标和填充起始 Y 坐标，再使用“截取此控件的背景图”自动截取，即自动从大背景图截取出一个背景图片，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg.bmp”。

  其中进度条长度 length 的计算公式为：

  length = 背景图长度 –（填充起始 X 坐标 – 背景图 X 坐标） * 2

版 权 所 有 侵 权 必 究

属性信息视图

Property	Value	
length	112	进度条长度
ProgressBarStyle	填充式	模式：水平+填充式
horizontal	☑	
bgPicX	16	
bgPicY	153	
bgPic	.\pic\Music\progress_bg_p.bmp	
useTransparentBg	☐	
progStartX	16	填充起始X坐标
fillPicY	153	填充起始Y坐标
progStartY	331	
fillPicX	388	
fillPic	.\pic\Music\progress1_p.bmp	填充高亮图片
unfillPic	.\pic\Music\progress_bg1_p.bmp	填充清除图片
tagPic		
tagPicY	153	
tagPicX	387	
scale	1	
WidgetbgPic	.\BgPicDoc\MUSIC_PLAY_PROGRESSBAR\MUSIC_PLAY_PROG...	自动截取的背景图

- 滑杆式：这种模式需要选择游标高亮图片，设置滑动杆起始 X 坐标和滑动杆起始 Y 坐标，再使用“截取此控件的背景图”自动截取，即自动从大背景图截取出一个包含背景图和游标的背景图片，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg.bmp”。

  其中进度条长度 length 的计算公式为：

  length = 背景图长度 – 游标宽度 –（填充起始 X 坐标 – 背景图 X 坐标） * 2

版 权 所 有 侵 权 必 究

属性信息视图

Property	Value	
length	111	进度条长度
ProgressBarStyle	滑杆式	模式：水平+滑动杆
horizontal	☑	
bgPicX	16	
bgPicY	153	
bgPic	.\pic\Music\progress_bg_p.bmp	
useTransparentBg	☐	
progStartX	16	滑动杆起始X坐标
fillPicY	153	
progStartY	331	
fillPicX	388	
fillPic		
unfillPic		
tagPic	.\pic\Music\progress1_p.bmp	游标高亮图片
tagPicY	153	滑动杆起始Y坐标
tagPicX	387	
scale	1	自动截取的背景图
WidgetbgPic	.\BgPicDoc\MUSIC_PLAY_PROGRESSBAR\MUSIC_PLAY_PROG...	

- 两者皆有：这种模式需要同时选择填充高亮图片和填充清除图片、游标高亮图片，设置填充和滑动杆起始 X 坐标、填充起始 Y 坐标、滑动杆起始 Y 坐标，再使用“截取此控件的背景图”自动截取，即自动从大背景图截取出一个包含背景图和游标的背景图片，放在 BgPicDoc 下以该控件名字为目录名的子文件夹下面，图片名字为“控件名字_bg.bmp”。

  其中进度条长度 length 的计算公式为：

  length = 背景图长度 – 游标宽度 –（填充起始 X 坐标 – 背景图 X 坐标） * 2

版 权 所 有 侵 权 必 究

属性信息视图

Property	Value	
length	111	进度条长度
ProgressBarStyle	两者皆有	模式：水平+两者皆有
horizontal	☑	
bgPicX	16	
bgPicY	153	
bgPic	.\pic\Music\progress_bg_p.bmp	
useTransparentBg	☐	
progStartX	16	填充和滑动杆起始X坐标
fillPicY	153	填充起始Y坐标
progStartY	331	
fillPicX	388	
fillPic	.\pic\Music\progress1_p.bmp	填充高亮图片
unfillPic	.\pic\Music\progress_bg1_p.bmp	填充清除图片
tagPic	.\pic\Music\progress1_p.bmp	游标高亮图片
tagPicY	153	滑动杆起始Y坐标
tagPicX	387	
scale	1	自动截取的背景图
WidgetbgPic	.\BgPicDoc\MUSIC_PLAY_PROGRESSBAR\MUSIC_PLAY_PROG...	

进度条方向为垂直，即从底部到顶部，与进度条方向为水平的差异在于：

- 背景图要选择竖向背景图片，填充高亮图片和填充清除图片，以及游标高亮图片也要跟着改变方向
- 填充和滑动杆的起始坐标
- 长度计算公式

7.2.3.6 ListBox 控件

控件定义

以列表形式展示选项（文件目录项或菜单项）的控件，用于文件浏览器和菜单列表等。

控件属性

- ListBox 包括主体，标题子部，列表子部，滑动条子部和 Ratio 子部。
- 主体包括背景图 ID 及坐标(一般为(0, 0))，列表行数，列表项间距。
- 标题包括标题字符串 ID 及显示区域。
- 列表包括项背景，项前置图标，项后置图标，项字符串，每一项都继续分为激活和非激活，每一项都须指定首项坐标或区域。
- 滑动条包括滑动条背景图片 ID 及坐标，滑动条游标图片 ID 及坐标。
- Ratio 类似于 NumberBox。

版 权 所 有 侵 权 必 究

7.2.3.7 ParamBox 控件

控件定义

以数字调整框或滑动条来进行参数设置的控件。

控件属性

- ParamBox 包括主体，标题子部，参数子部，单位子部和滑动条子部。
- 主体包括背景图 ID 及坐标(一般为(0, 0))，参数标志图 ID 及坐标，参数个数。
- 标题包括标题字符串 ID 及显示区域。
- 参数类似于 NumberBox。
- 单位包括单位字符串及显示区域。
- 滑动条类似于 Progressbar。

7.2.3.8 SliderBar 控件

7.2.3.9 DialogBox 控件

控件定义

用来进行消息提示或者询问用户的控件。

控件属性

- ParamBox 包括主体，选项列表子部。
- 主体包括背景图 ID 及坐标(一般为(0, 0))，对话框标志图 ID 及坐标，对话框描述说明字符串，对话框选项个数。
- 选项列表包括项背景非激活图片 ID 及坐标，激活图片 ID，项字符串显示区域(字符串内容由外部传入)。

7.2.3.10 AttributeBox 控件

版 权 所 有 侵 权 必 究

7.2.4 Common 工程说明

Common 工程是整个 case UI 的基础，几乎每个前台应用都会使用 Common 工程的控件模板，甚至一些应用就仅仅使用 Common 工程的控件模板，而不需要自己的场景 UI。

为了方便 case UI 的管理，我们把控件的 UI 设计统统都放到 Common 工程来做，也就是说，任何对控件模板的修改，都需要在 Common 工程进行。

Common 工程中的子场景模板可以分为两类：

- 公共 UI 事件场景，这种场景一般以"中断"方式插入到当前前台应用的显示中，比如音量调节，低电/充电满提示，按键锁提示，屏幕保护，USB 连接对话框，关机对话框，等等。
- ListBox，ParamBox，DialogBox 这三个综合类控件的公共模板，这些控件模板涵盖了多种不同使用场合，比如 DialogBox ，必须涵盖横竖屏、有无按钮（询问还是纯消息提示）。

7. 3 可配置化菜单

可配置化菜单使用 TreeLayer 和 Firmware Develop Kits 工具，实现可视化设计手段，操作简单。

为了方便描述，列举一些可配置化菜单的名词或概念：

菜单树：树形结构的菜单。

叶子菜单：不存在下级菜单的菜单项，比如音乐 EQ 设置的各种 EQ 选项。

根菜单：存在下级菜单的菜单项，比如音乐 EQ 设置。

入口菜单：没有上级菜单的根菜单，比如音乐播放菜单，一个入口菜单就是一颗菜单树或子树的根。

菜单资源：包括叶子菜单、根菜单和入口菜单，菜单树即由菜单资源通过菜单配置工具配置而成。其中所有叶子菜单和根菜单组成菜单项列表，所有入口菜单组成入口菜单列表。

7.3.1 需求概述

菜单可配置化是指应用开发者可以通过菜单配置工具 TreeLayer 和 Firmware

版权所有 侵权必究

DevelopKits 随意设置改变某个应用菜单树的层次结构，而不用修改和编译应用程序。

7.3.2 设计与实现要点

为了实现上面的需求，我们必须做到一点：叶子菜单的响应与其在菜单树中的具体位置无关。所以我们把叶子菜单的响应封装为回调函数，并且与叶子菜单菜单项结构体中的成员。另外，为了菜单配置工具能够读取并解释菜单资源，我们必须让菜单配置工具能够读取出其字符串编码内容，而在应用程序中我们只能够使用字符串资源 ID，所以重点要解决的问题包括：

1. 标识菜单资源：菜单树是依赖于应用程序的，应用程序最终会 make 生成*.exe 文件并打包为*.ap 文件，我们可以把菜单资源以 .rodata 段存放在这两个文件中，并且以特定的 const string 作为标志，这样菜单配置工具通过该标志就能够找到菜单资源了。其中，入口菜单列表标志为 ENTRY MENU ，菜单项列表的标志为 MENU ITEM 。

2. 菜单资源项对于工具的唯一性：入口菜单和菜单项在一个应用中可能会重名，比如菜单项“打开”，很明显应用中可能存在两个开关选项，而因为引进了动态菜单，菜单树和子树可能存在“变体”，所以入口菜单也可能会重名。所以我们菜单资源项的名字除了用于小机显示的主字符串 ID 外，还包含一个当出现重名时给工具识别的副字符串 ID。

3. 从第 2 点我们知道，工具从菜单资源项得到的用于显示的仅仅是字符串 ID，而非字符串编码，所以我们必须为工具指定字符串 ID 对应的编码来源，也就是 UI Editor 工具生成的 *.sty 文件。

7.3.2.1 菜单资源项

菜单资源项相关的数据结构如下所示：

```
//仅供应用开发人员编写菜单配置项使用，pc菜单配置工具可识别
/*!
 * \brief
 *  conf_entry_head_t 菜单配置脚本文件的入口菜单头部结构体
 */
typedef struct
{
    /*! 入口菜单标识字符串，表示入口菜单配置项的开始，默认为 ENTRY MENU */
    uint8  key string[14];
    /*! 入口菜单配置项总数 */
    uint16 total;
} conf_entry_head_t;
```

```
/*定义入口菜单*/
const conf_item_head_t entrymenu =
{ "ENTRY MENU  ", ENTRY_TOTAL };
```

```
/*!
 * \brief
 *  conf_menu_entry_t 入口菜单配置项结构体
 */
typedef struct
{
    /*! 入口菜单配置项的索引号 */
    uint16 menu_id;
    /*! 入口菜单配置项显示标题ID--主ID */
    uint16 major_id;
    /*! 入口菜单配置项显示标题ID--副ID, 如果minor_id非0,
    pc工具则使用minor_id显示, 否则使用major_id进行显示 */
    uint16 minor_id;
} conf_menu_entry_t;
```

```
/*!
 * \brief
 *  conf_item_head_t 菜单配置脚本文件的菜单项头部结构体
 */
typedef struct
{
    /*! 菜单项识别标志字符串, 表示菜单项定义的开始 */
    uint8  key_string[14];
    /*! 菜单项总数 */
    uint16 total;
} conf_item_head_t;
```

```
/*定义叶子菜单项*/
const conf_item_head_t item_head =
{ "MENU ITEM     ", ITEM_TOTAL };
```

```
/*!
 * \brief
 *  conf_menu_item_t 菜单项结构体
 */
typedef struct
{
    /*! 菜单项显示字符ID--主ID */
    uint16 major_id;
    /*! 菜单项显示字符ID--副ID, 如果minor_id非0,
    pc工具则使用minor_id显示, 否则使用major_id进行显示 */
    uint16 minor_id;
    /*! 菜单功能函数 */
    menu_cb_func menu_func;
    /*! 实时菜单回调函数,为NULL 表示非实时菜单 */
    menu_cb_leaf callback;
    /*! 菜单option的回调函数 */
    menu_cb_option menu_option;
    /*! 菜单项类型, 详细参见 menu_item_type_e 定义 */
    uint8 type;
    /*菜单项的索引号, 供菜单编辑保存时使用*/
    uint8 item_id;
    /*保留项*/
    uint16 reserve;
} conf_menu_item_t;
```

```
/*!
 * \brief
 *  menu_item_type_e: 菜单项类型枚举类型
 */
typedef enum
{
    /*! 普通菜单项 */
    NORMAL_MENU_ITEM           = 0,
    /*! 单选按钮菜单项 */
    RAIDO_MENU_ITEM            = 1,
} menu_item_type_e;
```

这里以 ap_setting 应用为例列出菜单资源列表的定义：

```
#define ENTRY_TOTAL 4
#define ITEM_TOTAL 129

/*定义入口菜单*/
const conf_item_head_t entrymenu =
{ "ENTRY MENU   ", ENTRY_TOTAL };
const conf_menu_entry_t entry[ENTRY_TOTAL] =
{
    { MENU_ENTRY_ID_MAIN, S_SETTING_TITLE, S_STORAGE_MSC },
    { MENU_ENTRY_ID_MAIN_MTP, S_SETTING_TITLE, S_STORAGE_MTP },
    { MENU_ENTRY_ID_SHUT_DOWN_ON, S_SHUT_DOWN_TIMER_TITLE, S_SHUT_DOWN_ON_TITLE },
    { MENU_ENTRY_ID_SHUT_DOWN_OFF, S_SHUT_DOWN_TIMER_TITLE, S_SHUT_DOWN_OFF_TITLE }
};

/*定义叶子菜单项*/
const conf_item_head_t item_head =
{ "MENU ITEM    ", ITEM_TOTAL };
const conf_menu_item_t item[ITEM_TOTAL] =
{
/*一级菜单项-播放模式*/
    { S_PLAY_MODE, 0, NULL, NULL, option_callback, NORMAL_MENU_ITEM ,0,0}, /*播放模式*/
/*播放模式-音乐来源（二级菜单）*/
    { S_MUSIC_FROM, 0, NULL, NULL, option_callback, NORMAL_MENU_ITEM ,1,0}, /*音乐来源于...*/
/*播放模式-音乐来源方式（三级菜单）*/
/*...来源于所有文件*/
    { S_FROM_ALL_SONGS, 0, set_music_from_all_songs_callback, NULL, option_callback, NORMAL_MENU_ITEM ,2,0},
/*...来源于此演唱者*/
    { S_FROM_ARTIST, 0, set_music_from_artist_callback, NULL, option_callback, NORMAL_MENU_ITEM ,3,0},
    ...
/*一级菜单项-声音设置*/
    { S_SOUND_SETTING, 0, NULL, NULL, option_callback, NORMAL_MENU_ITEM ,13,0 }, /*声音设置*/
/*声音设置-均衡器（二级菜单）*/
    { S_EQUALIZER, 0, get_eq_option_callback, NULL, option_callback, NORMAL_MENU_ITEM ,17,0 }, /*均衡器*/
/*声音设置-均衡器选项（三级菜单）*/
    { S_OFF, S_EQ_OFF, set_eq_normal_sure, set_eq_normal_callback, option_callback, RAIDO_MENU_ITEM ,18,0 }, /*均衡关*/
    { S_EQ_ROCK, 0, set_eq_rock_sure, set_eq_rock_callback, option_callback, RAIDO_MENU_ITEM ,19,0 }, /*摇滚*/
    { S_EQ_FUNK, 0, set_eq_funk_sure, set_eq_funk_callback, option_callback, RAIDO_MENU_ITEM ,20,0 }, /*放克*/
    ...
};
```

7.3.2.2 mcg 文件格式

可配置化菜单配置文件 *.mcg 文件相关的数据结构如下：

```
//小机内部使用的数据结构，由pc菜单配置工具生成
/*!
 * \brief
 *  menu_title_data_t 菜单头数据结构
 */
typedef struct
{
    /*! 菜单头资源字符串ID */
    uint16 str_id;
    /*! 当前菜单的菜单项个数 */
    uint16 count;
    /*! 菜单item data 开始地址 */
    uint16 offset;
    /*! 默认的菜单活动项，通过工具设置 */
    uint16 active_default;
    /*! 当前菜单的上一级菜单的索引，如果当前菜单为入口根菜单，此项为0x7fff */
    uint16 father_index;
    /*! 当前菜单列表对应的根菜单项在父菜单中的编号 */
    uint16 father_active;
} menu_title_data_t;
```

```
/*!
 * \brief
 *  menu_item_data_t 菜单项数据结构
 */
typedef struct
{
    /*! bit<15>表示菜单是否为radio按钮，0表示否，1表示是
     *  bit<14-0>表示菜单项类型：=0x7fff，表示叶子菜单；其他值表示根菜单，值表示指向的菜单头索引值
     */
    uint16 child_index;
    /*! 菜单项资源字符串ID */
    uint16 str_id;
    /*! 菜单显示字符副ID，在主ID相同的情况下，模拟器上需要使用此ID找到各个回调函数的地址 */
    uint16 str_id_sub;
    uint16 reserve;
    /*! 菜单项确定执行回调函数
     *  对于菜单头项，返回值不处理，返回 RESULT_NULL 即可；而叶子菜单项则需要谨慎选择返回值
     */
    menu_cb_func menu_func;
    /*! 即时叶子菜单回调函数，非NULL表示该叶子菜单为即时叶子菜单，NULL表示不是即时叶子菜单 */
    menu_cb_leaf callback;

    /*! PHILIPS特性支持，菜单项option回调函数 */
    menu_cb_option menu_option;
} menu_item_data_t;
```

*.mcg 文件格式如下所示：

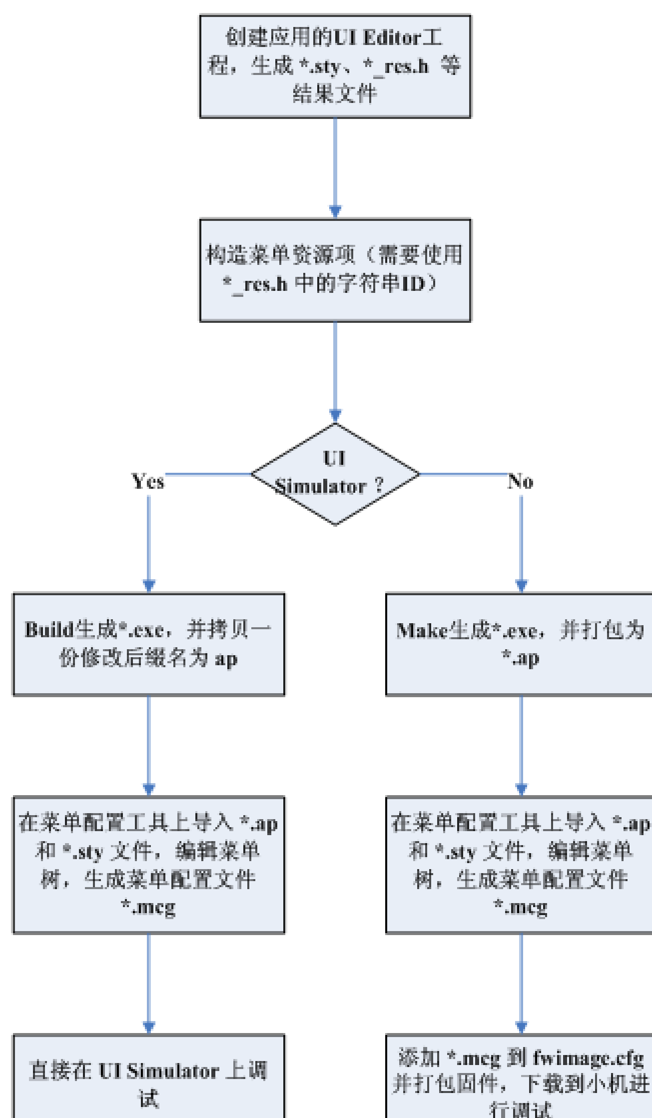
<table>
<tr><td>菜单配置文件名称（12bytes）
例如“music.mcg”</td><td>菜单列表头起始
位置偏移（2 bytes）</td><td>列 表 头 数 目
count（2 bytes）</td></tr>
<tr><td colspan="3">menu_title_data_t MENULIST 0，sizeof(menu_title_data_t) ：其中 offset 指向下面 menu_item_data_t 列表的某一项，表示该菜单列表头的第一个菜单项，并且连续 count 项组成该菜单列表。</td></tr>
<tr><td colspan="3">menu_title_data_t MENULIST 1，sizeof(menu_title_data_t)</td></tr>
<tr><td colspan="3">...</td></tr>
<tr><td colspan="3">menu_title_data_t MENULIST N, sizeof(menu_title_data_t)</td></tr>
<tr><td colspan="3">...</td></tr>
<tr><td colspan="3">menu_item_data_t MENUITEM 1, sizeof(menu_item_data_t) ：如果是根菜单项，child_index 是一个比较小的值，指向上面 menu_title_data_t 列表的某一项，这样就可以检索下一级菜单列表。</td></tr>
<tr><td colspan="3">menu_item_data_t MENUITEM 2, sizeof(menu_item_data_t)</td></tr>
<tr><td colspan="3">...</td></tr>
</table>

版 权 所 有 侵 权 必 究

menu_item_data_t	MENUITEM N, sizeof(menu_item_data_t)

7.3.3 可配置化菜单开发流程

可配置化菜单开发流程如下：



具体步骤如下：

版 权 所 有 侵 权 必 究

7.3.3.1 步骤 1：生成 *.sty 和 *_res.h

创建应用的 UI Editor 工程，导入字符串资源文件 *.xls 和图片，编辑场景 UI，生成 *.sty 、*_res.h 等结果文件。

其中，*.sty 文件是菜单配置工具的字符串编码的来源，所以必须保证 *_res.h 与 *.sty 文件是匹配的。

7.3.3.2 步骤 2：构造菜单资源项

先设计好应用程序的菜单树，确定入口菜单和动态菜单，其中每个动态菜单作为一棵菜单子树而需要一个入口菜单项。对于每个菜单项，最重要也是最难的是设计并实现其 3 个回调函数，回调函数可以为空。

完成以上步骤后，就可以按照上面 ap_setting 应用的例子编写菜单资源项列表了。

在编写菜单资源项列表时应该注意以下几点细节：

- 解决入口菜单和菜单项名字重名：可能需要额外增加本地语言的副字符串资源，这样就需要先修改应用的 *.xls 文件，并重新生成 *.sty 和 *_res.h 文件，然后才可以使用副字符串 ID。
- 叶子菜单要注意区分菜单项类型，即 NORMAL_MENU_ITEM 和 RAIDO_MENU_ITEM ，后者用于单选菜单列表，对应的菜单列表控件会在 select 项的后置图标位置上显示 select 图标，比如打勾，并且刚进入这种菜单列表会把 select 项作为激活项。
- 每个菜单项有个很特殊的成员 item_id ，是为菜单配置工具特别预留的信息，要求每个菜单项的 item_id 在菜单项列表中唯一。

7.3.3.3 步骤 3：Make 并打包为*.ap

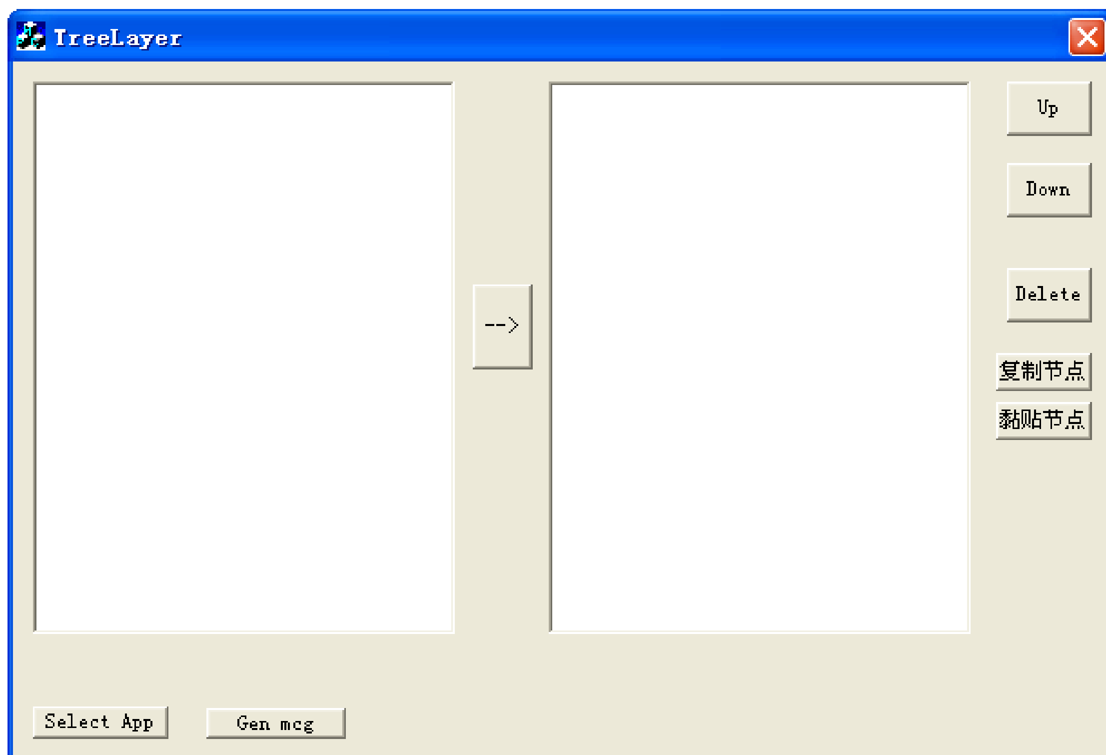
构造完菜单资源项，就可以 Make 应用程序，生成 *.exe ，并通过 ap_builder 工具打包为 *.ap 文件。菜单配置工具通过搜索 *.ap 文件中的菜单资源标志 MENU ENTRY 和 MENU ITEM ，就可以读取入口菜单头部信息及入口菜单列表，和菜单项头部信息及菜单项列表。

另外，如果是在 UI Simulator 上开发调试菜单，那具体步骤是这样的：build 应用程序工程，生成 *.exe ，直接拷贝一份并修改其后缀名为 ap，用来当做 *.ap 文件进行菜单配置。

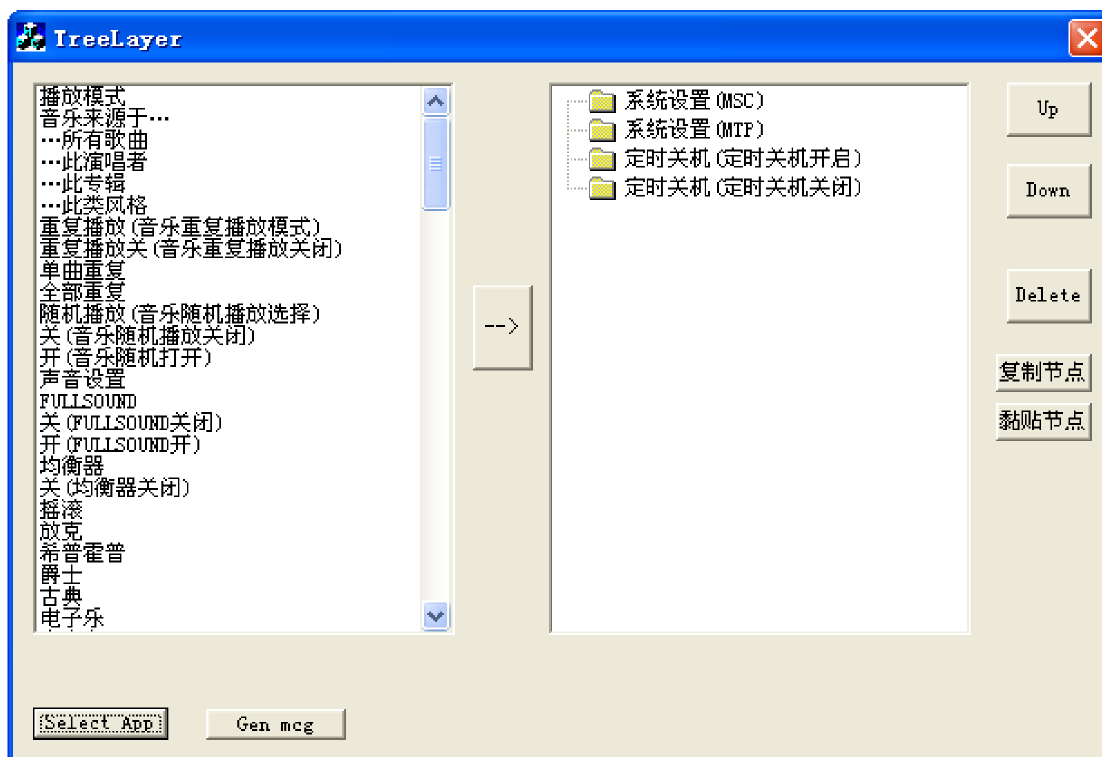
7.3.3.4 步骤 4：配置菜单树并生成 *.mcg

版 权 所 有 侵 权 必 究

打开 TreeLayer 工具，工具主界面如下如所示：



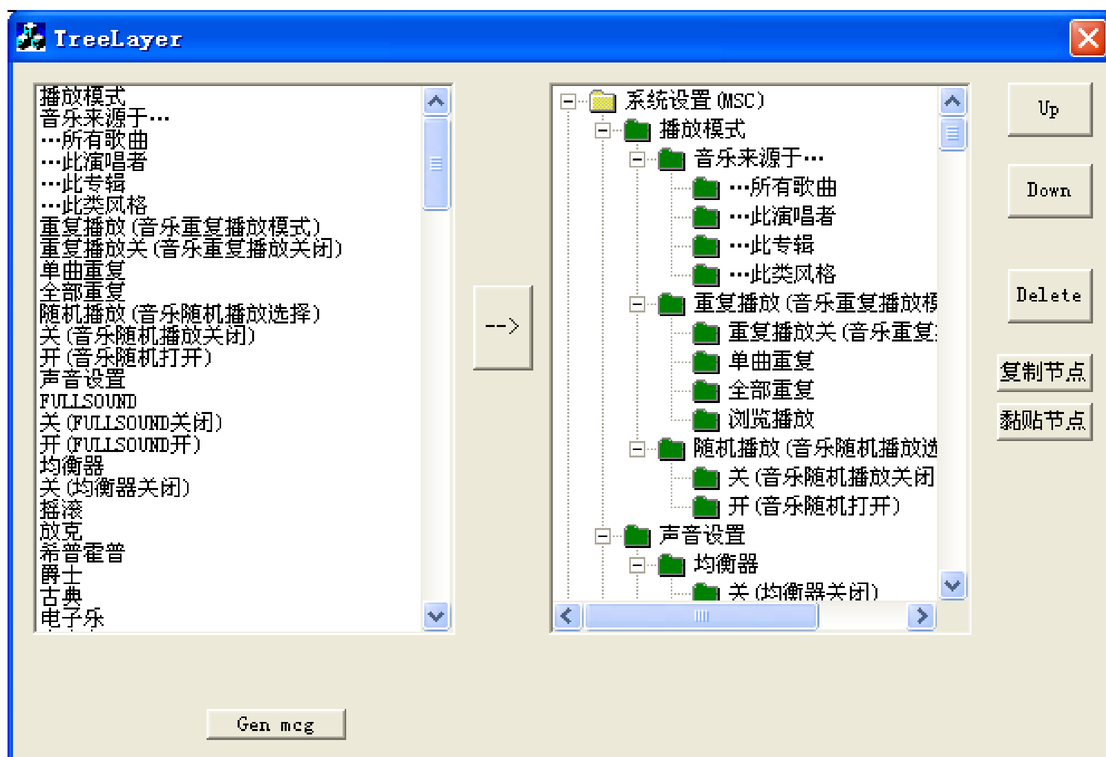
选择 *.ap 文件，工具要求 *.ap 所在目录必须存在名字一致的 *.sty 文件，然后就可以列出该应用的入口菜单列表（右边，每个入口菜单配置为一棵菜单树）和菜单项列表（左边）。



按照菜单树设计进行配置，配置操作简单介绍如下：

- 将菜单项添加到某一级菜单列表下：选择右边菜单树中的某一项，再选择左边的某个菜单项，单击中间的 --> 按钮，即可实现添加菜单项。
- 删除菜单树中的节点：选择右边菜单树中的某一项，单击右边框的 Delete 按钮，即可实现删除菜单项。
- 同级菜单列表项上下移动：选择右边菜单树种的某一项，单击右边框的 Up 或 Down 按钮，即可实现菜单项上下移动。入口菜单不能上下移动，最顶项的菜单项不能上移，最底项的菜单项不能下移。

配置好的菜单树如下所示：



最后单击底部的 Gen mcg 按钮，生成 *.mcg 文件。

注意：如果菜单配置导入的 *.ap 文件是直接从 UI Simulator 生成的 *.exe 拷贝过来的，那么最终生成的 *.mcg 文件只能作为 UI Simulator 开发调试菜单使用，并不能用来打包固件并在小机上使用。

7.3.3.5 步骤 5：调试菜单

将生成的 *.mcg 文件添加到固件打包配置文件 fwimage.cfg ，打包固件，即可在小机上调试菜单。

当然这样开发调试菜单效率会比较低。而如果是在 UI Simulator 上开发调试菜单，因为省去了生成 *.ap 文件，打包固件等步骤，并且可以借助 VC++ 强大的调试功能，效率会高很多。

当调试发现问题，或者完善菜单功能时，除了修改菜单资源项，即更新入口菜单列表和菜单项列表，需要更新 *.mcg 文件外，修改代码和修改 makefile 和 xn 脚本，也需要更新 *.mcg 文件，因为菜单项的回调函数的地址可能被修改掉了，如果没有更新，将很严重的错误甚至发生异常。

为了解决后面这个问题，我们做了两个方面的工作：

- 菜单配置工具可以更新 *.mcg ，并且提供批处理 build_mcg.bat 将所有前台应用

的 *.mcg 都更新，只要在打包固件之前执行该批处理，就可以避免该问题。

- 在 UI Simulator 上，通过实时匹配菜单项，并实时从菜单项列表中获取回调函数的地址，由于菜单项列表是应用程序的组成部分，其中的回调函数地址当然是实时的与真正的回调函数匹配，从根本上解决这个问题。但是因为这样做需要消耗很可观的内存资源和 CPU，所以这种解决方法只能在 UI Simulator 上实现。

首先在菜单资源定义文件里，添加下面 3 个实时匹配菜单项的函数：

```
#ifdef PC
menu_cb_func _get_menu_func(uint16 str_id, uint16 str_id_sub)
{
    uint16 item_cnt, i;
    item_cnt = item_head.total;
    for(i = 0; i < item_cnt; i++)
    {
        if((item[i].major_id == str_id) && (item[i].minor_id == str_id_sub))
        {
            return item[i].menu_func;
        }
    }
    return NULL;
}
menu_cb_leaf _get_callback_func(uint16 str_id, uint16 str_id_sub)
{
    uint16 item_cnt, i;
    item_cnt = item_head.total;
    for(i = 0; i < item_cnt; i++)
    {
        if((item[i].major_id == str_id) && (item[i].minor_id == str_id_sub))
        {
            return item[i].callback;
        }
    }
    return NULL;
}
menu_cb_option _get_menu_option(uint16 str_id, uint16 str_id_sub)
{
    uint16 item_cnt, i;
    item_cnt = item_head.total;
    for(i = 0; i < item_cnt; i++)
    {
        if((item[i].major_id == str_id) && (item[i].minor_id == str_id_sub))
        {
            return item[i].menu_option;
        }
    }
    return NULL;
}
```

并且需要在应用程序初始化时调用 load_menulist_sumfunc 接口将上面 3 个函数注册给 menulist 控件。

```
#ifdef PC
    load_menulist_simfunc(_get_menu_func, _get_callback_func, _get_menu_option);
#endif
```

7.3.4 可配置化菜单修改指南

7.3.4.1 增加/删除入口菜单

增加入口菜单，首先要定义一个入口菜单 ID 号，即在当前最大的入口菜单 ID 号续下去就可以了，然后填写入口菜单列表，并且把入口菜单项总数加 1，这一步很重要，如果忘记了，将不会把入口菜单项导入到菜单配置工具中。

删除入口菜单就是上述过程的逆过程，这里不再赘述。

增加/删除入口菜单后要重新配置菜单。

7.3.4.2 增加/删除菜单项

增加菜单项，首先要编写菜单项的回调函数，然后填写菜单项列表，填写时，要注意 步骤 2：构造菜单资源项 中所说的几个细节。

完整删除菜单项的操作是上述过程的逆过程，这里不再赘述。另外，我们还可以简单的在菜单配置工具上对应菜单树中删除指定的菜单项即可。

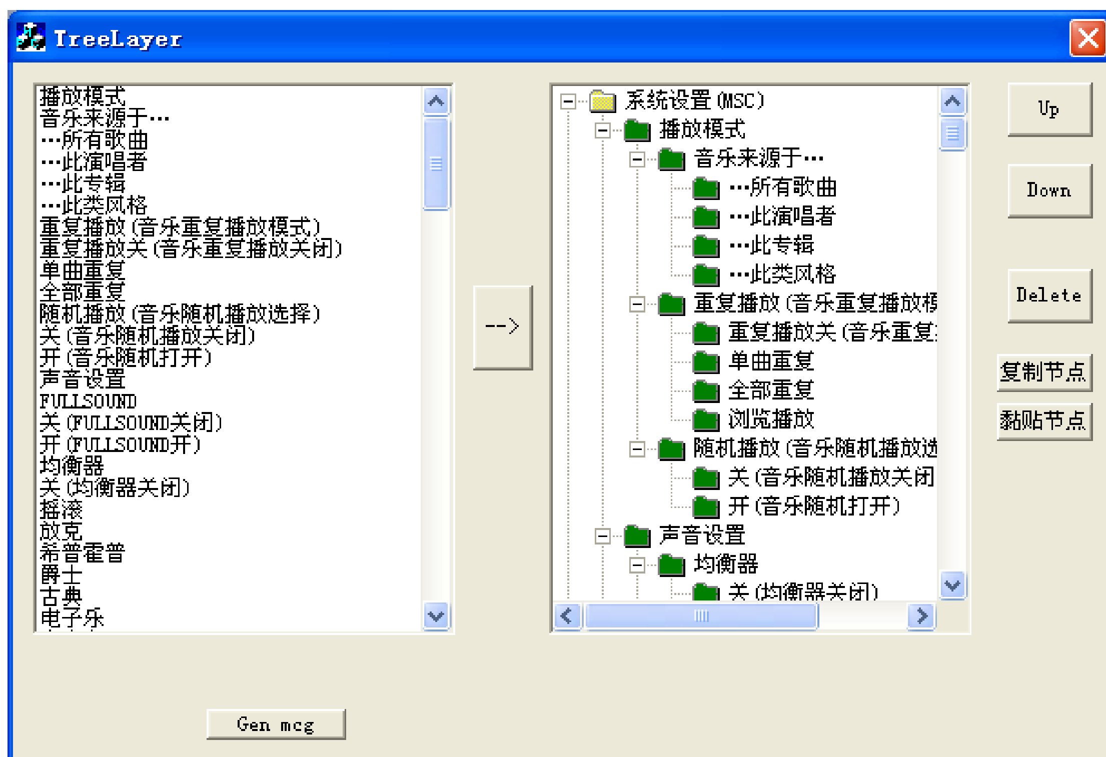
增加/删除入口菜单项后要重新配置菜单。

7.3.4.3 使用 Firmware Develop Kits 工具

在方案个性化阶段，对菜单的个性化修改，可以直接使用 Firmware Develop Kits 工具进行，操作步骤如下：

首先打开工具，选择固件：

接着选择要修改的应用程序，直接双击即可打开已经配置好的菜单树：



修改完成后，点击 Gen mcg 按钮，即可生成并自动保存更新了的 *.mcg 文件，关闭菜单配置窗口，返回主界面。

之后可以选择导出更新了的 *.mcg 文件，这样以后打包固件就可以把更新了的 *.mcg 文件打包进去；也可以选择保存固件，这样新的固件就更新了 *.mcg 文件。

8 ap_music 应用

US212A 的功能要求之一是，能够实现一边浏览图片，电子书，或者别的一些和音频无关的操作时，能够继续听音乐，也就是说，需要实现后台音乐播放的功能。为了实现这样的功能，Music 实现了播放控制应用和操作及显示的应用，其中，播放控制应用在后台运行，我们称之为音乐引擎，而操作及显示应用则作为 UI 应用，在前台运行，我们称之为 Music UI 应用。当在运行前台的 Music UI 应用的时候，用户可以主动控制音乐的播放和切换；而当在运行其它前台应用的时候，音乐引擎会根据当前的播放列表和重复方式，自动切换音乐播

放状态。

因此，音乐应用的实现，包括了两个子应用：音乐引擎应用和 Music UI 应用。

8.1 需求概述

music ui 用于实现音乐播放状态显示和参数设置等功能，完成音乐文件的列表和收藏，以及歌词、专辑图片等内容的显示，实现音乐的 ui 功能。

功能需求如下：

- ❖ 浏览功能：歌手/专辑/流派等 ID3 列表、书籍/作者等 audible 列表、目录列表浏览；
- ❖ 收藏夹功能
- ❖ 标签功能：添加/删除功能；
- ❖ 播放基本功能：播放界面文件信息、播放参数和播放状态显示和设置；
- ❖ LRC 歌词显示功能；
- ❖ album art 显示功能；
- ❖ 删除功能：目录文件及列表文件删除更新；

music 引擎用于后台的音乐播放。当界面实现其他功能时例如：播放电子书、播放图片、浏览文件、参数设置等功能时，实现音乐播放状态的自动转换，音乐文件的自动换曲等功能。

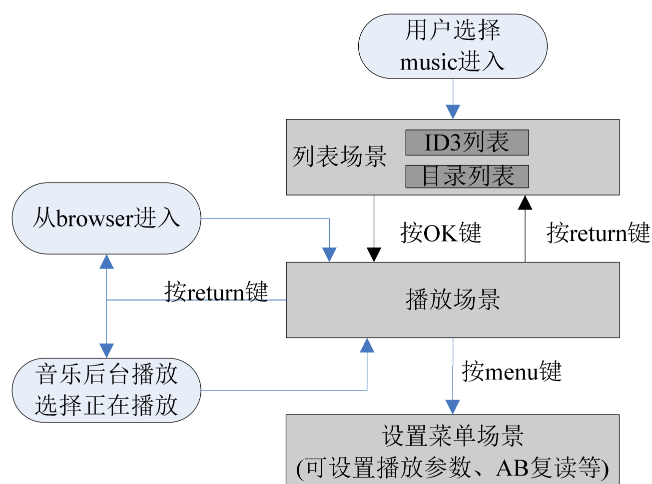
功能需求如下：

1. 能够完成音乐文件的播放。引擎播放的音乐来源有多种：播放列表、收藏夹、目录，引擎需要通过文件选择器的接口正确切换文件。
2. 能够自动切换音乐
3. 实现与 music ui 的通信，完成对 music ui 命令的响应，如接收 Music UI 的控制命令，提供当前的播放状态和相关播放信息给 Music UI 应用。音乐引擎不主动发消息给 music ui。
4. 能够完成对系统消息的响应

版 权 所 有 侵 权 必 究

8.2 总体架构设计

如下为 Music 的两个应用的结构图：



5.1总体架构图

其中 Music UI 包括列表场景，播放场景和设置场景三个主要部分，而音乐引擎应用则主要处理音乐的后台播放任务。

8.3 Music UI 应用的设计

8.3.1 Music UI 应用的功能模块划分

US212A 使用了一个设计理念：按照界面分类，对应用进行模块化的设计。每个应用，都抽象出具体的界面模块，并围绕界面模块进行资源设计，业务逻辑设计，数据流设计，消息流设计，每个界面模块都有独立的消息循化，有独立的场景资源和独立的业务逻辑。这些

版 权 所 有 侵 权 必 究

界面模块，在此我们称之为场景。

参考 Music 的架构图，Music 应用共分为三个场景：列表场景（SCENE_LSTMENU）、播放场景（SCENE_PLAYING）、设置场景（SCENE_SETMENU）。

场景名称	功能	入口函数	源文件
列表场景	音乐列表浏览功能，音乐列表分别按照 Artist，Album，Genre 进行分类，并提供了 All Songs 的列表。其中，每个分类里头，按照字母的升序进行排序。	app_result_e music_scene_playing(app_param_e enter_mode, file_path_info_t* path_locat, music_play_set_e paly_set)	Music_scene_playing.c
播放场景	音乐播放功能，包括，播放/停止控制，上一曲/下一曲，快进/快退，A-B 复读，Album art 显示，ID3 信息显示，播放模式显示，播放进度显示，A-B 复读显示等。	app_result_e music_scene_listmenu(bool menu_list, file_path_info_t* browser_path)	Music_scene_listmenu.c
设置场景	音乐播放的设置功能，设置重复方式，EQ	app_result_e music_scene_setmenu(bool music_audible)	Music_scene_setmenu.c

根据 music ui 应用的场景划分和 ap 架构，music ui 应用可以划分成如下几个模块：

模块名称	功能简述
初始化模块	对应用功能进行初始化
场景调度模块	对场景转移关系进行调度和处理
列表菜单场景	提供 ID3 列表和目录列表界面，供用户浏览音乐文件。
播放场景	提供播放界面，实现文件信息显示，文件上下首切换、暂停播放、快进快退，AB 复读等基本操作。
设置场景	在播放时按下 menu 键，进入设置菜单，实现播放参数的设置，标签设置，收藏夹设置等功能。
退出模块	退出应用的处理

版 权 所 有 侵 权 必 究

8.3.2 与其他模块的同步和交互

Ap_music 作为前台应用，在运行和系统的任务调度过程中，会涉及与 ap_manager 的通讯，与引擎应用的通讯。每个 AP 之间的通讯，其目的有如下几种：

- 使用相关 AP 的服务。比如，使用后台引擎提供的相关服务时，也是向后台引擎发送相应的消息。
- 任务调度过程中，通过消息通讯和信号量，去实现任务之间的同步和互斥。

任务之间的通讯，都通过消息传递去完成。US212A 为消息在应用之间的传递提供了如下系统接口：

```
#define sys_mq_send(a,b)                (int)sys_op_entry((void*)(uint32)(a), (void*)(b), (void*)0, MQ_SEND)
```

sys_mq_send(a,b)接口中，第一个参数是消息类型 ID，包括如下枚举类型的值：

```
MQ_ID_MNG = 0,/*进程管理应用消息队列*/
MQ_ID_DESK,/*UI（前台）应用消息队列*/
MQ_ID_EGN,/*引擎（后台）应用消息队列*/
MQ_ID_SYS,/*系统消息队列*/
MQ_ID_GUI,/*GUI 消息队列*/
```

而第二个参数是消息内容，一般是结构体 private_msg_t 类型的变量或者 msg_apps_t 类型的变量。当发送私有消息或者 GUI 消息时，使用的是 private_msg_t 类型的变量，当发送系统消息时，使用的是 msg_apps_t 类型的变量。

为了方便用户更友好的使用消息机制完成任务间通信，applib 提供了如下几个接口供用户使用:

- 同步发送私有消息队列的消息接口

bool send_sync_msg(uint8 *app_name, msg_apps_t *msg, msg_reply_t * reply, uint16 timeout) 其中 app_name 表示消息发送目标应用，msg 指向要发送的消息，replay 是消息应答结构体指针，timeout 用于控制超时发送。

- 异步发送私有消息队列的消息接口

bool send_async_msg(uint8 *app_name, msg_apps_t *msg)

其中 app_name 表示消息发送目标应用，msg 指向要发送的消息。

- 获取 gui 消息——GUI 消息队列

uint16 get_gui_msg(input_msg_t *input_msg)

其中 input_msg 是接收消息的 buffer 指针。

- 获取私有消息——私有消息队列

uint16 get_app_msg(private_msg_t *private_msg)

其中 private_msg 是接收消息的 buffer 指针。

详细用法可参考 applib 相关文档。

8.3.3 ap_music 应用依赖库及其接口说明

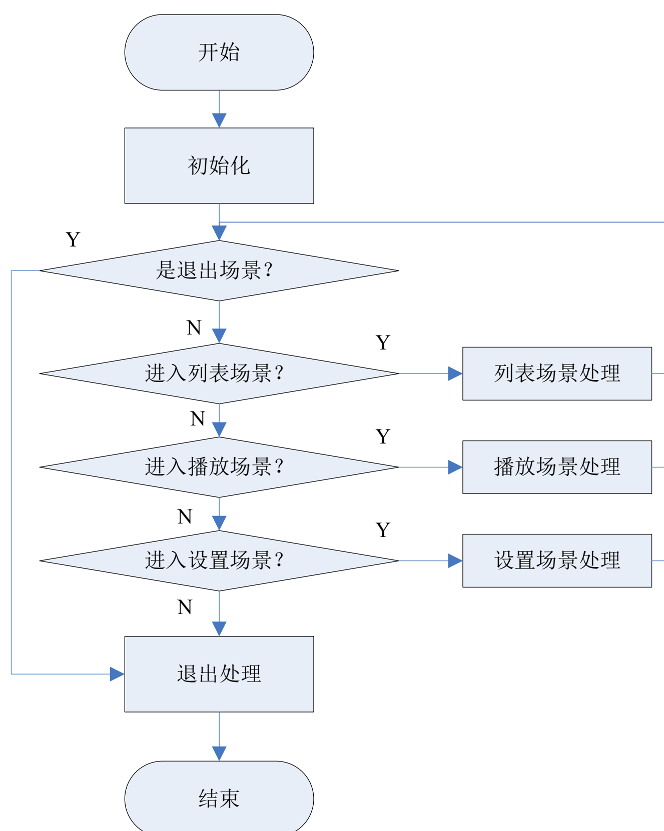
系统和 libc 的接口 api.a
应用运行时库 ctor.o
Applib 的全部函数
Common UI 的菜单，headbar，动画和其他公共提示模块
Enhanced 库中的文件浏览、列表和收藏夹、ID3、歌词等模块

8.4 ap_music 应用的业务流程

8.4.1 ap_music 应用的总流程

music ui 根据划分的场景，其总流程如下图所示:

版权所有 侵权必究



8.4.2 初始化模块的流程

Ap_music 的初始化模块，其所有的代码都编写在了接口_app_init()里头，该接口定义在文件 Music_main.c 里头。其主要功能如下：

- 装载保存在 VRAM 里头的配置数据到内存，包括全局的配置数据 VM_AP_SETTING，以及 ap_Music 专用的配置数据 VM_AP_MUSIC。其中，宏 VM_AP_SETTING 和宏数据 VM_AP_MUSIC 分别代表了 VRAM 的相对地址偏移。实现此功能的接口是_load_cfg()，此接口定义在 Music_main.c 里头。
- 加载 Style 文件，包括 ap_music 专用的 Style 文件 music.sty 和各个 AP 共用的 Style 文件 common.sty。music.sty 和 common.sty 都是系统区内的文件，打开之后，其句柄由 UI 驱动自行管理（在 UI 驱动内的 g_ap_res 变量去存储）。
- 加载菜单设计文件 music.mcg，调用的接口是 com_open_confmenu_file()，该接口

定义在 case\ap\common\common_func\common_confmenu.c 里头，加载之后的文件句柄，有 common 模块自行管理。music.mcg 是系统区内的文件，存储了 ap_music 使用的菜单配置信息。

- ❖ 初始化 ap_music 的各种全局变量，初始化软看门狗，并设定系统消息的处理句柄 applib_message_init(music_message_deal)。注：US212A 在每个 AP 里头实现了软看门狗的功能，AP 设定一个定时器，定时去喂狗，一旦由于某些特别的原因（如长时间的中断处理或者 IO 处理），导致 AP 设定的定时器没有去喂狗，则会导致系统重启。
- ❖ 创见系统计时软定时器，比如背光状态计时，standby 计时，sleep time 计时等；以及 Headbar 使用的周期更新定时器，如时间刷新，电量刷新等。
- ❖ 根据引擎的状态，设定当前正在播放的歌曲路径信息和歌曲浏览的路径，这两个信息分别保存在全局变量 g_file_path_info 和 g_browser_filepath 里头。当没有后台音乐引擎在运行的时候，则 g_file_path_info 填写了默认值：C 盘，而文件选择器模式为目录模式 FSEL_TYPE_COMMONDIR：按照目录浏览和播放。
- ❖ 根据进入 ap_music 的方式，如果是从 ap_browser 或者 ap_record 选择歌曲播放而运行 ap_music 的，由于 ap_browser 和 ap_record 在退出的时候，将歌曲文件名和路径信息存储在了 VRAM，因此在 ap_music 里头，可以从 VRAM 读取路径信息，为后续的播放作准备。
- ❖ 检查文件类型 play_scene_checktype(g_file_path_info.file_path.dirlocation.filename)，通过设定全局变量 g_audible_now，去确认是 Audible 文件的播放还是普通音乐文件的播放。
- ❖ 传入 g_file_path_info，初始化文件选择器 music_file_init(&g_file_path_info)。请注意，在 US212A 系统中，共有如下几种播放方式：（定义在文件 psp_rel\include\enhanced.h 里头）

FSEL_TYPE_COMMONDIR = 0, //目录播放

FSEL_TYPE_DISKSEQUNCE, //全盘序号播放

FSEL_TYPE_SDFILE,//sd 区文件

FSEL_TYPE_M3ULIST,//m3u 列表

FSEL_TYPE_PLAYLIST, //播放列表

FSEL_TYPE_LISTAUDIBLE, //AUDIBLE 播放列表

FSEL_TYPE_LISTVIDEO, //VIDEO 列表

FSEL_TYPE_LISTPICTURE, //PICTURE 列表

FSEL_TYPE_LISTEBOOK, //EBOOK 列表

FSEL_TYPE_LIST_PIC2, // PICTURE 表 2

FSEL_TYPE_LISTFAVOR1, //收藏夹 1

FSEL_TYPE_LISTFAVOR2, //收藏夹 2

FSEL_TYPE_LISTFAVOR3//收藏夹 3

以上所有的播放方式，其数据都是由播放列表文件去管理的，而播放列表文件的处理的接口，包括文件选择器，中间件等部分。

❖ 在用户区创建标签文件 MUSICBMK.BMK，用于存储用户创建的播放标签。

8.4.3 场景调度总流程图

us212a SDK 应用部分均是按照场景划分 ap 的模块组成，基于场景划分使得任务处理更为清晰明确。例如 music 就划分了列表，播放，设置三个场景，分别对应于 music 播放时文件浏览，文件播放以及参数设置三个方面的功能。场景调度模块的关键是对场景状态控制的一个状态机，用于实现各个场景间的转换和调度，控制应用的运行流程。该模块实现场景的调度，根据状态选择不同的场景。

一个最简单的场景调度状态机简化代码如下所示(定义在 case/ap/ap_music/music_main.c)：

```
//场景调度循环
while (g_music_scene != SCENE_EXIT)
{
    switch (g_music_scene)
    {
        //进入 listmenu 场景
```

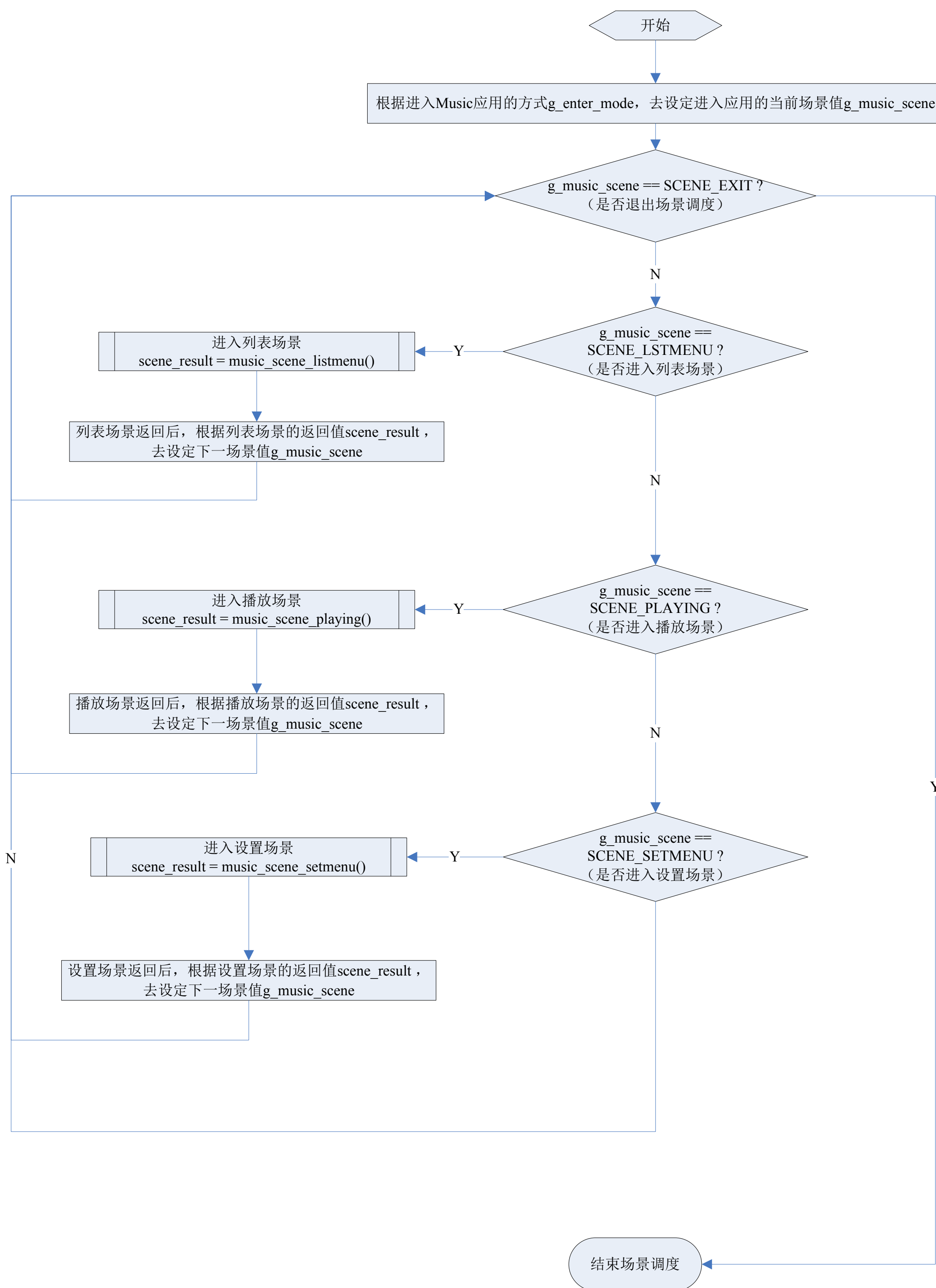
版 权 所 有 侵 权 必 究

```
            case SCENE_LSTMENU:
            //列表场景函数
            scene_result = music_scene_listmenu(list_menu, &g_browser_filepath);
            //场景返回值结果处理
            ......
            break;

            //进入播放场景
            case SCENE_PLAYING:
            //播放场景函数
            scene_result = music_scene_playing(g_enter_mode, &g_file_path_info, g_play_set);
            //场景返回值结果处理
            ......
            break;

            //进入 setmenu 场景
            case SCENE_SETMENU:
            //设置场景函数
            scene_result = music_scene_setmenu(music_audible);
            //场景返回值结果处理
            ......
            break;

            default:
            break;
    }
```

如果需要增加一个场景，只需要增加一个 case 分支；如果需要增加一个场景的返回值处理，也只需要在结束场景函数后，在返回值结果处理中增加一个分支处理。便于用户进行修改和扩展。

music 场景调度详细流程如下图所示：

版 权 所 有 侵 权 必 究

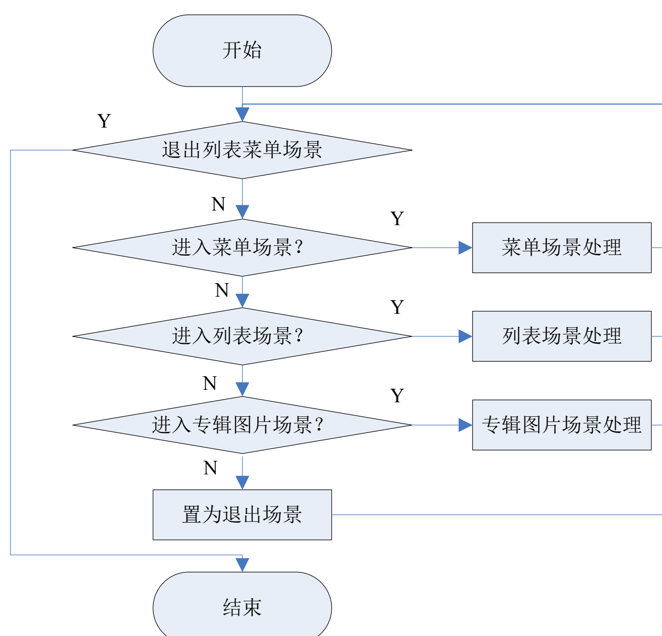


在场景调度中，最重要的变量是 g_music_scene，在以上场景调度流程中，是根据

g_music_scene 的变化，然后进入不同的场景的。g_music_scene 是个在 ap_music 里头的全局变量，在 ap_music 的任何地方，都可以访问。在其它场景退出的时候，通过设定 g_music_scene 的值，去指定退出当前场景之后，是进入哪个场景。

8.4.4 列表场景流程图

列表菜单场景是进入 music ui 的主要场景，负责菜单的处理和文件列表显示功能。列表菜单场景是由三个子控件场景组成，包括菜单控件，列表控件以及专辑图片控件。对这三个控件的调用实际上就是一个简单的场景调度，该部分的子场景调度流程图如下图所示:



可以看出，这部分的场景调度流程和 music 场景调度总流程是很相似的，事实上，该流程图可看做是对前一个流程图的简化。由于列表菜单场景只需要处理在这三个控件中的进入和退出处理，因此程序较为简单。程序在 App_music.h 定义了三个子场景状态机编号:

```
//定义 music 列表子场景所包含的控件状态机
#define     MUSIC_LISTSCENE_LIST          0x0
#define     MUSIC_LISTSCENE_MENU           0x1
#define     MUSIC_LISTSCENE_ALBUMLIST      0x2
#define     MUSIC_LISTSCENE_EXIT          0x3
```

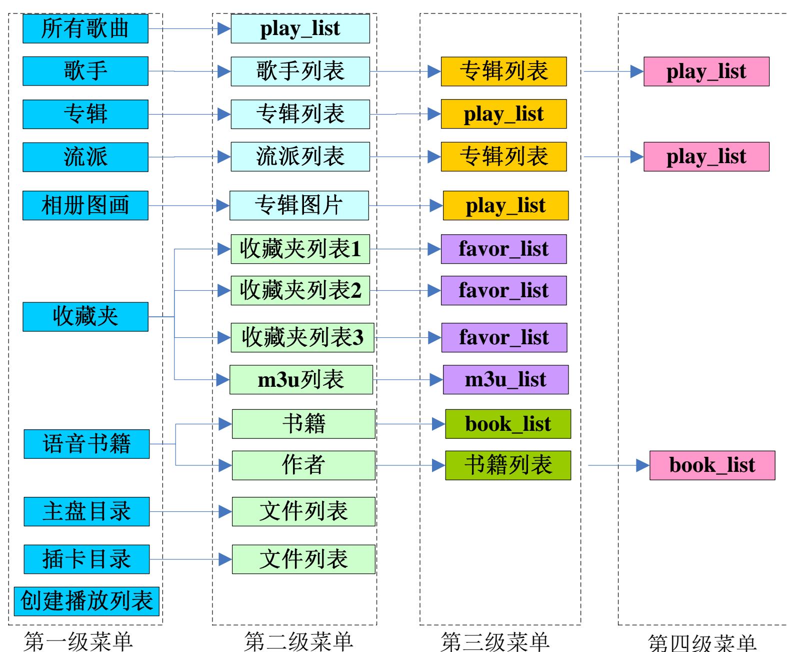
在 music_scene_listmenu.c 中实现了该状态机循环，循环部分代码如下:

```
    //场景循环
    while (sub_scene_cur != MUSIC_LISTSCENE_EXIT)
```

```
    {
        //选择子控件场景
         switch (sub_scene_cur)
        {
            case MUSIC_LISTSCENE_LIST:
            //列表场景
            ret_vals = music_scene_listmenu_list(list_browser_mode, root_layer, browser_path);
            break;

            case MUSIC_LISTSCENE_MENU:
            //菜单场景
            ret_vals = music_scene_listmenu_menu(menu_browser_mode, browser_path);
            break;

            case MUSIC_LISTSCENE_ALBUMLIST:
            //专辑图片场景
            ret_vals = music_scene_listmenu_albumlist(albumlist_browser_mode, browser_path);
            break;

            default:
            break;
        }

        switch (ret_vals)
        {
            //各个子控件返回值处理
        }
    }
```

这种设计，使得用户如果需要在列表菜单场景增加一个自己需要实现的子控件，例如这里的专辑图片子控件，就变得很容易。各个 ap 的场景调度循环是很类似的，熟悉了一个 ap 的场景调度循环，再理解其它 ap 的处理流程就很简单。

music 列表菜单场景的菜单列表结构如下图所示:



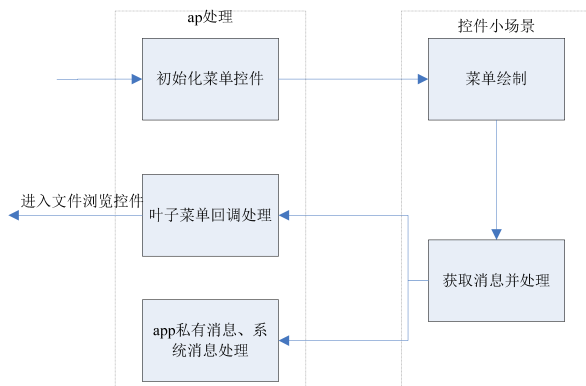
菜单列表场景主要实现音乐应用下的目录浏览和 ID3 列表（歌手 / 专辑 / 流派等列表）的分类浏览。该场景仅列出播放器的有效磁盘内的所有音频文件，返回用户在该场景下的操作结果。

该模块需要用到三个 common 控件：菜单控件，文件列表控件以及专辑图片列表控件。当前架构下，菜单为可配置结构，通过工具进行菜单树形结构设计。各种菜单的结构相似，可通过 common 控件（菜单解析器）解析。菜单解析器用于分析菜单结构，回调叶子菜单执行功能函数。菜单解析器是独立子场景，因此可以处理 gui 消息和系统消息。叶子菜单的功能可能是进行文件列表，文件列表通过文件浏览器进行解析，文件浏览器同样是独立子场景，可处理 gui 消息和系统消息。专辑图片列表控件实际是解码一类专辑的专辑图片，并以九宫格形式显示的图片控件。下面分别描述三个控件的实现过程及相关细节。

8.4.4.1 menu 控件部分

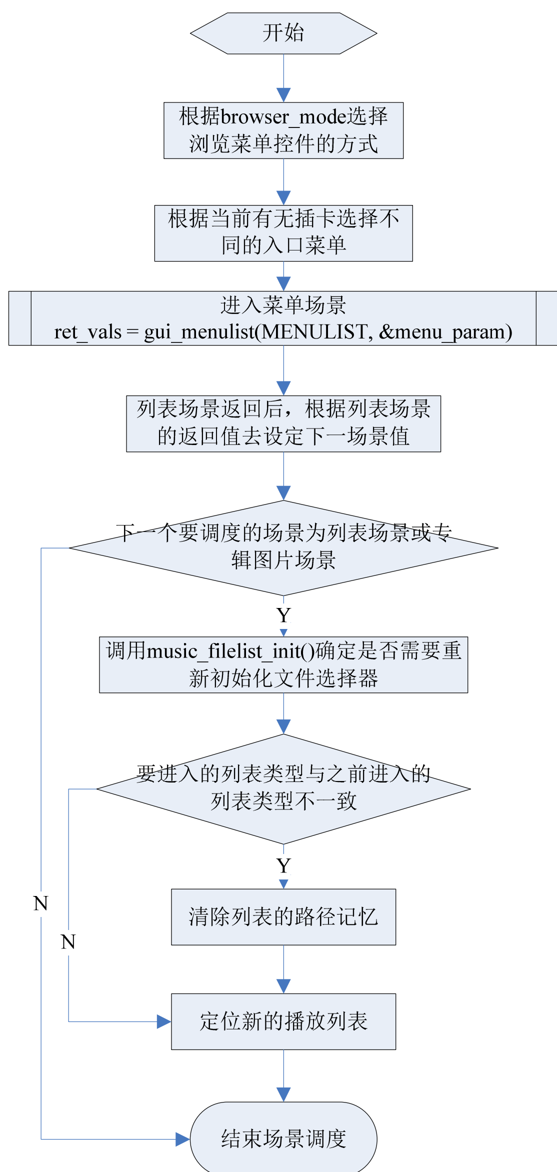
menu 控件主要从配置文件中读出菜单的树形结构，绘出菜单界面，并回调叶子菜单的回调处理。控件部分主要实现菜单的绘、按键和系统消息的处理。菜单控件部分的流程图

版 权 所 有 侵 权 必 究



从图中可以看出，ap 部分则主要完成对菜单结构的获取以及菜单处理回调的功能实现。music 菜单控件的场景流程图如下图所示:

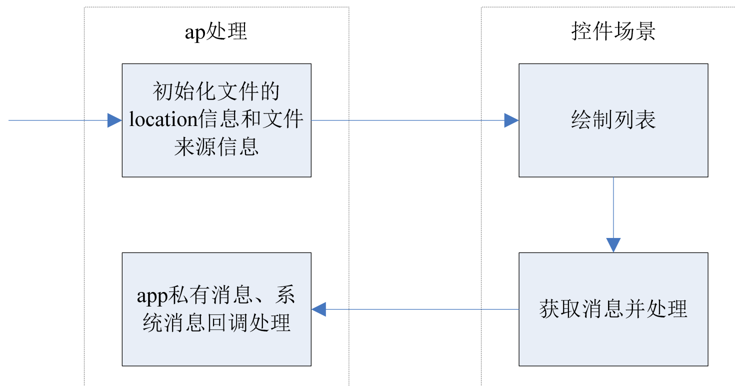
版 权 所 有 侵 权 必 究



8.4.4.2 列表控件部分

该场景下的有些叶子菜单是需要进行文件浏览的，需要调用文件浏览控件，只是各个菜单的参数不同。叶子菜单的功能是文件浏览时，叶子菜单功能处理函数需要调用文件浏览控件，进行文件浏览和 gui 消息处理。

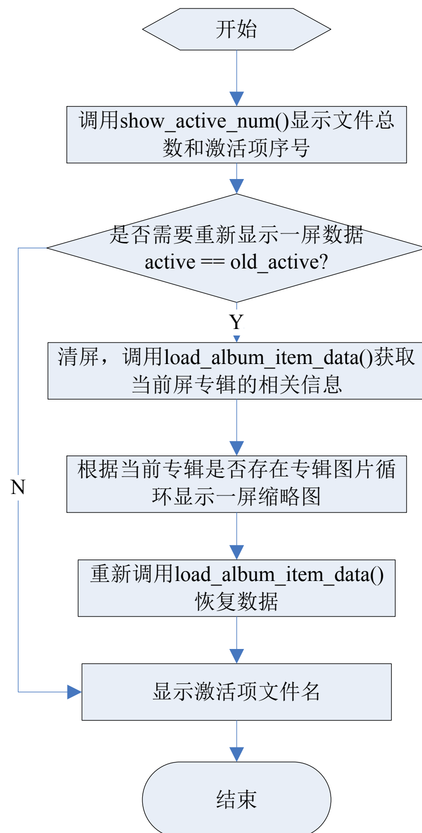
版 权 所 有 侵 权 必 究



列表控件的处理就简单一些，ap 只需要初始化 ui_directory()控件所需的数据结构，然后调用该控件进行文件浏览，根据浏览的结构进入相应的子场景。这里就不再详述。

8.4.4.3 专辑图片控件

专辑图片控件结合了 music 的专辑图片列表以及图片的缩略图实现模块，将 music 的专辑按照专辑类型，以专辑图片缩略图的形式浏览文件。关于图片解码及缩略图实现，请参考图片应用有关部分的说明。这里分析一下显示一屏专辑图片的流程，对应函数为 prev_view_one_screen()(位于 ap_music/music_listmenu_albumlist.c)



可以看出，这里文件的显示依赖于专辑图片文件信息的获取，因此重点函数为load_album_item_data(),该函数会调用 fsel_browser_get_albums()根据专辑的排序号获取当前专辑是否存在专辑图片。专辑图片的判断依据是当前专辑排序为 1 的文件是否存在命名为Foldre.jpg 的专辑图片。如果存在，则获取该图片的位置信息，并设置标志位。有关结构体为 album_brow_t,定义如下:

```
//专辑图片记录结构
typedef struct
{
    uint8 album_flag;   //是否有专辑图片标志，0 - 无  1 - 有
    uint8 name_len;      //名字的长度，单位（bytes）
    uint8 reserve[2];   //4 字节对齐
    uint8 *name_buf;     //名字缓冲区
    uint32 cluster_no; //  图片文件目录项簇号
    uint32 dir_entry;   //  图片文件目录项偏移
} album_brow_t;
```

调用 enhanced 的 fsel_browser_get_albums()就会根据需要获取的专辑序号获取当前一类专辑是否存在专辑图片，如果存在设置 cluster_no 和 dir_entry 值，同时设置 album_flag 标志。

8.4.5 播放场景流程图

Playing 场景（播放场景）是 music 应用的一个重要场景，主要实现音频文件的播放/停止、快进快退、上下首切换等基本功能，以及播放文件信息和播放状态显示和实现与用户的交互。

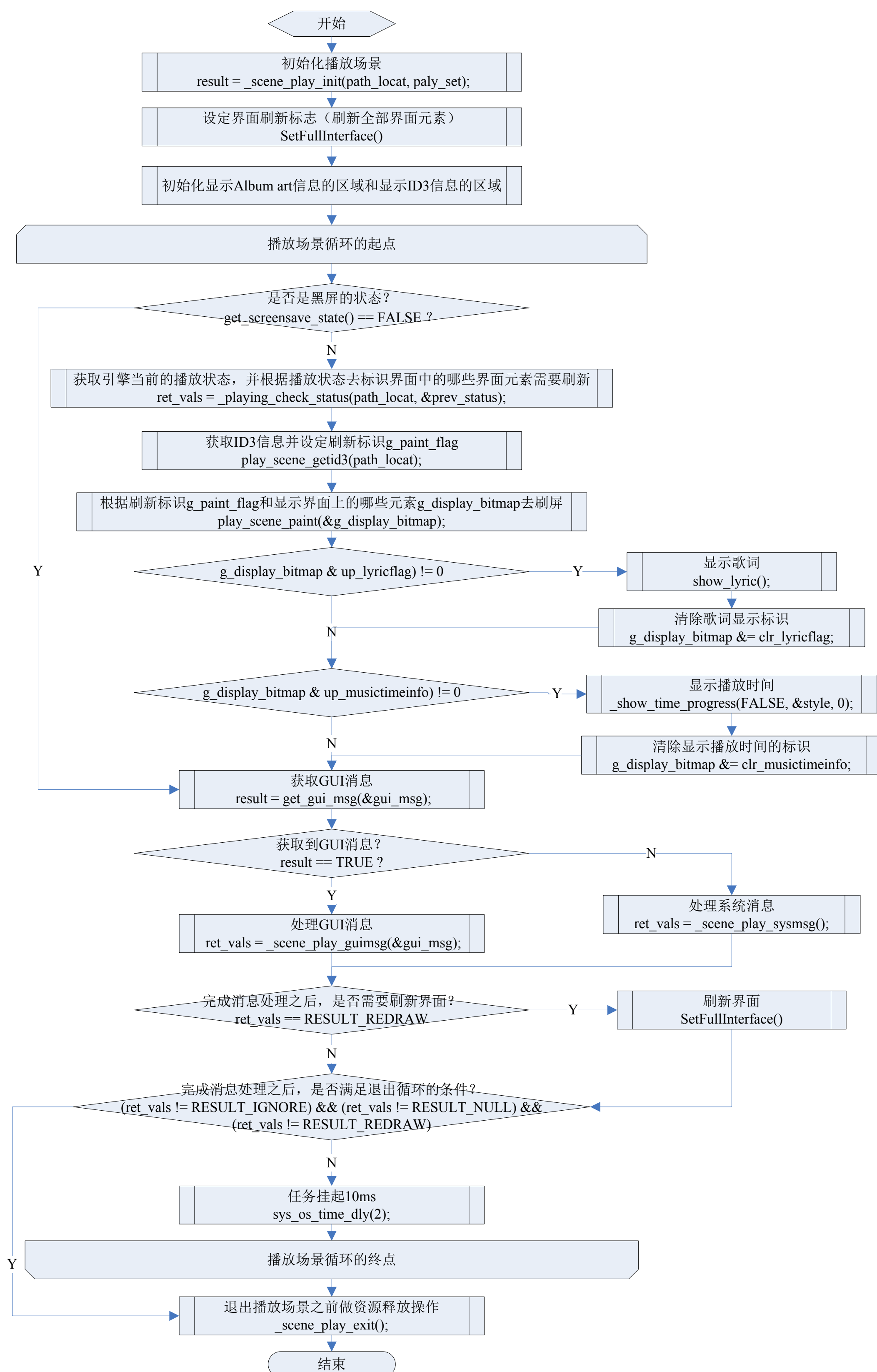
主要有如下几种方式会进入播放场景(通过 app_param_e 类型的枚举变量 g_enter_mode 去标识，app_param_e 的全部枚举值见 case_type.h 文件里头的定义)：

- ❖ 用户从列表场景中选择一首歌播放，从而进入播放场景
- ❖ 用户在 ap_browser 或者 ap_record 选择一个音乐文件播放，从而进入播放场景
- ❖ 在后台播放的时候，用户从其它界面切换到播放场景（如，从列表场景或者设置场景手动切换到播放场景）
- ❖ 在后台播放的时候，用户没有作任何操作，UI 设定的时间到了，系统自动切换回播放场景

进入播放场景，调用的接口是 music_scene_playing(app_param_e enter_mode,

file_path_info_t* path_locat, music_play_set_e paly_set)，该接口的三个参数中，enter_mode 标识进入播放场景的方式，path_local 标识播放方式和路径信息，play_set 标识进入播放场景的操作（开始播放/恢复播放/保持现状）。接口的流程图如下：

版权所有 侵权必究



版 权 所 有 侵 权 必 究

对以上流程，其主线描述如下：

- 第一步：

  首先进行初始化，调用的接口是 bool _scene_play_init(file_path_info_t* path_locat, music_play_set_e play_set)和 SetFullInterface()。以上两个接口实现了如下功能：

  （1）设定在播放场景需要使用的各个全局变量的初始值 g_audible_now，g_setting_comval 等，从而设定播放模式，重复方式，音量，EQ，播放的文件系列等等信息；

  （2）根据音频引擎互斥的一些原则，关闭了其它音频引擎，并创建了 music 的引擎 music_engine（如果 music_engine 本来就在后台运行，则无需再创建）；在创建 music_engine 的时候,将,将播放的第一首歌定位到接口 music_scene_playing()的参数里头指定的歌曲上。

  （3）如果文件选择器没有初始化，则初始化文件选择器。文件选择器没有初始化的情况，往往都是在后台没有 music_engine 运行时选择 ap_music 应用的状况。

  （4）设定需要刷新界面的所有元素（因为是第一次刷新界面），也就是设定 g_paint_flag 和 g_display_bitmap 的值。

- 第二步：

  完成初始化之后，进入播放场景的消息循环。在每一轮的循环里头，消息循环的处理顺序如下：

  （1）在背光亮着的时候，调用_playing_check_status()接口，更新播放状态；并根据播放状态的变化去设定 g_paint_flag 和 g_display_bitmap 的数值,从而设定了哪些界面元素需要刷新；

  （2）获取 ID3 信息，歌词信息等；

  （3）按照 g_paint_flag 和 g_display_bitmap 的设定刷新界面，比如歌曲的播放时间，歌曲播放进度，ID3 信息，Album Art，播放状态，重复方式等等；

版 权 所 有 侵 权 必 究

（4）读取 GUI 消息（主要是按键消息），如果读取到了 GUI 消息则调用接口 _scene_play_guimsg()进行处理，比如上下曲切换，快进快退，退出等操作，处理的同时，更新界面刷新标记 g_paint_flag 和 g_display_bitmap；如果本次循环没有读取到 GUI 消息，则调用接口_scene_play_sysmsg()读取系统消息，并处理系统消息，如 USB 拔插，SD 卡拔插，应用退出等消息。

（5）根据获取的播放状态和 GUI 消息或者系统消息的处理结果（返回值）去判断如何刷新界面元素；

（6）根据获取的播放状态和 GUI 消息或者系统消息的处理结果（返回值）去判断是否退出播放场景的消息循环；

（7）每次循环，都必须调用 sys_os_time_dly(2)，主动将应用挂起一个时间片，否则，本循环必然占用过多的 CPU 资源导致其它任务调度无法实现。

❖ 第三步：

退出播放场景的时候，释放相关的系统资源。

针对以上流程，特别需要说明的内容如下：

❖ 以上流程中，涉及关键逻辑的，有两个全局变量：g_paint_flag 和 g_display_bitmap。其中 g_paint_flag 标识是否需要刷新界面元素，而 g_display_bitmap 的每个 Bit 分别代表了需要刷新哪个界面元素。

❖ 在运行播放场景的消息循环的同时，应用尚有几个定时器也同时在运行，比如，软看门狗，背光状态计数，standby 计数器，用于字符滚动的定时器等等。

❖ 在播放场景的消息循化里头，需要与后台引擎 music_engine 打交道，是通过发送同步消息去实现的，调用的接口是 music_send_msg()，所谓的发送同步消息，指的是发送出去之后，必须等到接收方的回复才返回的消息发送。比如，在 GUI 消息处理里头，接收到歌曲切换，快进快退等消息时，会发送相应的消息给后台引擎 music_engine，后台引擎受到消息并处理完毕后，回复给发送者。

❖ 在系统消息处理里头，当接收到要求 AP 退出的消息 MSG_APP_QUIT 时，必须无条件

版 权 所 有 侵 权 必 究

返回合适的值，让 AP 退出，否则，任务调度会发生问题。

❖ 循环内部调用 sys_os_time_dly(2)挂起应用一段时间，是关键的操作。

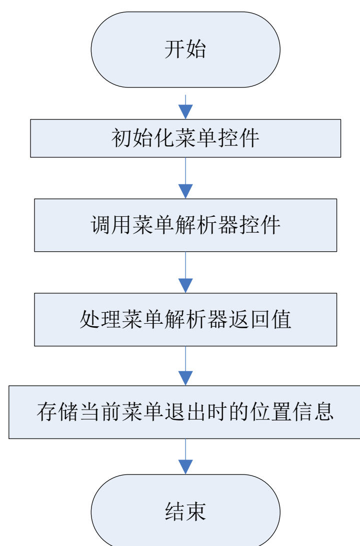
8.4.6 设置场景流程图

设置场景与列表场景类似，都是菜单处理模块，他们的不同处在于设置场景只能从播放场景进入。设置场景主要处理在播放过程中对音乐的播放模式、参数、状态等的设置，以及对当前播放音乐的收藏、设置书签等操作。场景列表如下：

版 权 所 有 侵 权 必 究

该

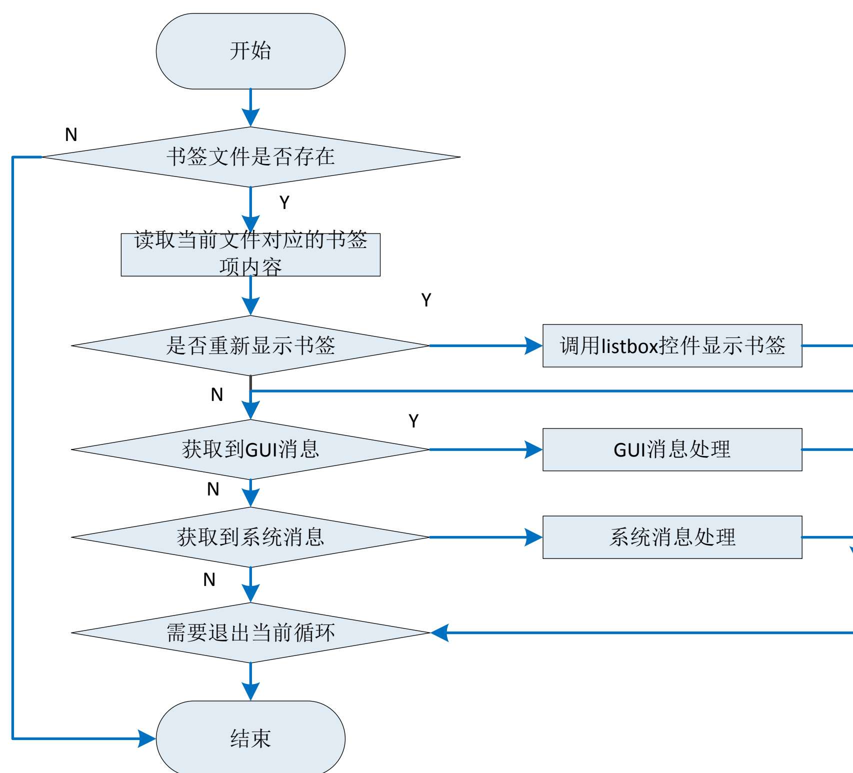
场景处理的流程示意图如下：



8.4.7 书签模块处理流程

书签模块用于记录当前 music 歌曲或 audible 歌曲的播放断点。书签模块提供对书签的添加，删除，浏览操作。添加书签将当前播放的断点信息写到相应的书签项，删除书签将相应的书签项删除，浏览书签将根据记录的断点信息设置到特定时间播放。书签模块其实就是一个子场景，其处理流程如下图所示:

版 权 所 有　侵 权 必 究



书签模块的组织结构分为三个部分，包括头部的记录信息，文件索引区和文件数据区。如下图所示:

书签头部记录有效书签总数，以及按照双向链表记录的第一个书签项和最后一个书签项的索引序号，以及删除过的书签总数，该结构体定义如下:

```
typedef struct
{
    uint16 magic; //魔数
    uint16 total_index; //书签总数
```

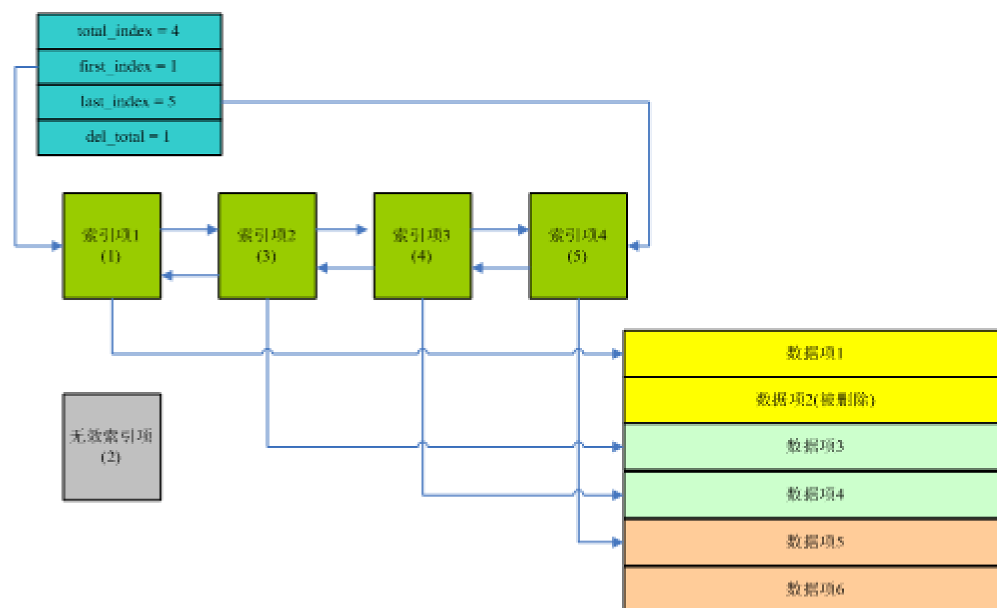
版 权 所 有 侵 权 必 究

```
    uint16 first_index;
    uint16 last_index;
    uint16 del_total; //有删除操作的书签总数
} music_bm_head;
```

魔数用于判断当前的书签文件是否有效，如果文件无效，则重新创建书签文件。书签总数记录的是有效书签总数，如果添加过 3 个文件的书签项，但又删除了两个，则 total_index 为 1，del_total 为 2。del_total 用于当索引区数据分配完后回收曾经删除过的索引项。first_index 和 last_index 用于定位索引区双向链表的头和尾。文件索引区数据结构如下:

```
typedef struct
{
    uint32 bmk_clus_no; //4bytes
    uint32 bmk_dir_eny; //4bytes
    uint16 prev;          //2bytes
    uint16 next;          //2bytes
} mbmk_index_t;//索引项数据结构
```

每一个书签文件在索引区都有这样的索引项存在。索引项的结构如下图所示:



图中形象了表示了书签文件的组织结构，索引项在索引区的位置序号代表该索引区所在数据区的数据序号。每个音乐文件的书签都占用了 256bytes 的存储空间。数据区的数据结构如下:

```
typedef struct
{
    music_bm_item mfile_bmk_items[MAX_BOOKMARKS];//160bytes
    uint8 mfile_name[64];//64bytes
```

版 权 所 有 侵 权 必 究

```
    uint8 reserved[32];
} mfile_bm_items_t; //224byte
```

数据区共记录了 10 个书签项，另外包含书签文件的文件名(ID3 文件名)。文件名在这里用于匹配当前的数据项是否是对应的 music 文件的数据项。因为完全根据索引区的 cluster_no 和 dir_entry 并不能完全表示文件的位置信息。书签项的数据结构如下:

```
typedef struct
{
    mmm_mp_bp_info_t break_param; //断点位置信息 12bytes
    time_t break_time;//断点时间信息 3bytes
    uint8 reserved;
} music_bm_item; // 16byt
```

以搜索书签索引项为例分析书签模块的工作原理。书签索引项的查找有如下几个原则:

（1） 如果当前书签文件为初始化状态，则所有数据均为 0xff,按照初始化进行分配，此时不需要查找

（2） 如果书签文件总数有效，则有两种查找模式，一种是增加书签项，此时如果查找失败，需要分配一个新的索引项，在查找的过程中如果遇到有删除的文件，回收该删除文件的索引项。另一种是删除文件时需要清除对应的书签文件，只需要查找有效的书签索引项。

对于第一种情况，在函数 get_bookmark_index,程序判断如下(music_setmenu_bookmark_deal.c)：

```
    if (bm_head_ptr->total_index != 0xffff)
    {
        //需要遍历索引项数据区
        index = find_index(cluster_no, dir_entry, bm_fp, 1);
    }
    else
    {
        //记录项为 0
        bm_head.total_index = 0;
        bm_head.del_total = 0;
        index = 0xffff;
        is_write_bmk = TRUE;
    }
```

对于第二种情况，程序在 find_index()函数，程序如下:

```
    retry: read_sector(0, bm_fp);    //从第一个索引扇区开始遍历

    bmk_index_item += 2; //每一个索引项占用 12 字节，但前面 24 字节分配给书签头部，因此加 2

    loop_cnt = bm_head.total_index + bm_head.del_total; //计算总共的书签总数
```

```
    //循环读取每一个索引项，判断是否有效
    for (i = 2; i < (loop_cnt + 2); i++)
    {
        if (search_mode == 0) //search_mode = 0 只搜索有效的书签项，对于删除过的书签项会跳过
        {
            if ((cluster_no == bmk_index_item->bmk_clus_no) && (dir_entry ==
bmk_index_item->bmk_dir_eny))
            {
                //获取数据区索引项地址
                file_index = i - 2;
                break;
            }
        }
        else
        {
            if ((0xffffffff == bmk_index_item->bmk_clus_no) && (0xffffffff ==
bmk_index_item->bmk_dir_eny))
            {
                //当前项处于删除状态，则回收利用该空闲项
                file_index = i - 2;
                bm_head.del_total--;
                break;
            }
        }

        bmk_index_item++;

        if ((i % BM_SECTOR_PER_SORT_INDEX) == 0) //如果当前扇区数据搜索完，搜索下一个扇区
        {
            file_read(bm_buf, SEC_SIZE_USE, bm_fp);
            bmk_index_item = (mbmk_index_t*) (bm_buf);
            bmk_cache_sector++;
        }
    }
```

如果找到了书签的索引项，则会将对应的数据项写入数据。写入书签项的函数为handle_bookmark(),位于 music_setmenu_bookmark_deal_sub.c.以添加书签项为例进行说明:

```
        //获取当前的播放时间
        music_get_playinfo(&play_info);
        //播放时间转换为秒为单位
```

```
time = (play_info.cur_time / 1000);
//组织书签项的时间信息
bm_item.break_time.hour = (uint8)(time / 3600);
bm_item.break_time.minute = (uint8)((time % 3600) / 60);
bm_item.break_time.second = (uint8)(time % 60);
//获取书签项的断点位置信息
music_get_bkinfo(&bm_item.break_param);
```

以上程序就得到待添加书签项的内容，下面的程序是将书签项写入文件中:

```
libc_memcpy(&(bm_data_ptr->mfile_bmk_items[active]), &bm_item, sizeof(music_bm_item));
write_sector(cur_index / 2 + BM_HEADER_SECTOR_NUM, g_bookmark_fp);
```

书签项的删除和添加类似，唯一的不同在于需要添加的文件内容为全 0xff 数据，也就是无效数据，这样就达到了删除的目的。

关于书签项的删除和删除 music 文件时删除对应的书签文件，由读者自己分析程序是如何实现的。

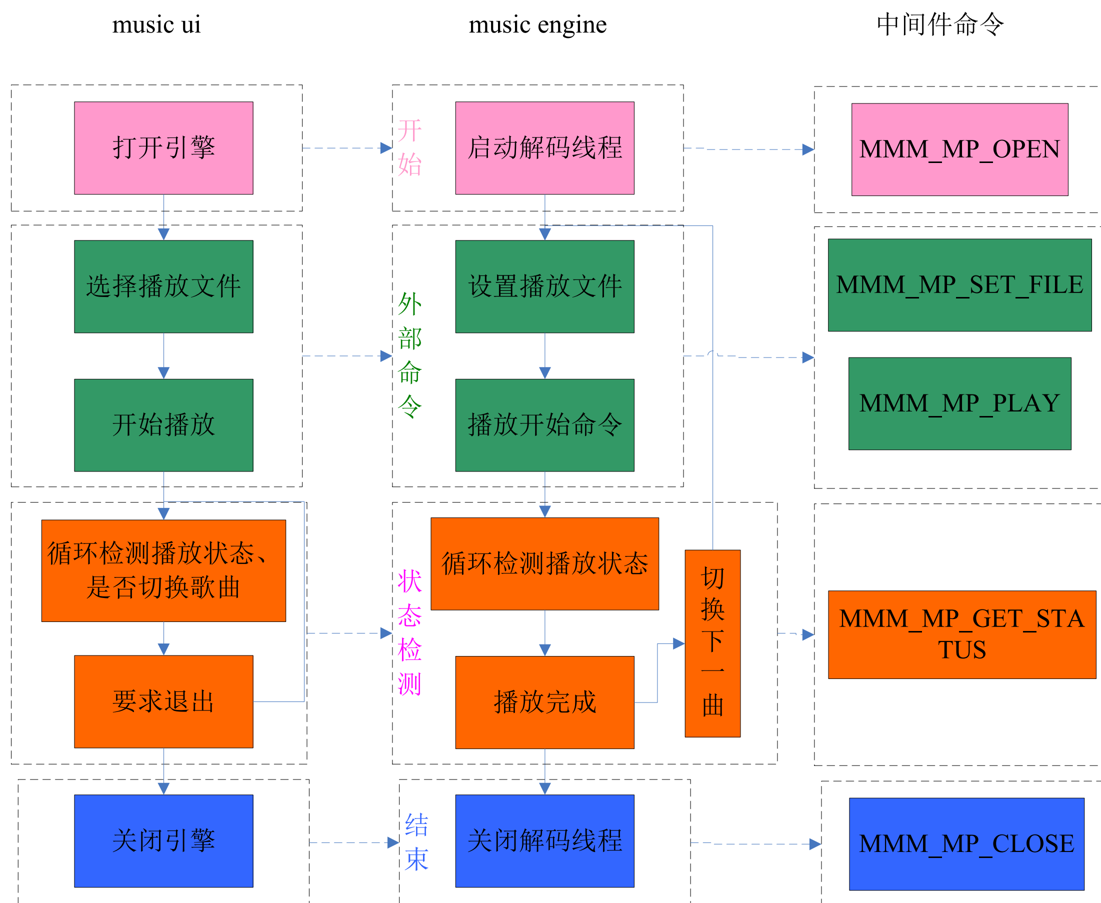
8.4.8 应用退出模块流程图

收到退出消息 MSG_APP_QUIT 后退出。该模块主要完成保存一些参数设置和断点信息以及一些资源的释放等功能。

music ui 在初始化文件选择器时，盘符是通过后台引擎获取或从其它 ap 获取，music ui 本身并不记忆盘符。如果从 music ap 退出创建播放列表后返回 music ap，需要读取之前的盘符状态，因此在创建播放列表之前，将当前的文件路径写入到 browser ap 的 VM 中，这样在从 playlist ap 退出回到 music ap 时，就可以得到之前的盘符。

8.5 播放一个音频文件的接口调用系列和顺序说明

下图描述了从列表界面选择一首歌曲播放到退出 music 应用，music ui 应用和 music 引擎应用以及解码线程的交互过程。



从图中可以看出如下几个特点:

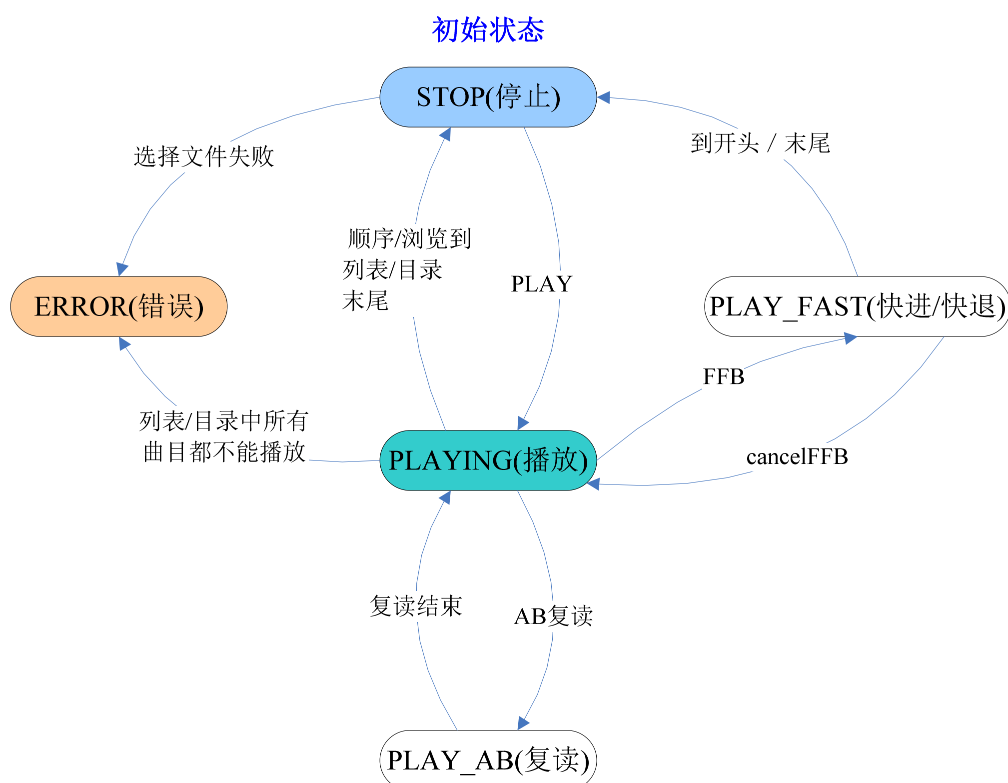
✧ 命令的发起方都是 music ui，也就是前台应用控制后台引擎的操作。例如启动解码线程，开始文件播放，获取播放状态等
✧ 三个应用(线程)实际上为单独的调度单元，优先级顺序为:解码线程优先级>music engine>music ui.这样可以保证高优先级的任务有足够的时间片响应低优先级任务的请求
✧ 前台与后台是通过发送消息的形式实现任务间通信，而后台和解码线程通过系统接口 mmm_mp_cmd，调用不同的接口命令字实现相应的功能。当需要新增加一种命令时，需要在前台与后台间增加一个消息种类，如果需要解码线程参与，同时需要新增加一个接口命令字

根据上图所示，实现文件播放具体的流程如下:

✧ 首先 music 前台向 manager 进程发送消息要求创建 music 引擎，同时将 music 引擎需要的参数传给 manager
✧ manager 在创建 music 引擎时将参数传递给 music 引擎
✧ music 引擎在创建后立即使用 MMM_MP_OPEN 命令字打开 music 解码线程，该操作实

际上是在music中间件创建一个music解码线程,music解码线程是一个独立的调度单元，也就是一个任务

- ✧ music 解码线程创建后由于优先级最高，因此立即得到调度执行。但此时无文件播放，因此等待设置待播文件播放
- ✧ 如果此时 music ui 选择文件播放，则 music ui 发送 SET_FILEPATH 消息给 music 引擎
- ✧ music 引擎收到消息后，解析文件路径，并通过 MMM_MP_SET_FILE 设置待播文件
- ✧ 解码线程收到该命令后，根据传递的参数，立即打开文件并对文件进行格式解析(format check)，获取文件格式，时间信息等
- ✧ music ui 发送 PLAY 消息给 music 引擎，music 引擎通过 MMM_MP_PLAY 播放文件
- ✧ 解码引擎收到 MMM_MP_PLAY 后真正装载解码库，设置相应的工作频率，并启动解码，此时可以听到文件播放
- ✧ 在播放过程中，对播放时间，引擎状态的获取都是由前台主动发消息进行查询的。引擎收到消息后会通过 MMM_MP_GET_STATUS 获取解码线程的状态，并据此设置引擎的状态。引擎的状态转换流程图如下图所示:



如果播放过程中或播放结束后需要退出 music 应用，则也是通过前台发送消息到后台，后台收到消息后关闭解码线程，然后再退出。manager 查询到后台退出后会回复消息给前台，前台再自行退出并创建新的前台应用。

下面从 music 解码线程，music 引擎以及前台 ap 列举实现歌曲播放的函数调用顺序，用户可以据此实现自己的个性化需求。

```
1 int main(int argc, char *argv[])
2 {
3     void *mp_handle = NULL;
4     mmm_mp_aout_setting_t music_aout_setting;
5     mmm_mp_fs_para_t fs_para;
6     mmm_mp_file_info_t fileinfo;
7     mmm_mp_status_t music_status;
8     char filename[PATH_MAX] = "demo.wav";
9     mmm_mp_err_t mp_err;
10    int ret = 0;
11
12    //通过加载MMM_MP_API，可以调用mmm_mp_cmd                装载解码线程
13    sys_load_mmm("mmm_mp.al",1);
14
15    ret = mmm_mp_cmd(&mp_handle, MMM_MP_OPEN, NULL);
16    if (mp_handle == NULL)                                 打开解码线程，
17    {                                                      获取解码句柄
18        goto exit1;
19    }
20
21    music_aout_setting.drive_mode = 0;
22    music_aout_setting.default_volume = 24;
23    music_aout_setting.default_samplerate = 44;            打开声音输出
24    music_aout_setting.default_channels = 2;
25    ret = mmm_mp_cmd(mp_handle, MMM_MP_AOUT_OPEN, (unsigned int) &music_aout_setting);
26    if (ret != 0)
27    {
28        goto exit1;
29    }
30
31
```

上述 11-30 行程序为解码库初始化函数，相应的程序位于 mengine_main.c 的_app_init(),程序首先装载解码线程，然后通过 MMM_MP_OPEN 命令字打开解码线程，并获取了该线程的句柄。此后与该线程的交互都是通过句柄完成的。最后打开声音输出，此操作为非必须完成的操作，因为在播放文件时，仍然会根据文件的播放类型设置 EQ,音量等参数。

```
32    fs_para.fs_type = 1;//比如sd文件系统
33    fs_para.file_mount_id = 0;                             设置播放文件
34    fs_para.file_name = filename;
35    ret = mmm_mp_cmd(mp_handle, MMM_MP_SET_FILE, (unsigned int) &fs_para);
36    if (ret != 0)
37    {
38
39        goto exit2;
40    }
41    ret = mmm_mp_cmd(mp_handle, MMM_MP_MEDIA_INFO, (unsigned int) &fileinfo);
42                                                           获取文件信息并播放
43    ret = mmm_mp_cmd(mp_handle, MMM_MP_PLAY, NULL);
44    if (ret != 0)
45    {
46        goto exit2;
47    }
```

32-35 行为设置播放文件，这里给出的示例程序是播放 SD 区文件，因此直接设置播放文件即可。对于普通的音乐文件，则需要初始化文件选择器，至少需要装载文件系统和相应的设

备驱动。在 music 引擎中实现了这一函数，mengine_set_filepath()(位于 mengine_event.c),前台通过消息将文件路径传递给后台引擎后，引擎根据相应的判断确定是否需要初始化文件选择器。在这里，这一操作被省略。设置完播放文件后，第 41-43 行获取文件信息并通过 PLAY 命令播放文件。在 music 引擎中相应的函数为_set_file()和_play()(位于 mengine_play_deal.c)。

```
49      while (1)
50      {
51          mmm_mp_cmd(mp_handle, MMM_MP_GET_STATUS, (unsigned long) &music_status);
52          if (music_status.status == MMM_MP_ENGINE_ERROR)
53          {
54              goto exit2;
55          }
56          if (music_status.status == MMM_MP_ENGINE_STOPPED)
57          {
58              break;
59          }
60          else
61          {
62              //mdelay(100);
63          }
64      }
65
```

49-64 行为播放时的状态循环，这里一直在查询解码线程的状态，如果解码线程为出错或停止状态(歌曲播放结束)，则结束状态循环。在 music 引擎中有类似的状态循环，函数为 mengine_status_deal()(mengine_control.c),该函数查询解码线程状态并根据不同的状态做出处理。该函数可看作对上述函数的扩充和完善。

```
66      mmm_mp_cmd(mp_handle, MMM_MP_STOP, NULL);
67      mmm_mp_cmd(mp_handle, MMM_MP_CLEAR_FILE, NULL);
68      mmm_mp_cmd(mp_handle, MMM_MP_AOUT_CLOSE, NULL);
69      mmm_mp_cmd(mp_handle, MMM_MP_CLOSE, NULL);
70      //卸载MMM_MP_API
71      sys_free_mmm(1);
72      return 0;
73
74      exit2: mmm_mp_cmd(mp_handle, MMM_MP_GET_ERRORNO, (unsigned long) &mp_err);
75      mmm_mp_cmd(mp_handle, MMM_MP_STOP, NULL);
76      mmm_mp_cmd(mp_handle, MMM_MP_CLEAR_FILE, NULL);
77      mmm_mp_cmd(mp_handle, MMM_MP_AOUT_CLOSE, NULL);
78      mmm_mp_cmd(mp_handle, MMM_MP_CLOSE, NULL);
79      exit1:
80      //卸载MMM_MP_API
81      sys_free_mmm(1);
82      //exit0:
83      return -1;
84 }
```

66-84 是出错或正常退出处理。关闭解码线程并卸载解码库。

根据上述描述，如果是 music 引擎，实现播放一个音频文件的接口调用顺序为:

```
 86 //music engine初始化
 87 _app_init();
 88 
 89 //初始化文件选择器
 90 fsel_init();
 91 
 92 //定位播放文件
 93 fsel_set_location(&location, FSEL_TYPE_COMMONDIR);
 94 
 95 //开始播放, 内部封装_set_file()函数
 96 _play();
 97 
 98 while(1)
 99 {
100     //获取引擎状态
101     mengine_status_deal();
102     if ((g_eg_status_p->play_status == StopSta) || (g_eg_status_p->play_status == PauseSta))
103     {
104         break;
105     }
106 }
107 
108 //关闭文件播放
109 _stop();
110 
111 //music engine退出处理
112 _app_deinit();
113 
```

该程序只是一个示例程序，方便用户更好的理解 music ap 的实现思路。可以看出该程序和上述解码线程的调用函数是很类似的，只是接口都封装在相应的函数中。

对于 music ui，实现 music 播放的方法只是将上述函数在前台以消息发送的形式提供。前台不需要关心后台是如何实现播放细节的，只需要根据消息的要求，传递相应的参数即可。下面以一个实例说明前台是如何实现 music 文件播放的。

如何实现带开机声音的 welcome? 普通的开机画面是在 config 应用调用 draw_logo()函数实现的(位于 config_main_sub.c),draw_logo()调用 common 控件 gui_logo()函数显示开机 logo。如果要求开机 logo 在显示图片的同时有声音输出，需要开启 music 播放。相应的示例程序如下:

```
void draw_logo(uint8 param)
{
    animation_com_data_t temp_animation;
    style_infor_t temp_sty;

    temp_animation.direction = 0;
    temp_animation.interval = 200;

    if (param == SWITCH_ON)
    {
        temp_sty.style_id = STY_POWER_ON;
```

```
    }
    else
    {
        temp_sty.style_id = STY_POWER_OFF;
    }
    temp_sty.type = UI_AP;
    ui_res_open("config.sty", UI_AP);

    music_playing_init();

    gui_logo(&temp_sty, &temp_animation);

    music_close_engine();
    ui_res_close(UI_AP);
}
```

重点看 music_playing_init()的实现，这里播放的文件位于 SD 区，因此处理很简单，代码如下:

```
void music_playing_init(void)
{
    uint8 cur_mode;
    music_open_engine(ENTER_ALARM);
    g_file_path.file_source = FSEL_TYPE_SDFILE;
    libc_memcpy(g_file_path.file_path.dirlocation.filename,g_sd_filename,12);
    if(music_set_filepath(&g_file_path) == FALSE)    //设置文件路径
    {
        return ;
    }
    set_current_volume(g_setting_comval.g_comval.volume_current);

    //设置循环模式
    cur_mode = FSEL_MODE_LOOPONE;
    music_set_playmode(cur_mode);

    music_play();
}
```

这些函数在 ap 都有实现，因此就不列举出来。可以看出，通过自底向上层层封装，我们提供给用户一种很简单快捷的方法实现 music 文件播放。用户可以据此实现更多更个性化的需求。

8.6 如何减少一种音频格式的支持

在 case/inc/case_type.h 中定义了 music 支持的文件类型的宏定义 MUSIC_BITMAP，修改该 bitmap 即可减少音频格式的支持。修改其它宏定义可修改相应应用支持的文件格式.

```
#define MUSIC_BITMAP            0x7e010e00
#define AUDIBLE_BITMAP          0x00003000
#define VIDEO_BITMAP            0x01004000
#define PICTURE_BITMAP          0x00780000
#define TEXT_BITMAP             0x00800000
#define RECORD_BITMAP           0x50000000
```

8.7 歌词的解析和显示是如何实现的

歌词模块的解析和处理流程大致如下：

- ❖ music engine 调用 enchanced 的歌词解析接口解析歌词，如果歌词存在，设置标志位。代码位于_set_file()(mengine_play_deal.c)：

```
g_eg_playinfo_p->cur_lyric = 0;
if (g_is_audible == FALSE)
{
    vfs_get_name(file_sys_id, g_file_name, 32);
    g_eg_playinfo_p->cur_lyric = lyric_open(g_file_name);
}
result = _get_file_info(TRUE);
```

- ❖ music ui 查询到该标志位，则在播放过程中每秒钟均会调用歌词获取接口试图获取当前时间点的歌词。
  代码位于_playing_check_status()(music_playing_playdeal.c)：

```
//获取播放信息
music_get_playinfo(&playinfo);
......
if (g_music_config.lrc_support == TRUE)
{
    g_lyric_flag = playinfo.cur_lyric;
}
```

- ❖ 如果当前时间恰好位于两个歌词标签之间，则解析并显示当前歌词
  代码位于 show_lyric()(music_playing_playdeal.c)：

```
if (lyric_check_query_time(g_cur_time) != 0)
{
```

版 权 所 有 侵 权 必 究

```
        if (lyric_seek_for(lrc_buf, LRC_BUF_LEN, g_cur_time) == TRUE)
        {
            lyric_param_init();
            libc_memset(display_buf, 0, LRC_BUF_LEN);
            parse_lyric(&g_lyric_decode);
            display_lrc_str(g_lyric_decode.output_buffer);
        }
    }
```

可以看出，这里歌词解析分成了两步，首先由 enhanced 读取歌词文件，并根据歌词的时间 tag 整理成若干条歌词信息。music ui 再调用 enhanced 接口获取某个时间点的歌词。此时歌词还需要再次整理成可以在 LCD 上显示的多行甚至多屏歌词。选择 music engine 去解析歌词而不是 music ui，原因在于 music engine 能够确切知道何时音乐停止和播放，可以在播放之前将歌词解析完毕。而由于空间冲突问题，music ui 不能在歌曲未播放之前调用歌词解析函数。因此安排歌词解析函数在 music engine 实现。在 music ui 获取歌词后讲歌词整理成输出字符串时，支持方案配置，根据配置值歌词支持多行，多屏和断词处理。歌词模块支持如下几种功能配置：

- 是否过滤歌词开始位置的换行符和空格符
- 是否支持歌词的断词功能(即对于英文字符，选择单词作为解析的对象)
- 是否支持歌词多屏显示

可通过设置歌词的显示模式来控制歌词显示方式，该结构体定义如下：

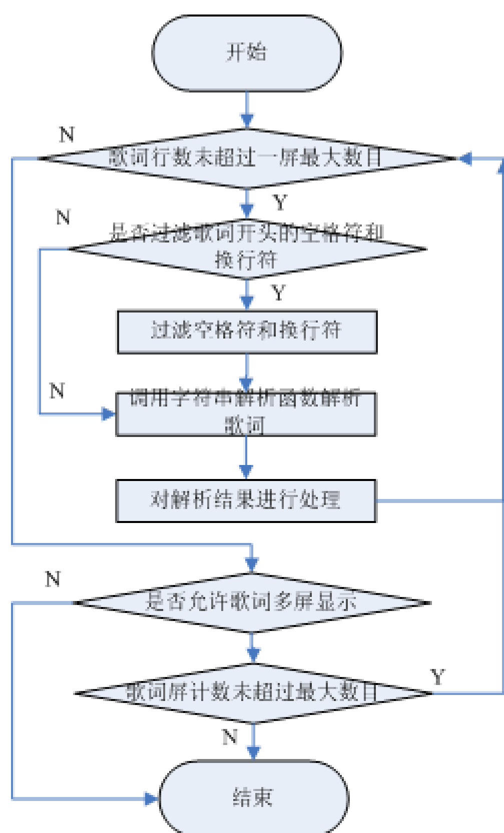
```
typedef struct
{
    /*! 显示一页行数 */
    uint8 line_count_one_page;
    /*! 显示一行像素点宽度 */
    uint8 max_width_one_line;
    /*! 显示模式，包括过滤空行，分词显示等 */
    uint8 mode;
    /*! 语言类型 */
    int8 language;
} lyric_show_param_t;
```

方案对歌词的相关配置位于 lyric_param_init()(music_playing_show_lyric.c),用户如果需要修改歌词的显示模式，可以根据注释修改相应的配置项。

```
void lyric_param_init(void)
{
    g_lyric_decode.input_buffer = lrc_buf;
    g_lyric_decode.output_buffer = display_buf;
    g_lyric_decode.input_length = 0;
```

```
    g_lyric_decode.input_remain = (uint16) libc_strlen(lrc_buf);
    g_lyric_decode.output_length = 0;

    g_lyric_decode.param.line_count_one_page = LYRIC_LINE_COUNT_ONE_PAGE;
    g_lyric_decode.param.max_width_one_line = LYRIC_MAX_WIDTH_ONE_LINE;
    g_lyric_decode.param.mode = (LRC_FILTER_BLANK_ENABLE | LRC_DIVIDE_WORD_ENABLE |
LRC_DIVIDE_SCREEN_ENABLE);
    g_lyric_decode.param.language = (int8) g_setting_comval.g_comval.language_id;

}
```

歌词解析模块实现歌词的断行，分屏显示等功能，函数为 parse_lyric(),相应的函数流程如下图所示：



8.8 music 是如何获取和显示专辑图片的

us212a 方案支持 music 专辑图片和 audible 内嵌图片的显示。。对于普通的 music 文件，专辑图片是由 music ui 通过调用 enhanced 中间件获取的，而 audible 专辑图片，分辨图也是通过中间件获取的，而 aax 文件其它内嵌图片是通过音频中间件解析的。在 music_engine 中存在

如下变量:

audible_para_t *g_audible_info_p = (audible_para_t *) 0x9fc2f000;

其中 audible_para_t 结构体中就有内嵌图片数量，位置，大小的记录信息。music 引擎申请一个 1s 的定时器用于查询当前时间点是否存在专辑图片，相应的函数为_check_audible_image()(mengine_event.c)。如果存在，则设置相应标志位并读取图片位置信息。music ui 查询到标志位设置之后，就根据当前的位置信息实现图片解码。相应的函数位于_playing_check_status()(music_playing_playdeal.c) :

```
//for audible  图片信息
    if (g_audible_now == TRUE)
    {
        //对于 audible 文件，是一定有专辑图片的，只是对于 AA 文件只有一张
        //对于 AAX 文件可能有多张，通过 get_playinfo 只能表示当前的 AAX 文件
        //是否有专辑图片，不能把 g_album_art 标志位清掉
        if ((playinfo.cur_imag_flag != 0) && (g_music_config.album_art_support == TRUE))
        {
            //专辑图片在文件中的偏移
            album_art_info.offset = playinfo.cur_imag_pos;

            album_art_info.apic_type = 0;

            g_display_bitmap |= up_musicalbumart;

            g_paint_flag = TRUE;//绘制

            g_album_art = TRUE;
        }

        if (g_music_config.section_mode == SWITCH_SECTION)
        {
            if ((playinfo.cur_file_switch & 0x02) != 0)
            {
                music_get_section_info(&g_file);
                //切换歌曲
                g_display_bitmap |= up_musictracks;
                g_paint_flag = TRUE;//绘制
            }
        }
    }
```

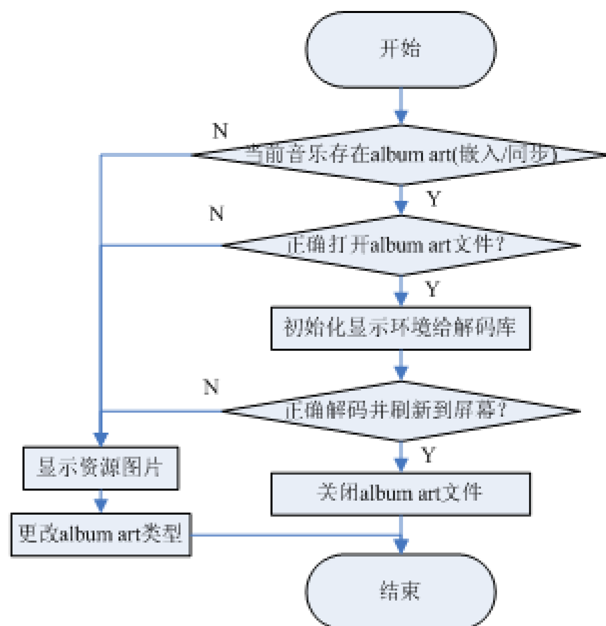
该程序片段在查询到 cur_img_flag 为 TRUE 之后就设置刷新专辑图片的标志位，在播放场景

版 权 所 有 侵 权 必 究

主循环中，根据该标志位显示专辑图片。函数位于 play_scene_paint(),程序如下：

```
//专辑图片
if ((*display_flag & up_musicalbumart) != 0)
{
    if (g_album_art == TRUE)
    {
        play_paint_albumart(&album_art_info);//如果当前有歌词,则将歌词显示标识置位
    }
    else
    {
        //显示光盘图标
        style.style_id = ALBUM_ART_PICBOX;
        ui_show_picbox(&style, NULL);
    }
    //清标志
    *display_flag &= clr_musicalbumart;
}
```

解码显示部分功能流程示意图如下：



8.9 music 应用支持哪些类型文件的 ID3 显示

music 应用支持 MP3/WMA/APE/FLAC/OGG/AAC/AAX/AA 类型文件的 id3 解析和显示，其

中 mp1/mp2/mp3 均是按照 mp3 的规格进行 id3 解析，m4a 文件也是按照 aac 格式进行 id3 解析的。从 enhanced 中间件获取的 ID3 内容有三种格式:内码，unicode，UTF8，对于 UTF8 格式的 ID3 会将其转换成 unicode，因此对于应用而言，获取的 ID3 就只有内码和 unicode 两种形式。enhanced 提供接口 get_id3_info()用于获取当前类型歌曲的 ID3:

```
bool get_id3_info(id3_info_t *id3_info, const char *filename, id3_type_e file_type)
```

通过该接口获取 ID3 后，需要根据获取的字符串特征设置相应的语言 ID。转换判断如下:

1. 如果 ID3 为 unicode，则设置语言 ID 为 UNICODEDATA
2. 如果 ID3 为 UTF8,则首先调用 UI 驱动接口函数 ui_utf8_to_unicode()将 UTF8 类型的 ID3 转换成 unicode,然后设置语言 ID 为 UNICODEDATA
3. 如果 ID3 为内码，则设置语言 ID 为 ANSIDATAAUTO

相应的函数为 deal_id3_sub(),位于 music_playing_getid3.c,用户可以参考该函数实现自己的 ID3 显示功能。

8.10 小机开机如何实现 music 断点播放

断点续播需要实现两个功能,首先是断点的记忆和恢复,其次是开机后自动播放 music 文件。前者比较好处理，music 引擎在关机前保存断点，在开机后读取 VRAM 中保存的断点，通过发送断点续播命令就可以实现断点播放。后者较为负责,需要 config ap 和 music ui 及 music 引擎交互完成开机歌曲自动播放。这里需要解决两个问题:

- ❖ 如何实现开机直接进入 music 应用
- ❖ 如何实现歌曲的自动播放

首先看第一个问题的解决，小机开机和关机都会执行 config ap，因此开机进入哪个 ap 是由 config ap 决定的。在关机之前，manager ap 杀死当前执行的前台 ap 和后台 ap 后，根据当前的引擎状态设置传给 config ap 的参数,其参数由两部分组成,last_app_id 和 last_engine_state,前者占低 16bit，代表关机时的前台应用 ID，后者占高 16bit 表示后台引擎的状态。config ap 据此记忆开机后要创建的 ap 以及传递给相应的 ap 参数。

例如如果此时后台正在播歌，则 manager ap 传递给 config ap 的参数为 last_app_id = APP_ID_MUSIC, last_engine_state 为正在播放或暂停。config ap 在关机前保存这些参数。再次开机时读取参数后创建的第一个前台 ap 就是 music ui，同时根据引擎参数传递给 music ui 不同的参数。由此完成开机直接进入 music 应用的功能。

对于第二个问题，music ui 在启动后，根据传入的参数进入不同的场景。music ui 暂时使用了如下几种进入模式，下面描述各种模式下的含义及入口场景:

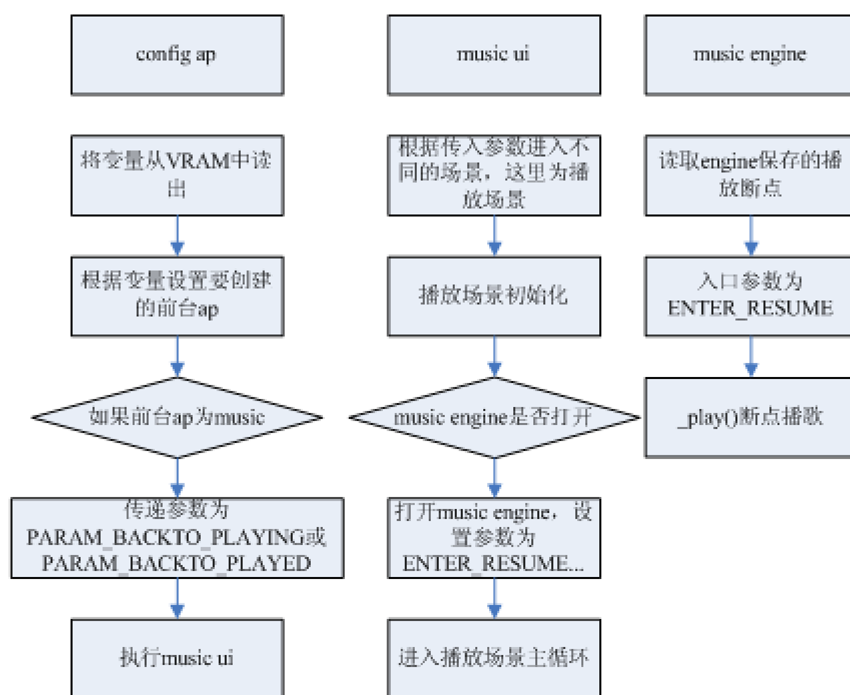
名称	含义	入口场景

版 权 所 有 侵 权 必 究

PARAM_FROM_MAINMENU	从主界面选择 music 图标进入	菜单列表场景
PARAM_FROM_BROWSER	从文件浏览器选择 music 文件播放	播放场景
PARAM_FROM_RECORD	从 record 录音库或 FM 录制选择录音文件进入	播放场景
PARAM_BACKTO_PLAYING	从主界面选择正在播放图标或其它 ap 选择正在播放菜单进入	播放场景
PARAM_BACKTO_PLAYED	从主界面选择上一次播放图标或其它 ap 选择上一次播放菜单进入	播放场景
PARAM_FROM_PLAYLIST	从 playlist ap 创建完播放列表后返回 music ui	菜单列表场景

对于参数为 PARAM_BACKTO_PLAYING 或 PARAM_BACKTO_PLAYED，如果当前引擎没有打开，则表明属于 standby 起来播歌，因此会执行打开引擎。music engine 在加载后会自动执行播歌流程。所以 config ap 在创建 music 时，传递的参数就是 PARAM_BACKTO_PLAYING 或 PARAM_BACKTO_PLAYED。

整个过程实现的流程图如下图所示:



版 权 所 有 侵 权 必 究

8.11 music 如何选择某个专辑，某个歌手的所有专辑播放

对某个专辑，某个歌手歌曲的选择类似于选择一个目录播放，如果当前激活项为 list_no,则只需要调用 fsel_browser_select(list_no)就可以选择整个专辑或整个歌手列表。而在歌手下一级会有一个所有歌曲，对该菜单的选择实际上是歌手所有专辑的总和，如果将 list_no 设置为 0xffff,则 enhanced 就可以选择某个歌手的所有专辑。相关示例程序如下所示:

```
    fsel_browser_enter_dir(CUR_DIR, 0, &browser);
    if((active == 0) && (browser.dir_total != 0))
    {
        //选择所有相同类型的专辑播放
        active = 0xffff;
    }
    else
    {
        if(browser.dir_total != 0)
        {
             //选择某种风格，歌手，专辑播放
        }
        else
        {
            if (active == 0)
            {
                //allsong 的嵌套菜单全部随机播放处理
                active = 1;
            }
        }
    }
   fsel_browser_select(active);
   fsel_browser_get_location(plist_locatp, location->file_source);
```

函数为_music_listmenu_play_file(),位于 music_listmenu_listoption.c。

8.12 如何定位专辑，歌手等播放列表

艺术家，专辑，流派和歌曲都属于 playlist 的一个子表，因此 location 类型为 plist_location,该结构体定义如下:

```
//播放列表下的文件信息
typedef struct
{
```

```
    uint8 disk;                    //DISK_C;DISK_H;DISK_U
    uint8 list_layer;              //列表层数
    uint16 list_type;              //子表类型  plist_type_e
    uint8 filename[4];             //文件的后缀名，全填 0 代表文件信息为空
    uint32 cluster_no;             //文件的目录项所在的簇号
    uint32 dir_entry;              //文件的目录项在所在簇号内的偏移
    pdir_layer_t dir_layer_info;//目录层次信息
    uint8 res_3byte[3];            //保留对齐
    uint16 file_total;             //当前文件总数 (在当前列表下文件层的总数)
    uint16 file_num;               //当前文件序号 (在当前列表下文件层的序号)
    uint16 file_index;             //当前文件索引号（在数据区的序号）
    uint16 reserved;               //保留对齐
    uint16 list_layer_no[4];       //记录文件所在的每层列表中的位置，（分别记录在每层排序后的位置）
    uint16 layer_list_offset;      //list 偏移位置
    uint16 reserve;                //保留对齐

} plist_location_t;
```

其中 list_layer 表明当前路径在播放列表的层级，例如对于 artist 子表，如果 list_layer 为 1，表明在专辑这一层次。而 list_type 正是用于说明当前子表的类型。使用 fsel_browser_set_location()用于定位某一个子表。例如在 music_scene_listmenu.c 中选择专辑子菜单进入，程序如下:

```
browser_path->file_source = FSEL_TYPE_PLAYLIST;
browser_path->file_path.plist_location.list_type = PLIST_ALBUM;
//定位子表
fsel_browser_set_location(&(browser_path->file_path), browser_path->file_source);
```

函数为 music_scene_listmenu_menu(),位于 music_scene_listmenu.c,用户可以参考该函数实现定位相应播放列表的功能。

8.13 music 遇到格式不支持文件是如何处理的

music 播放过程中遇到非法文件会根据场景的不同做不同处理，如果前台为 music ui 且处于播放场景，则 music ui 会查询当前引擎的状态，因此引擎可以将当前出错的状态反馈给前台并提示格式错误，此时引擎只负责处理状态，并根据前台发送的消息切下一曲或上一曲。如果前台在其它场景，则没有前台会关心当前歌曲是否错误，因此引擎主动完成歌曲切换的过程，重新选择下一曲进行播放。引擎有关格式错误的处理函数为 mengine_status_deal()(mengine_control.c)，代码如下:

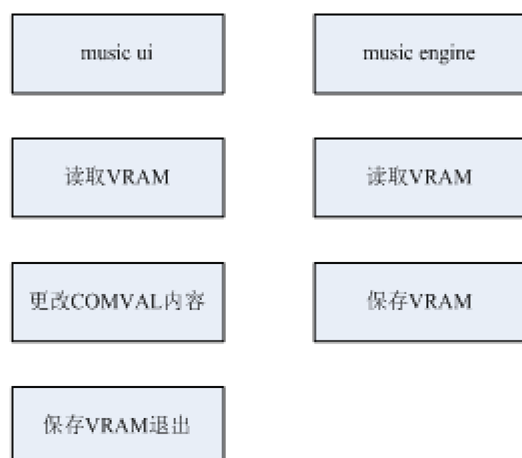
```
case MMM_MP_ENGINE_ERROR://出错
//前台不是 music UI 或者前台进入屏保状态，出现错误直接切下一曲
if ((g_app_info_state.app_state != APP_STATE_PLAYING) || (g_app_info_state.screensave_state == TRUE))
```

```
{
    //前台不是 music ui,则直接切下一曲
    _error_timeout_handle();
}
else
{
    //否则等待前台响应 UI 状态
    bret = _stop(STOP_NORMAL);//停止播放

    //未开始播放便报错的情况
    if ((g_eg_status_p->play_status == StopSta) || (g_eg_status_p->play_status == PauseSta))
    {
        //关闭文件
        mmm_mp_cmd(mp_handle, MMM_MP_CLEAR_FILE, (unsigned int) NULL);
    }

    //恢复成默认状态,防止快进退遇到格式错误文件再次快进退
    g_eg_status_p->play_status = StopSta;
    g_eg_status_p->fast_status = FBPlay_Null;
    g_eg_status_p->ab_status = PlayAB_Null;
    mengine_save_errno_no();
}
break;
```

8.14 music 前后台读写 VRAM 有何注意事项

在本方案中，music 应用划分成前后台两个应用，因此前后台均会访问 comval 的 VRAM 变量，而按照一般的逻辑，应用在进入时都会读取 VRAM，同时在退出的时候写入 VRAM，但此时存在一个问题，即较新的 VRAM 值有可能被覆盖。如下图所示:

版 权 所 有 侵 权 必 究



如图所示，由于 music ui 有可能在 music engine 还没退出时就已经退出，因此当 music engine 退出时，其保存的内容由于是旧的，导致 music ui 更新的 VRAM 值被覆盖。
因此 music 引擎对于 comval 的 VRAM 保存需要特别注意，只保存 comval 中 music engine 会用到，且会改变的成员，其它成员都在保存之前，重新从 VRAM 读取。

8.15 music 正在播放时长按 play 键关机的流程需要注意什么

music 正在播放时，前台会不断发消息给后台询问当前的播放状态，播放时间等信息，如果长按 play 键关机或接入 USB 线需要进入 U 盘，需要先关闭 music 应用，然后再进入 config 关机流程或创建 udisk 应用。
在关闭 music 应用时，需要注意前后台的退出顺序。由于消息的发送都是由前台发起，后台接收处理，因此需要先退出前台，然后再关闭后台，否则会因为前台等不到消息的回复而陷入阻塞状态。
manager 应用在收到类似的消息时，先广播消息给前台应用并等待前台退出后释放资源；前台退出之后再给后台发送消息，等待后台退出之后，manager 应用再创建相应的应用。
相应的函数位于 manager_msg_callback_sub()(manager_msg_handle2.c),用户根据上述解释可以对照代码的实现，更好的理解多应用场景下 ap 的交互原则。

8.16 播歌过程中 VRAM 读写有何注意事项

播歌过程中存在 VRAM 读写的场景如下:

5) 对主盘收藏夹，书签的操作
6) audible 播放过程中保存 audible 断点
7) 前台应用切换时会保存当前应用的 VRAM 变量和全局 VRAM 变量

由于 VRAM 的读写使用的 buffer 和进行 bank 切换时使用的 buffer 为同一块 buffer，因此二

者为互斥使用的关系。为保证数据一致性，在 VRAM 读写时系统将中断关闭。为尽量减小因关闭中断导致的响应不及时问题，对播歌过程中 VRAM 的读写需要注意如下几个方面:

- 尽量减小 VRAM 的读写次数，以减小开关中断的延迟和写 nandflash 的次数
- 读写 VRAM 时，使用 512 字节作为缓冲区大小，防止大批数据写入
- 优化程序，防止在读写 VRAM 的同时还有 bank 切换的情形发生

8.17 audible 如何实现专辑图片和章节的切换

在 music engine 中申请了两个 UI 定时器，专门用于查询当前 audible 播放时是否发生了专辑图片切换或章节切换。audible codec 提供了一个 buffer 用于读取当前 audible 文件内包含的专辑图片数目和每张专辑图片的时间，位置，章节的总数和章节切换的时间。定时器通过比较当前的时间点和 codec 提供的时间点确定当前是否发生了专辑图片切换或章节切换。一旦发生了切换，则会设置相应的标志位。而前台会不断地查询这些信息，一旦有标志位设置，则会显示相应的专辑图片或章节数目。引擎提供的播放信息定义如下:

```
/*!
 * \brief
 *   当前播放文件信息
 */
typedef struct
{
    /*!  当前时间  */
    uint32 cur_time;
    /*!  比特率  */
    uint32 cur_bitrate;
    /*! audible 图片位置偏移  */
    uint32 cur_imag_pos;
    /*! audible 图片存在标志(非分辨图) */
    uint8 cur_imag_flag;
    /*!  歌词存在标志  */
    uint8 cur_lyric;
    /*!  文件切换标志  bit0:  歌曲是否切换  bit1:audible 章节是否切换   */
    uint8 cur_file_switch;
} mengine_playinfo_t;
```

前台通过查询 cur_file_switch 相应的标志位，就可知道是否有章节切换。查询 cur_image_flag 是否为 TRUE,可知是否需要显示相应的专辑图片。

版 权 所 有 侵 权 必 究

8.18 audible 如何实现断点续播功能

当前 SDK 支持记忆最后一个播放的 audible 文件的时间断点，一旦 audible 歌曲播放过，则该时间断点就会记录。前台 g_music_config 用一个标志位记录是否有 audible 歌曲播放，该标志位只有在创建播放列表后才会被清除。audible 断点结构体定义如下:

```
/*!
 * \brief
 *   audible 续播功能结构体定义
 */
typedef struct
{
    uint16 magic;
    //uint8 title[30];
    /*! 断点信息 */
    mmm_mp_bp_info_t bk_infor;
    /*! 路径信息 */
    file_path_info_t locat_info;
} audible_resume_info_t;
```

其中 magic 用于判断当前的 audible 断点是否有效，当创建播放列表后会设置 magic 为无效状态。而真正的断点记录需要记录当前播放的 audible 的断点信息和路径信息。这些内容实际上是由引擎写入的。为了减小写入断点 VRAM 的次数，对该时间断点的写入规则如下:

- 前台从语音书籍进入时如果没有 audible 播放，而断点标志存在，则从 VRAM 读出断点内容显示断点续播，否则直接获取当前的断点信息
- 后台引擎负责真正断点的写入，只有两种情形会写入 audible 断点，music engine 退出前如果是 audible 播放，会写入断点；music engine 从 audible 播放切换到 music 播放，也会将当前的 audible 断点保存。其它情况下并不写入断点。对于第一种情形，代码位于 mengine_control_block()，代码如下：

```
if (g_mengine_result == RESULT_APP_QUIT)
{
    if (g_is_audible == TRUE)
    {
        mengine_save_audible_breakinfo();
    }
    _stop(STOP_NORMAL);
    break;
}
```

对于第二种情形，代码位于 mengine_set_filepath()，代码如下：

```
if (loop_cnt == 2)
{
```

```
        //是否从 audible 切换到 music 播放
        if (g_is_audible == TRUE)
        {
            mengine_save_audible_breakinfo();
            g_is_audible = FALSE;
            need_fsel_init = TRUE;
        }
    }
```

8.19 audible 的 pos 文件是如何保存和读取的

audible 在播放之前检测当前的 audible 歌曲是否存在 pos 文件，如果文件不存在，则创建文件，否则从文件读取当前的 pos 信息，并在播放之前将 pos 信息传递给 audible 解码库。在歌曲播放结束后，会从解码库获取当前的时间断点，并将断点写入到 pos 文件中。每个 pos 文件为 512 字节大小。相应的代码位于 _deal_audible_posfile() 函数中(mengine_audible_deal.c)。

8.20 audible 是如何读取和保存激活文件的

audible 的激活文件是通过 audible manager 工具从网上下载而来，对于不同方案，该激活文件存储的位置和文件名不尽相同。music ui 在第一次播放 audible 歌曲时，会在指定目录读取该 sys 文件(激活文件)的内容，并将其保存到 VRAM 中，后续对激活文件的读取都是通过读取 VRAM 的内容实现的。一般激活文件为 560 字节大小，在 VRAM 中，前 560 字节为激活文件内容，第二个扇区的最后两个字节用于存储标志位，表示当前的 VRAM 中是否已保存激活文件。

将激活文件保存到 VRAM 中而不是直接读取激活文件，避免了卡盘播放文件时需要在不同存储介质切换的操作，简化了 audible 播放的流程。

music ui 保存激活文件程序位于 music_load_sysfile()(music_audible_sys.c),用户可以修改相关激活文件的路径，实现播放 DRM 的 audible 文件。

8.21 如何在列表控件嵌套菜单项

当前的列表控件是支持在 ui_direcotry()的某一级嵌套一个或若干个菜单，当前只支持一个入口菜单，但该入口菜单可包含多个叶子菜单。例如 music 的所有歌曲列表包含一个“全部随机播放”菜单，要在该列表实现一个嵌套菜单的方法很简单，示例代码如下:

```
menu_item_insert.app_id = APP_ID_MUSIC; //嵌套菜单属于 AP 还是 common
menu_item_insert.layer = 0;             //嵌套菜单层级
```

```
menu_item_insert.list_menu = SHUFFLE_PLAY; //入口菜单 ID
list_param.menulist = &menu_item_insert;    //设置私有成员
list_param.menulist_cnt = 1;
```

需要注意，即使嵌套菜单不止一个，但由于这些菜单属于同一个入口菜单，因此 menulist_cnt 仍为 1。

8.22 列表控件的嵌套菜单激活项是怎么计算的

普通列表的激活项计数是从 1 开始的，对于嵌套菜单项，如果只有一个嵌套的叶子菜单，激活项序号为 0，如果有多个，则激活项序号为 1 – 第几个嵌套菜单。例如如果有三个叶子菜单，则激活项序号分别为-2,-1,0。

对于嵌套菜单的回调也是由 ui_directory()控件的 option 回调函数完成，该回调函数由 ap 层完成。因此应用开发人员可以根据激活项的序号确定选择的激活项是列表还是菜单，以及是第几个菜单。

相关示例程序可参考函数 m3ulist_option_result_proc()(位于 music_listmenu_listoption.c).该函数实际为 ui_direcotry()的回调函数，其中前三个菜单对应收藏夹 1/2/3 的回调后面，后面的对应 m3u 播放列表的回调函数。这里就是根据激活项区分相应的列表项的。

8.23 列表和菜单控件是如何实现路径记忆的

列表和菜单控件通过 com_get_history_item ()读取路径记忆，通过 com_set_history_item()设置路径记忆。路径记忆的内容由结构体 history_item_t 进行封装，其成员如下:

```
/*!
 * \brief
 *    history_item_t 路径记忆项结构体
 */
typedef struct
{
    /*! 当前显示列表首项编号，0xffff 表示无效 */
    uint16 top;
    /*! 激活项在当前显示列表中的位置，0xffff 表示无效 */
    uint16 list_no;
} history_item_t;
```

菜单的路径记忆比较好处理，一旦菜单确定，路径记忆也就确定。当前一个 ap 的菜单控件最多支持 8 个不同的菜单控件路径记忆。需要注意的是，ui_menulist()控件需要传入一个

版 权 所 有 侵 权 必 究

path_id,表明使用哪个路径记忆。而 ui_menulist_simple 则不包换路径记忆。对于简单的只有一级菜单的场景，使用 ui_menulist_simple()可以减少对路径的记忆，同时增加程序的灵活性。

列表的路径记忆需要在读取之前判断当前路径是否有效，如果当前的路径记忆为非法路径，需要设置默认的路径记忆内容。例如 music 不同的列表切换，需要清除当前的列表记忆，然后再进入新的列表进行浏览。

common 的列表路径记忆需要将当前的路径记忆写入 VRAM，为了减少浏览上下级时读写 VRAM 的次数，ui_direcotry()控件在子场景中实际是将路径记忆写入从系统堆申请的内存空间中，在控件退出之前才真正将路径记忆写入 VRAM。因此如果用户实现的控件也需要记忆路径，可以仿照列表控件的实现方法。例如专辑图片退出时保存路径记忆的代码如下（位于 music_listmenu_alnumlist_sub.c）:

```
static void _pic_save_history(dir_control_t *p_dir_control)
{
    if (p_dir_control->list_no < 6)
    {
        p_dir_control->top = 1;
    }
    else
    {
        p_dir_control->top = p_dir_control->list_no - 5;
    }
    //这里只是将路径记忆写入缓存，注意在退出时调用该接口再次写入，参考接口说明
    write_history_item(g_dir_browser.layer_no, p_dir_control, FALSE);
    return;
}
```

8.24 music 函数在头文件声明时为什么要加__FAR__

mips 中跳转指令为 J 型指令，跳转地址占 26 位，考虑地址四字节对齐，最多支持 28bit，即 256MB 范围内地址跳转。如果超过该地址范围，则需要将目标地址放在一个寄存器中，然后跳转到该寄存器内容所在的地址执行。

Figure 4-1 Immediate (I-Type) CPU Instruction Format

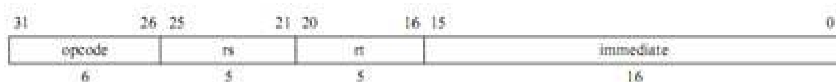
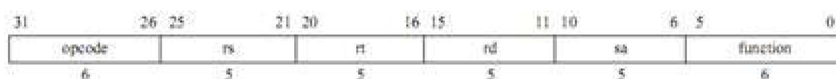


Figure 4-2 Jump (J-Type) CPU Instruction Format

Figure 4-3 Register (R-Type) CPU Instruction Format



如下为一个子函数分别为不采用__FAR__声明和采用__FAR__声明编译后的汇编代码：

```
            music_set_eq(&eq_info);
400a0070:        1b40 804c        jal        406a0130 <music_set_eq>
400a0074:        6500             nop
400a0076:        1060             b          400a0138 <_scene_play_seteq+0x104>
400a0078:        b232             lw         v0,400a0140 <_scene_play_seteq+0x10c>
```

```
            music_set_eq(&eq_info);
400a0070:        b236             lw         v0,400a0148 <_scene_play_seteq+0x114>
400a0072:        eac0             jalrc      v0
400a0074:        1060             b          400a0136 <_scene_play_seteq+0x102>
400a0076:        b232             lw         v0,400a013c <_scene_play_seteq+0x108>
```

可以看出，如果不使用__FAR__，则函数跳转会直接使用 jal 指令，mips J 类型指令可以跳转的地址是有一定长度限制的。而采用__FAR__声明后，跳转指令由 jal 转变为 jalrc,其中 r 即代表跳转地址为寄存器中的值。

因此如果 bank 调用常驻代码或常驻代码调用 bank 代码会出错，需要使用__FAR__声明。这样编译器会自动先将跳转地址加载到一个寄存器中，然后再实现跳转。对于普通函数，没有必要都加__FAR__，这样会增加代码编译空间。

一般情况下，如果 bank 代码调用 rcode 代码，或 rcode 调用 bank 代码，被调用的函数声明都需要加__FRA__。

版 权 所 有　侵 权 必 究

9 music_engine 引擎

9.1 需求概述

music 引擎用于后台的音乐播放。当界面实现其他功能时例如：播放电子书、播放图片、浏览文件、参数设置等功能时，实现音乐播放状态的自动转换，音乐文件的自动换曲等功能。其总体功能需求如下：

- 能够完成音乐文件的播放
- 能够自动切换音乐
- 实现与 music ui 的通信，完成对 music ui 命令的响应
- 能够完成对系统消息的响应

9.2 总体设计

music 引擎由四个功能模块组成：初始化模块、状态处理模块、消息处理模块、应用退出模块。其中初始化模块主要实现进入音乐引擎时一些配置参数的获取、断点信息的获取，播放状态的初始化等功能。状态处理模块主要实现在音乐播放过程中对音频解码器的状态检测和处理，例如音乐播放当到尾、播放出错等状态的获取与处理。消息处理模块主要实现获取消息，并对相应消息处理，例如 music ui 应用发送切换歌曲的命名时，引擎需要根据命名切换歌曲；再如：music ui 应用发送复读消息，则引擎应使解码器状态进入复读状态。应用退出模块，主要是当引擎接收到退出的消息时，保存一些参数设置和断点信息以及一些资源的释放等功能。

模块名称	功能简述
初始化模块	对应用功能进行初始化
状态处理模块	实现在音乐播放过程中对音频解码器的状态检测和处理
消息处理模块	实现获取消息，并对相应消息处理

版 权 所 有 侵 权 必 究

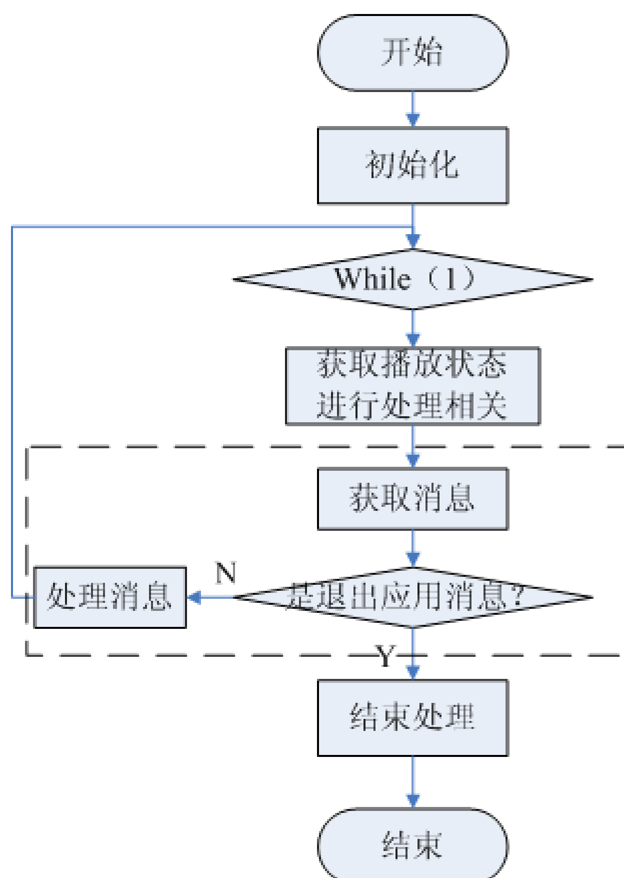
应用退出模块	保存一些参数设置和断点信息以及一些资源的释放等功能

9.3 音乐引擎的业务流程

music 引擎的业务流程比较简单，主要实现歌曲的播放控制，包括快进退处理，复读处理，格式错误报错，歌曲播放等，此外对于 audible 文件，会实现 audible 专辑图片的检测，pos 文件的读取与保存。

9.3.1 音乐引擎的总体流程

music 引擎的总体流程如下图所示：



9.3.2 音乐引擎的状态处理流程

在 music 引擎主循环中，并没有类似前台 ap 的场景调度，因为 music 引擎只负责获取解码器状态，并对相应的状态进行处理。music 引擎一般不关心显示相关问题。因此 music 引擎主体结构为状态处理模块。

该模块主要是对音乐解码过程中的一些状态进行处理，在 music 引擎中定义了如下几种播放状态，前台根据不同的播放状态实现不同的功能，其定义如下:

```
/*!
 * \brief
 *   引擎状态
 */
typedef struct
{
    /*!  播放状态   */
    play_status_e play_status;
    /*! AB 复读状态   */
    ab_status_e ab_status;
    /*!  快进退状态   */
    fast_status_e fast_status;
    /*!  错误状态,获取后清除   */
    eg_err_e err_status;
} mengine_status_t;
```

其中定义了如下几种播放状态:

```
/*!
 * \brief
 *   当前播放状态
 */
typedef enum
{
    /*!  停止  */
    StopSta    = 0,
    /*!  暂停  */
    PauseSta,
    /*!  播放  */
    PlaySta,
    /*! AB 复读  */
    PlayAB,
    /*!  快进退  */
    PlayFast
```

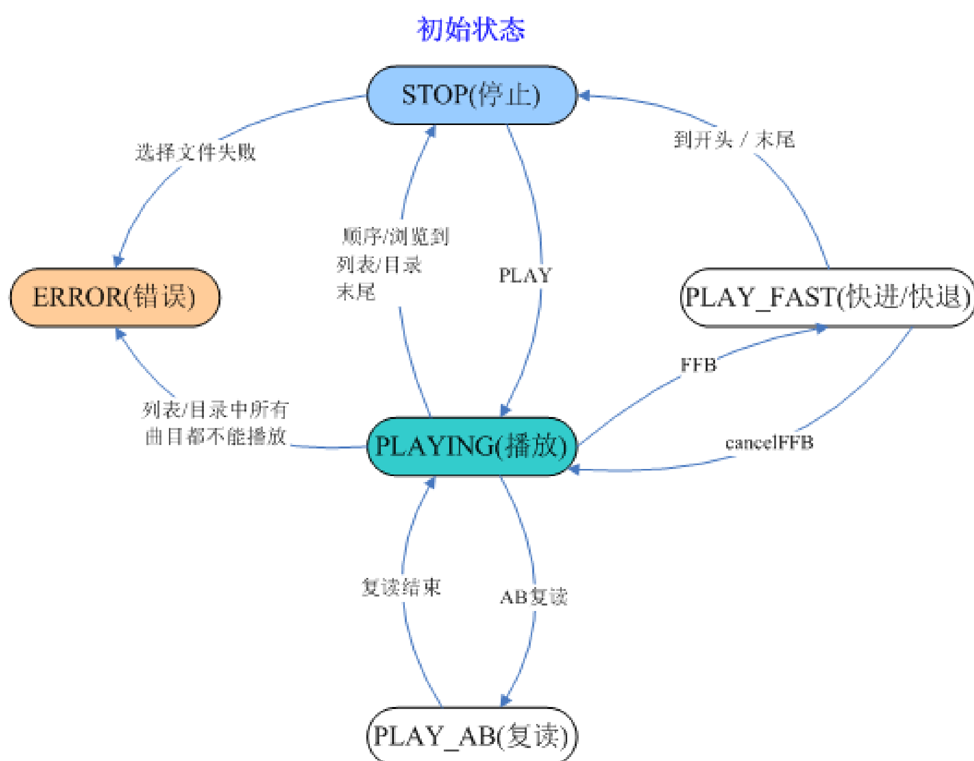
版 权 所 有 侵 权 必 究

```
} play_status_e;
```

类似的还有快进退状态和 AB 复读的状态。如果将播放状态看作一个总类的话，则快进退和 AB 复读属于单独的一个子类播放状态。引擎是根据解码线程的播放状态在不同的播放状态中转移的，解码线程定义了如下几种播放状态信息:

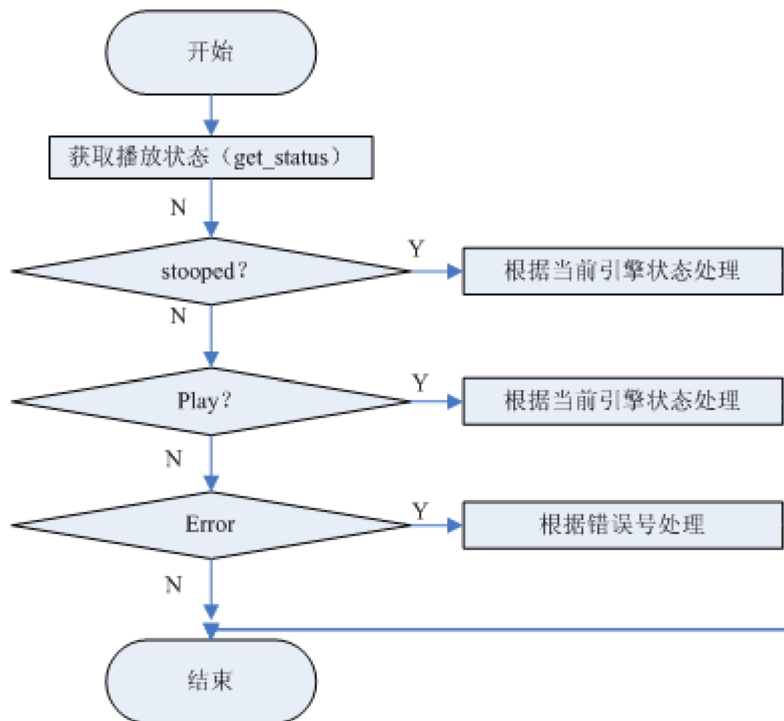
```
typedef enum
{
MMM_MP_ENGINE_STOPPED,
MMM_MP_ENGINE_PLAYING,
MMM_MP_ENGINE_PAUSED,
MMM_MP_ENGINE_ERROR,
} mmm_mp_engine_t;
```

可以看出，解码线程定义的播放状态信息较引擎的播放状态要简单的多，因此引擎需要根据这三个抽象的状态增加对快进退和 AB 复读的支持。引擎的播放状态循环图如下图所示，理解了该图的意义就可以掌握引擎的整个播放状态迁移。



music 引擎的状态处理其实就根据不同的播放状态进行不同分支的处理，例如 stopped 状态处理歌曲的切换，播放状态控制 AB 复读实现，出错状态实现错误状态显示等。总体流程图如下图所示:

版 权 所 有 侵 权 必 究



播放状态处理：

✧ 当前是否在 AB 复读，AB 复读是否结束
✧ 当前是否是浏览播放，浏览播放时间到切换到下一曲

停止状态处理：

❖ 当前是快进/正常播放 切换到下一曲
❖ 当前是快退 切换到上一曲/停止
❖ 当前是停止状态，不做操作

Error 状态工作：

❖ 判断出错类型，修改 engine 状态
❖ 关闭当前播放文件

music 引擎除了支持普通 music 播放外，还支持 audible 歌曲播放以及闹铃音乐播放。因此在状态处理模块为这两种类型文件作了特殊处理，体现在：

❖ 对于闹铃音乐，播放完毕(转入 stop 状态)后，不会切换上下曲
❖ 对于普通 music 音乐，除了暂停状态外，都通过该模块获取当前时间。而 audible 由于 pos 文件记录有断点时间，因此只有在播放状态(PLAYING)时，通过该分支获取当前时间

版 权 所 有 侵 权 必 究

- 普通 music 文件支持快进到尾切换下一曲以及快进到头切换上一曲，但 audible 文件快进到尾和快退到头只是结束快进退状态。对于快进到文件尾，会清除当前文件断点信息并切换下一曲播放；对于快退到头，会重新从当前文件起始位置播放
- 引擎处于错误状态时有两个分支，如果前台处于 music ui 播放场景，引擎停止 music 播放并关闭文件，然后等待前台处理；如果前台处于其它 ap 或其它场景，引擎自动切换下一曲播放

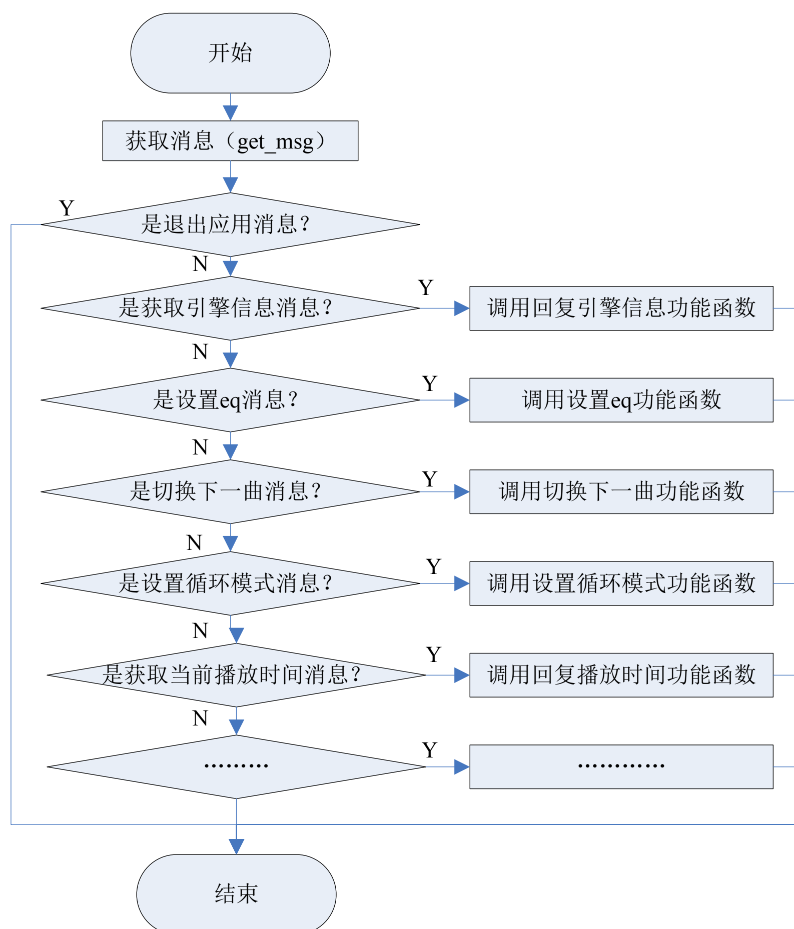
9.3.3 音乐引擎的消息处理流程

music engine对外提供的功能，主要通过外部消息来体现。Music UI应用播放音乐时通过发送不同的消息来控制引擎的行为，其它应用也可以通过发送消息获取/设置播放信息。该模块主要实现获取消息，并对相应的消息进行处理。

主要的功能消息有：

- 获取引擎信息（状态信息、文件信息、配置信息等）
- 获取当前文件路径信息
- 设置 EQ
- 设置循环模式
- 设置 AB 复读
- 设置音量
- 设置播放速度
- 开始播放 （用于选择文件开始/暂停后开始）
- 停止播放（用于暂停）
- 设置播放文件(用于选择歌曲播放：选择列表/选择目录/选择收藏夹，可返回总数和序号?)
- 获取当前时间标签
- 设置书签播放（选择书签选项）
- 播放下一首(按键切歌)
- 播放上一首(按键切歌)
- 快进快退
- 取消快进退
- 删除当前文件(同时更新列表)
- 播放上一次断点

消息处理流程示意图如下：



9.4 与其他模块的同步和交互

音乐引擎作为后台服务，会与其交互的模块包括：ap_manager，ap_music，以及中间件。其中，通过与 ap_manager 的交互，实现任务调度；与 ap_music 的交互，主要体现在被动的接收 ap_music 应用发过来的同步消息，并做相应的处理，与中间件的交互，主要是通过调用中间件的接口，实现播放系列，定位文件，ID3 信息处理等功能。

music 引擎与其它 ap 交互均是通过同步消息实现的，以 music ui 向引擎发送 SET_FILEPATH 消息为例，说明同步消息发送及处理过程。

发送方在发送同步消息时，需要组织私有消息结构体的内容，该私有消息定义了发送消息的内容以及接收方需要的消息回执。其定义如下:

```
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;
```

msg 成员即是发送消息实体，其定义如下:

```
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;
```

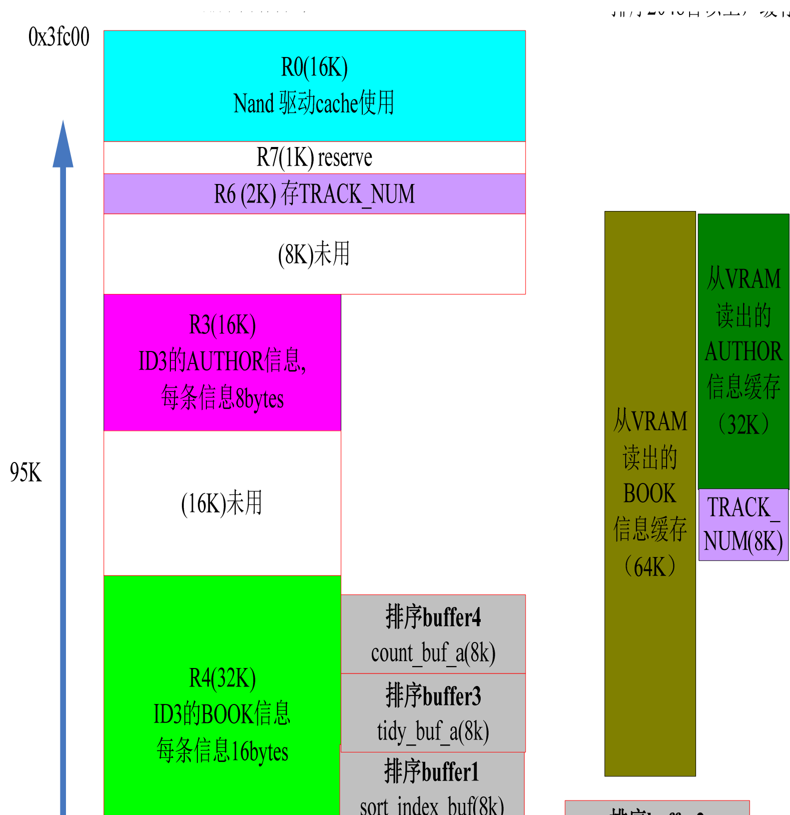
发送方需要设置发送的消息类型以及需要发送的消息内容，内容既可以是数据(4 个字节)，也可以指向某个地址空间，用于发送方和接收方之间传递内容。

发送方需要关心接收方收到消息的处理结果，结果记录在 reply 成员中，其定义如下：

```
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;
```

消息接收方会设置 type 类型，content 的内容，发送方在获知消息处理完毕后，可以根据 type 类型得知处理结果。

music ui 和 music engine 使用同步消息进行消息通信的流程图如下所示：



9.5 应用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

Applib 的全部函数

Enhanced 库中的文件选择、收藏夹和列表功能模块

9.6 如何增加一条引擎消息

在 case/inc/App_msg.h 中有关 msg_apps_type_e 的枚举类型中定义了 music 引擎所需要的消息类型。如果要增加一条引擎消息，需要先增加一个消息类别，然后在 music 引擎 ap 中增加对该消息的处理。由于 music 引擎设计原则为只能接受消息，无法发送消息，因此需要在前台 ap 增加发送该消息的函数。music 引擎对消息的接收位于

mengine_control_block()函数中，对于需要特别频繁处理的消息，直接在该函数处理，否则位于子函数 mengine_message_done_bank()中处理。

10 ap_record 应用

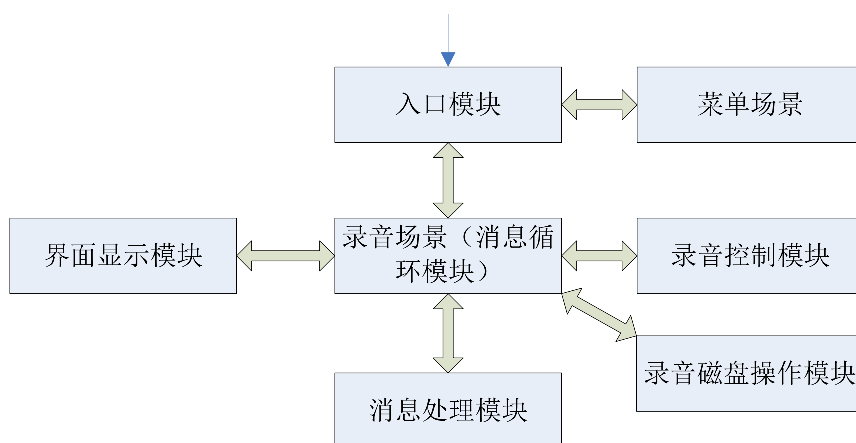
10.1 需求概述

Record 应用主要实现的功能包括：

1. 支持 PCM 编码 512Kbps、768Kbps、1024Kbps、1536Kbps 的 WAV 格式录音；支持 32kbps、64kbps、128kbps、192kbps 的 mp3 格式录音。
2. 录音的音源可以是 FM 输入或麦克风输入。
3. 所有的录音文件保存在 Record 目录。
4. 录音文件可以在菜单中浏览，选择播放后进入音乐应用播放。

10.2 总体架构设计

10.2.1 总体架构图



10.2.2 功能模块划分

模块名称	功能简述	对应文件
入口模块	负责应用初始化和退出，包括资源文件，菜单配置文件的加载和卸载，以及应用的场景管理	record_main.c
录音场景模块	提供录音界面显示、GUI 消息处理、录音过程处理，用户按键交互等功能	record_scene_recording.c
消息处理模块	处理应用的私有消息和系统消息	record_message.c
界面显示模块	根据显示标识位进行界面显示和刷新	record_paint.c
录音命令控制模块	录音参数设置、开始/暂停/结束录音的命令控制	record_control.c
录音磁盘操作模块	录音应用的文件系统初始化、wav 录音回写文件头功能	record_disk.c
录音文件操作模块	录音应用的新建文件名、创建文件、查找磁盘剩余空间等操作	record_file.c
菜单模块	调用菜单控件以及菜单项的执行函数	record_menu.c
菜单项配置模块	菜单配置的数据	record_menu_config.c

10.3 与其它应用的同步和交互

从主界面进入录音并选择开始录音后，如果后台引擎存在，会发送 MSG_KILL_APP_SYNC 的消息给 manager，将后台杀死。

退出录音应用时，发送创建应用的消息给 manager，然后退出应用。

10.4 应用依赖库及其接口

系统和 libc 的接口 api.a

应用运行时库 ctor.o

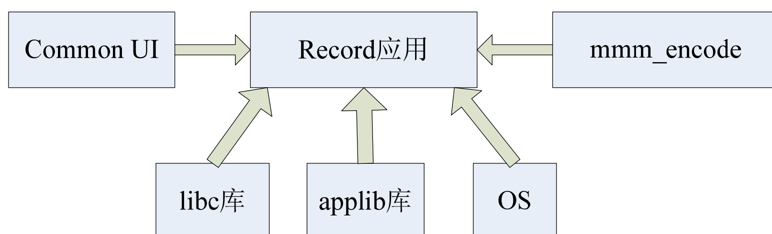
Applib 的全部函数

Common UI 的菜单，headbar，progress，timerbox 和其他公共提示模块

Codec 的 encode 模块

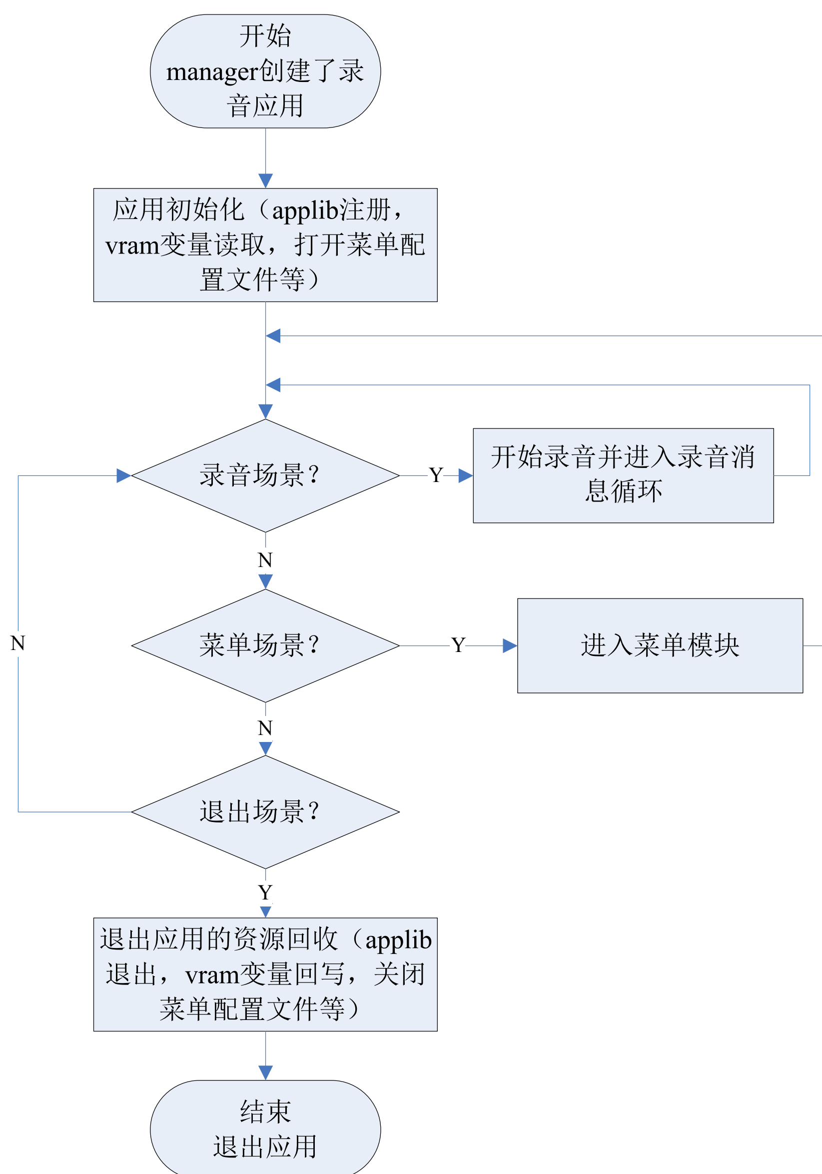
Enhanced 的文件选择器

文件系统接口


Common UI
Record应用
mmm_encode
libc库
applib库
OS

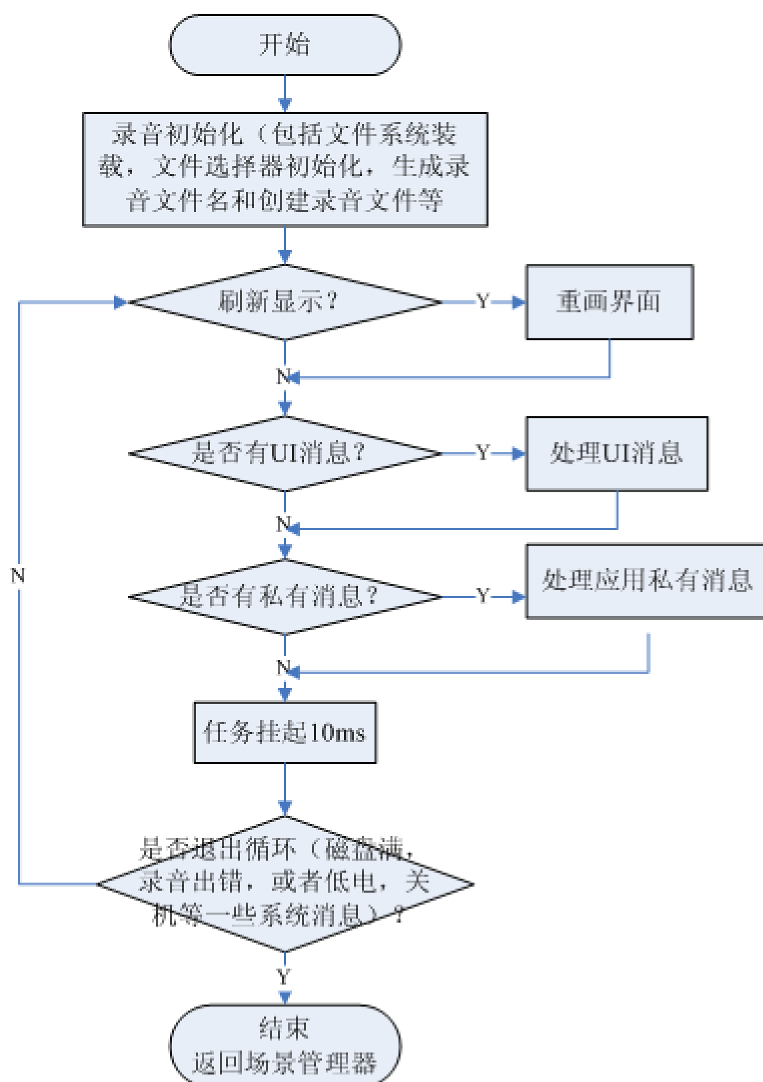
10.5 应用的业务流程

10.5.1 应用的总流程和场景调度流程



10.5.2 主菜单场景流程图

10.5.3 录音场景流程图



10. 6 如何实现录音监听功能

在开始录音前，即发送开始录音命令 MMM_MR_START 给中间件前，应用调用 set_pa_val 来打开监听，即可实现录音监听。在退出录音后调用 clear_ls_reg 关闭。

```
typedef volatile uint32*          REG32;
void set_pa_val(volatile uint32 val)//打开监听
```

```
{

        int i;
        /**********************init pa and fm in**********************/
        typedef volatile unsigned int*          REG32;
        //set dac clock
        *((REG32) (CLKENCTL)) |= (1 << CLKENCTL_DACCLKEN);

        //enable DAC Module, so dac reg(include pa reg can be set ?)
        *((REG32) (MRCR)) |= (1 << MRCR_DACIISReset);

        //MIXEN, analog mixer and PA enable
        *((REG32) (DAC_ANALOG1)) |= 1 << DAC_ANALOG1_PMP_PAEN;
        /*pavcc connect vcc*/
        *((REG32) (DAC_ANALOG1)) |= 1 << DAC_ANALOG1_PVCTV;
        //PMP mode AA MUTE disable
        *((REG32)(DAC_ANALOG1)) |= 1 << DAC_ANALOG1_PMP_AAMUTE;
        /*32 for PA volume level : 0 - 41*/
        *((REG32) (DAC_ANALOG1)) |= (val << DAC_ANALOG1_Volume_SHIFT) &
DAC_ANALOG1_Volume_MASK;

        /*pa output stage enable, can output now*/
        *((REG32) (DAC_ANALOG2)) |= 1 << DAC_ANALOG2_OSEN;

}

void clear_ls_reg(void)//关闭监听
{
        int i;
        /**********************init pa and fm in**********************/
        typedef volatile unsigned int*          REG32;
        //set dac clock
        *((REG32) (CLKENCTL)) &= ~(1 << CLKENCTL_DACCLKEN);

        //enable DAC Module, so dac reg(include pa reg can be set ?)
        *((REG32) (MRCR)) &= ~(1 << MRCR_DACIISReset);

        //MIXEN, analog mixer and PA enable
        *((REG32) (DAC_ANALOG1)) &= ~(1 << DAC_ANALOG1_PMP_PAEN);
```

版 权 所 有 侵 权 必 究

```
        /*pavcc connect vcc*/
        *((REG32) (DAC_ANALOG1)) &= ~(1 << DAC_ANALOG1_PVCTV);
        //PMP mode AA MUTE disable
        *((REG32)(DAC_ANALOG1)) &=~( 1 << DAC_ANALOG1_PMP_AAMUTE);
        /*32 for PA volume level : 0 - 41*/
        *((REG32) (DAC_ANALOG1)) &= ~( (0 << DAC_ANALOG1_Volume_SHIFT) &
DAC_ANALOG1_Volume_MASK);

        /*pa output stage enable, can output now*/
        *((REG32) (DAC_ANALOG2)) &= ~(1 << DAC_ANALOG2_OSEN);

}
```

10.7 录音命令的调用系列和顺序说明

开始录音调用录音中间件的命令顺序如下：
MMM_MR_OPEN：打开录音中间件
MMM_MR_AIN_OPEN：设定录音增益等参数
MMM_MR_SET_FILE：将文件系统句柄，录音文件句柄等参数传给录音中间件
MMM_MR_AUDIO_PARAM：设置录音文件类型/采样率/通道数/比特率等
MMM_MR_SET_DENOISE：设定降噪等级
MMM_MR_START：开始录音

获取录音状态命令：
MMM_MR_GET_STATUS，录音过程中需要循环发送此命令来获取已经录制的文件扇区，应用需要判断文件是否超过文件系统限制的单个文件大小，磁盘是否已满，是否需要分曲等处理。

暂停录音命令：
MMM_MR_PAUSE

恢复录音命令：
MMM_MR_RESUME

停止录音命令：
MMM_MR_STOP

版 权 所 有 侵 权 必 究

关闭录音调用录音中间件的命令顺序如下：

MMM_MR_CLEAR_FILE

MMM_MR_AIN_CLOSE：关闭录音通道

MMM_MR_CLOSE：关闭录音中间件

注意：录音文件的创建和关闭是应用去调用 vfs_file_create 和 vfs_file_close 来实现的。录音源的选择也是应用调用 enable_adc 和 enable_ain 去设置的。

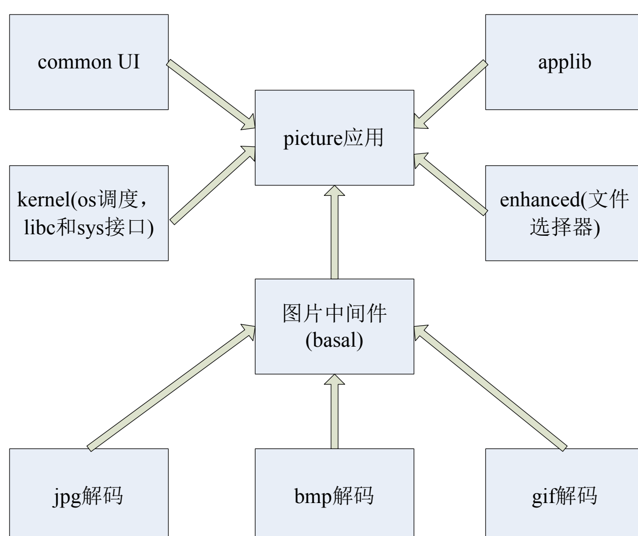
11 ap_picture 应用

11.1 需求概述

picture 应用通过调用系统接口和图片解码库完成图片文件的浏览，播放和菜单设置等功能。支持 JPG/BMP/GIF 解码，自动/手动播放控制，缩略图预览及多任务支持，同时支持通过配置工具完成 picture 应用的功能配置，图片旋转设置等。

11.2 总体架构设计

11.2.1 总体架构图



11.2.2 功能模块划分

模块名称	功能简述
初始化模块	负责应用初始化和退出，包括资源文件，菜单配置文件的加载和卸载，以及应用的场景管理
场景调度模块	对场景转移关系进行调度和处理
列表场景	调用文件列表模块显示 picture 文件列表
option 菜单场景	调用菜单列表模块显示 picture option 菜单项，完成 picture 应用功能设定
播放模块场景	实现图片的手动/自动播放功能
缩略图场景	以九宫格形式显示 9 张图片的缩略图
菜单场景	显示主盘目录和卡目录
退出模块	实现应用的退出处理

11.3 与其它应用的同步和交互

如果后台有音乐或 FM 播放，根据播放状态(播放/暂停)，在 optionmenu 菜单有正在播放/上一次播放菜单项，选择相应菜单项可转入相应的应用。如果后台为音乐播放，前台选择 bmp/gif 解码，会给出提示信息，查询是否关闭音乐并浏览图片。

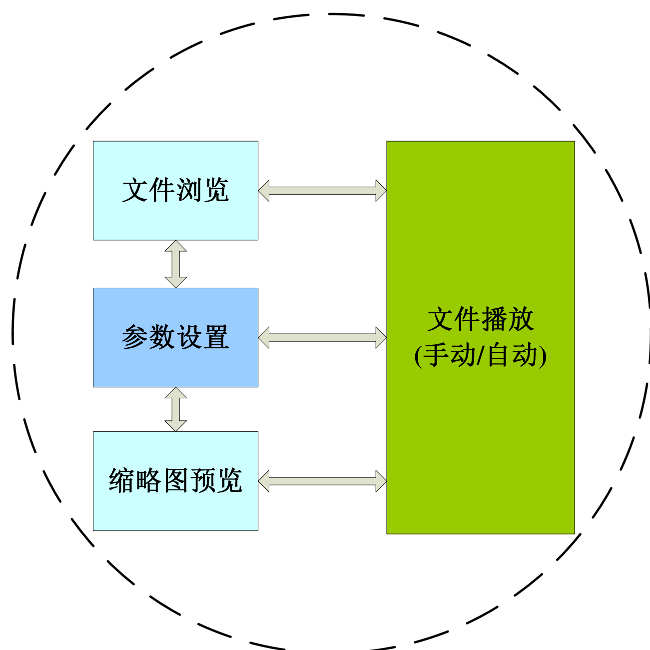
11.4 应用依赖库及其接口

系统和 libc 的接口
应用运行时库
applib 的全部函数
common ui 的菜单，文件列表，headbar 和其它公共模块
中间件文件选择器模块
图片中间件

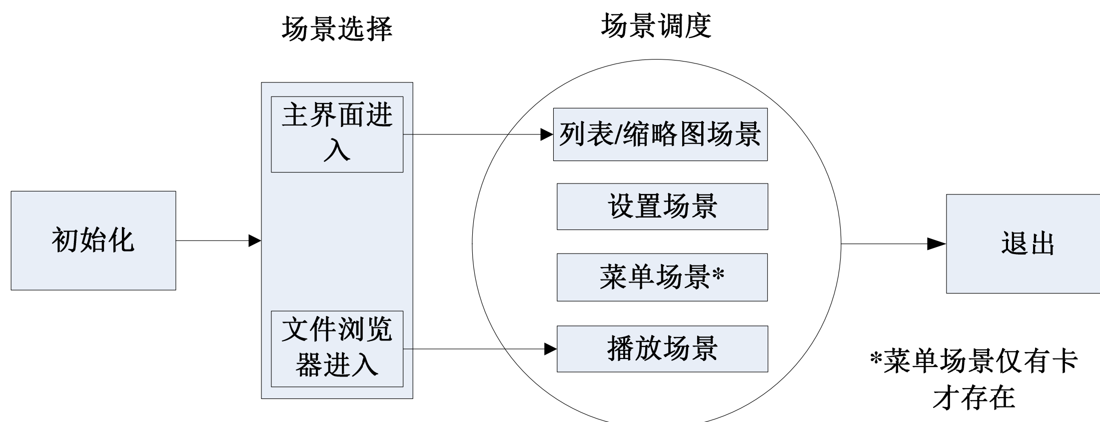
11.5 应用的业务流程

picture 需要实现文件浏览，文件播放，缩略图预览以及应用参数设置，因此模块划分如下所示:

版 权 所 有 侵 权 必 究

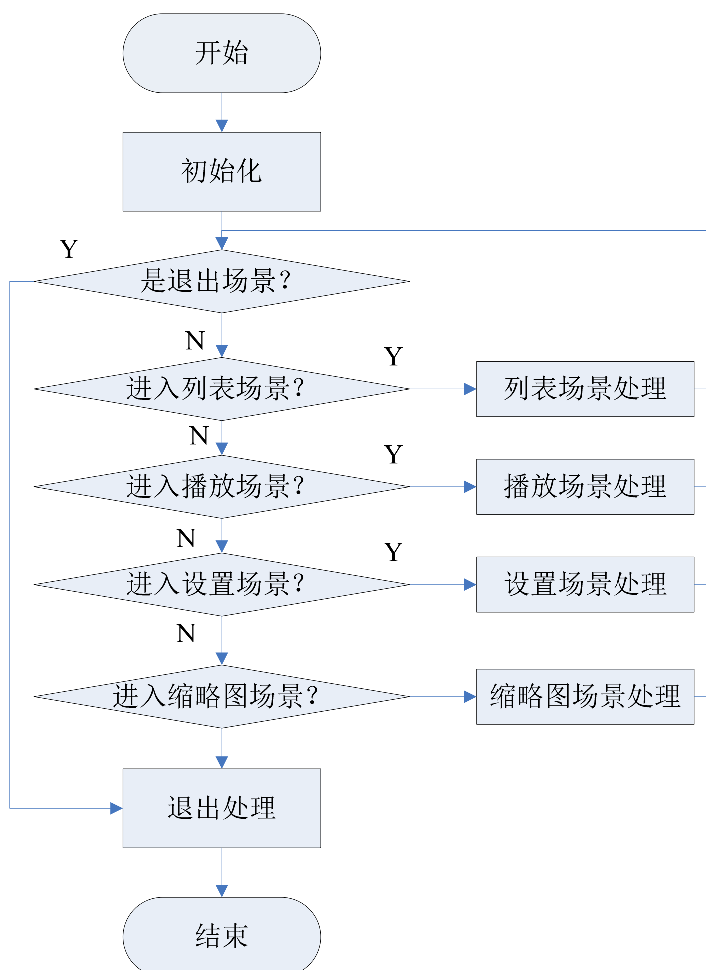


考虑与其他 ap 交互，picture 应用可以选择两种进入模式，从主界面 picture 图标进入和从文件浏览器应用选择图片文件播放进入，相应的流程图如下所示:



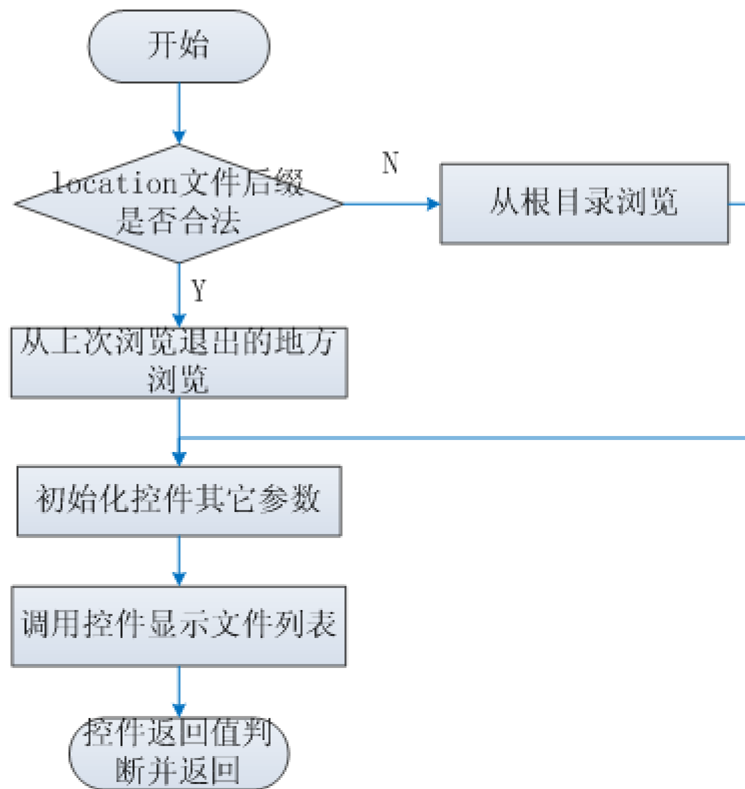
11.5.1 应用的总流程和场景调度流程

场景调度模块的关键是对场景状态控制的一个状态机，用于实现各个场景间的转换和调度，控制应用的运行流程。该模块实现场景的调度，根据状态选择不同的场景。us212a 应用均由场景组成，一个独立场景完成相对独立的一个功能。而主模块(main)负责应用的进入退出处理，以及场景调度选择。picture 主要划分了四个场景，分别对应上面描述的文件浏览，缩略图预览，文件播放以及参数设置 4 个功能模块。相应的场景调度循环如下图所示:



11.5.2 文件列表场景流程图

列表场景以列表形式列出介质上存储的图片文件。与文件浏览器应用(browser)不同，本模块对文件的解析是基于 picture 的播放列表(playlist)实现的。列表场景通过调用 common UI 的文件浏览控件实现。



11.5.3 图片预览场景流程图

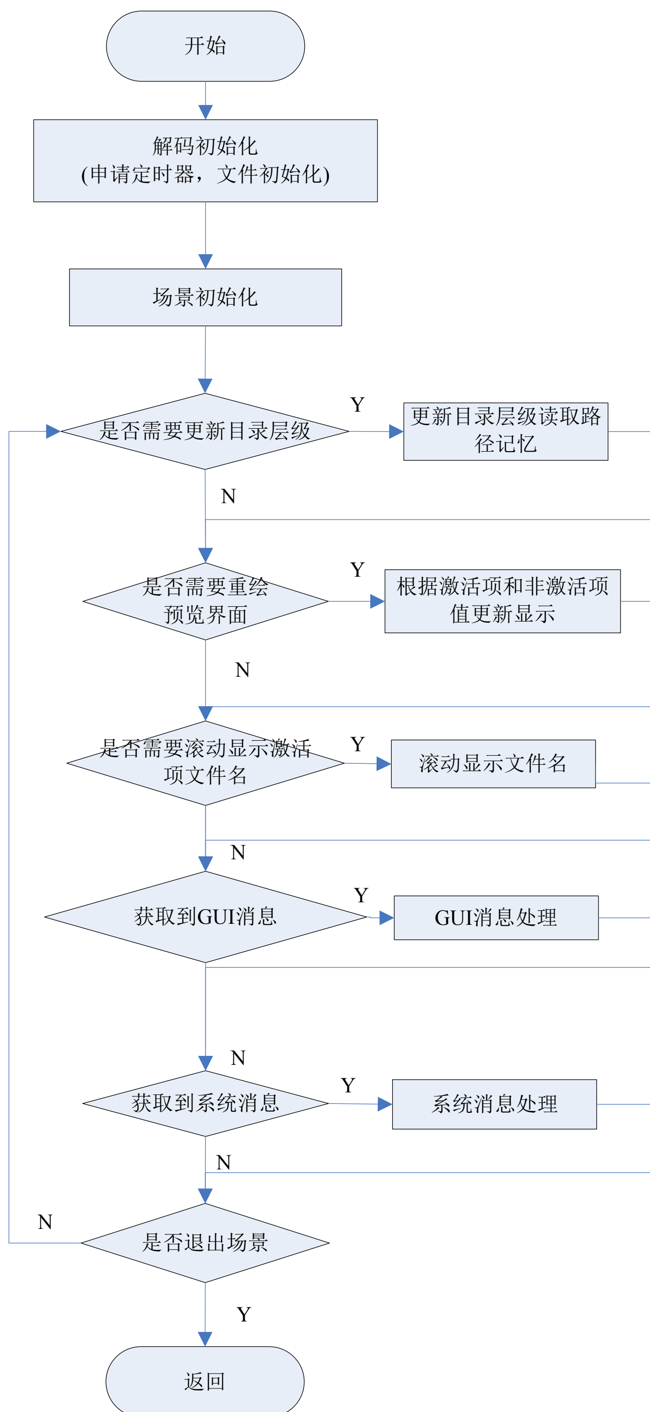
九宫格形式显示图片缩略图。顶部显示 headbar 和文件序号，底部显示文件名，中间每张图片区域为 42*42，每个图片窗大小为 40*40。.

缩略图模块功能与文件列表模块类似，提供图片文件的预览。对于激活项文件，以白色边框，底部文件名作为提示。在实现流程上，也类似于 common ui 的文件浏览控件。

缩略图模块按函数功能，可分为如下几个功能模块:

- ✧ 图片窗口计算及激活项和非激活项边框的绘制
- ✧ 激活项文件或目录的文件序号及文件名的显示
- ✧ 一屏图片文件的缩略图解码
- ✧ 按键响应处理
- ✧ 文件的浏览，选择，路径记忆

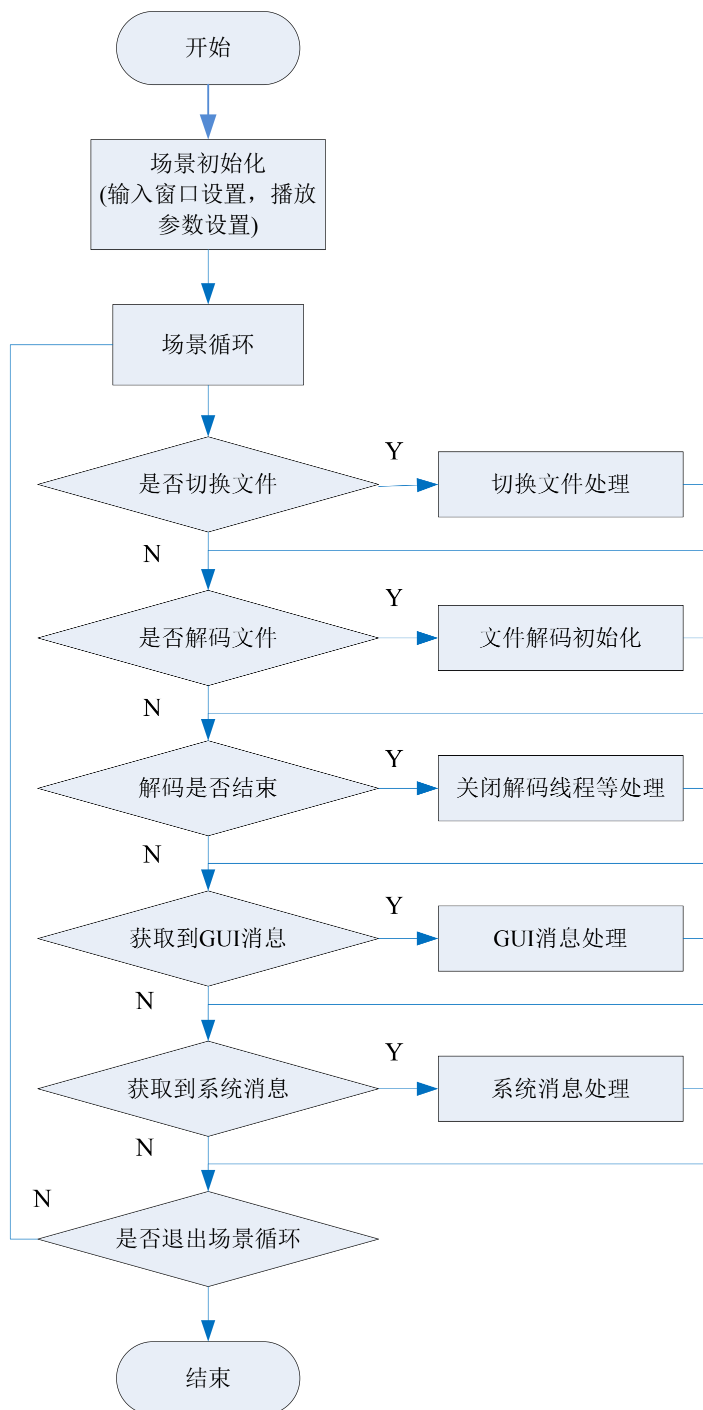
版 权 所 有 侵 权 必 究



11.5.4 图片播放场景流程图

图片播放模块实现图片的自动/手动播放，上下曲切换，对按键响应等功能。

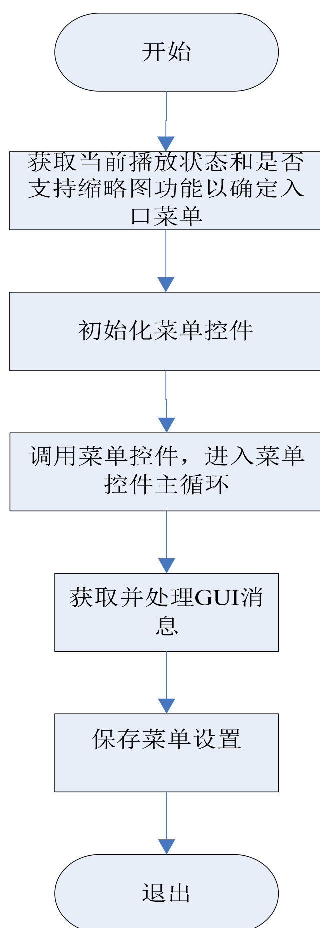
版 权 所 有 侵 权 必 究


开始
场景初始化
(输入窗口设置，播放
参数设置)
场景循环
是否切换文件
Y
切换文件处理
N
是否解码文件
Y
文件解码初始化
N
解码是否结束
Y
关闭解码线程等处理
N
获取到GUI消息
Y
GUI消息处理
N
获取到系统消息
Y
系统消息处理
N
N
是否退出场景循环
结束

11.5.5 弹出菜单场景流程图

菜单模块用于设置亮度，幻灯片播放时间，文件删除，动态转入音乐/FM,创建播放列表和切换盘符功能。由于后台存在三种情形:无后台，后台正在播放，后台暂停，，缩略图功能存在两种:无缩略图功能，有缩略图功能，因此会有 6 种组合组成 picture option 菜单入口。此外加上插卡显示的卡目录和主盘目录菜单，共 7 个菜单入口:

菜单功能依赖于 common ui 的菜单控件和菜单编辑工具生成的 mcg 文件。根据上述三种动态项确定入口菜单，初始化菜单控件结构体，调用菜单控件实现该模块功能。



11.5.6 图片菜单场景流程图

picture 的菜单场景不是一个一直存在的场景。只有在检测到卡存在的情形下才会出现该场景。该场景实现流程与 option 菜单场景类似，只是该场景不是通过在其它场景下按

option 键出来的。通过选择不同的菜单项，可以进入不同的 picture 列表。

菜单场景也是调用 ui_menulist()控件实现的，其流程与 option 菜单场景类似。

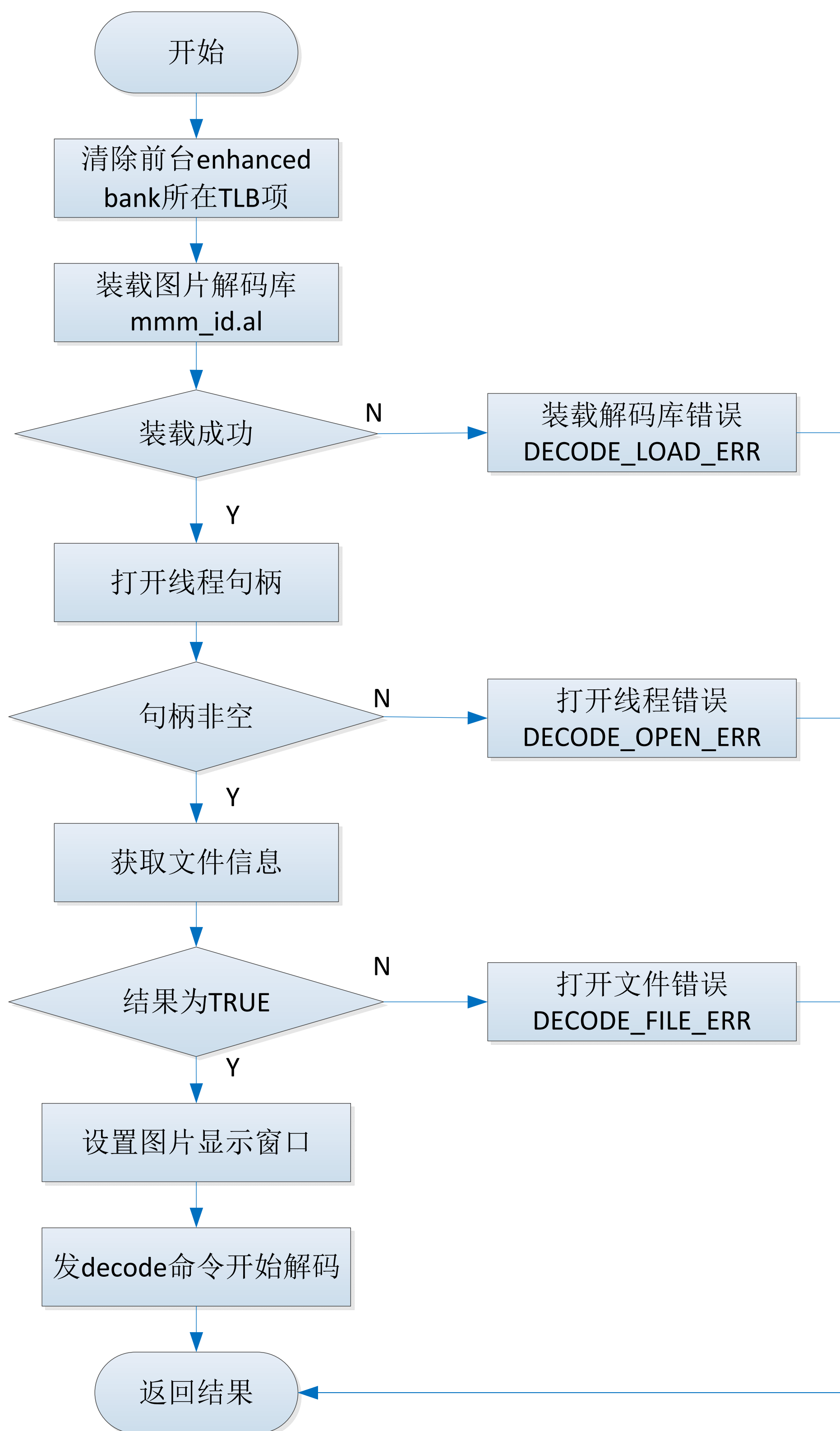
11.6 图片解码的数据流图

图片解码流程如下所示:

（1） 定位播放文件
（2） 装载解码库模块，创建解码线程并获取解码句柄
（3） 向解码库发送 set_file 命令开始解析文件
（4） 根据获取的解码信息设置刷屏模式和设置图片显示窗口
（5） 向解码库发送 decode 命令开启图片解码
（6） 前台 UI 进行消息循环，获取 GUI 消息和系统消息并作出处理
（7） 如果解码完毕或要求解码退出，关闭解码线程，释放解码库模块

由于解码库现在也实现为一个线程，因此发送 decode 命令开启图片解码后，在前台进程就存在两个线程:主线程为 UI 线程，实现消息获取和响应，次线程为解码线程，实现图片解码。次线程在解码结束后会挂起该线程，但线程所占用的资源需要应用发出关闭该线程之后才会释放。当前台响应按键要求切换上下曲时，主线程主动发 close 命令要求结束解码线程，解码线程查询到要求退出条件时会终止本线程，之后主线程释放次线程所占用的资源并调用文件选择器选择待播放文件。关键逻辑流程/序列图

版 权 所 有 侵 权 必 究


开始
清除前台enhanced bank所在TLB项
装载图片解码库 mmm_id.al
装载成功
N
装载解码库错误 DECODE_LOAD_ERR
Y
打开线程句柄
句柄非空
N
打开线程错误 DECODE_OPEN_ERR
Y
获取文件信息
结果为TRUE
N
打开文件错误 DECODE_FILE_ERR
Y
设置图片显示窗口
发decode命令开始解码
返回结果

11.7 如何修改图片预览的行和列的数量

在 ap_picture/Picture_preview.h 头文件中定义了图片预览的行和列的数量，修改 PIC_PER_OW 和 PIC_PER_COLUME 就可以修改图片预览的行列数量

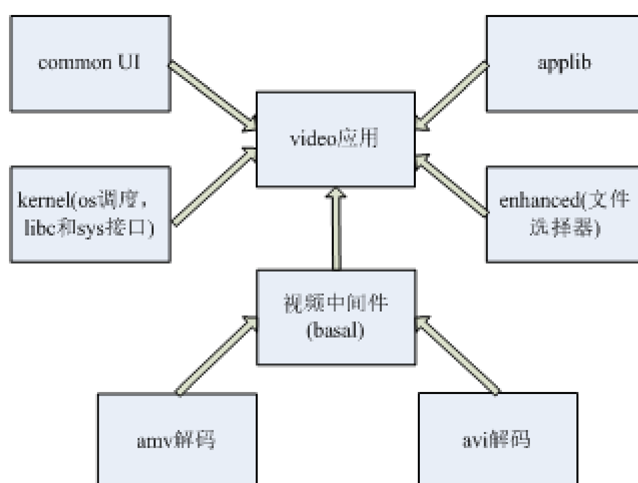
12 ap_video 应用

12.1 需求概述

video 应用通过调用系统接口和视频解码库完成视频文件浏览，播放和菜单设置等功能。支持 amv/avi 格式解码，支持视频进度条显示。

12.2 总体架构设计

12.2.1 总体架构图



12.2.2 功能模块划分

模块名称	功能简述
主模块	负责应用初始化和退出，包括资源文件，菜单配置文件的加载和卸载，以及应用的场景管理
系统消息处理模块	处理接收到的 video 系统消息
文件列表模块	调用文件列表模块显示 video 文件列表
option 菜单模块	调用菜单列表模块显示 video option 菜单项，完成 video 应用功能设定
视频播放模块	实现视频播放控制
菜单项配置模块	菜单配置的数据
视频公有模块	实现一些公共子函数，完成文件选择器初始化，盘符切换功能

12.3 与其它应用的同步和交互

从主界面进入 video 应用，未播放视频之前，如果后台有音乐或 FM 播放，根据播放状态(播放/暂停)，在 optionmenu 菜单有正在播放/上一次播放菜单项，选择相应菜单项可转入相应的应用。选择视频播放后，video 会主动发消息给 manager 进程，要求后台引擎退出。因此 video 视频播放时与其它后台进程是互斥的。

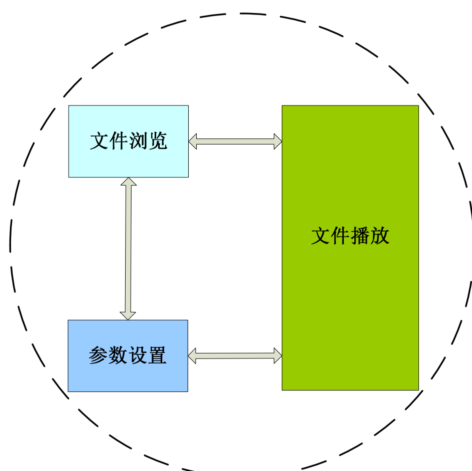
12.4 应用依赖库及其接口

系统和 libc 的接口
应用运行时库
applib 的全部函数
common ui 的菜单，文件列表，headbar 和其它公共模块
中间件文件选择器模块
视频中间件

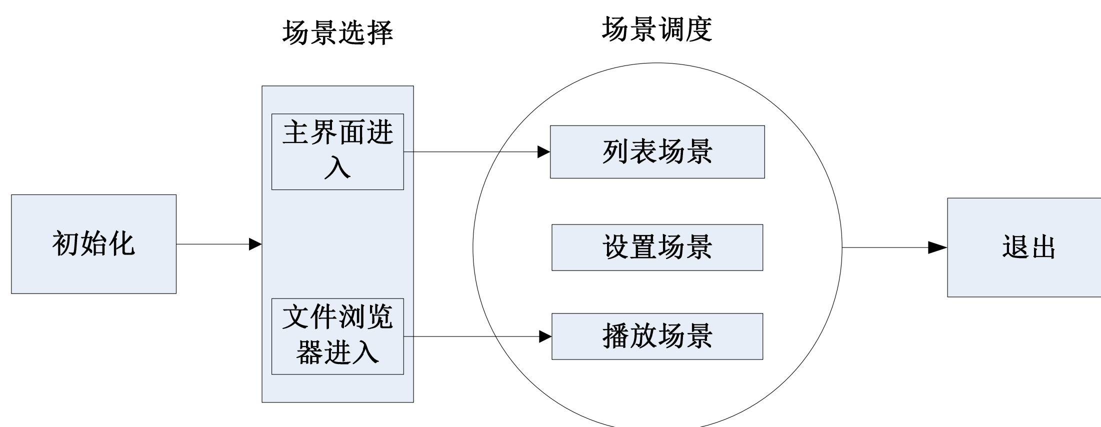
12.5 应用的业务流程

video 需要实现两个主要功能，即 video 文件浏览和 video 文件播放，此外还应包括对播放参数的设置，因此主模块划分如下图所示:

版 权 所 有 侵 权 必 究



考虑与其他 ap 交互，video 可以选择两种进入模式，即从主界面 video 图标进入和从文件浏览器应用选择 video 文件播放进入，相应的流程如下图所示：



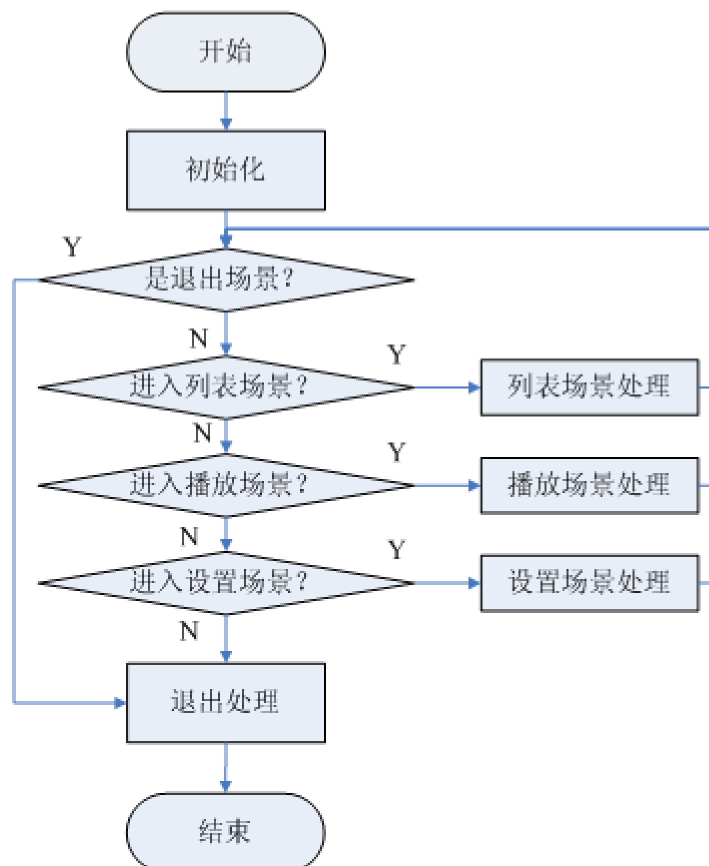
us212a 应用均是由场景组成，一个独立的场景完成相对独立的功能，主模块负责 ap 进入和退出，以及场景调度选择。video 主要划分了三个场景，分别对应文件浏览，文件播放和参数设置三个部分。

12.5.1 应用的总流程和场景调度流程

video 应用属于前台 UI 进程。应用在进入时通过_video_read_var()读取 VM 变量，在_video_app_init()函数中完成 applib 的初始化，软件定时器初始化，消息模块初始化，系统计时器初始化，消息初始化，打开资源文件和菜单配置文件，最后调用文件文件选择器初始化模块加载设备驱动和文件系统驱动。

video 应用退出时会根据得到的返回值结果和进入 video 应用的模式判断要返回哪个应用，然后发消息给 manager 进程要求创建相应的应用。然后与初始化顺序相反，在退出时会保存 VM 变量，调用_video_app_deinit()完成文件选择器退出，系统定时器注销，菜单配置

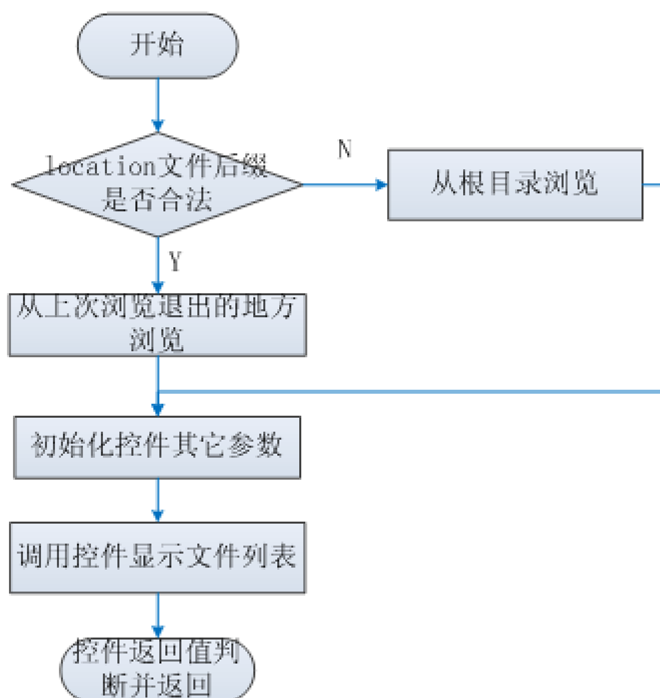
文件和资源文件关闭，最后执行 applib 库的注销操作并返回。应用的总流程如下图所示:



12.5.2 文件列表场景流程图

文件浏览模块以列表形式列出介质上存储的图片文件。与文件浏览器应用(browser)不同，本模块对文件的解析是基于 video 的播放列表(playlist)实现的。文件列表模块通过调用 common UI 的文件浏览控件实现。

文件列表场景流程图如下图所示:



12.5.3 视频播放场景流程图

视频播放模块实现 amv，avi 格式视频文件播放，三种显示界面的切换和 UI 显示，具体的实现流程如下::

（1） 定位播放文件

（2） 装载解码库模块，获取解码句柄

（3） 向解码库发送 set_file 命令开始解析文件

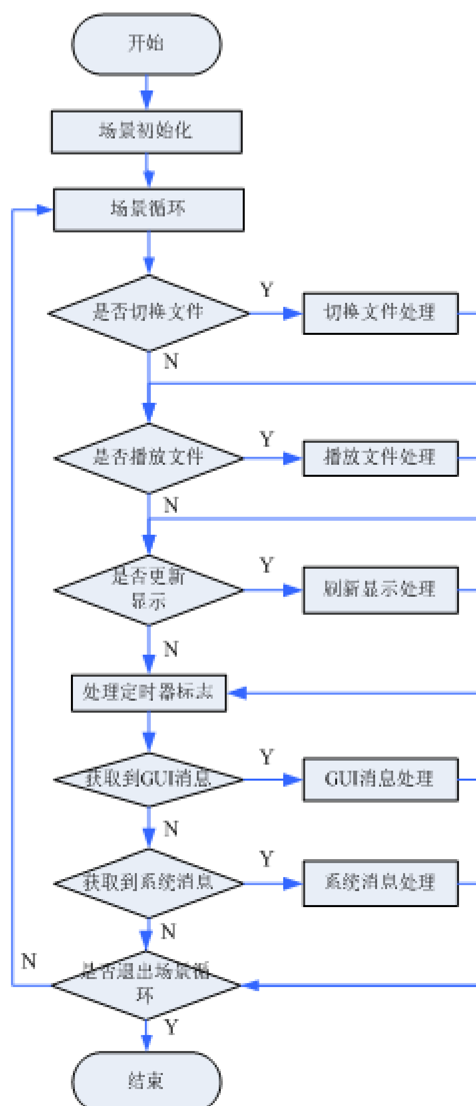
（4） 根据获取的解码信息设置 amv，avi 窗的坐标参数，codec 据此设窗，设刷屏模式

（5） 前台 UI 进行消息循环，获取 GUI 消息和系统消息并作出处理

（6） 根据处理结果判断是否继续解码，如果需要解码，发送解码命令解一帧视频帧或音频帧，如果需要快进退，发送快进退命令

（7） 其它情况，结束当前解码，切换上下曲或退出 play 模块

版 权 所 有 侵 权 必 究

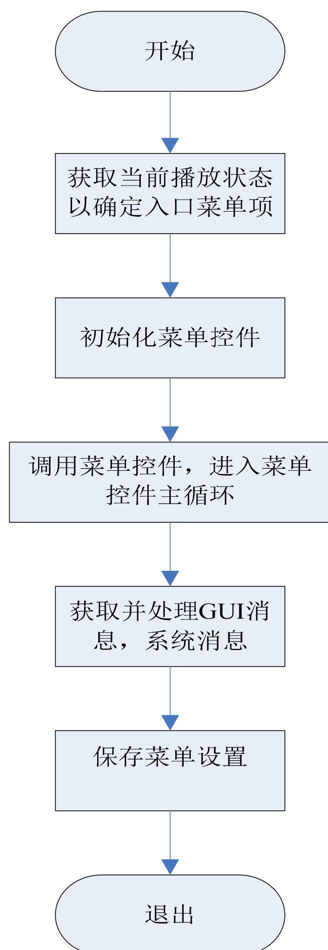


12.5.4 弹出菜单场景流程图

菜单模块用于设置亮度，文件删除,创建播放列表和切换盘符功能。video 共定义了 4 个菜单入口，分别对应于有卡时存在的选择盘符菜单，无播放/正在播放/上一次播放对应的三种动态菜单。

菜单功能依赖于 common ui 的菜单控件和菜单编辑工具生成的 mcg 文件。根据确定的入口菜单，初始化菜单控件结构体，调用菜单控件实现该模块功能。

版 权 所 有 侵 权 必 究



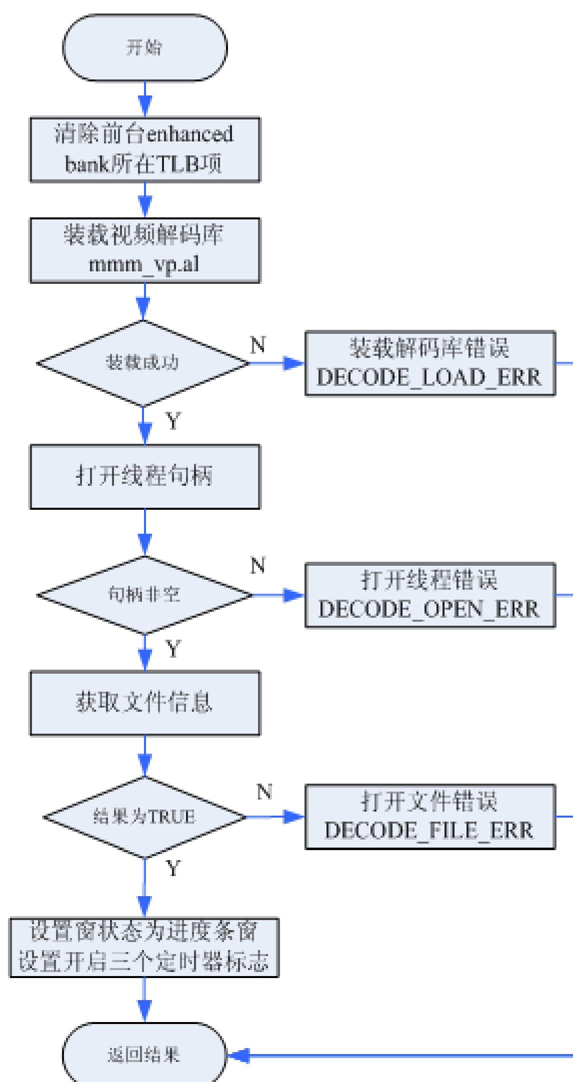
12.5.5 视频菜单场景流程图

video 的菜单场景不是一个一直存在的场景。只有在检测到卡存在的情形下才会出现该场景。该场景实现流程与 option 菜单场景类似，只是该场景不是通过在其它场景下按 video 键出来的。通过选择不同的菜单项，可以进入不同的 video 列表。

菜单场景也是调用 ui_menulist()控件实现的，其流程与 option 菜单场景类似。

12.6 视频解码的数据流图

视频解码和图片解码，音乐解码有类似的地方。要实现文件解码，首先要将解码库装载进内存，然后通过发送 SET_FILE 命令让解码中间件解析视频文件，最后循环发送解码命令实现视频解码。整个视频解码初始化的流程图如下:



12.7 video 是如何实现显示视频同时显示进度条等 UI 界面的

video 解码实际是串行执行的，在 video 播放过程中，每解完一帧 video 视频，在播放场景大循环中就检测是否有需要显示的界面元素，例如时间的显示，进度条的显示。需要注意的是，在 video 解码和刷新 UI 界面时，二者使用的刷屏模式，lcd 控制器的初始化参数是不同的。因此需要注意在刷界面之前需要先将 lcd 控制器初始化成 RGB 模式，同时设置刷屏模式为 DRAW_MODE_V_DEF,待刷新完毕之后，再恢复成 video 默认的刷屏模式。video 解码中间件在每次解码之前都会将控制器重新初始化成 YUV 模式，因此应用只需要恢复刷屏模式，不需要重新初始化 lcd 控制器。

版 权 所 有 侵 权 必 究

12.8 video 如何优化播放过程中显示性能

video 播放过程中支持边播视频边显示进度信息，由于视频播放可能码流较大，而视频解码和界面显示实际上是串行执行的，如果显示界面占用时间过多，就有可能出现声音有断音的情形。而一般情况下，显示驱动的控件实现都更为复杂，函数层次也很多，在效率上与 video 解码有冲突，因此 video 在 ap 层实现了效率更高的显示界面的方法。
video 在播放过程中，需要实现进度条，音量条，时间显示，文件名显示等控件。为了提高效率，将相关图片，文件名缓存到 RAM 中。视频播放过程中使用了从 0x3c000 开始的 15K buffer 作为缓存 buffer。显示驱动提供了接口函数 ui_read_pic_to_buffer()用于将相应的资源文件读入到指定 buffer。

12.9 视频播放过程中如何实现全屏到进度条，音量条的显示

视频解码模块没有像图片解码做成一个单独的线程，因此需要应用在主循环调用解码命令实现视频的不间断播放。又由于视频在播放界面存在三种状态转变:进度条窗，音量窗，全屏窗。为此定义三种窗类型:

```
typedef enum
{
    VIDEO_VOLUMEBAR = 0,
    VIDEO_PROGRESSBAR,
    VIDEO_FULLSCREEN
} video_window_e;
```

利用变量 g_video_prev_window 和 g_video_cur_window 记录其状态。

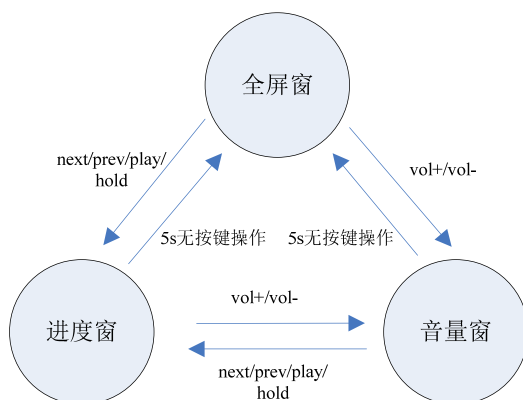
在进度条窗需要时间刷新和文件名滚动显示，5s 之后需要自动从播放窗转为全屏窗，因此需要三个定时器，分别用于时间刷新，文件名滚动和全屏转换。其 ID 为 timer_flush_progress,timer_flush_filename,timer_flush_fullscreen。为了控制这三个定时器的开启和关闭，定义变量 g_video_timer_flag，定义如下:

```
#define up_video_fullscreen_timer    0x01
#define up_video_filename_timer      0x02
#define up_video_progress_timer      0x04
#define clr_video_fullscreen_timer   0x08
#define clr_video_filename_timer     0x10
#define clr_video_progress_timer     0x20
```

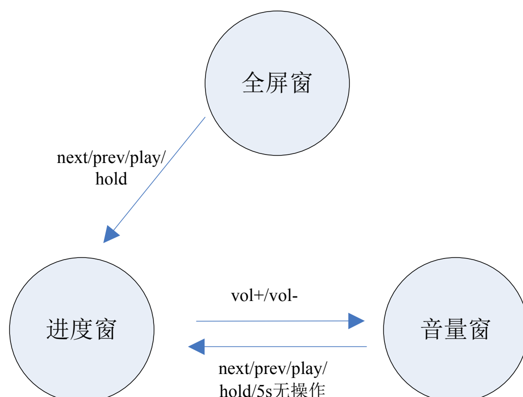
用 6 个 bit 表示三个定时器的开启和关闭。

视频存在三种窗状态，三种状态下视频文件的有效播放区域是不同的，在前面流程图可看出，在显示刷新模块会根据刷新标志位发送不同的设窗命令，视频解码据此算出需要保留的区域供 UI 显示。三种窗状态在视频处于播放状态转换图如下所示:

版 权 所 有 侵 权 必 究



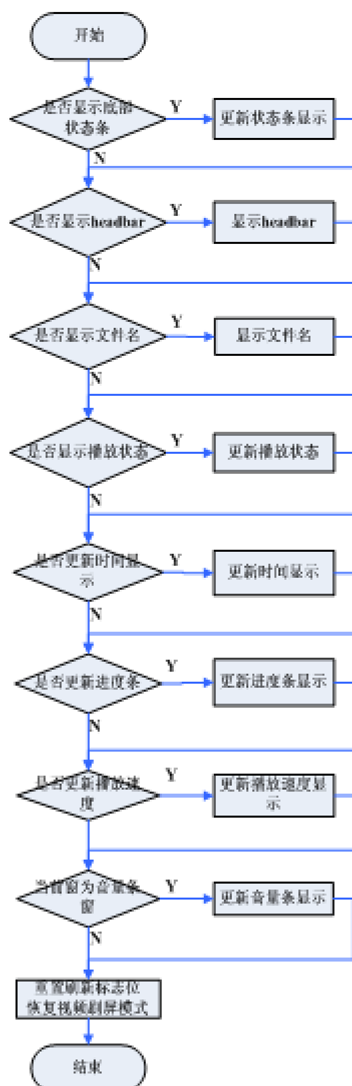
当视频处于暂停状态时，根据上面的转换图可知，视频窗状态一定为进度窗，且暂停状态进度窗不会在 5s 后转为全屏窗，因此暂停状态转换图为:



如果正常播放，则开始状态为进度条窗，调用显示刷新模块显示 video 播放界面。显示刷新模块完成播放界面的显示，向解码中间件发送显示进度条窗，音量条窗，全屏窗的命令。在 amv 格式播放时，视频的刷屏模式和 UI 界面元素的刷新模式有冲突，因此在调用完显示刷新模式后，要重新恢复视频的刷屏模式。

整个显示刷新模块的处理流程如下:

版 权 所 有 侵 权 必 究



需要注意，为了提高视频显示时 UI 刷新的效率，对时间，进度条，headbar，清屏等函数没有使用 UI 驱动的相关函数，而是通过将图片缓存到 RAM，然后使用 DMA 刷屏输出。这样可以最大限度提高视频显示性能。

12.10 视频菜单配置文件为何编译模式不同

视频应用为了减少 bank 切换，提高解码效率，程序都采用了 O2 编译模式，这样编译的代码量更小，执行效率更高。但菜单配置文件记录了相应的菜单头和叶子菜单的内容，如果采用 O2 编译，会打乱这种排序，导致菜单配置工具无法解析该文件的内容。因此菜单配置文件单独采用 O0 编译。实际上该文件只是供菜单配置工具使用，在小机端并不真正使用该文件，因此不需要 2k 代码量的限制。

版 权 所 有 侵 权 必 究

13 ap_radio 应用

13.1 需求概述

Radio 应用用于实现对 FM 电台的播放，搜索，保存和删除等功能。主要实现的是显示层相关功能，和 FM 硬件模组相关的功能由 FM 驱动单独提供接口实现。

功能需求如下：

1. 支持预设电台和用户自定义电台的播放
2. 支持预设电台和用户自定义电台列表的显示
3. 支持预设电台的保存，删除，包括删除全部预设电台
4. 支持手动调谐和自动调谐的界面显示，有效电台的保存。
5. 支持三种电台频段：普通频段，日本频段和欧洲频段
6. 支持菜单跳转到 FM 电台录音或录音文件播放功能应用。

性能需求如下：

1. 能够稳定搜索到有效电台
2. 满足 FM 输出幅度，搜台灵敏度等硬件指标要求。

13.2 总体架构设计

按界面划分，radio 应用的界面包括主菜单场景，FM 播放场景，电台列表场景（包括预设电台列表和用户电台列表）和 option 子菜单场景（包括预设列表的子菜单和播放场景的子菜单）共四个主要部分。 另外，半自动搜台和全自动搜台，作为比较独立的模块，其场景的绘制，由单独模块处理，不包含在场景调度器中调度。

13.2.1 总体架构图

应用的总体架构设计图如下：

版 权 所 有 侵 权 必 究

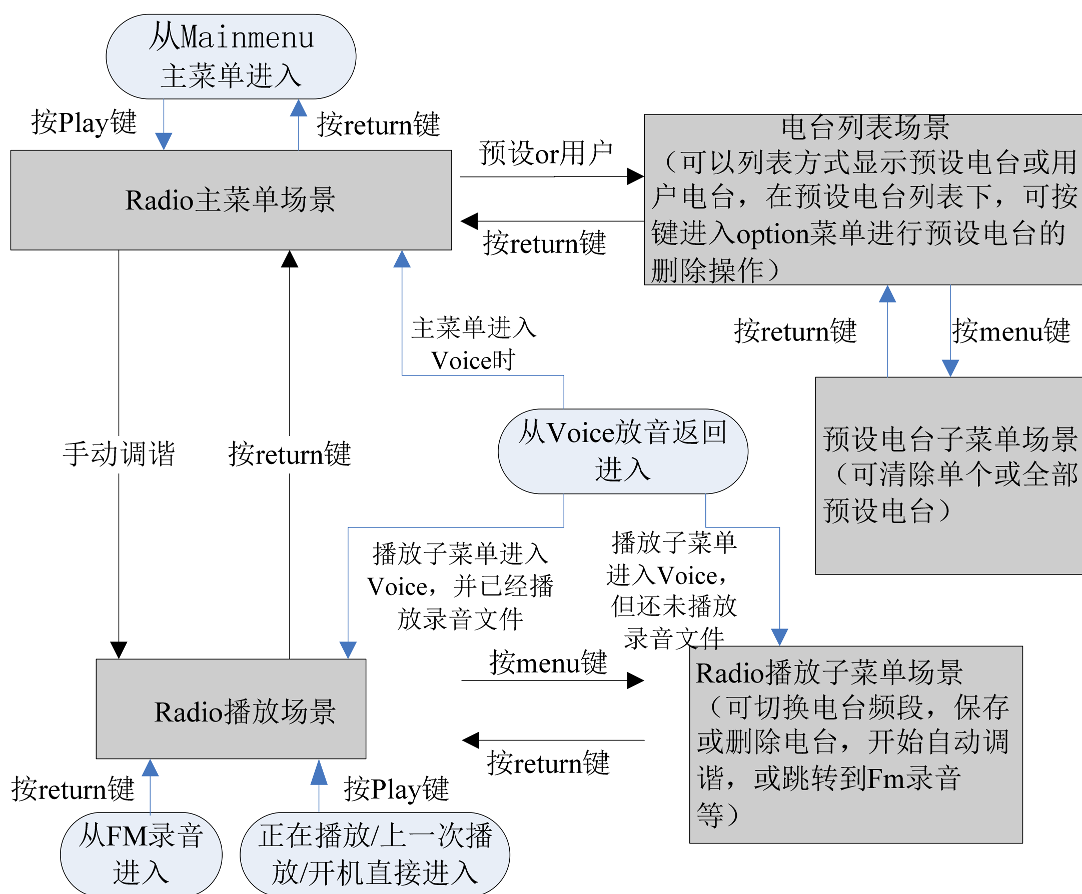


图 15-1 Radio 应用总体架构图

13.2.2 功能模块划分

按功能模块划分，Radio 应用的主要模块可分为：主模块(包括场景调度器模块)，全自动搜台模块（包括硬件全自动搜台，软件全自动搜台，以及相关的处理子函数），半自动搜台模块，电台列表模块，菜单功能模块，播放功能模块，用户电台解析模块，引擎层命令模块，常驻代码功能模块以及全局数据模块。

具体实现功能如下：

表 13-1 功能模块划分表

模块名称	功能简述	对应文件
主模块	主要是实现应用初始化，参数恢复，备份，	app_radio_main.c

	资源开关，Radio 场景调度等。	app_radio_scene_dispatch.c
全自动搜台模块	主要是进行全自动搜台功能的初始化，退出，界面的刷新，根据搜台模式，实现硬件或软件全自动搜台功能	app_radio_autoseek.c app_radio_autoseek_bank.c app_radio_softseek.c
半自动搜台模块	实现硬件或软件半自动搜台功能	app_radio_search.c
电台列表模块	主要实现预设电台列表和用户电台列表的浏览，以及预设电台的保存和删除功能。	app_radio_saved_preset.c app_radio_delete_preset.c app_radio_list_bank.c app_radio_list_common.c app_radio_list_keydeal.c
菜单功能模块	主要包括 Radio 主菜单的显示，Radio 播放场景子菜单以及预设列表子菜单的显示，入口菜单和叶子菜单的配置，菜单功能函数的实现等。	app_radio_mainmenu.c app_radio_menu_config.c app_radio_menu_process.c app_radio_option.c
播放功能模块	主要实现 Radio 播放场景的显示，播放场景下按键消息的处理等。	app_radio_playing.c app_radio_playing_deal.c app_radio_playing_user.c
用户电台解析模块	主要是解析用户输入的电台文件，转换为可供显示的用户电台列表。支持对 Ascill 码及 Unicode 码电台文件的解析。	app_radio_userlist_parse.c app_radio_userlist_parse_uni.c
引擎层命令模块	主要是将给 Radio 引擎发送消息的相关函数封装为一个模块。	app_radio_control.c
常驻代码功能模块	主要是将使用频率比较高的函数存放在常驻空间里，减少 bank 切换，提高效率。	app_radio_rcode.c
全局数据模块	主要是统一管理 Radio 应用使用的全局数据	app_radio_global_data.c

13.3 与其它应用的同步和交互

manager：

1. 当有后台播放音乐时，如果切换到 Radio 播放，需要先发送关闭后台引擎的消息给 manager，将 music 引擎进程关闭，然后再创建 Radio 引擎进程。
2. 当退出 Radio 应用时，需考虑进入哪一个 AP。需要向 manager 发送创建新 AP 的消息。比如从 Radio 进入 FM 录音，需创建 Record 进程；从 Radio 进入 Voice，需创建 Browser 进程；从 Radio 回到 Music，需创建 Music 前台进程；从 Radio 返回 MainMenu，需创建 MainMenu 进程等。

Fm engine:

版 权 所 有 侵 权 必 究

1. 当开始播放 Radio 时，Radio UI 进程需首先向 manager 发送创建 Radio 引擎进程的消息，在 Radio 播放过程中，如有操作 FM 模组硬件的操作，需要 Radio UI 进程发送命令给 FM Engine 进程，再由 Engine 调用 FM 驱动接口，实现硬件操作功能。Radio UI 与 Fm Engine 交互相关的函数，基本都在引擎层命令模块中。

13.4 应用依赖库及其接口

Radio 应用使用了如下库：

1. 系统和 libc 的接口 api.a
2. 应用运行时库 ctor.o
3. Applib 的全部函数
4. Common UI 的菜单，headbar，动画和其他公共提示模块

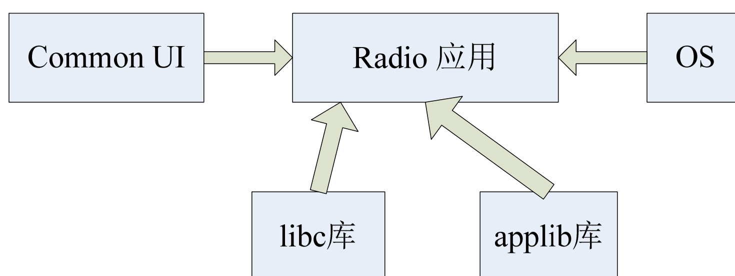


图 15-2 应用依赖关系图

13.5 应用的业务流程

13.5.1 应用的总流程和场景调度流程

主模块所实现的功能主要是承载应用的启动，应用的退出，场景切换；本模块非常驻代码，对应的源码文件为 app_radio_main.c 和 app_radio_scene_dispatch.c.

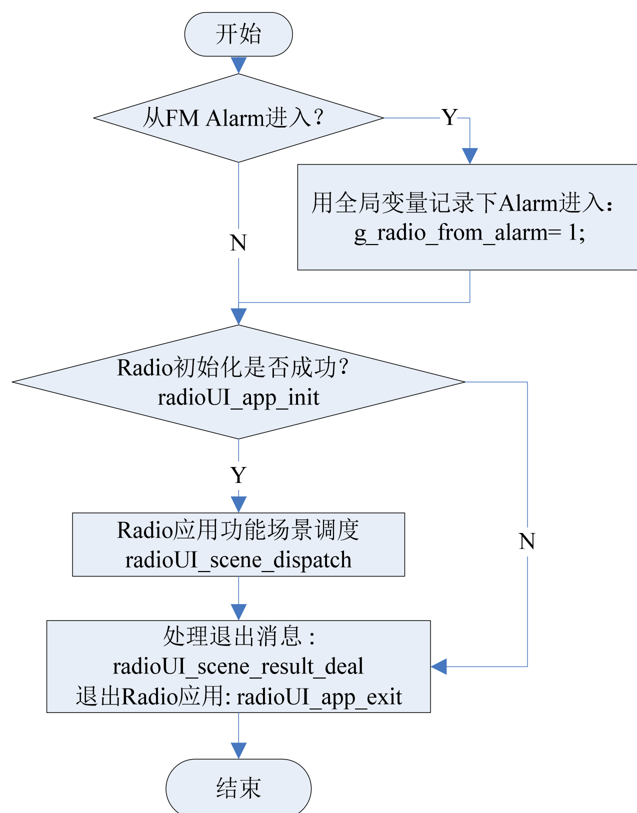


图 15-3 Radio 总流程图

Radio 应用的场景调度器主要实现在如下四种场景下的调度：

a) RadioUI 主菜单场景

b) RadioUI 播放场景

c) 电台列表场景，包括预设电台列表和用户电台列表场景。两者通过参数 radio_listtype_e g_radio_listtype 区分。另外，由于进入预设电台列表的方式有多种，比如正常浏览，选择保存到预设，或者删除预设进入，为进行区分，使用变量 radio_list_entry_e g_stalist_entry

d) Option 子菜单场景，包括预设电台列表下的子菜单，以及 Radio 播放场景下的子菜单。两者通过参数 radio_opt_subscene_e g_option_type 区分。

App_radio_scene_dispatch.c 用于实现 RadioUI 的场景调度功能，场景调度流程图如下：

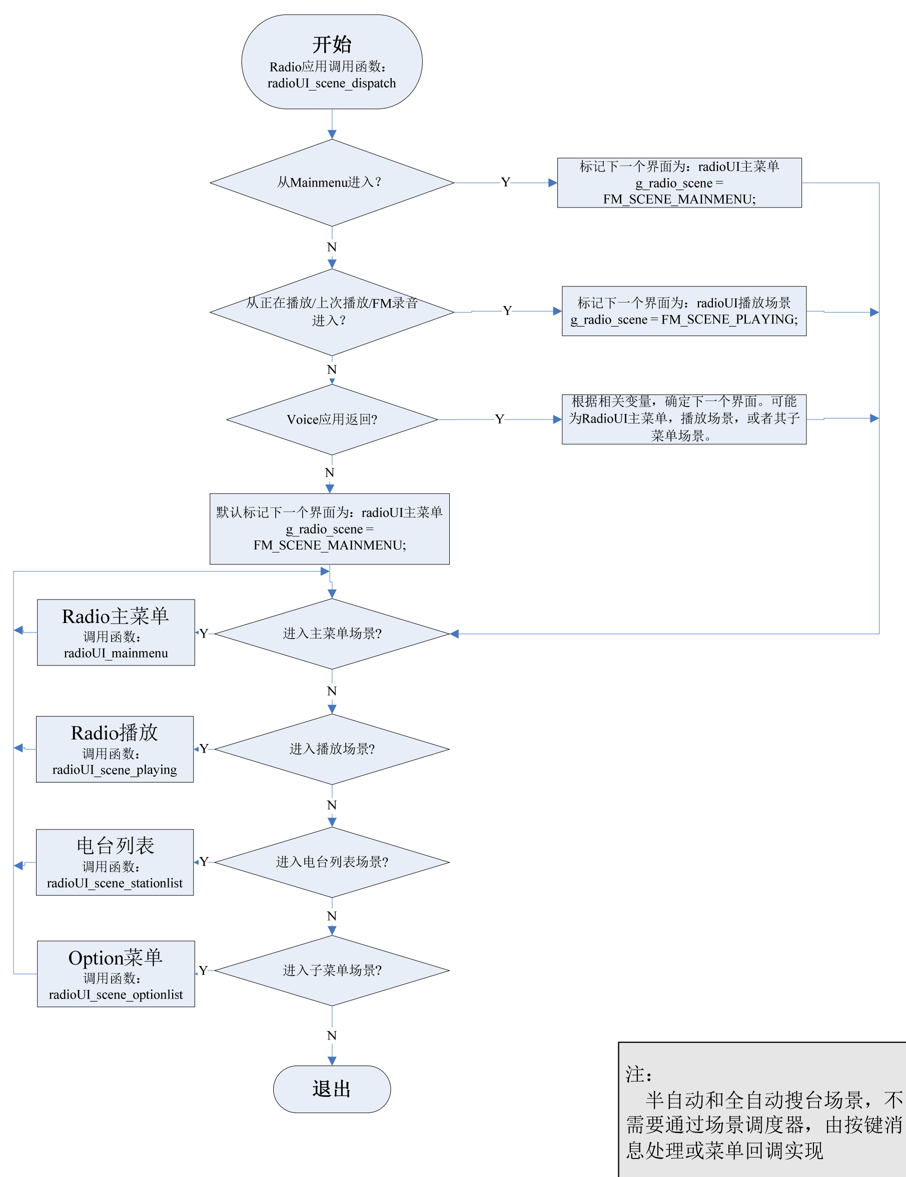


图 15-4 Radio 场景调度流程图

13.5.2 电台播放场景流程图

电台播放场景，除了需要考虑正常的播放状态外，也需要考虑半自动搜台时的界面刷新，以及播放用户电台时的界面刷新。因此，将频率和电台号等与当前频率相关项的显示，作为单独的函数提供，以方便半自动搜台时的调用。界面需要刷新的各个区域，以 bitmap 区分。

版 权 所 有 侵 权 必 究

另外，由于可能从各种场景下进入电台播放，其返回时需要回到不同的场景，所以应用中使用了一个全局的变量 g_playwin_mode，用于标识进入播放时的场景。

电台播放场景流程图如下：

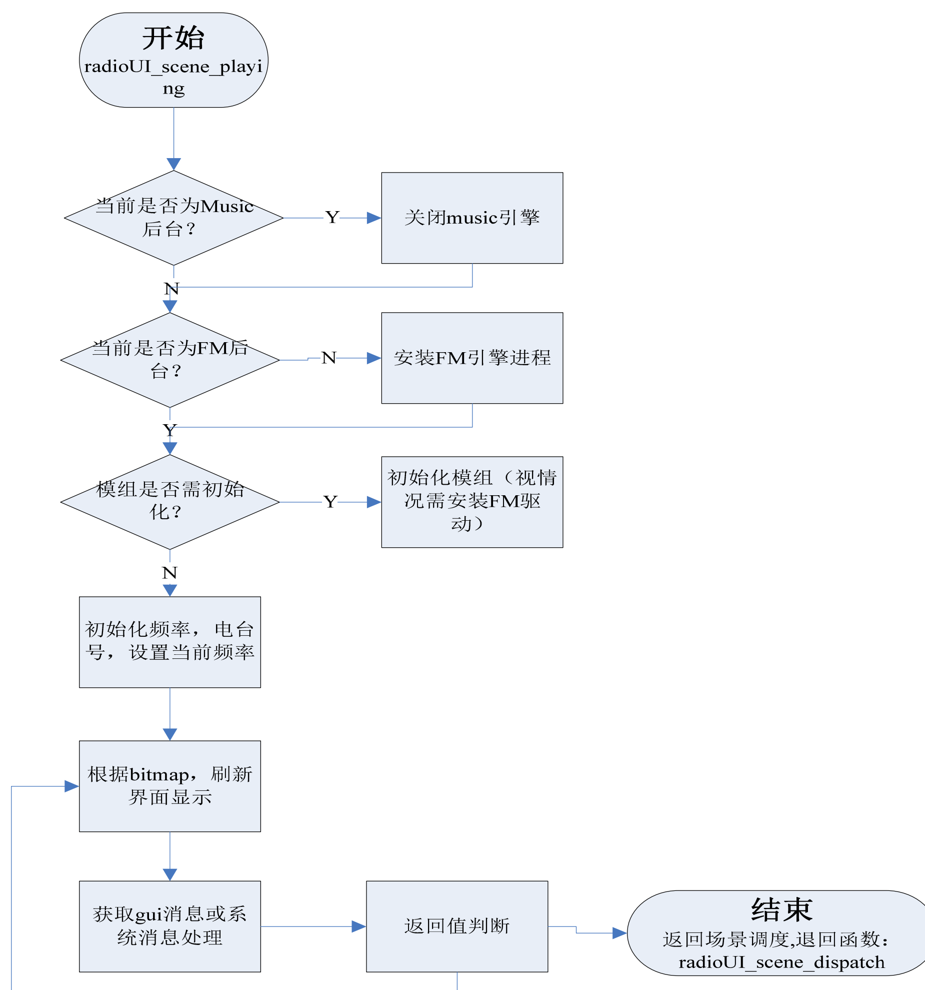


图 15-6 Radio 电台播放场景流程图

注： 在播放场景的 gui 消息处理中，如果长按 prev 或者 next，会进入半自动搜台场景。该场景由 gui 消息处理中直接调用 scene_play_search 进入，不经过场景调度器管理。

在半自动搜台中，会根据所配置的搜台模式（硬件搜台或软件搜台），分为不同的处理流程分开处理，从而将软件搜台和硬件搜台完全独立开来，如下：

```
if(g_radio_config.seek_mode==HARDSEEK)
{       //处理硬件半自动搜台
        ret=deal_hard_search(dir);
}
else
{       //处理软件半自动搜台
        ret=deal_soft_search(dir);
}
```

13.6 如何增加预置电台的数量

为方便修改所支持的保存电台数量，RadioUI 应用中使用了一个宏，用于定义支持的电台数。如下：

```
/*
定义预设电台列表中所支持的保存电台数，需开出相应大小 buffer*/
#define   MAX_STATION_COUNT   30
```

需要注意几点：

1. 这个宏同时定义了预设电台和用户电台支持的电台数量，目前每个频段的预设电台，以及用户电台支持的电台数量默认为是一样的。
2. 如果增加预设电台的数量，数据空间的使用量会相应增加，需注意空间是否超出。包括如下：

版 权 所 有 侵 权 必 究

```
//radio ui 配置参数
typedef struct
{
    /* 魔术数*/
    uint16 magic;
    /* 预设列表( 普通频段)*/
    uint16 fmstation_us[MAX_STATION_COUNT];
    /* 预设列表( 日本频段)*/
    uint16 fmstation_jp[MAX_STATION_COUNT];
    /* 预设列表( 欧洲频段)*/
    uint16 fmstation_eu[MAX_STATION_COUNT];
    /* 当前播放的电台信息*/
    fm_play_status_t FMStatus;
    /* 当前播放波段, Band_MAX 表示当前播放为用户电台*/
    radio_band_e band_mode;
    /* 进入放音的入口，决定了返回到 radioUI 的入口*/
    enter_voice_mode_e enter_voice;
    /* 配置模组采用的搜台模式( 默认模式可通过配置项配置) */
    FM_SeekMode_e seek_mode;
} radio_config_t;

/* radio 用户电台结构*/
typedef struct
{
    /* 用户电台列表频点，数组下标为电台号-1*/
    uint16 fmstation_user[MAX_STATION_COUNT];
    /* 当前播放的用户电台名称 */
    char station_name[40];
} radio_userlist_t;

//用户电台列表索引号和电台号映射表，
//数组下标为索引号，数组值为电台号
//用户电台最多支持编辑数 MAX_STATION_COUNT
uint8   g_userlist_table[MAX_STATION_COUNT];

//自动搜台中用于保存有效电台的临时 buffer
uint16 Auto_tab[MAX_STATION_COUNT];
```

14 fm_engine 引擎

14.1 需求概述

Fm 引擎作为后台进程，是 RadioUI 进程对 FM 模组进行控制的衔接层。通过 FM 引擎，可以将 RadioUI 和 FM 硬件剥离开，如果有需要，可以在 RadioUI 进程退出的情况下，由其他进程向 Fm 引擎发送 Radio 模组控制消息，实现电台切换等功能。

Fm 引擎的功能需求如下：

1. 向上正确接收来自 RadioUI 进程或其他进程的消息，并作出处理。
2. 向下调用 FM 驱动提供的各个接口，对 Fm 模组进行控制，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。
3. 引擎应用的退出不由 RadioUI 进程控制，而是由相关进程（如 Music，Video）向 manager 发送消息，要求 Fm 引擎进程退出。

14.2 总体设计

FM 引擎应用相对比较简单，首先将 Fm 引擎需要处理的消息分为 3 类： FM 模组初始化及卸载相关消息，FM 模组设置类消息，FM 模组状态读取类消息。 接着按照这 3 类消息，将模块划分如下：

表 16-1 功能模块划分表

模块名称	功能简述	对应文件
主模块	主要是实现引擎应用初始化，参数恢复，备份，fm 驱动的安装等。	Fmengine_main.c
消息处理入口模块	Fm 引擎消息处理的总入口，当判断到引擎有私有消息需处理时，调用 fmengine_message_deal，按消息分类处理。	Fmengine_control.c
消息处理模块 1	处理初始化，卸载以及模组设置类消息	Fmengine_message_deal.c
消息处理模块 2	处理 Fm 模组状态获取类消息	Fmengine_message_deal_2.c

版 权 所 有 侵 权 必 究

14.3 FM 引擎的业务流程

14.3.1 FM 引擎的总体流程

按照上述所说的模块划分，FM 引擎的总体流程图如下：

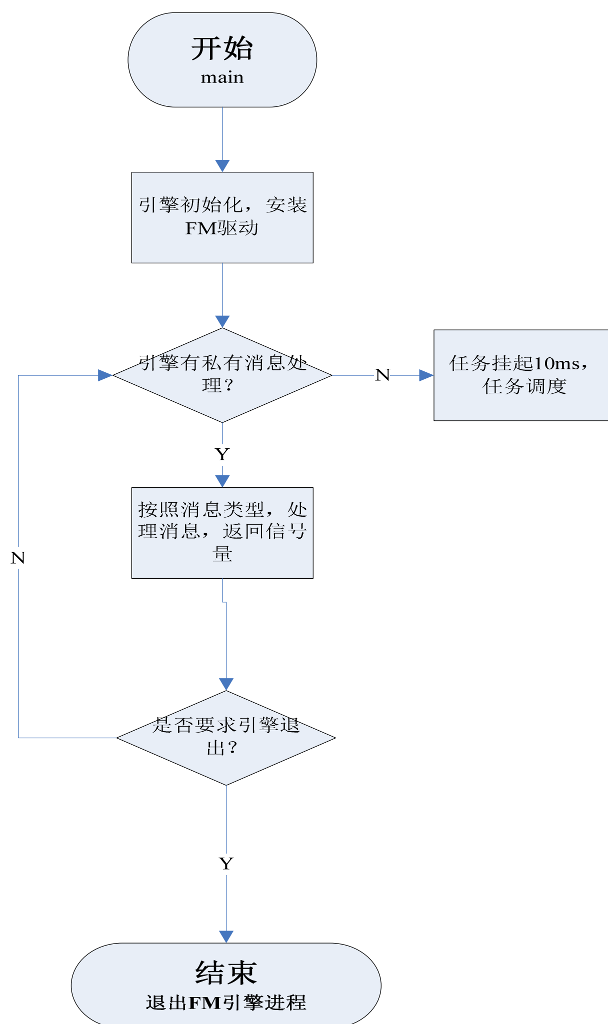


图 16-1 FM 引擎总体流程图

14.3.2 FM 引擎的消息处理流程

FM 引擎的消息处理，总的思路是： 当引擎有私有消息需要处理时，处理私有消息，这时根据消息的类型，在引擎层处理完毕，或者调用 FM 驱动接口，操作硬件； 当引擎没有消息需要处理时，挂起 FM 引擎进程，以进行任务调度。

消息处理流程图和图 16-1 FM 引擎总体流程图类似，不再重复。

14.4 与其他模块的同步和交互

FM 引擎与 ap_manager 的交互如下：

各前台应用可以向 manager 发送 MSG_APP_QUIT 消息，要求 FM 引擎进程退出。

FM 引擎与 ap_radio 的交互如下：

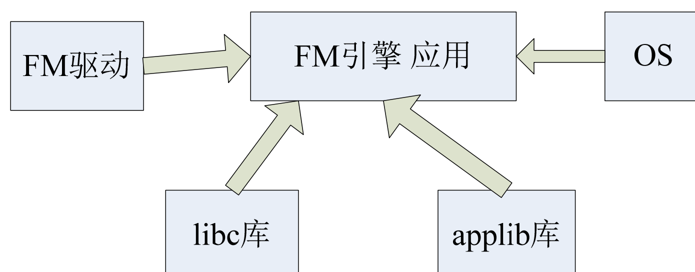
目前的方案，由于设置频率，搜台等常规的 FM 操作都在 RadioUI 中进行，所以和 FM 引擎交互最为频繁的是 RadioUI 应用。radioUI 中通过给 FM 引擎进程发送消息，实现模组的初始化，频率设置，搜台，获取状态等。RadioUI 中 app_radio_control.c 文件内的函数，即负责和 FM 引擎的交互。

FM 引擎与 FM 驱动的交互如下：

FM 引擎向下调用 FM 驱动提供的各类接口，实现硬件操作，它和 FM 驱动的交互就是在各个消息处理中，调用所需的 FM 驱动接口，实现对硬件的操作。

版 权 所 有 侵 权 必 究

14.5 应用依赖库及其接口说明



14.6 如何增加一条引擎消息

增加引擎消息可参照如下步骤：

1. 在 App_msg.h 的 msg_apps_type_e 中，添加一条 FM 引擎消息，注意 FM 引擎消息占位符的位置。
2. 在 fmengine_message_deal 函数中，添加对消息的处理。

在需要发送该消息的地方，比如 RadioUI 应用中，添加消息的发送（同步或异步）。

15 FM 驱动程序

15.1 需求概述及设计原则

FM 驱动实现需求如下：

1. 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。
2. 实现部分代码对各款 FM 模组的通用性
3. 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）。

针对如上的设计需求，FM 驱动设计时，主要的设计原则有：

1. 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, (fm_op_func) sFM_Standby, (fm_op_func) sFM_SetFreq,
            (fm_op_func) sFM_GetStatus, (fm_op_func) sFM_Mute, (fm_op_func)
sFM_Search,
            (fm_op_func) sFM_HardSeek, (fm_op_func)sFM_SetBand,
            (fm_op_func)sFM_SetThrod, (fm_op_func)sFM_BreakSeek,
            (fm_op_func)sFM_GetHardSeekflag, (fm_op_func)sFM_GetBand,
            (fm_op_func)sFM_GetFreq, (fm_op_func)sFM_GetIntsity,
            (fm_op_func)sFM_GetAnten, (fm_op_func)sFM_GetStereo,
};
```

2. 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。
3. 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。
4. 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，这就使得有一种情况需要考虑：**当进行 FM 录音后，直接回到 FM，由于录音时 codec 代码已将 FM 驱动常驻代码覆盖，这时就需要重新安装 fm 驱动。**
   因为 FM 引擎后台需要的数据空间不多，所以可从后台数据空间中分配一块作为 FM 驱动的数据空间。

```
*------------------------------code-------------------------------------*
*---rcode         :0xbfc29000-0xbfc29fff   length:0x1000                 *
*I2C 代码使用固化时,占用如下 rom_I2C 空间，否则，也可放在上述 rcode 空间
*
*---rom_I2C       :0xbfc08000-0xbfc087ff   length:0x800                  *
*---bankA         :(0x18**0000+0x24c00)-(0x18**0000+0x24fff)   length:0x400 *
*---bankB         :(0x28**0000+0x25000)-(0x28**0000+0x257ff)   length:0x800 *
*------------------------------data-------------------------------------*
*--rdata: 0x9fc1dd00-0x9fc1dfff length:0x300                           *
*************************************************************************
***/
```

15.2 总体设计

15.2.1 FM 驱动在系统结构中的位置

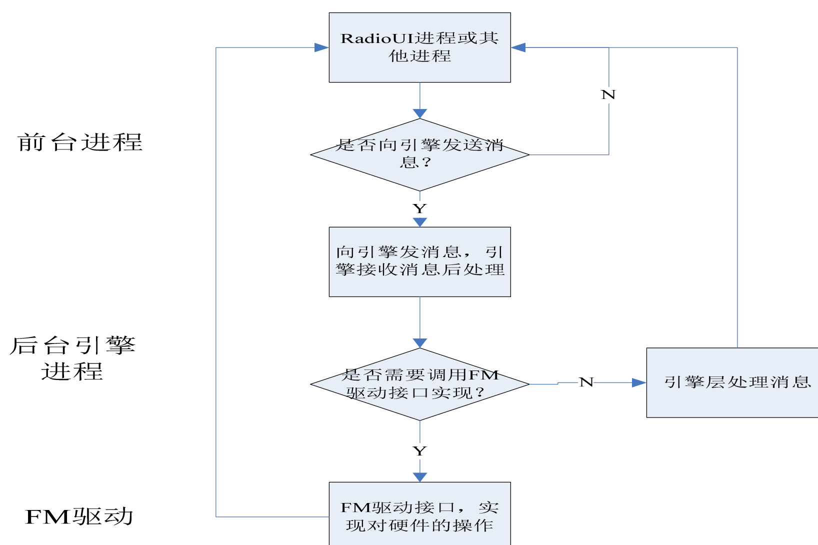


图 30-1 FM 驱动在系统结构中位置图

FM 驱动是 FM 硬件操作层，需要时，调用 sys_drv_install(DRV_GROUP_FM, 0, "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 RadioUI 进程或者其他进程需要操作 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： RadioUI 播放场景下，需要步进调节频率，此时 RadioUI 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。

15.2.2 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：

版 权 所 有 侵 权 必 究

模块名称	功能简述	对应文件
驱动初始化模块	主要是驱动的装载初始化以及卸载函数，为方便各模组公用，这两个函数无实际内容	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_i2cdrv1.c
驱动 BANK 模块	FM 驱动其他的接口实现，放在 BANK 空间。	bank_b_fm_i2cdrv2.c bank_b_fm_i2cdrv3.c

15.3 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：

```
/* I2C 所用模拟 GPIO 信息结构*/
gpio_init_cfg_t gpio_i2c[2] =
{
        //SCL==GPIO_A15
       { GPIO_AINEN, GPIO_AOUTEN, GPIO_ADAT, GPIO_SCL_BIT },
       //SDA==GPIO_A16
       { GPIO_AINEN, GPIO_AOUTEN, GPIO_ADAT, GPIO_SDA_BIT }
};
/* I2C 操作延时*/
uint8 delay_i2c = PARAM_DELAY_I2C;
```

I2C 固化代码向上提供若干接口，对所有模组可以通用：

版 权 所 有 侵 权 必 究

1. uint8 I2C_Trans_Bytes(uint8 *buf, uint8 length, gpio_init_cfg_t* gpio, uint8* delay)
实现将指定 buffer 中长度为 length 字节的数据通过 I2C 送出。至于长度为 length 字节中是否包含设备地址，寄存器起始地址等，可根据不同的模组修改。
2. uint8 I2C_Recev_Bytes(uint8 *buf, uint8 address, uint8 length, gpio_init_cfg_t* gpio, uint8* delay)
发送设备地址，并从设备端将长度为 length 的字节数据读回到指定 buffer 中。
3. void IIC_Init(gpio_init_cfg_t* gpio, uint8* delay)
I2C 的初始化，是 GPIO 处于初始状态
4. void IIC_Start(gpio_init_cfg_t* gpio, uint8* delay)
I2C 启动条件
5. void IIC_WriteByte(uint8 dat, gpio_init_cfg_t* gpio, uint8* delay)
通过 I2C 发送指定的一个字节数据，可以是设备地址，寄存器地址或寄存器值等。
6. uint8 IIC_GetAck(gpio_init_cfg_t* gpio, uint8* delay)
获取从设备端返回的响应信号。
7. uint8 IIC_ReadByte(gpio_init_cfg_t* gpio, uint8* delay)
从模组读回一个字节的数据
8. void IIC_SendAck(uint8 ack, gpio_init_cfg_t* gpio, uint8* delay)
向模组发送响应信号
9. void IIC_Stop(gpio_init_cfg_t* gpio, uint8* delay)
I2C 停止条件。

15.4 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。

目前，FM 驱动的应用接口，采用接口表形式列明如下：

```
/*FM 驱动对外接口函数*/
fm_driver_operations fm_drv_op =
{ (fm_op_func) sFM_Init, (fm_op_func) sFM_Standby, (fm_op_func) sFM_SetFreq,
          (fm_op_func)   sFM_GetStatus,   (fm_op_func)   sFM_Mute,   (fm_op_func)
sFM_Search,
          (fm_op_func) sFM_HardSeek, (fm_op_func)sFM_SetBand,
          (fm_op_func)sFM_SetThrod, (fm_op_func)sFM_BreakSeek,
          (fm_op_func)sFM_GetHardSeekflag, (fm_op_func)sFM_GetBand,
          (fm_op_func)sFM_GetFreq, (fm_op_func)sFM_GetIntsity,
          (fm_op_func)sFM_GetAnten, (fm_op_func)sFM_GetStereo,
};
```

15.5 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 IIC 总线 GPIO 配置宏*/
#define   IIC_SCL_BIT   13
#define   IIC_SDA_BIT   12
#define   GPIO_SCL_BIT   (0x00000001<<IIC_SCL_BIT)
#define   GPIO_SDA_BIT   (0x00000001<<IIC_SDA_BIT)
//模拟 I2C 延时参数，与主频相关，需传参调整
#define   PARAM_DELAY_I2C   20
```

FM 驱动向上提供的接口介绍：

1. int sFM_Init(uint8 band, uint8 level, uint16 freq)
   FM 模组初始化
2. int sFM_Standby(void* null1, void* null2, void* null3)
   FM 模组进入 Standby
3. int sFM_SetFreq(uint16 freq, void* null2, void* null3)
   设置一个频率开始播放
4. int sFM_GetStatus(void * pstruct_buf, uint8 mode, void* null3)
   获取当前模组的状态信息
5. int sFM_Mute(FM_MUTE_e mode, void* null2, void* null3)
   模组静音或者解除静音模式
6. int sFM_Search(uint16 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
7. int sFM_HardSeek(uint16 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
8. int sFM_SetBand(radio_band_e band, void* null2, void* null3)

版 权 所 有 侵 权 必 究

设置电台波段
9. int sFM_SetThrod(uint8 level, void* null2, void* null3)
设置搜台门限值
10. int sFM_BreakSeek(void* null1, void* null2, void* null3)
退出硬件搜台操作。
11. int sFM_GetHardSeekflag(void* flag, void* null2, void* null3)
获取硬件搜台是否完成的标记
12. int sFM_GetBand(void* band, void* null2, void* null3)
获取当前波段信息
13. int sFM_GetFreq(void* freq, void* null2, void* null3)
获取当前频率信息
14. int sFM_GetIntsity(void* intensity, void* null2, void* null3)
获取当前电台的信号强度
15. int sFM_GetAnten(void* antenna, void* null2, void* null3)
获取天线（耳机）连接状态
16. int sFM_GetStereo(void* stereo, void* null2, void* null3)
获取当前电台的立体声信息。

15.6 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。

15.7 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，因此给 FM 驱动分配的空间如下：

版权所有 侵权必究

```
*------------------------------code-------------------------------------*
*---rcode            :0xbfc29000-0xbfc29fff   length:0x1000                   *
*I2C 代码使用固化时,占用如下 rom_I2C 空间，否则，也可放在上述 rcode 空间          *
*---rom_I2C          :0xbfc08000-0xbfc087ff   length:0x800                     *
*---bankA            :(0x18**0000+0x24c00)-(0x18**0000+0x24fff)   length:0x400 *
*---bankB            :(0x28**0000+0x25000)-(0x28**0000+0x257ff)   length:0x800 *
*------------------------------data-------------------------------------*
*--rdata: 0x9fc1dd00-0x9fc1dfff length:0x300                                  *
**********************************************************************/
```

说明如下：

a) FM 驱动常驻代码空间和 codec 空间复用，需要考虑 FM 录音。 由于 FM 录音时，codec 代码会将驱动常驻代码覆盖，所以从 FM 录音直接返回 FM 时，需考虑重装 FM 驱动。
b) FM 驱动常驻的数据空间，可从后台数据空间中划分出 0.75K，供 FM 驱动使用。因为 FM 后台引擎数据量较少，不需要 1.5K 的后台数据空间。

15.8 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 和电源配置。

步骤 1：根据硬件设计，修改 GPIO 和电源的配置宏。配置宏前面已有描述，不再赘述。

标案的电源配置宏暂未使用，如有需要，可修改

```
/* FM 电源 GPIO 定义，与硬件相关，待定*/
#define REGFM_POWER_CTRL        GPIO_ADAT
#define     OpenFMPower()     act_writel((act_readl(REGFM_POWER_CTRL)|0x00000040), REGFM_POWER_CTRL)
#define     CloseFMPower()     act_writel((act_readl(REGFM_POWER_CTRL)&0xffffffbf), REGFM_POWER_CTRL)
```

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。

15.9 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。前者在 I2C.H 中，通过

```
#define    IIC_SCL_BIT            13
#define    IIC_SDA_BIT            12
```

以及 Rcode_fm_op_entry.c 中

```
gpio_init_cfg_t gpio_i2c[2] =
{
    { GPIO_AINEN, GPIO_AOUTEN, GPIO_ADAT, GPIO_SCL_BIT },
    { GPIO_AINEN, GPIO_AOUTEN, GPIO_ADAT, 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_I2C     20
```

即可修改。

15.10 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：

（8） 在 Rcode_fm_op_entry.c 的 fm_drv_op 中，增加接口函数

（9） 在 fm_interface.h 的 fm_driver_operations 中，相应位置添加新的接口成员。

（10） 在 fm_interface.h 的 fm_cmd_e 中，相应位置添加新的接口命令。

（11） 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。

16 ap_mainmenu 应用

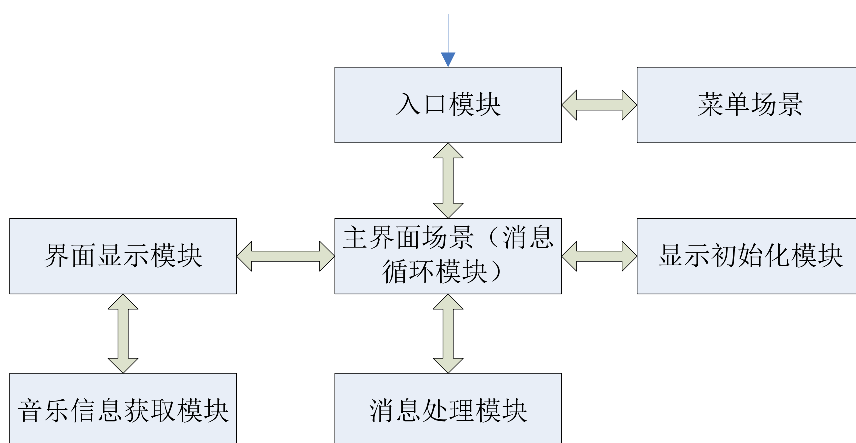
16.1 需求概述

Mainmenu 应用将方案支持的应用以制定风格的菜单方式显示在屏幕上，实现应用的功能导航功能，同时支持通过 FW Modify 工具修改方案支持的应用个数以及各应用在主界面

中的排列顺序。

16.2 总体架构设计

16.2.1 总体架构图



16.2.2 功能模块划分

模块名称	功能简述	对应文件
入口模块	负责应用初始化和退出，包括资源文件，菜单配置文件的加载和卸载，以及应用的场景管理	mainmenu_main.c
消息循环模块	查询 UI 消息和应用的私有消息，并查询是否需要更新界面，如果停留在正在播放的界面，还需要查询后台歌曲是否有切换，如果切换需要更新歌曲标题的显示	mainmenu_msg_loop.c
消息处理模块	处理 UI 消息和应用的私有消息	mainmenu_msg_handle.c
界面显示模块	根据当前激活的应用序号进行界面显示	mainmenu_paint.c
音乐信息获取模块	获取后台音乐的标题	mainmenu_get_info.c
显示初始化模块	根据配置项初始化应用图标的排列顺序	mainmenu_display_init.c
菜单模块	调用菜单控件以及菜单项的执行函数	mainmenu_option_menu.c
菜单项配置模块	菜单配置的数据	ap_cfg_mainmenu.c

版 权 所 有 侵 权 必 究

16.3 应用的生命周期

16.3.1 应用的启动

任何应用需要返回主界面，发送创建应用的消息给 manager，由 manager 来创建。

应用的初始化在_app_init 函数中完成，主要是 applib 的初始化，软件定时器初始化，系统计时器初始化，消息初始化，打开资源文件和菜单配置文件。

16.3.2 应用的退出

在主界面中选中了某个应用，按键进入时，mainmenu 应用主动退出；

在低电，关机和用户通过 usb 连接电脑后确认进入 u 盘时，manager 会发 MSG_APP_QUIT 的消息，这时应用也会退出。

退出应用需要做的事情与进入应用时相反，即退出系统计时器，保存 VRAM 信息，关闭菜单配置和资源文件，applib 退出，这些动作都在_app_exit 函数中完成。

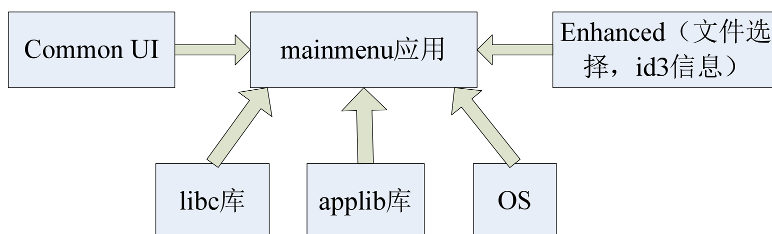
16.4 与其它应用的同步和交互

如果有后台音乐播放，主界面在切换到正在播放时，需要获取文件路径的同步消息给后台引擎，同时通过文件选择器获取到当前播放音乐的标题信息来显示，如果界面一直停留在正在播放，需要查询后台音乐是否有切换，一旦切换后需要更新音乐标题的显示。

16.5 应用依赖库及其接口

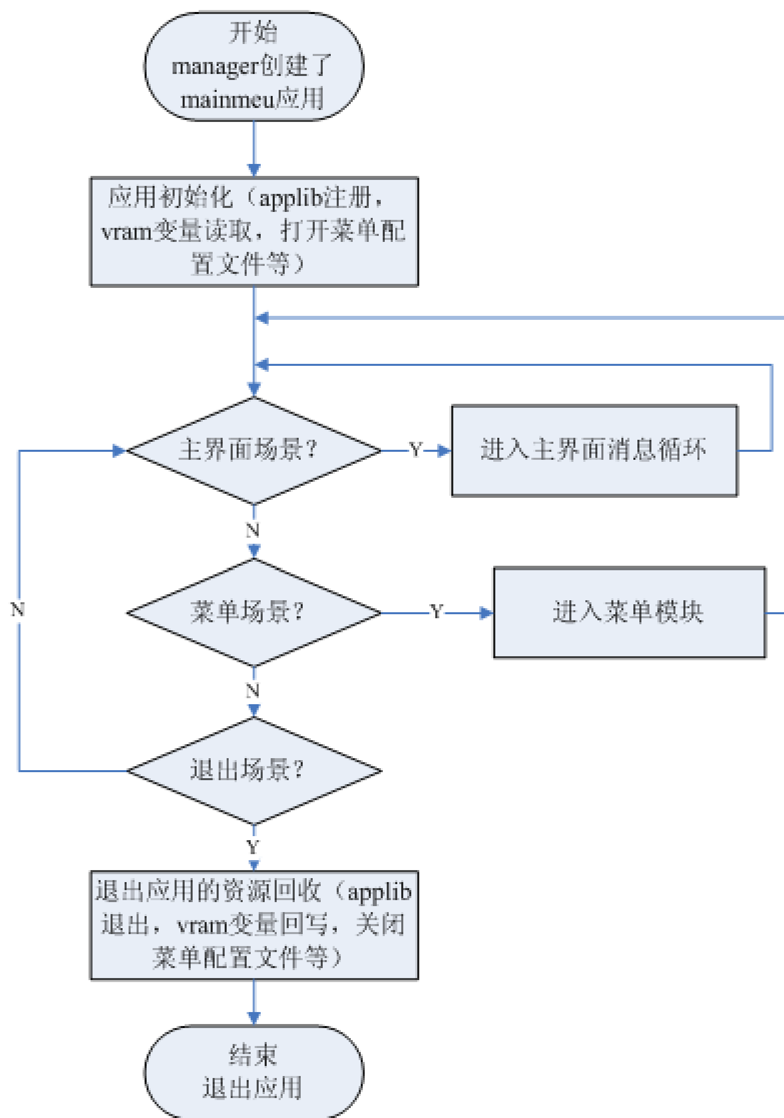
系统和 libc 的接口 api.a
应用运行时库 ctor.o
Applib 的全部函数
Common UI 的菜单，headbar，动画和其他公共提示模块
Enhanced 库中的文件选择和 ID3 获取模块

版 权 所 有 侵 权 必 究

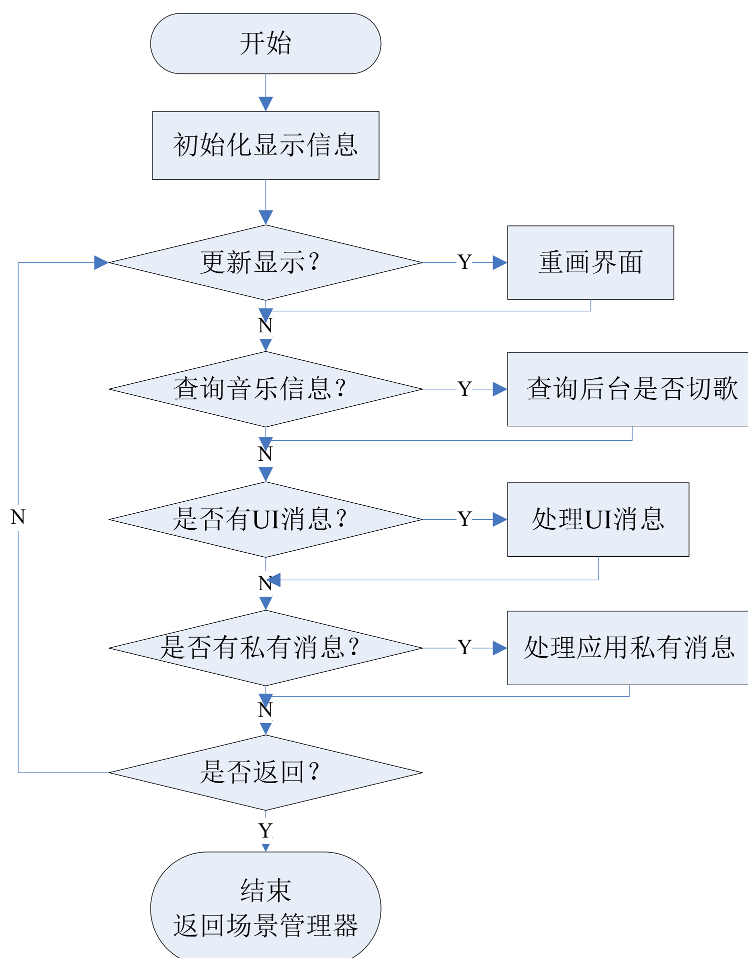


16.6 应用的业务流程

16.6.1 应用的总流程和场景调度流程



16.6.2 桌面场景流程图（mainmenu_desktop）



16.6.3 弹出菜单场景流程图（option_menulist）

调用 Common ui 的公共函数，在此不必描述。

16.7 如何增加一个应用的入口

步骤 1：在 case_type.h 中增加新应用的 ID 宏定义，

```
#define APP_ID_MUSIC            0x00
#define APP_ID_AUDIBLE          0x01
```

```
…
#define APP_ID_NEWAP              0x??
```

步骤 2：在 manager ap 中 manager_get_name.c 文件中 app_name_ram 数组以新 apd 的 ID 为下标，增加新 ap 的名字，注意数组的大小需要同时增加来匹配

```
const char app_name_ram[25][12] =
{
    "music.ap", "audible.ap", "video.ap", "picture.ap", "ebook.ap", "browser.ap", "voice.ap",
    "record.ap", "radio.ap", "tools.ap", "setting.ap", "user1.ap", "user2.ap", "user3.ap",
     "manager.ap", "mainmenu.ap", "playlist.ap", "config.ap", "udisk.ap", "mengine.ap",
     "fmengine.ap", "alarm.ap", "mtp.ap", "mtpsync.ap", "fwupdate.ap"，”newap.ap”
};
```

步骤 3：如果应用是前台 ap，修改 config.txt 文件中 mainmenu ap 的配置项，在需要的应用顺序下增加新 ap 的 ID，如 MAINMENU_AP_ID_ARRAY = 0[0, 2, 7, 3, 8, ??, 4, 5, 9, 10]; 然后执行 case\tools\Gen_config\ genconfig.bat，生成新的 config.bin

步骤 4：在 case\fwpkg\fwimage.cfg 增加新的 ap，将其打包进固件，这样修改后在主界面下就能看到新的前台应用了。

16.8 如何删除一个应用的入口

修改 config.txt 文件中 mainmenu ap 的配置项，将需要删除的应用对应的 ID 从 MAINMENU_AP_ID_ARRAY 配置项中删除，然后执行 case\tools\Gen_config\ genconfig.bat，生成新的 config.bin 即可。

17 ap_browser 应用

17.1 需求概述

browser 用于实现对各种物理存储介质(flash 盘，卡盘，U 盘)目录项(目录和文件)的显示，选择支持文件进行播放，并可删除选定的任意文件。

17.2 总体架构设计

browser 应用较为简单，需要完成如下几个功能:

（1） 浏览功能：对非隐藏文件和目录的显示，具备浏览记忆功能；
（2） 对任意文件的删除功能
（3） 选定文件的播放，从播放界面返回后仍进入 browser 文件浏览界面

因此 browser 共划分了三个场景，文件列表场景，option 菜单场景和菜单场景。文件列表场景实现文件的浏览，option 菜单场景实现菜单操作，如跳转到 music 播放界面，实现文件的删除操作，而菜单场景只有在插卡情况下才会出现。

17.2.1 总体架构图

browser 的总体架构比较简单，从 browser 选择文件播放就会退出 browser 应用，当结束文件播放后返回时，又会重新进入 browser 应用。

17.2.2 功能模块划分

browser 共划分了如下几个模块:

模块名称	功能简述
初始化模块	负责应用初始化和退出，包括资源文件，菜单配置文件的加载和卸载，以及应用的场景管理
场景调度模块	对场景转移关系进行调度和处理
列表场景	调用文件列表模块显示 browser 文件列表
option 菜单场景	调用菜单列表模块显示 browser option 菜单项

菜单场景	显示主盘目录和卡目录
退出模块	实现应用的退出处理

17.3 与其它应用的同步和交互

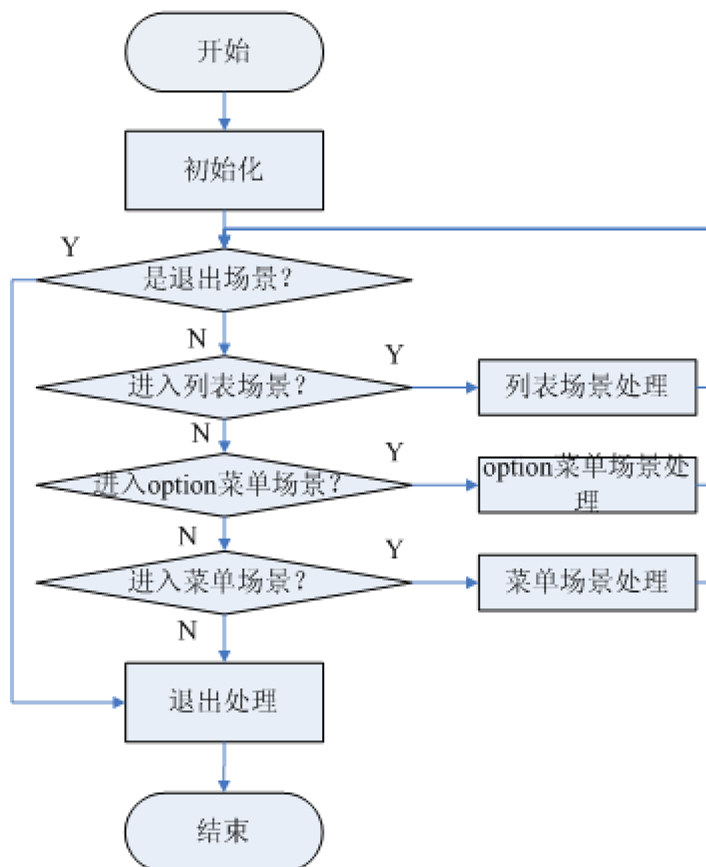
从主界面进入 browser 应用，如果后台有音乐或 FM 播放，根据播放状态(播放/暂停)，在 optionmenu 菜单有正在播放/上一次播放菜单项，选择相应菜单项可转入相应的应用。选择文件播放后，browser 会主动发消息给 manager 进程，要求创建相应的前台应用并退出 browser。

17.4 应用依赖库及其接口

系统和 libc 的接口
应用运行时库
applib 的全部函数
common ui 的菜单，文件列表，headbar 和其它公共模块
中间件文件选择器模块

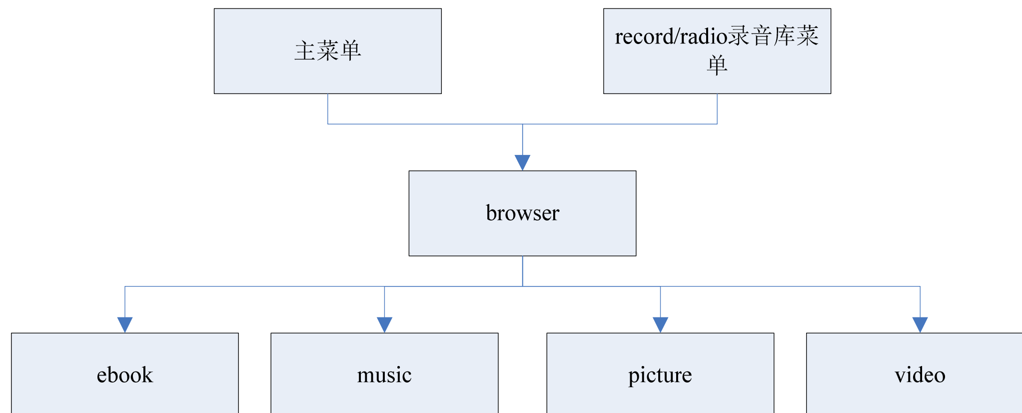
版 权 所 有 侵 权 必 究

17.5 应用的总流程和场景调度流程



17.6 browser 是如何保存进入和退出 ap 的

browser 提供文件浏览，选择的功能，当前 SDK，browser ap 进入和退出路径如下图所示:



从主界面和从录音库进入 browser 的区别在于:

- ❖ 主界面进入 browser 从根目录开始浏览，录音库则到 record 子目录浏览
- ❖ 主界面进入 browser 无嵌套菜单项, 录音哭进入 record 包含一个嵌套菜单项
- ❖ 主界面进入选择 music 播放时，music 的设置菜单和从 record 目录进入选择 music 播放时，music 的设置菜单不同

因此 browser 需要记忆从哪个 ap 进入，并区分是从 ebook/music/picture/video 返回还是从 main/record 进入。browser 的 VRAM 变量 g_browser_var 中有一个成员 prev_enter_mode 用于记录 browser 进入是从 main 进入还是从录音库进入。而从其他 ap 返回时，可以根据传入 browser 的参数进行判断。

17.7 browser 如何实现删除整个目录操作

browser 支持删除单个文件，也支持删除整个目录操作。根据当前列表目录总数和激活项序号可以判断删除模式是文件还是目录。对于目录删除，从当前目录的最顶层开始，采用了深度遍历方式遍历每一个目录的文件。其伪代码如下所示:

```
for eachdir
   search file and delete
   if(dir_total != 0)
   {
       enter sub dir //recurse
   }
   else
   {
       back to parent and delete //delete dir
   }
```

在遍历的过程中，如果发现目录层级超过 9 级，则停止删除删除操作并返回出错状态。应用需要据此做恢复目录项的操作(因目录层级已修改)。遍历和删除操作有 ui_delete()控件实现。

版 权 所 有 侵 权 必 究

browser 应用需要在删除前初始化好当前的目录环境。

17.8 如何进入指定目录浏览文件

进入指定目录浏览文件和进入根目录浏览文件的差异在于前者需要先进入一个默认的目录，因此需要实现进入指定目录的方式。enhanced 提供两种接口函数进入指定目录，第一种是使用 fsel_browser_get_items()获取当前目录的目录项个数，然后逐个进行目录名的匹配，这是一种效率较低的方法，但做到了 bs 接口的统一；另外一种是使用 fsel_enter_dir()传入要进入的目录名，然后文件系统在底层 cd 到要进入的目录。该方法是一种效率较高的方法。使用 fsel_browser_get_items 参考代码如下:

```
//进入根目录
fsel_browser_enter_dir(ROOT_DIR, 0, brow);
...
//获取根目录目录项，查找是否有匹配的目录名
for (i = 1; i <= brow->dir_total; i++)
{
    dec_para.top = i;
    fsel_browser_get_items(&dec_para, &file_brow);
    //长名比较
    if (libc_memcmp(dentry_name, dir_name, file_brow.name_len) == 0)
    {
        break;
    }
    //短名比较
    if (libc_memcmp(dentry_name, test_dentry, name_len) == 0)
    {
        break;
    }
}

if (i > brow->dir_total)
{
    //找不到指定目录，退出
    gui_dialog_msg(DIALOG_MSG, DIALOG_INFOR_WAIT, S_FILE_NO_FILE);
    return FALSE;
}

//进入所要搜索的目录
result = fsel_browser_enter_dir(SON_DIR, i, brow);
```

版 权 所 有 侵 权 必 究

使用 fsel_enter_dir()参考代码如下:

```
fsel_enter_dir(CD_ROOT, NULL);

//先找长名目录
if (fsel_enter_dir(CD_SUB, (char *)dir_name) == TRUE)
{
    fsel_browser_enter_dir(CUR_DIR, 0, &brow);
    if ((brow.dir_total + brow.file_total) == 0)
    {
        result = FALSE;
    }
}
else
{
    //找不到长名目录，寻找短文件名目录
    if (fsel_enter_dir(CD_SUB, (char *)dir_short_name) == TRUE)
    {
        fsel_browser_enter_dir(CUR_DIR, 0, &brow);
        if ((brow.dir_total + brow.file_total) == 0)
        {
            result = FALSE;
        }
    }
    else
    {
        result = FALSE;
    }
}
```

17.9 如何备份和恢复目录项

如果备份和恢复目录项不需要改变 enhanced 的环境，例如文件操作时对当前目录项的备份，可以直接使用 vfs_file_dir_offset(),在操作之前获取当前的目录偏移和目录层次，然后在操作完成之后使用 vfs_file_dir_offset(),恢复当前的目录项。如果备份和恢复同时需要恢复 enhanced 的工作环境，则需要使用 get_location 和 set_location 备份和恢复当前的目录项。例如在删除目录时，如果目录层次超过 8 层，则无法删除当前目录。但遍历 8 级目录会改变当前的 enhanced 层级关系，因此返回时需要使用 fsel_browser_set_location()恢复之前的目录项。参考代码如下所示:

```
//使用 vfs_file_dir_offset
```

版 权 所 有 侵 权 必 究

```
//save current path
vfs_file_dir_offset(g_music_mount_id, &sys_layer_buf, cur_offset_save, 0);
...
//restore path
vfs_file_dir_offset(g_music_mount_id, &sys_layer_buf, cur_offset_save, 1);
```

```
//使用 location 接口
fsel_browser_get_location(&(g_browser_var.path.file_path.dirlocation), g_browser_var.path.file_source);
...
fsel_browser_set_location(&(g_browser_var.path.file_path.dirlocation), g_browser_var.path.file_source);
```

18 ap_udisk 应用

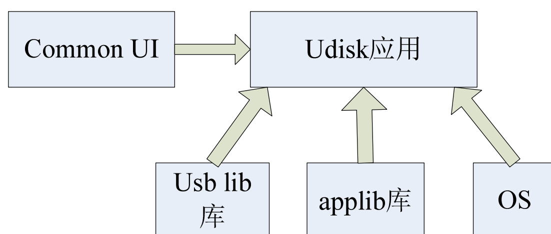
18.1 需求概述

Udisk 应用主要实现的功能包括：(1) 实现 U 盘功能（2）界面显示（3）按键控制

18.2 总体架构设计

Udisk 应用需要系统和公共函数库的支持：

18.2.1 总体架构图



18.2.2 功能模块划分

由本应用的主要有由 4 个模块组成，每个模块都负责单独的功能，分别为模块的调度、系统消息处理，应用初始化，应用的退出处理和界面显示。这 4 个模块配合其他辅助模块，实现全部 U 盘功能。

模块名称	功能简述	对应函数	对应文件
主模块	主导程序运行，决定其它模块调度。	main	main_udisk.c
初始化模块	应用初始化。	udisk_init_all	udisk_enter2exit.c
退出模块	应用退出。	udisk_exit_all	usbdisk_enter2exit.c
显示模块	负责显示。	display	usbdisk_task.c

18.3 与其它应用的同步和交互

阐述如何从各个应用进入 ap_udisk 应用的过程以及从 ap_udisk 退出到应用的过程。

18.3.1 应用的启动

插入 USB 线时，在弹出的 USB 连接对话框中选择数据传输模式，那么 COMMON 就会发送 MSG_USB_TRANS 消息给 ap_manager，在杀死当前前台应用和后台应用后，只由 ap_manager 应用启动。

18.3.2 应用的退出

在 USB 空闲状态下按下 return 键或 play 键，又或者拔掉 USB 电缆，又或者选择电脑的弹出操作，都可以退出 Udisk 应用。

版 权 所 有 侵 权 必 究

18.4 应用依赖库及其接口

Udisk 应用只由 manager 创建，在完成开机需要的动作后根据需求发送创建应用的消息给 manager。

由于退出 udisk 应用时需要知道上次进入 udisk 前台运行的应用。根据这些信息创建对应的应用。而 udisk 应用都是 manager 创建的，所以只有 manager 知道。进入 udisk 时，这些信息只能在 manager 启动 udisk 的时候通过参数传入。由于系统限制，应用只有第一个 int argc 的参数有效，所以对传入参数作了如下定义：低 16 位表示进入 udisk 前正在运行的前台应用 ID，高 16 位表示后台的状态。

当退出 udisk 时，如果 u 盘有写操作，则进入 playlist ap 创建播放列表。如果按 play 键退出，则进入 mtp；其他情况退出，则运行进入 udisk 之前的前台 ap。

系统和 libc 的接口 api.a

应用运行时库 ctor.o

Applib 的全部函数

18.5 应用的业务流程

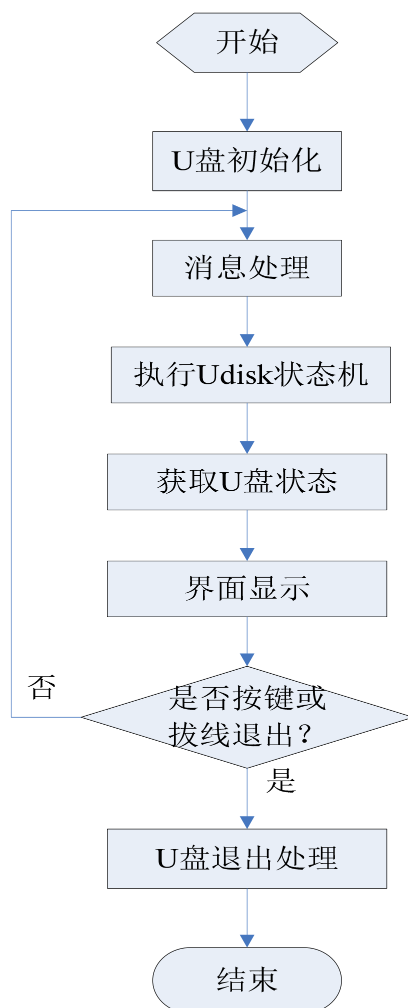
18.6 主模块设计说明

18.6.1 模块描述

主模块负责 AP 的初始化、module 状态分析处理、系统消息和 GUI 消息处理、退出资源释放等。其常态运行于不断查询处理循环中，当收到 module 退出状态指示后才打破循环退出 AP，USB 中断在这个过程随时发生。

18.6.2 模块功能

主导程序运行，决定其他模块调度。



18.6.3 场景调度流程

18.6.4 消息处理流程图

版 权 所 有 侵 权 必 究

18.6.5 充电流程图

18.7 初始化模块设计说明

18.7.1 模块描述

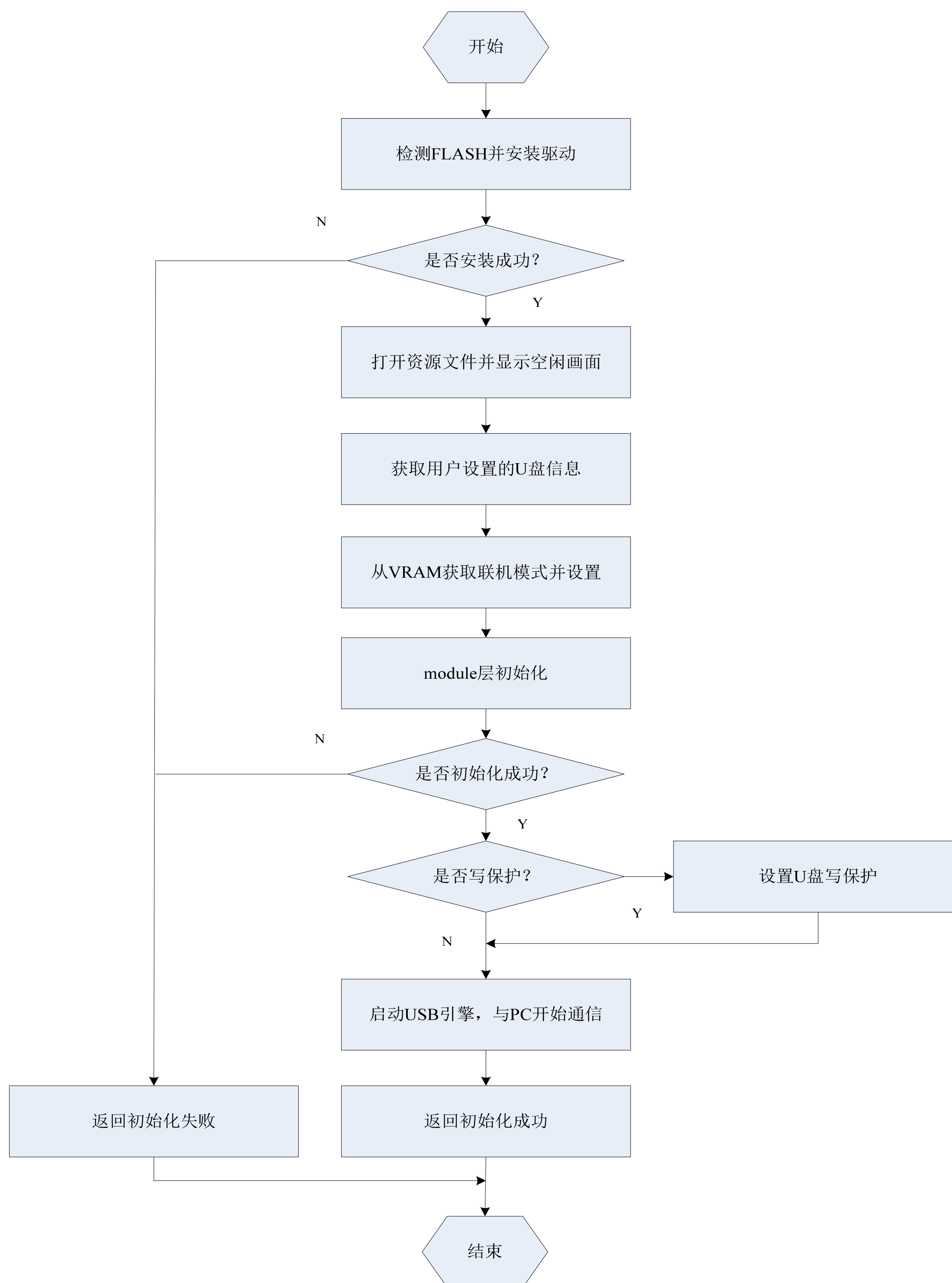
AP 初始化模块，包括 AP 层初始化和模块层初始化，最后启动 USB 引擎。

18.7.2 模块功能

初始化 udisk AP，启动 USB 引擎，完成 U 盘枚举过程。

版 权 所 有 侵 权 必 究

18.7.3 流程逻辑



18.8 退出模块设计说明

18.8.1 模块描述

Udisk ap 退出处理模块。

18.8.2 模块功能

Udisk ap 退出流程处理，根据退出的方式，选择需要进入的前台应用，创建前台 AP，并退出 udisk 应用。

18.8.3 流程逻辑

当 udisk 收到需要退出的消息时，进行 udisk 的退出处理。当退出 udisk 时，如果 u 盘有写操作，则进入 playlist ap 创建播放列表。如果按 play 键退出，则进入 mtp；其他情况退出，则运行进入 udisk 之前的前台 ap。

18.9 显示模块设计说明

18.9.1 模块描述

根据 U 盘当前状态显示不同界面。

18.9.2 模块功能

显示 U 盘各状态下的界面。

18.9.3 流程逻辑

用全局变量记录当前 U 盘的状态，如果 U 盘状态发生变化，则根据当前 U 盘状态绘制相应的界面；如果 U 盘界面不发生变化，则不需要刷新界面。

版 权 所 有 侵 权 必 究

18.10 如何修改 USB 设备的属性

修改 fwpkg 目录下的 config.txt 中和 usb 有关的配置字段，具体在字段后面有注释。

18.11 如何实现只充电而不上报 USB 盘符

现在的做法，只要进入 udisk 应用且连上 USB host，udisk 应用就会正常工作且一定会上报 USB 盘符，所以要实现该功能，只要在只充电的时候不进 udisk 应用即可。

18.12 如何实现没插卡的时候，不上报卡盘符

现在的做法，没有实现不插卡不上报卡盘符，但在 win7 以后的系统，不插卡，虽然上报了卡盘符，PC 端也不会显示。

19 ap_playlist 应用

19.1 需求概述

1、PLAYLIST 应用是基于 US212A 的 SDK 上，负责创建各个播放应用的播放列表 LIB 文件。
主要需求如下：

1）查找磁盘上所有的音乐格式文件和 AUDIBLE 文件，获取文件的 ID3 信息，经排序 ID3 信息后生成播放列表给 MUSIC 应用。支持 4000 首音乐和 1000 首 AUDIBLE。

2）另外还会分别查找磁盘上的所有的 VIDEO,PICTURE，EBOOK 文件，根据其文件名排序，创建其播放列表供 VIDEO,PICTURE，EBOOK 应用使用。每个表支持 1000 个文件。

3）提高 PLAYLIST 运行性能，要求 2000 首歌<60s。

2、PLAYLIST 分音乐歌曲、语音书籍、视频、图片和电子书共 5 个 LIB 列表。

1）音乐歌曲列表 分别按照艺术家 Artist、专辑 Album、流派 Genre 和歌名 Allsong 共 4

个类别来分类和排序；具体排序规格如下：

A、allsong　一级：title

B、album　二级：album->title

C、artist　三级：artist->album->title

D、genre　四级：genre-> artist->album->title 或 genre-> album->title

2）语音书籍列表 只按书籍名 book 和作者 author 分类和排序，具体排序规格如下。

A、book　一级：book

B、author 二级：author-> book

3）video、picture 和 ebook 的排序只排序文件名(优先长名，后短名)。

A、video 一级：file_name

B、picture 一级：file_name

C、ebook 一级：file_name

4）列表 LIB 中包含的信息。

A、列表的总文件个数。

B、每个文件的基本信息 plist_f_info_t（具体定义见下文）。

C、 所有文件的排序顺序。

B、每级包含的信息： 当前级的子级的个数和文件的总个数，文件排序顺序。

5）各排序的关键要素

排序要求是以内码方式（字符比较，从小到大）：中文简体以拼音排序，数字和英文（统一转换为大写再排序）从小到大排序；排出来的结果应该是 先中文，然后数字，最后是英文。若比较的两者相同则不改变两者的先后顺序。

A、title：排序 16 个字节 (若 title 在 album 下级排序则内容为 track_num+title=16 字节)

B、album：排序 8 个字节

C、artist：排序 8 个字节

D、genre：排序 8 个字节

E、book：排序 8 个字节

F、author：排序 8 个字节

G、file_name：排序 16 个字节

5、PLAYLIST 支持文件格式

1）音乐歌曲格式（带 ID3 信息）：MP3、WMA、ASF、OGG、FLAC、APE、AAC

2）audible 格式（带 ID3 信息）：AAX、AA

3）视频格式：AMV、AVI。

4）图片格式：JPEG、BMP、GIF 格式

版 权 所 有　侵 权 必 究

5）电子书格式：TXT

6、PLAYLIST 查找文件的方式

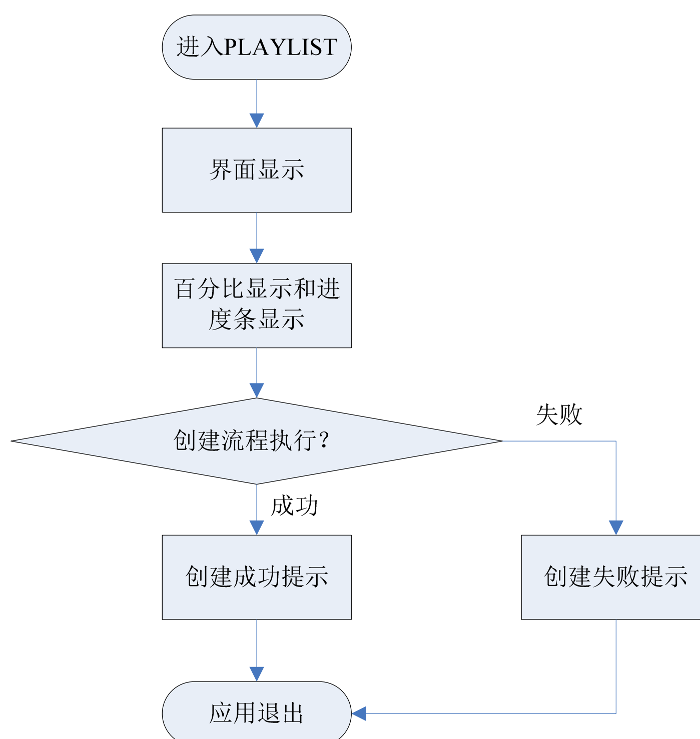
深度优先，先找文件，再找目录。最多支持 8 层目录

7、PLAYLIST 支持删除文件，并能更新到包含删除文件的列表。

19.2 总体架构设计

19.2.1 总体架构图

描述 PLAYLIST 应用的总体流程结构图：



19.2.2 功能模块划分

模块名称	功能简述	对应文件
系统模块	负责应用初始化，打开资源文件；整个应用流程处理 ；以及应用退出处理	plist_main.c plist_sys_deal.c
界面显示模块	负责 UI 界面的显示	plist_ui_deal.c
文件遍历模块	负责查找磁盘上对应格式的文件	plist_fsel.c
文件操作模块	负责创建文件，写入和保存	plist_operate_list.c
算法排序模块	负责排序算法的实现	plist_sort_method.c plist_sort_al.c
音乐列表建表模块	负责扫描音乐文件，按需求排序建表	plist_scan_music.c plist_sort_music.c
audible 列表建表模块	负责扫描 audible 文件，按需求排序建表	plist_scan_audible.c plist_sort_audible.c
video，picture，ebook 列表建表模块	负责扫描文件，按需求排序建表	plist_other_list.c
创建收藏夹和书签模块	处理创建对应磁盘的收藏夹和书签文件	plist_creat_userpl.c plist_creat_userpl_sub.c plist_creat_bmk.c
创建 M3U 播放库	根据*.M3U 列表生产用于播放的列表库	plist_m3u_deal.c plist_m3u_deal_sub.c plist_m3u_deal_sub1.c

19.3 PLAYLIST 应用的生命周期

19.3.1 应用的启动

系统调用场景：
- ✧ DEVICE 应用 USB 退出。
- ✧ 系统第一次上电。
- ✧ 各应用主动调用创建。

系统第一次上电调用代码示例：

```
msg_apps_t config_msg;
```

```
//消息类型，为创建新应用
config_msg.type = MSG_CREAT_APP;

//要创建 PLAYLIST 的应用
config_msg.content.data[0] = APP_ID_PLAYLIST;

//应用的输入参数
//bit0~bit4 表示来源于那个应用的 ID
//bit5~bit7 指示创建 PLAYLIST 的存储介质
config_msg.content.data[1] = PARAM_FROM_CONFIG | PLIST_DISK_C;

//向 process manager 发送消息
send_async_msg(APP_ID_MANAGER, &config_msg);
```

19.3.2 应用的退出

应用的退出处理：

- ✧ 若是由 UDISK 和 CONFIG 应用调用 PLAYLIST 的，则退出到 MAINMENU 应用。
- ✧ 若是 MUSIC，VIDEO，PICTURE，EBOOK 应用调用 PLAYLIST 的,则回到对应的应用。

退出示例代码：

```
msg_apps_t msg;
//给 process manager 发送创建其它进程消息

//消息类型，为创建新应用
msg.type = MSG_CREAT_APP;

msg.content.data[1] = (uint8) PARAM_FROM_PLAYLIST;

//判断入口参数，决定返回那个应用
switch (enter_mode)     {
    case PARAM_FROM_MUSIC:
    msg.content.data[0] = APP_ID_MUSIC;
    break;
    case PARAM_FROM_PICTURE:
    msg.content.data[0] = APP_ID_PICTURE;
    break;
    case PARAM_FROM_VIDEO:
```

```
        msg.content.data[0] = APP_ID_VIDEO;
        break;
        case PARAM_FROM_EBOOK:
        msg.content.data[0] = APP_ID_EBOOK;
        break;
        case PARAM_FROM_CONFIG:
        msg.content.data[0] = APP_ID_MAINMENU;
        break;
        case PARAM_FROM_UDISK:
        break;
        default:
        msg.content.data[0] = APP_ID_MAINMENU;
        break;
}

//向 process manager 发送消息
send_async_msg(APP_ID_MANAGER, &msg);
```

19.4 应用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
Applib 的全部函数
Common UI 的菜单，headbar，动画和其他公共提示模块
Enhanced 库中的 ID3 解析等模块

19.5 PLAYLIST 应用的内存空间分配说明

由于 PLAYLIST 是以应用的方式实现，代码可用前台的代码和数据空间。另因需要链接 ENHANCED 模块，保留前台的两个 ENHANCED BANK 代码；而 nand 需要 16K 的 cache 空间。所以可用的 RAM 空间为 79K，起始地址为 0x28000。

1、可用数据区域

  1）前台数据区域

[0x1d200--- 0x1d9ff]
AP_FRONT_DATA (AP+BASAL+ENHANCED)：(2K)

存放应用全局数据

2）连续 RAM 空间

PLAYLIST 建表RAM
0x28000~ 0x3BBFF: 79K

存放文件 ID3 信息和排序接口

3）nand 的 16k cache 空间

UD_WRITE_CACHE : 16K

Nand 写缓存

2、可用代码区域

1）常驻代码区

[0x1ee00--- 0x1f5ff] AP_FRONT_RCODE: (2K)

存放 VRAM 读写操作，和遍历文件处理

2）AP bank 空间

[0x1fe00--- 0x205ff]
AP_BANK_FRONT_CONTROL：(2K)
[0x1f600--- 0x1fdff]
AP_BANK_FRONT_UI：(2K)

存执行代码

3)ENHANCED 模块 bank 空间

[0x27800--- 0x27fff] AP_FRONT_ENHANCE2 :2K

[0x27000--- 0x277ff] AP_FRONT_ENHANCE1:2k

存 解析文件 ID3 代码

版 权 所 有 侵 权 必 究

2、xn 文件表述

1）ap_plist.xn

规划 AP_FRONT_DATA,AP_FRONT_RCODE,AP_BANK_FRONT_CONTROL, AP_BANK_FRONT_UI空间使用

2）id3_link.xn

规划 ENHANCED 模块 ID3 解析代码使用

3）common_front_no_selector.xn

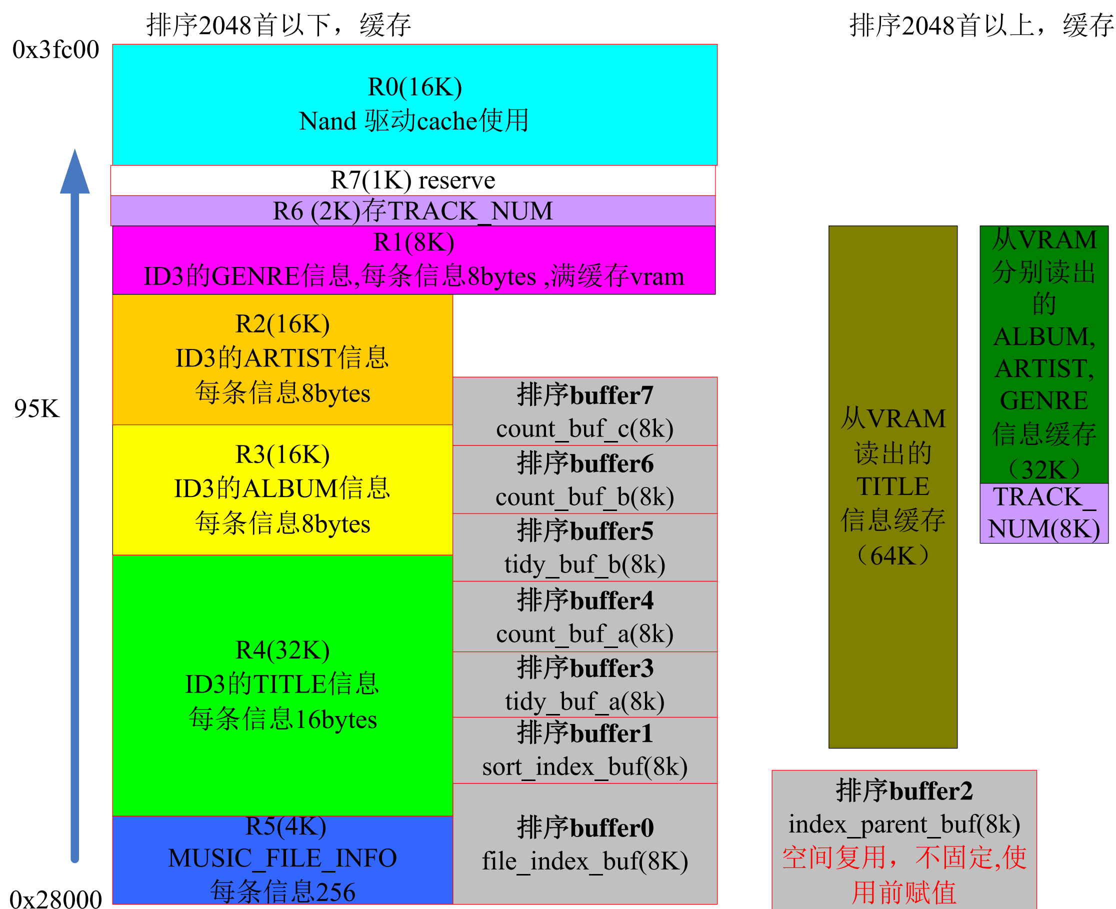
规划 applib 模块代码使用

19.6 应用中 RAM 空间的划分

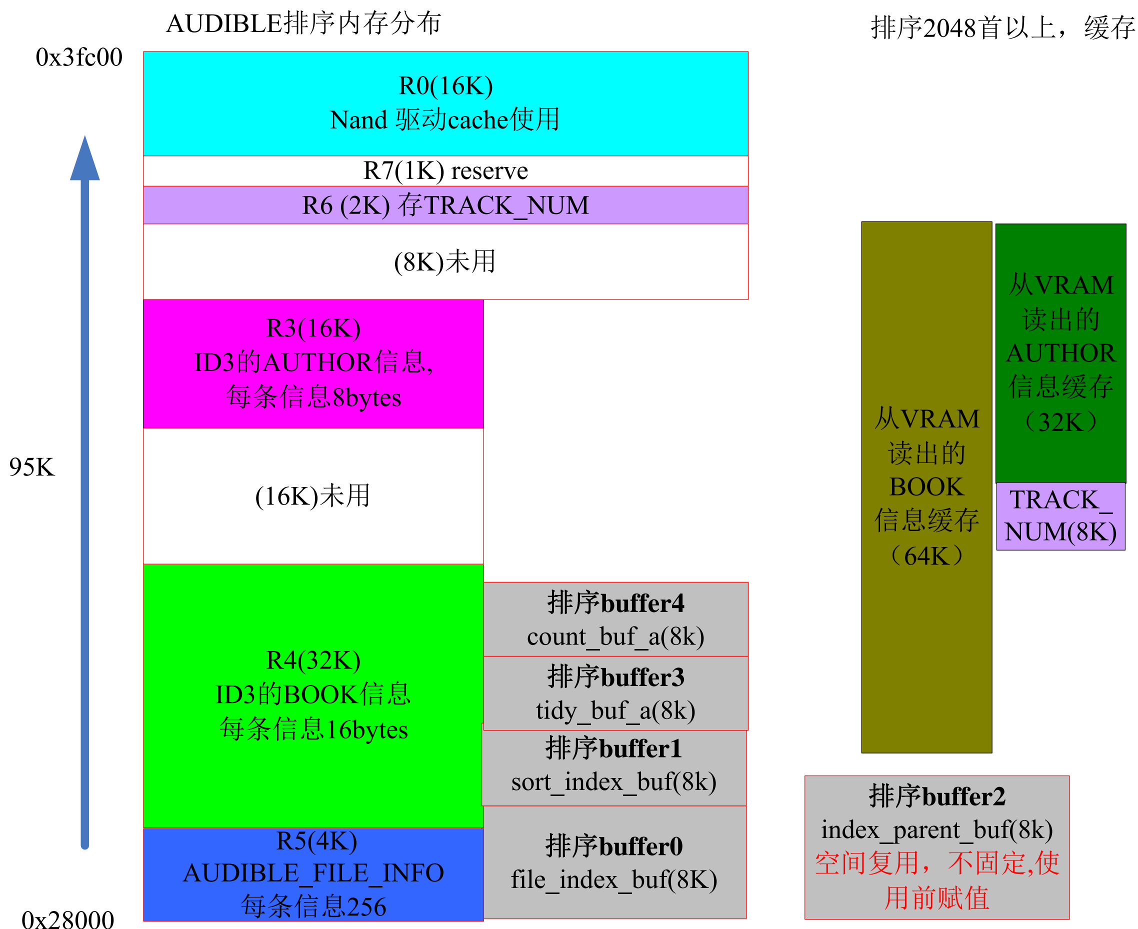
保证排序 2000 的歌曲时，不用使用 VRAM 作缓存，提高性能；先排序 MUSIC 文件，再排序 AUDIBLE 文件。

另因为排序的规格，列表之间排序有依赖性，MUSIC 排序顺序：TITLE， ALBUM，ARTIST ， GENRE ； 而 AUDIBLE 序顺序 ： BOOK ， AUTHOR ； 其他 PICTURE ,VIDEO,EBOOK 只排序文件名。

1、 生成 MUSIC 的 PLAYLIST 划分。

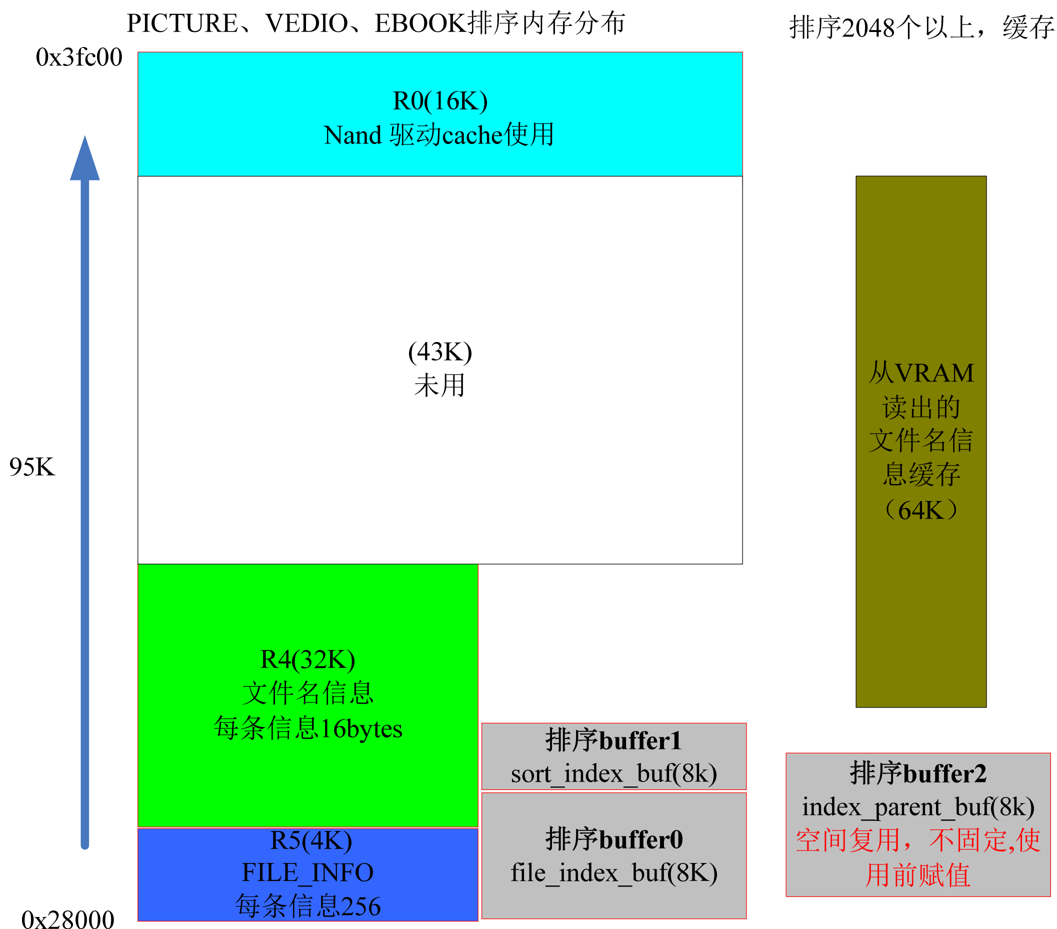


2、 生成 AUDIBLE 的 PLAYLIST 划分。



3、 生成 PICTURE、VIDEO、EBOOK 排序内存分布。

版 权 所 有 侵 权 必 究



4、 解析排序 buffer 的使用

file_index： 为磁盘查找文件时，按文件查找到的先后分配给文件的序号。

sort_index： 为排序后的 file_index,其对应的排序位置。

index_parent：为排序后的 file_index 器所属的父级偏移位置。

8K 的 buffer 的使用定义，如下表：

buffer 名称	buffer 存放的内容
file_index_buf	sort_index 为索引，存放排序后的 file_index
sort_index_buf	file_index 为索引，存放排序位号 sort_index
index_parent_buf	file_index 为索引，存放 file_index 对应的父 tree 的位置偏移号
tidy_buf_a	file_index 为索引，存放归类的编号（若两个文件 ID3 信息相同的话，编号相同）
tidy_buf_b	file_index 为索引，存放归类的编号（若两个文件 ID3 信息相同的话，编号相同）
count_buf_a	tree 的个数为索引，存放每个 tree 包含的文件的个数

版 权 所 有 侵 权 必 究

count_buf_b	tree 的个数为索引，存放每个 tree 包含的文件的个数
count_buf_c	tree 的个数为索引，存放每个 tree 包含的文件的个数

各类排序时，buffer 的使用情况，如下表：

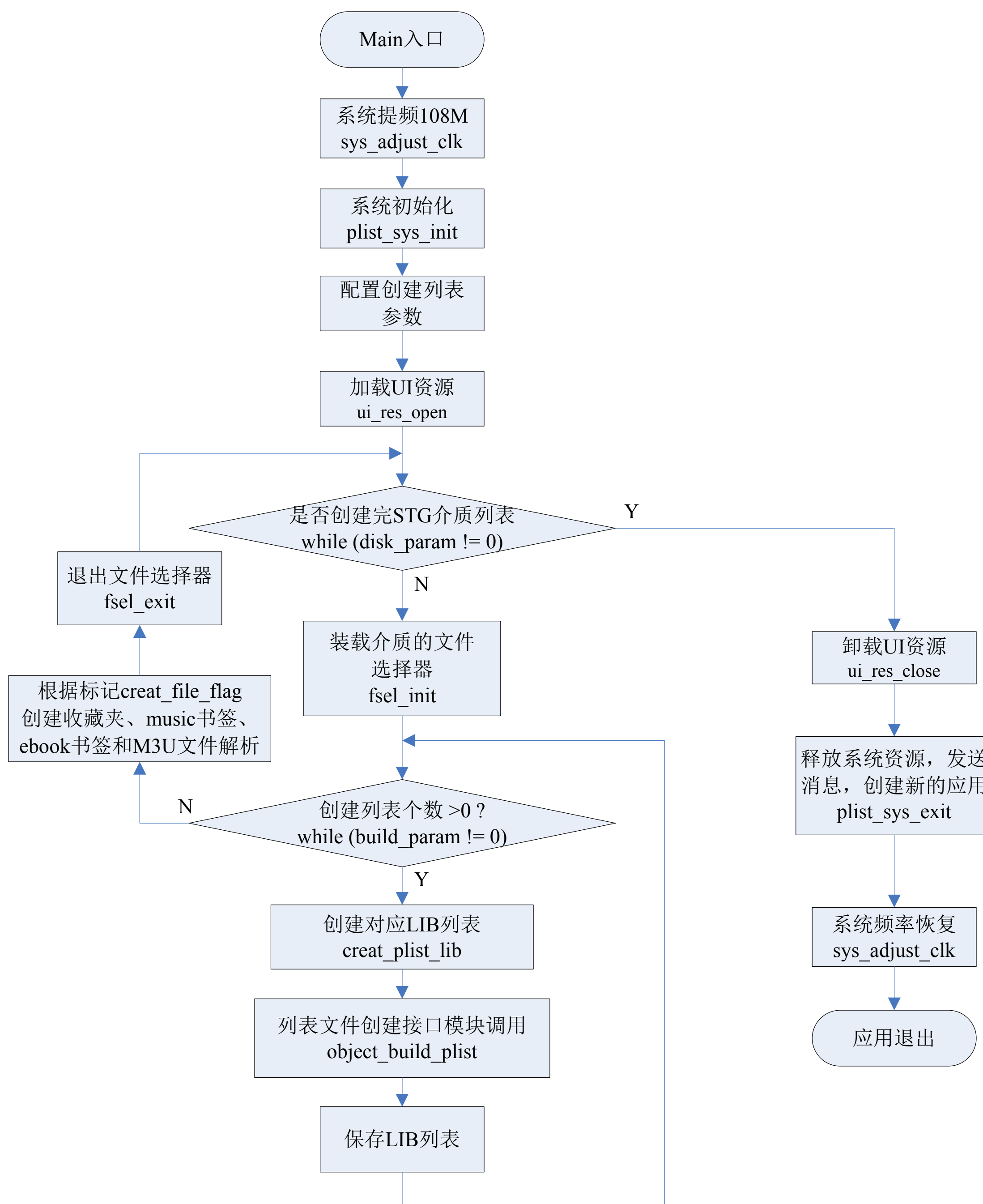
排序类别\buffer 名称	TITLE	ALBUM	ARTIST	GENRE	BOOK	AUTHOR
file_index_buf	used	used	used	used	used	used
sort_index_buf	used	used	used	used	used	used
index_parent_buf	used	used	used	used	used	used
tidy_buf_a		used	used	used		used
tidy_buf_b			used	used		
count_buf_a		used	used	used		used
count_buf_b			used	used		
count_buf_c				used		

备注：当排序完某一类的 file_index 后，对应的 ID3 信息存放 buffer 就可以被使用。例如：排序完 title 信息的 file_index，那 R4 空间的 RAM 就可以作为排序 buffer 使用，因为后续排序 album，artist，genre 不需要 title 信息排序了。

19.7 PLAYLIST 应用的业务流程

19.7.1 应用的总流程

应用流程的总流程如图下：



代码流程结构：

*int main(int argc, const char *argv[])*

{

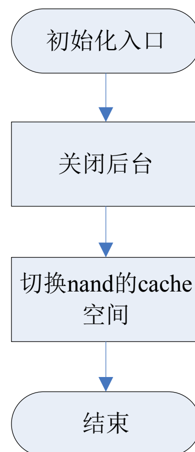
。。。。。。

```
        //系统提频处理
        save_freq_level = sys_adjust_clk(FREQ_108M, 0);
       。。。。。。
       //系统初始化
       plist_sys_init();
       。。。。。。
       //配置系统参数
       plist_param = (uint16) ((uint8) argc & PLIST_DISK_ALL) << 8;
      。。。。。。
      //加载 Ui 资源
      。。。。。。
      ui_res_open(plist_ui_sty, UI_AP);
     。。。。。
//是否创建完 STG 介质列表
while (disk_param != 0)
{
。。。。。。
//装载介质的文件选择器
 ap_vfs_mount = fsel_init(&fsel_param, MODE_NORMAL);
。。。。。。
//创建列表个数 >0 ?
 while (build_param != 0)
 {
    //创建对应 LIB 列表
        if (FALSE != creat_plist_lib(lib_num))
         {
          //列表文件创建接口模块调用
              object_build_plist[lib_num]();
            。。。。。。
          }
 }
//根据标记 creat_file_flag
if (creat_file_flag == TRUE)
{
 //创建电子书
 creat_ebook_bmk();
。。。。。。。
 }
```

版 权 所 有 侵 权 必 究

```
stg_exit:
    //退出文件选择器
     fsel_exit();
    。。。。
     }

    //卸载 UI 资源
    ui_res_close(UI_AP);
    。。。。。。
    //系统退出
    plist_sys_exit(enter_mode);
  。。。。。
    //恢复频率
  sys_adjust_clk(save_freq_level, 0);
  。。。。。。
    }
```

19.7.2 初始化模块的流程



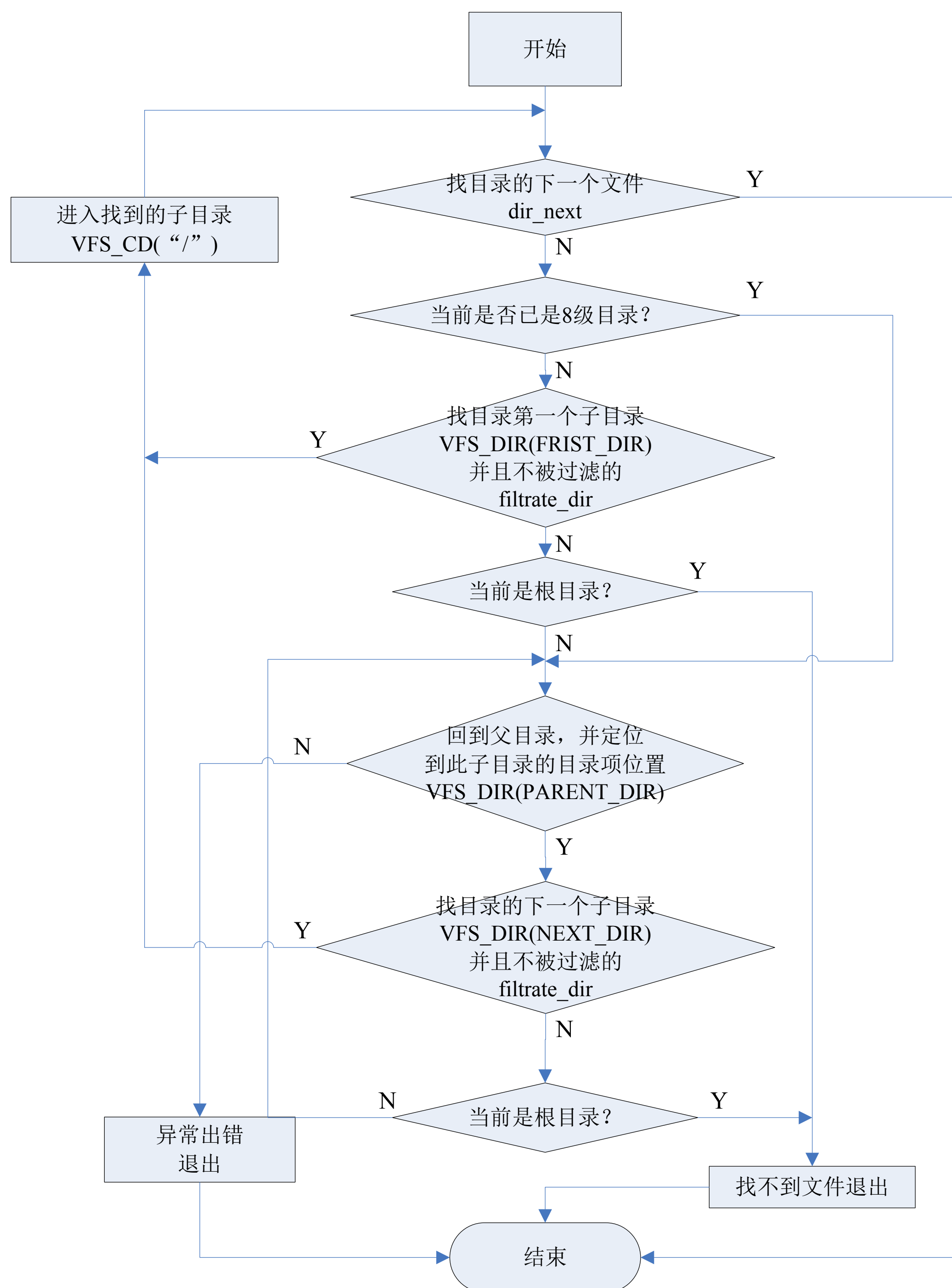
代码结构流程:

```
void plist_sys_init(void)
{
```

```
        //先关闭后台
        close_engine();

        //安装 nand 存储驱动，切换 nand 的 cache 空间
        if (sys_drv_install(DRV_GROUP_STG_BASE, MODE_NORMAL, stg_drv_name[0]) != -1)
        {
            /****
             因为要用到 nand 默认的 cache 页面地址 0x9fc34000 来建表，
             所以需要把 nand 的 cache 切到别的页面
             ****/
    #ifndef PC
            base_op_entry((void*) 1, (void*) NAND_CACHE_ADDR, 0, BASE_UPDATE);
    #endif
            sys_drv_uninstall(DRV_GROUP_STG_BASE);
        }

    }
```

19.7.3 遍历文件方式

举例接口 plist_fsel_get_nextfile(uint8 *strfile) 处理流程

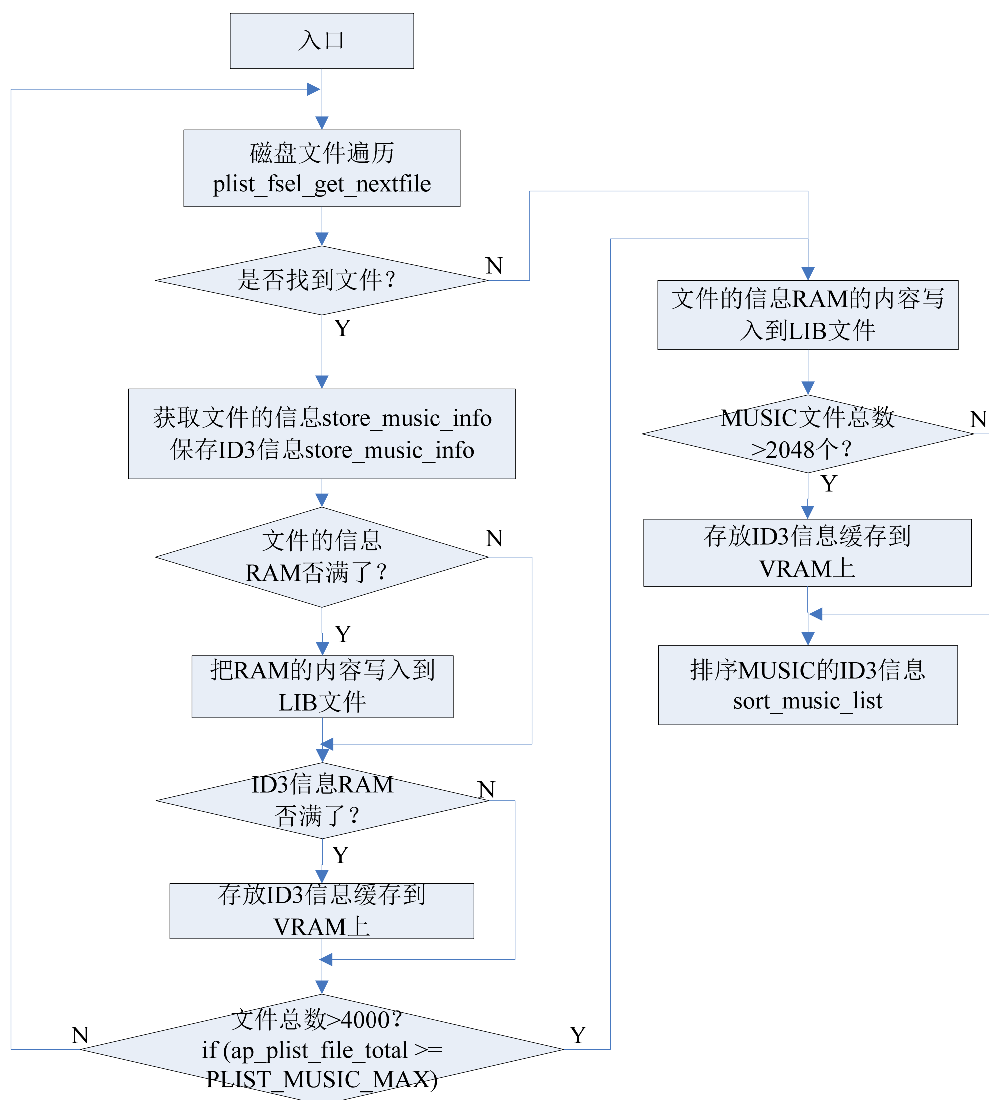


代码结构流程：

具体代码详见接口 bool plist_fsel_get_nextfile(uint8 *strfile)。

19.7.4 获取文件信息存储

举例：遍历查找文件，保存 MUSIC 文件的信息和 ID3 信息保存。



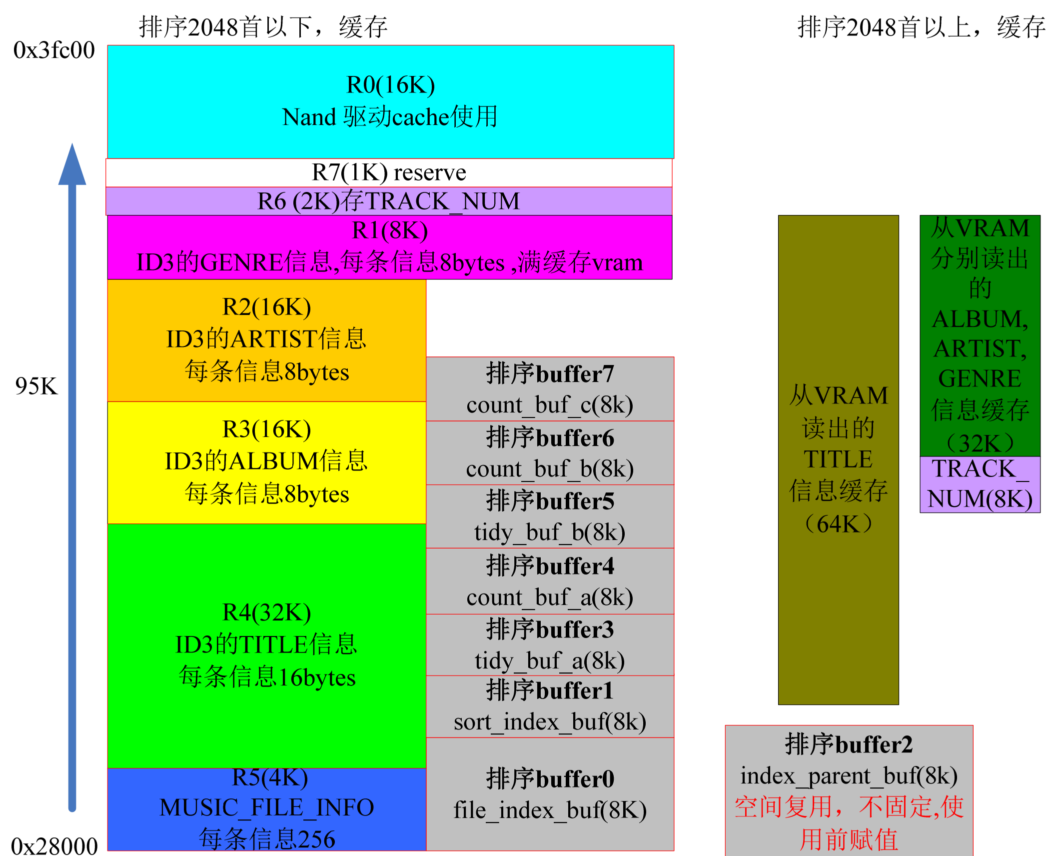
代码结构流程:

```
uint16 scan_music_file(void)
{
    。。。。。。
//磁盘遍历文件
    while (FALSE != plist_fsel_get_nextfile((char*) &ext_name))
    {
```

```
        。。。。。。
        //获取文件信息和 ID3 信息
        get_music_info(ext_name);
        //存储文件信息
        store_music_info(ap_plist_file_total);
        ap_plist_file_total++;
        if (ap_plist_file_total >= PLIST_MUSIC_MAX)
        {
            break;
        }
    }

    if (ap_plist_file_total > PLIST_SAVE_MAX)
    {
        。。。。。。
    }

    if (ap_plist_file_total > PLIST_GENRE_MAX)
    {
        。。。。。。
    }

    //RAM 中是否有未写入的数据，保存到文件中
    if (ap_plist_file_total > 0)
    {
        temp = ap_plist_file_total % (FILE_INFO_BUF_SIZE / PLIST_FILE_SIZE);
        if (temp == 0)
        {
            //刚好写满
            temp = FILE_INFO_BUF_SIZE / PLIST_FILE_SIZE;
        }
        save_to_file((uint8*) FILE_INFO_ADDR, temp * PLIST_FILE_SIZE);
    }
    return ap_plist_file_total;
}
```

19.7.5 列表排序

RAM 空间划分如下：

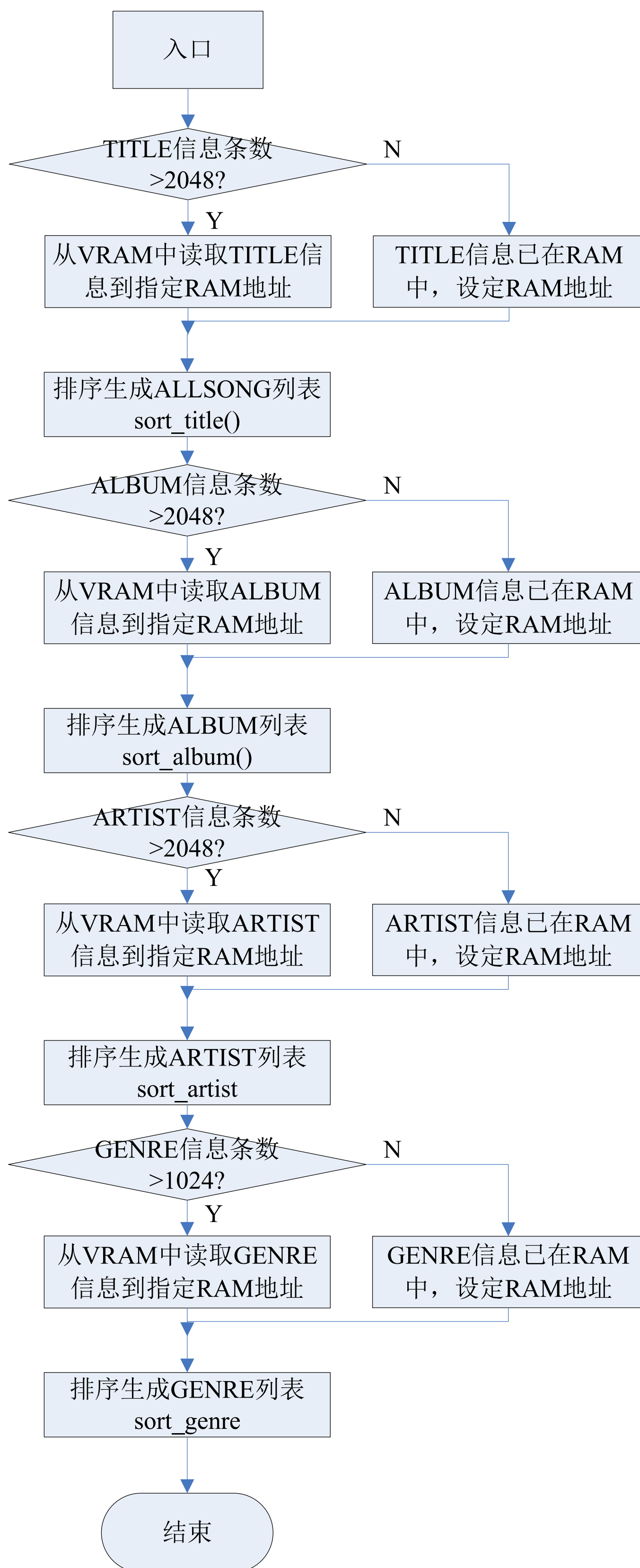


如果搜索到的文件不足 2048 个文件时，则系统没有使用 VRAM 来缓存 ID3 信息；直接可在 RAM 中比较排序。

file_index： 为磁盘查找文件时，按文件查找到的先后分配给文件的序号。
sort_index： 为排序后的 file_index,其对应的排序位置。
index_parent：为排序后的 file_index 其所属的父级 tree 节点偏移位置。

1、MUSIC 文件的排序流程如下：

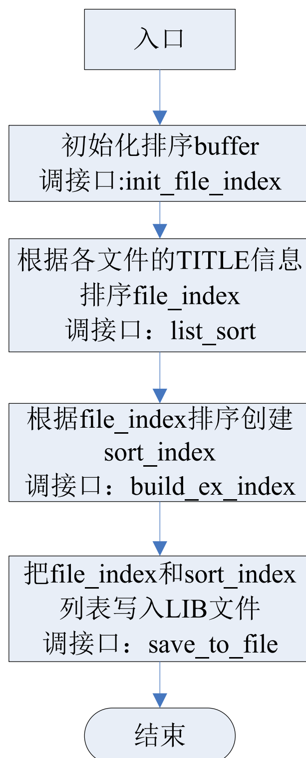
总体框架设计流程如下：



版 权 所 有 侵 权 必 究

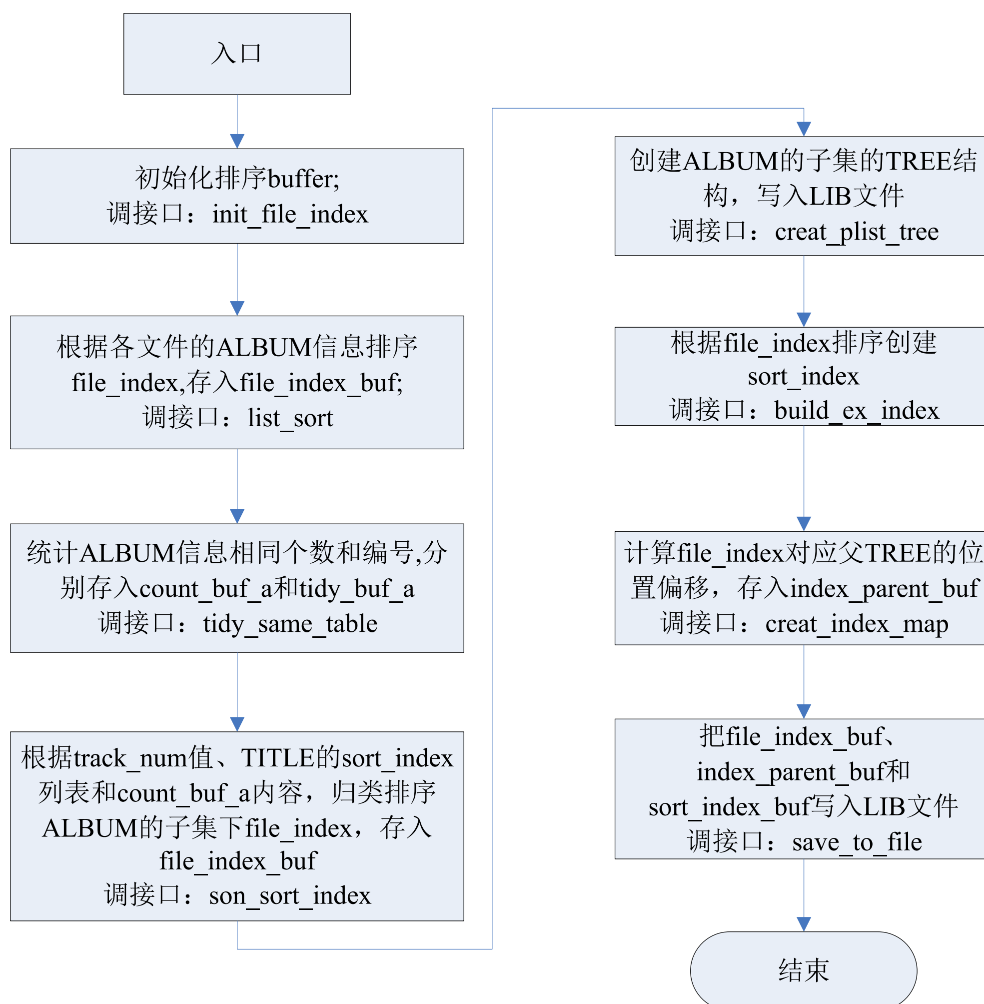
具体代码实现流程见接口 sort_music_list。

(1)TITLE 排序



具体代码实现流程见接口 sort_title。

(2)ALBUM 排序



具体代码实现流程见接口 sort_album。

(3)ARTIST 排序

具体代码实现流程见接口 sort_artist。

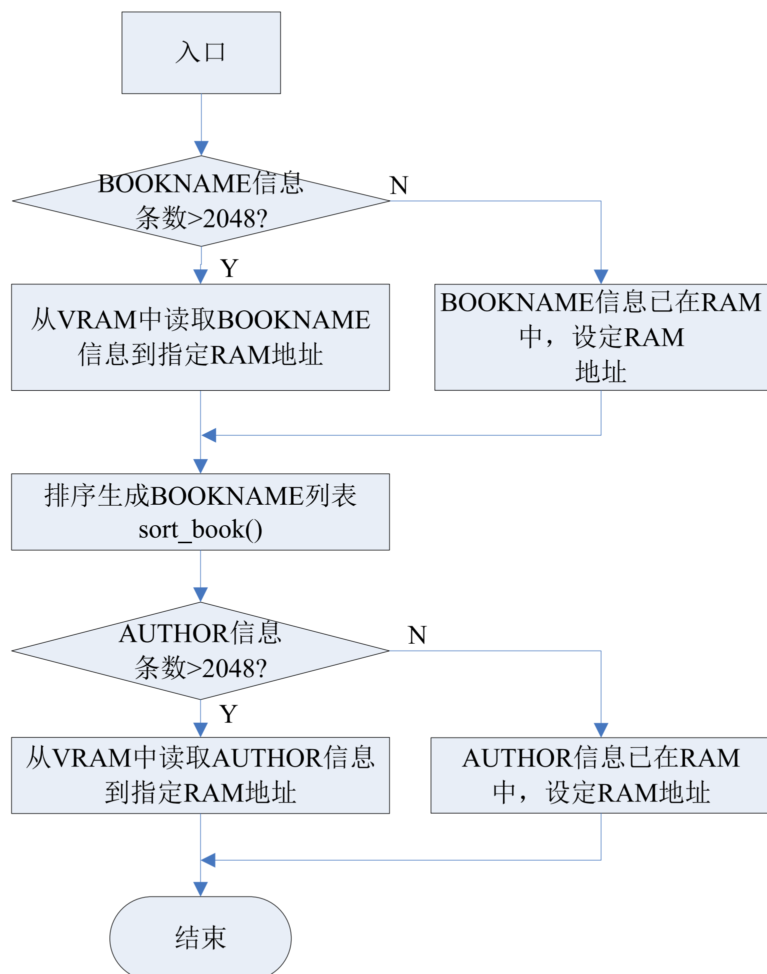
(4)GENRE 排序

版 权 所 有 侵 权 必 究

具体代码实现流程见接口 sort_genre。

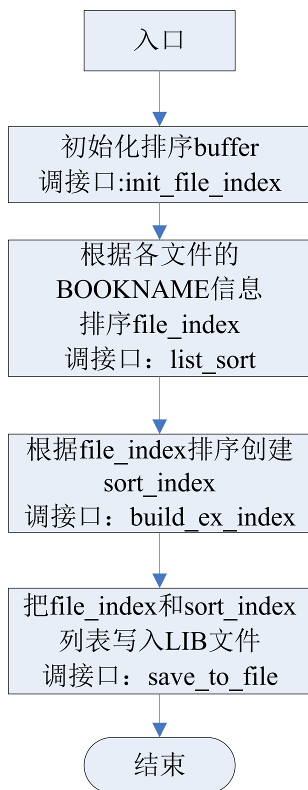
2、AUDIBLE 文件的排序流程如下：

总体设计框架流程如下：



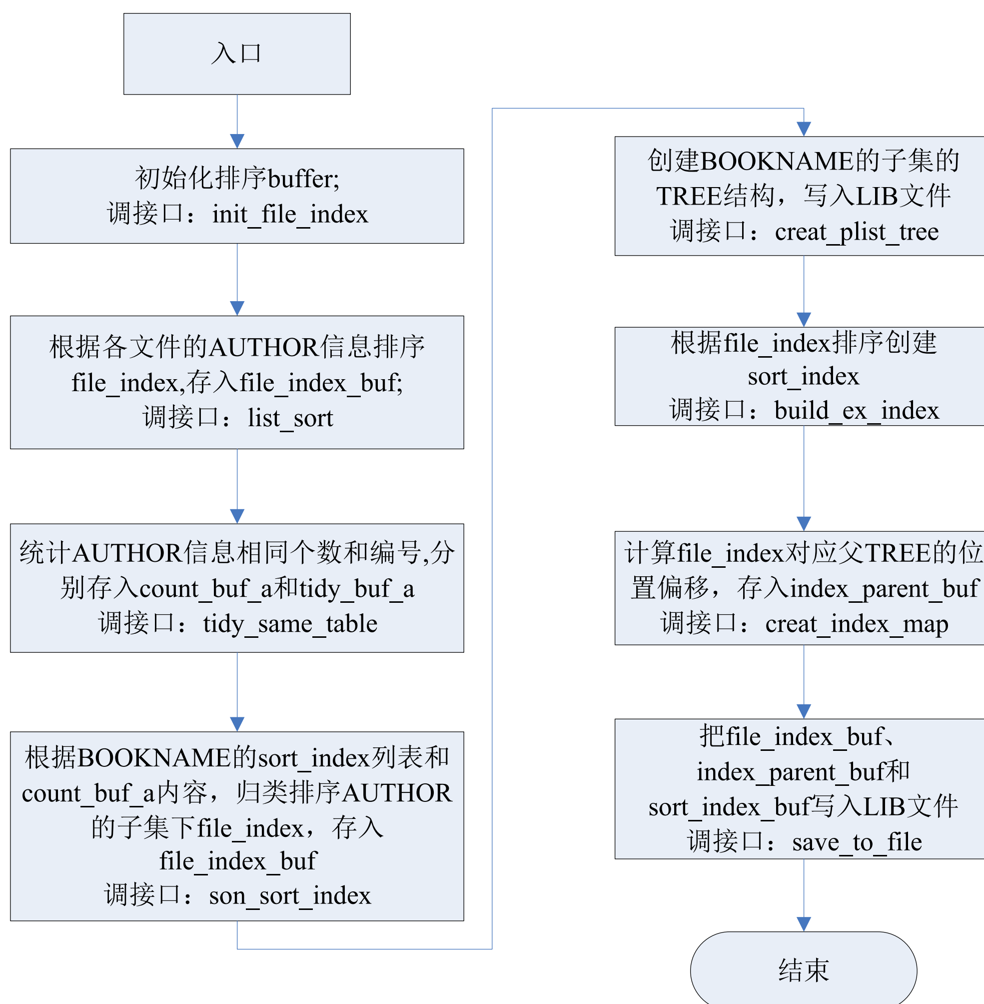
具体代码实现流程见接口 sort_audible_list。

(1)BOOK 排序



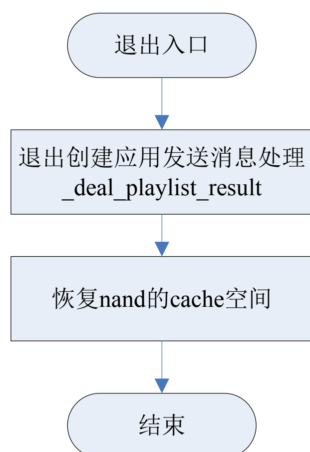
具体代码实现流程见接口 sort_book。

(2)AUTHOR 排序



具体代码实现流程见接口 sort_author。

19.7.6 应用退出模块流程图



代码流程结构：

```
void plist_sys_exit(app_param_e enter_mode)
{
    //退出创建应用发送消息处理
    _deal_playlist_result(enter_mode);
    //重装 nand 驱动，恢复 nand 的 cache 默认地址 0x9fc34000
    if(sys_drv_install(DRV_GROUP_STG_BASE,MODE_NORMAL, stg_drv_name[0]) !=
-1)
    {。。。。。
    }


}
```

19.8 Playlist 的数据结构说明

19.8.1 结构体数据结构

1、存放在 LIB 的文件信息结构，预占 256 字节

版 权 所 有 侵 权 必 究

```
//文件 ID3 信息结构
typedef struct
{
    char title_string[ID3_ITEM_SIZE * 2];   //标题&文件长名  （排序 16bytes）
    char artist_string[ID3_ITEM_SIZE];      //艺术家名称     （排序 8bytes）
    char album_string[ID3_ITEM_SIZE];       //专辑名称       （排序 8bytes）
    char gener_string[ID3_ITEM_SIZE];       //流派名称       （排序 8bytes）
    char track_string[8];                   //track num string
    uint8 filename[4];                      //文件的后缀名
    uint8 id3_tag_flag;                     //是否存在 ID3 信息的标志：1--有，0--无
    uint8 flag;                             //该文件是否存在 1-有，0-无，2-已删除
    uint16 track_num;                       //专辑曲目号
    uint32 cluster_no;                      //文件目录项所在簇号
    uint32 dir_entry;                       //文件的目录项在当前簇内的偏移
    uint16 prev_offset;                     //上一个存放位置的偏移
    uint16 next_offset;                     //下一个存放位置的偏移
    pdir_layer_t dir_layer_info;            //目录层次信息
} plist_f_info_t; // 256 bytes
```

2、树结构，指示上下级层次关系，预占 16 字节

```
//播放列表 tree 结构
typedef struct
{
    uint8 flag;              //tree 标志  :0-no used,1-exist,2-deleted
    uint8 type;              //当前列表显示 ID3 信息类
    uint16 son_num;          //下级子列表的个数
    uint16 file_total;       //列表下所有文件的个数
    uint16 file_index;       //起始文件的序号，可读出出 sort_index 值
    uint16 son_offset;       //子 list 的存放位置偏移个数
    uint16 parent_offset;    //父 list 的存放位置偏移个数
    uint16 prev_offset;      //上一个的偏移
    uint16 next_offset;      //下一个的偏移
} plist_tree_t;
```

19.8.2 列表 LIB 文件结构

5 个列表 LIB 的结构相同，具体如下：

1、 头信息，在文件首扇区（512 字节）

（1）前 16 个字，存放 LIB 主要信息

```
typedef struct
{
      char plist_name[8];              //LIB 识别标记
      uint16 file_total;               //文件的总数
      uint16 file_info_offset;       //文件信息存放的偏移，以扇区为单位
      uint16 reserve[2];               //保留对齐
} plist_head_t;

plist_name 可取值：
plist_name={"MUSIC"};
plist_name={"AUDIBLE"};
plist_name={"VIDEO"};
plist_name={"PICTURE"};
plist_name={"EBOOK"};
```

（2）接着 28*6 字节，存放各个 item 信息

```
typedef struct {
   char item_name[8]; //item 名称
   uint16 son_tree_num[3]; //级子表的 tree 的个数
   uint16 son_tree_offset[3]; // 3 级子表的 tree 存放位置偏移，以扇区为单位
   uint16 file_index_offset; //排序后的 file_index 表扇区偏移，存 file_index
   uint16 sort_index_offset; // map 表扇区偏移，file_index 为索引，存排序位号 sort_index
   uint16 index_parent_offset; // parent 表扇区偏移，file_index 为索引，存 parent_offset
   uint16 reserve;
} plist_item_t; //（28bytes）
音乐歌曲列表 item_name 的取值：
allsong_name[]={“ALLSONG”};
album_name[]={“ALBUM”};
artist_name[]={“ARTIST”};
genre_name[]={“GENRE”};

audible 列表 item_name 的取值：
a_book_name[]={“BOOK”};
a_author_name[]={“AUTHOR”};
video、picture 和 ebook 列表 item_name 的取值分别是
char allfile_name[]={"ALLFILE"};
```

（3）512 字节结构如图下：

plist_head_t 16 字节
plist_item_t 28 字节
plist_item_t 28 字节
plist_item_t 28 字节
plist_item_t 28 字节
。。。
reserve

2、 第 2 个扇区起存放所有文件的信息

文件信息占 256 字节，存放以扇区对齐结束。存放是以链表的方式，方便查找。若其中有文件被删除，将从链表中删除；置数据结构无效标记。

如表：

plist_f_info_t file_1
plist_f_info_t file_2
plist_f_info_t file_3
plist_f_info_t file_4
。。。
(以上 512 字节对齐)

3、 接着存放是各个 item 的具体 tree 内容。

tree 结构如下：

```
typedef struct
{
    uint8 flag;              //tree 标志 :0-no used,1-exist,2-deleted
    uint8 type;              //当前列表显示 ID3 信息类
    uint16 son_num;          //下级子列表的个数
    uint16 file_total;       //列表下所有文件的个数
    uint16 file_index;       //起始文件的序号，可读出出 sort_index 值
    uint16 son_offset;       //子 list 的存放位置偏移个数
    uint16 parent_offset;    //父 list 的存放位置偏移个数
    uint16 prev_offset;      //上一个的偏移
    uint16 next_offset;      //下一个的偏移
} plist_tree_t;
```

版 权 所 有 侵 权 必 究

各部分存放的时候也是以扇区对齐结束。

1）tree 结构是以链表的方式存放，若其中有文件被删除，将从链表中删除；置数据结构无效标记。

2）file_index 为查找的文件序号；若对应文件被删除则置为 0xFFFF 标记。

3）sort_index 为文件序号对应的排序号，若对应文件被删除，置为 0xFFFF 标记。

4）index_parent 为文件序号对上级 tree 位置偏远号，若对应文件被删除，置为 0xFFFF 标记。

如下表：

ALLSONG - plist_tree_t son_tree1
ALLSONG - plist_tree_t son_tree2
ALLSONG - plist_tree_t son_tree3
ALLSONG - uint16 file_index
ALLSONG - uint16 sort_ index
ALLSONG - uint16 index_parent
。。。
GENRE - plist_tree_t son_tree1
GENRE - plist_tree_t son_tree2
GENRE - plist_tree_t son_tree3
GENRE - uint16 file_index
GENRE - uint16 sort_ index
GENRE - uint16 index_parent
。。。

19.9 如何在列表上增加或者删除一种文件格式

各列表对应的 LIB 和文件格式值：

列表类型	LIB 文件	包含文件格式(BITMAP)
MUSIC	MUSIC.LIB	file_type_bitmap[0]
AUDIBLE	AUDIBLE.LIB	file_type_bitmap[1]
VIDEO	VIDEO.LIB	file_type_bitmap[2]

版 权 所 有 侵 权 必 究

PICTURE	PICTURE.LIB	file_type_bitmap[3]
TEXT	EBOOK.LIB	file_type_bitmap[4]
M3U	M3U.LIB	file_type_bitmap[5]

如上，各列表对应的文件格式在数组 file_type_bitmap 中定义，用户可自行修改定义。

file_type_bitmap 可以指示为 bitmap 值也可以指示为扩展名组的内存地址。

若想自己定义一组查找的文件后缀名组合，可如下实现:

```
//定义扩展名数组（最好为常驻的，因为要提供给 FS 使用）
Char creat_file_ext[3] [4]= {“MP3”,”WMA”,”ABC”};

//把扩展名数组的地址填入 file_type_bitmap
file_type_bitmap[0] = & creat_file_ext;
```

19.10 如何去掉 VIDEO 或者 AUDIBLE 的播放列表

现系统默认创建列表项由宏定义 PL_BUILD_ALL 决定，如下：

```
#define PL_BUILD_ALL              0x3f
```

各列表创建对应使能位如下：

```
#define PL_BUILD_MUSIC            0x01
#define PL_BUILD_AUDIBLE          0x02
#define PL_BUILD_VIDEO            0x04
#define PL_BUILD_PICTURE          0x08
#define PL_BUILD_EBOOK            0x10
#define PL_BUILD_M3U              0x20
```

1）若删除 VIDEO 表创建则可修改 PL_BUILD_ALL 值为 0x3b。

版 权 所 有 侵 权 必 究

2）若删除 AUDIBLE 表创建则可修改 PL_BUILD_ALL 为 0x3b：

19.11 如何过滤掉录音应用产生的文件

过滤录音产生的文件是通过过滤指定的文件夹来实现，现系统默认过滤指定的文件名为“RECORD”，过滤名字定义为：

```
static const uint8 rec_dir_name[] = "RECORD   ";
```

用户也可自行修改指定过滤的文件夹名称。

20 ap_setting 应用

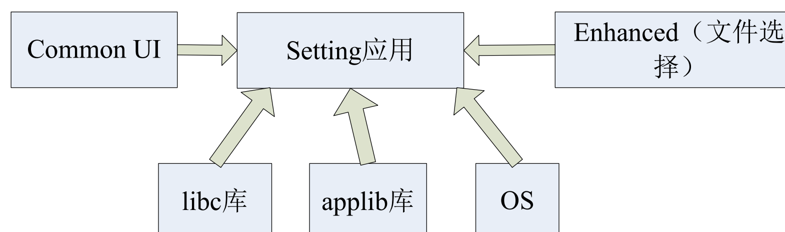
20.1 需求概述

Setting 应用将方案实现对系统各参数的配置和管理，以及一些系统信息的显示。主要实现对音乐播放参数设置、声音设置、显示设置、日期与时间设置、幻灯片放映设置、语言设置等等。

20.2 总体架构设计

20.2.1 总体架构图

Setting 应用需要系统和公共函数库的支持：



20.2.2 功能模块划分

模块名称	功能简述	对应文件
主模块	负责应用初始化和退出，包括资源文件，菜单配置文件的加载和卸载，以及应用的场景管理	main_setting.c
菜单项处理模块	时间参数相关菜单处理	menu_callback_time_setting.c
	日期参数相关菜单处理	menu_callback_date_setting.c
	显示参数相关菜单处理	menu_callback_display_setting.c
	语言参数相关菜单处理	menu_callback_language.c
	Music 播放设置参数相关菜单处理	menu_callback_play_mode.c
	幻灯片播放设置参数相关菜单处理	menu_callback_slide_show_setting.c
	Fullsound 和 eq 设置参数相关菜单处理	menu_callback_sound_setting.c
	Srs eq 设置参数相关菜单处理	menu_callback_sound_setting2.c
	定时关机、pc 连接首选项、播放器信息、法律信息等参数相关菜单处理	menu_callback_others.c
	格式化、出厂设置、cd 安装驱动等相关参数菜单处理	menu_callback_others2.c
菜单项配置模块	菜单配置的数据	ap_cfg_menu_setting.c

20.3 与其它应用的同步和交互

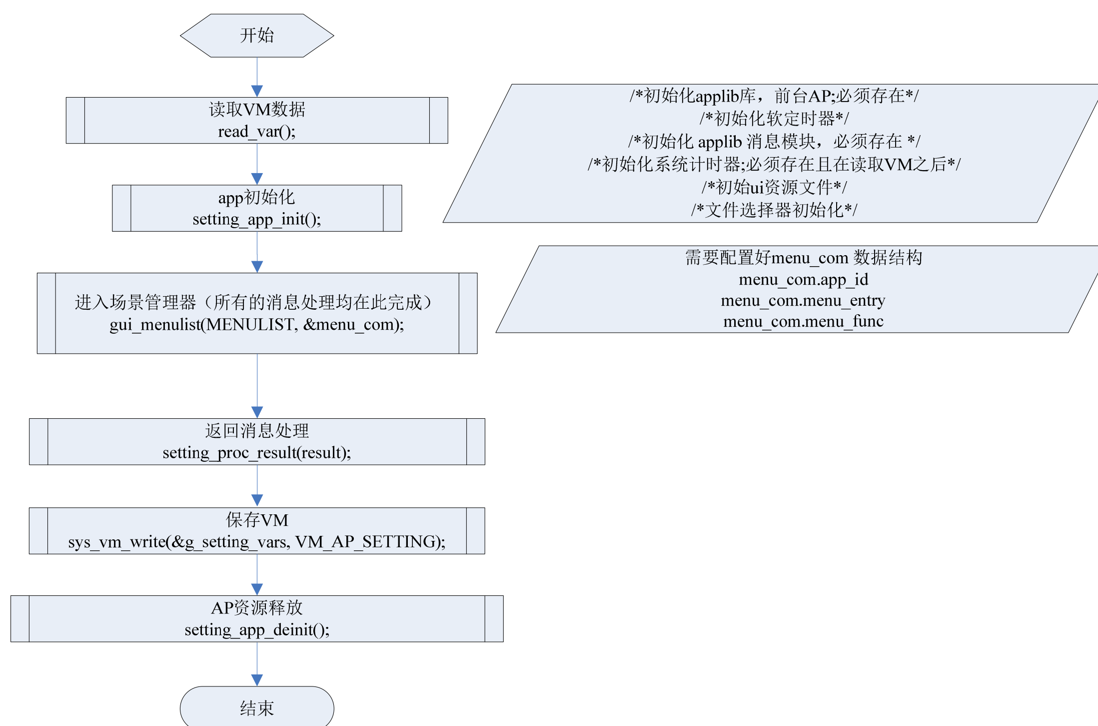
如果有后台音乐播放，在 music 设置相关的菜单确定后，需要发送相关信息的同步消息给后台引擎，如音乐来源/EQ 参数等等。

版 权 所 有 侵 权 必 究

20.4 应用依赖库及其接口

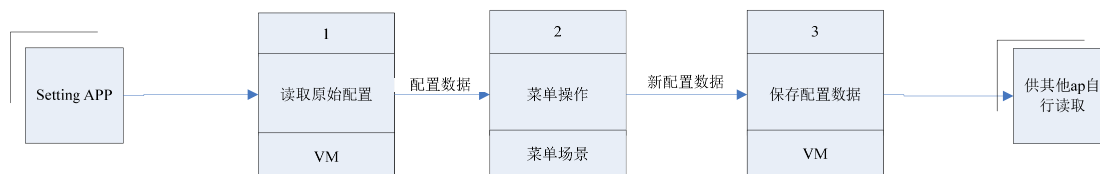
系统和 libc 的接口 api.a
应用运行时库 ctor.o
Applib 的全部函数
Common UI 的菜单，headbar，动画和其他公共提示模块
Enhanced 库中的文件选择和 ID3 获取模块

20.5 应用的业务流程



版 权 所 有 侵 权 必 究

20.6 配置数据的数据流图



20.7 关键的数据结构

（1）保存到 VRAM 的全局信息，此信息被分成两个部分，其中一部分为系统公共变量组合结构体信息，另外一部分为 music 相关设置信息：

```
typedef struct
{
    /*! magic，值为 0x55AA 时有效 */
    uint16 magic;
    //显示相关项
    /*! 背光亮度等级 */
    uint8 lightness;
    /*! 主题 0：蓝色 1：绿色 2：灰色 3：粉红色 4：红色 */
    uint8 theme;
    //+4bytes
    /*! 屏幕保护 0: 无 1: 数字时钟 2：相册图画 3：关闭屏幕 4：演示模式 */
    uint8 screen_saver_mode;
    /*! 背光时间 以 0.5 秒为单位，0 表示常亮 */
    uint8 light_timer;
    /*! 屏幕保护时间 以 0.5 秒为单位，0 表示无屏幕保护 */
    uint8 screen_saver_timer;
    /*! 省电关机时间 以 0.5 秒为单位，0 表示无省电关机 */
    uint8 poweroff_timer;
    //+8bytes
    /*! 定时返回正在播放界面 以 0.5 秒为单位，0 表示无须返回 */
    uint8 playing_timer;
    /*! 定时关机 9 睡眠）时间 以分钟为单位，0 表示不定时关机 */
    uint8 sleep_timer;
```

```
    //日期和时间
    /*! 时间格式 0：12 小时制 1：24 小时制 */
    uint8 time_format;
    /*! 日期格式 0：DD_MM_YYYY 1：MM_DD_YYYY 2：YYYY_MM_DD */
    uint8 date_format;
    //+12bytes
    /*! 界面语言，此值由显示驱动决定 */
    uint8 language_id;
    /*! 电脑连接首选项 0：MSC 1：MTP */
    uint8 online_device;
    /*! CD 安装程序 autorun 支持选择 0 ：autorun，1：  支持 autorun */
    uint8 autorun_set;
    /*! 支持卡选择  0:不支持  1:支持 */
    uint8 support_card;
    //+16bytes
    /*! 音量限制 */
    uint8 volume_limit;
    /*! 当前音量值 */
    uint8 volume_current;
    //图片设置
    /*! 幻灯片间隔时间 */
    uint8 slide_time;
    /*! 幻灯片重复模式 */
    uint8 slide_repeat_mode;
    //+20bytes
    /*! 幻灯片 shuffle 功能 */
    uint8 slide_shuffle;
    uint8 reserved[3];
} comval_t;


/*!
 * \brief
 *  music_comval_t 音乐设置变量组合结构体
 */
typedef struct
{
    /*! 音乐来源 */
    uint8 music_from;
```

```
    /*! 音乐循环方式 */
    uint8 music_repeat_mode;
    /*! 音乐随机不重复开关 */
    uint8 music_shuffle;
    /*! fullsound 音效开关 */
    uint8 fullsound;
    /*! srs 音效开关 */
    uint8 srs;
    /*! eq 模式设定 */
    uint8 eq_setting;
    /*! 变速播放 */
    int8  variable_playback;
    uint8 reserved;
    /*! eq 用户参数表 */
    int8 eq_parameter[12];
    /*! srs 用户参数表 */
    int8 srs_parameter[12];
} music_comval_t;

/*!
 * \brief
 *  setting_comval_t 设置应用全局参数结构体
 */
typedef struct
{
    /*! 系统公共变量组合结构体 */
    comval_t g_comval;
    /*! 音乐设置变量组合结构体 */
    music_comval_t music_comval;
} setting_comval_t;
```

20.8 如何增加一个配置项

1. 在 setting_common.h 的 comval_t 数据结构中添加该配置项，注意尽可能的保持该数据结构为 4 字节的整数倍；
2. 根据新增菜单的个数在 ap_cfg_menu_setting.c 的 conf_menu_item_t 数据结构类型的 item [ITEM_TOTAL]数组中添加菜单叶子项；
3. 更新 ITEM_TOTAL 数值；

4. 编写菜单确认键对应的功能函数、实时菜单回调函数（不一定需要）、按 option 键后回调函数
5. 为相关菜单准备字符串资源，并重新生成 setting.sty 和 setting_res.h 及 setting_sty.h
6. 编译并生成 AP
7. 用 modify 工具菜单编辑

21 ap_tools 应用

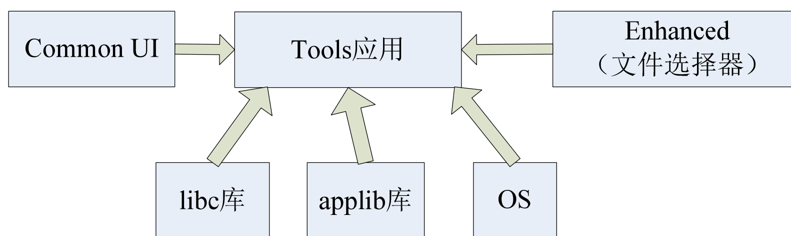
21.1 需求概述

TOOLS 应用是在基于 us212a 平台，实现了一些独立的功能及设计，例如日历、秒表、闹钟等。这些功能相对独立，与其他 ap（除 common、manager 外）基本无交互。

21.2 总体架构设计

21.2.1 总体架构图

Tools 应用需要系统和公共函数库的支持



21.2.2 功能模块划分

模块名称	功能简述	对应文件
主模块	TOOLS ap 程序入口、负责应用初始化和退出，包括资源文件，菜单配置文件的加载和卸载，	tools_main.c

	以及应用的场景管理	
	常驻代码段。消息处理及秒表计数等功能	tools_rcode.c
菜单项处理模块	TOOLS 主菜单处理；alarm 某些功能	tools_menu.c
	日历主要函数处理	menu_callback_calendar_tools.c
	秒表相关菜单处理	menu_callback_stopwatch_tools.c
	Alarm 相关菜单处理	menu_callback_alarm_tools.c
	定时 fm 相关菜单处理	menu_callback_fm_tools.c
	万年历的公历与阴历转换等函数	solar2lunar.c solar2lunar_data.c solar2lunar_main.c
菜单项配置模块	菜单配置的数据	ap_cfg_menu_tools.c

21.3 与其它应用的同步和交互

因该应用的独立性，除 common 与 manager 外，与其他应用基本无交互

21.4 应用依赖库及其接口

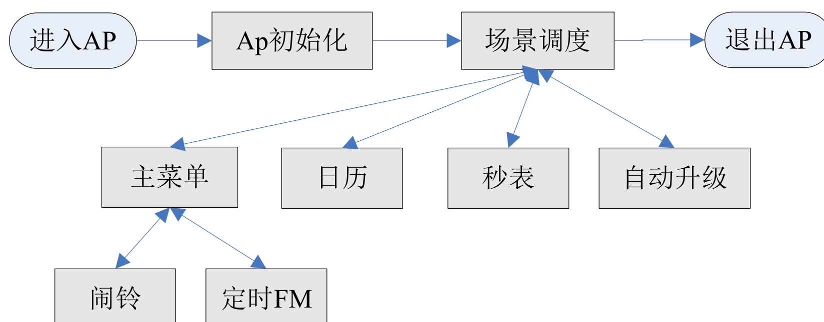
系统和 libc 的接口 api.a
应用运行时库 ctor.o
Applib 的全部函数
Common UI 的菜单，headbar，动画和其他公共提示模块
Enhanced 库中的文件选择

21.5 应用的业务流程

21.5.1 应用的总流程和场景调度流程

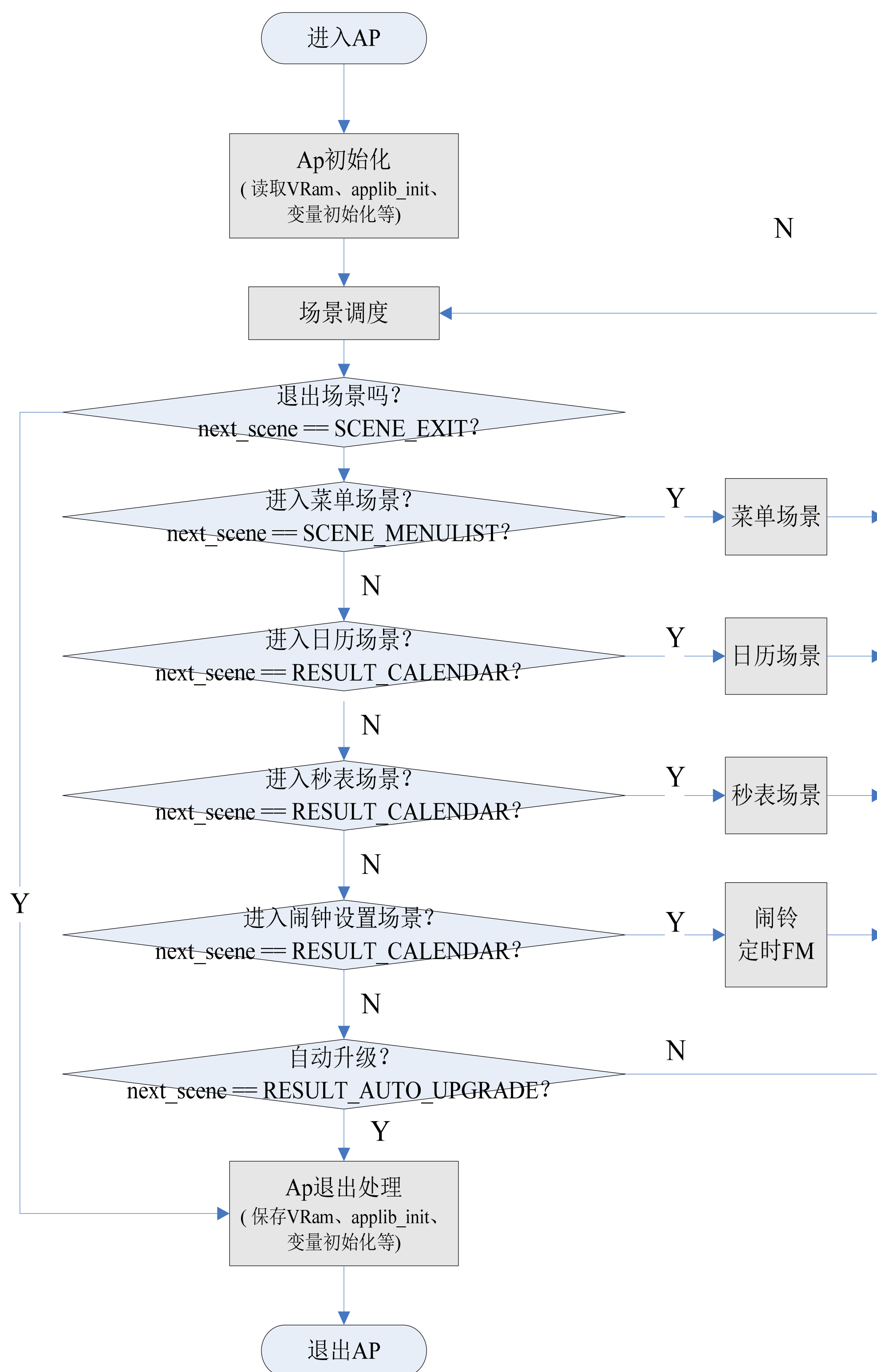
Tools 应用的场景调度流程如下图示：

版 权 所 有 侵 权 必 究



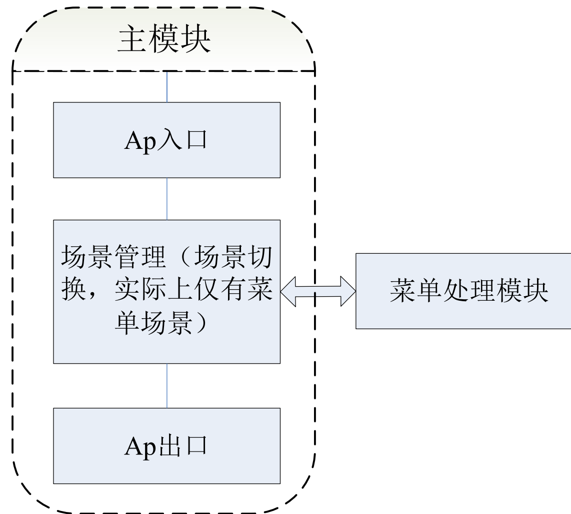
日历、秒表作为独立的场景在 main 中调度；而闹铃和定时 fm 基本上是菜单结构，因此作为 tools_menu 的下级菜单调用。

场景调度流程图如下图示



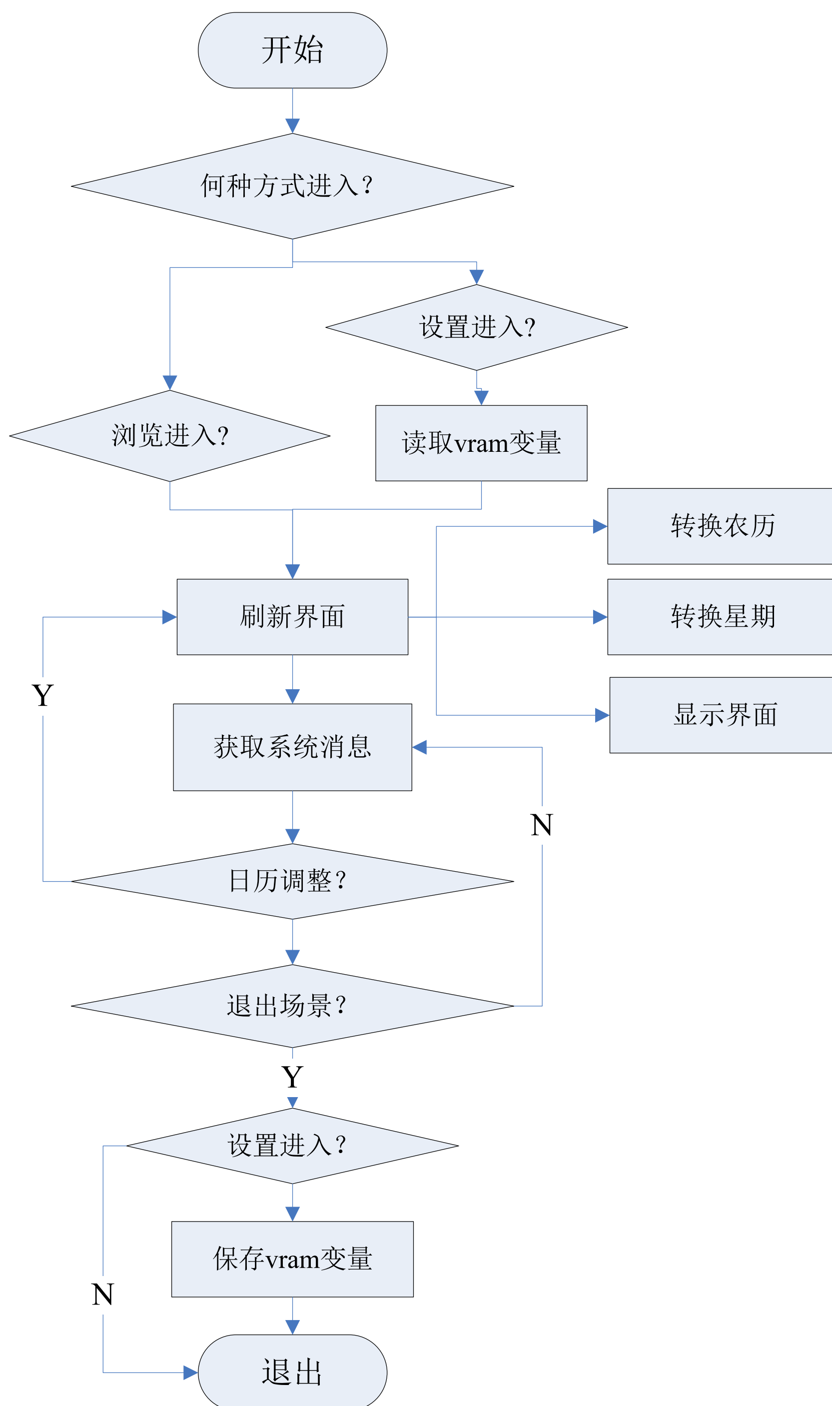
版 权 所 有 侵 权 必 究

21.5.2 工具列表场景流程图

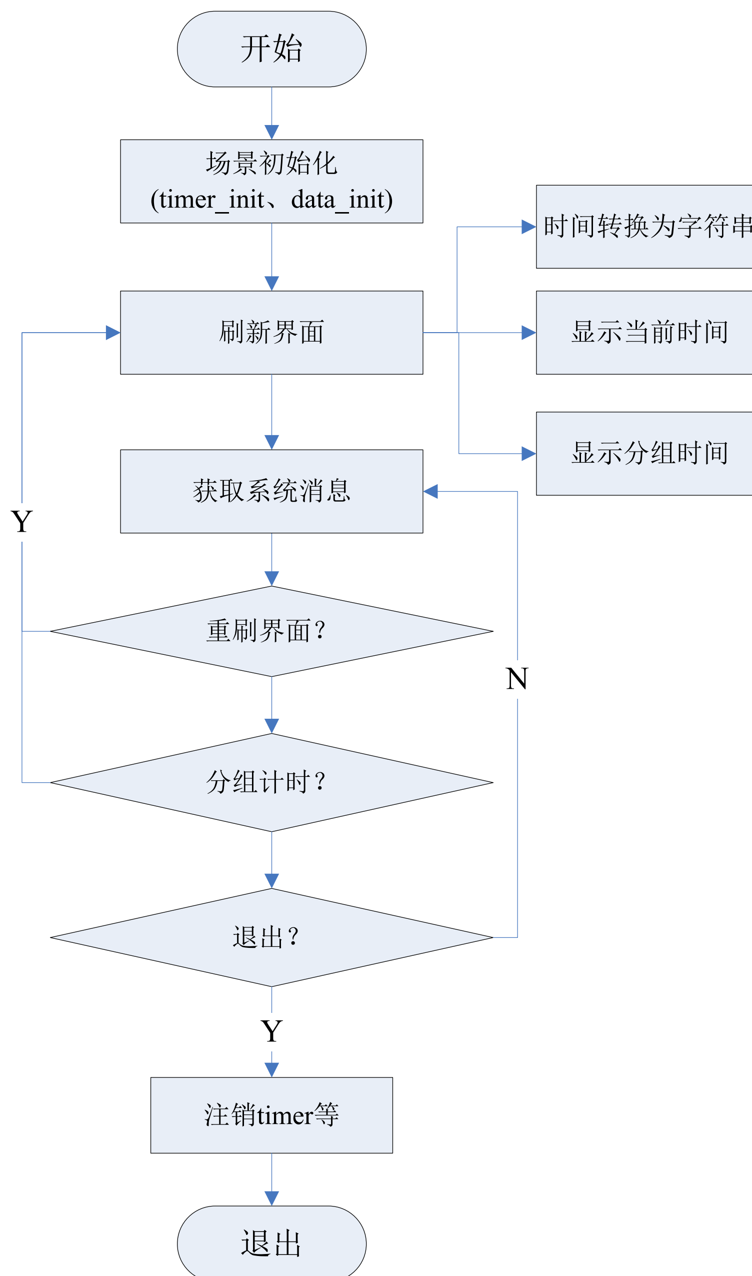


版 权 所 有 侵 权 必 究

21.5.3 日历场景流程图



21.5.4 秒表场景流程图



版 权 所 有 侵 权 必 究

21.6 如何增加一组 alarm

1、 在 alarm_common.h 中增加一组 alarm 结构体，并在 alarm_vars_t 数据结构中添加该项；
2、 添加菜单。在 ap_cfg_menu_tools.c 中，根据新加的菜单为根菜单或者二级菜单更改响应的菜单项；
3、 编写菜单确认键对应的功能函数、实时菜单回调函数、按 option 键后回调函数；
4、 在 alarm_msg_dispatch.c 中，增加新加 alarm 的比较处理
5、 对新加 alarm 的响应处理。

22 ap_alarm 应用

22.1 需求概述

闹钟响应应用。主要对内置闹铃文件或 flash 中音乐文件的的播放及处理。

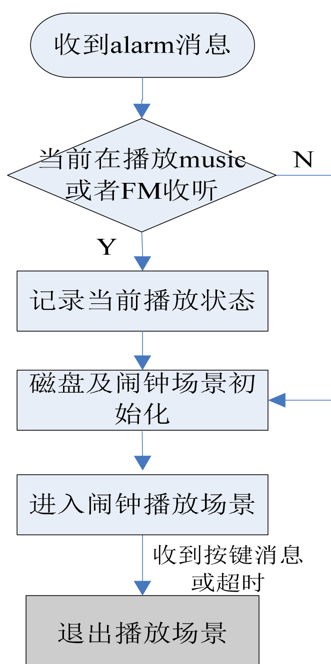
22.2 闹钟的设计原理和启动

Alarm ap 只负责闹钟的播放及处理。闹钟的设置在 Tools ap。闹钟的比较处理及消息发送在 common 控件。

版 权 所 有 侵 权 必 究

22.3 总体架构设计

22.3.1 总体架构图



总体架构图

22.3.2 功能模块划分

参考总体架构图，alarm 应用共分为三个模块：主模块（main）、初始化模块（init）、播放模块（Play）。

模块名称	功能	入口函数	源文件
主模块	Alarm 应用的初始化，退出处理等	Main()	Alarm_main.c
初始化模块	场景初始化、fm 初始化、music engine 的打开等	_scene_play_init()	alarm_playing_init.c
播放模块	播放场景循环、消息及按键处理、engine	playing_loop_deal()	alarm_playing_loopdeal.c

版 权 所 有 侵 权 必 究

	的关闭等		

22.4 与其它应用的同步和交互

Alarm 消息作为系统消息，在其他应用收到时，必须无条件退出。因此与其他应用基本无交互。如果是从 music 及 fm 场景进入，需对 engine 状态等进行记录，以便退出 alarm 时进行断点续播。

22.5 应用依赖库及其接口

系统和 libc 的接口 api.a
应用运行时库 ctor.o
Applib 的全部函数

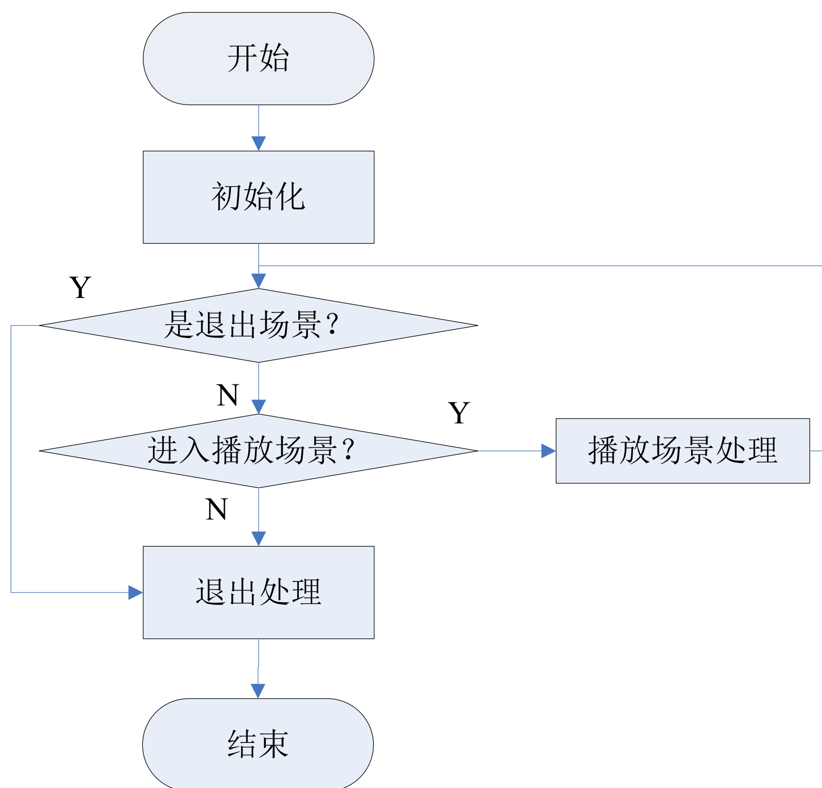
22.6 应用的业务流程

22.6.1 应用的总流程和场景调度流程

Alarm 应用属于前台进程。应用在进入时通过_read_var()读取 VRam 变量，在_app_init()函数中完成 applib 的初始化，软件定时器初始化，消息模块初始化，系统计时器初始化，消息初始化，打开资源文件和菜单配置文件，最后调用文件文件选择器初始化模块加载设备驱动和文件系统驱动。

Alarm 应用退出时会根据得到的返回值和进入应用时的模式判断要返回哪个应用，然后发消息给 manager 进程要求创建相应的应用。然后与初始化顺序相反，在退出时会保存 VRam 变量，调用_app_deinit()完成文件选择器退出，系统定时器注销，菜单配置文件和资源文件关闭，最后执行 applib 库的注销操作并返回。

Alarm 应用的场景调度比较简单，只有一个播放场景。流程图如下图示：



22.6.2 超时退出之后的流程图

22.7 关键的数据结构

（1）保存到 VRAM 的 alarm 变量信息，共分成两个部分，其中一部分为定时闹钟变量的结构体信息，另外一部分为定时 FM 相关变量的结构体信息：

```
typedef struct
{
    /* MAGIC 标志 */
    uint16 magic;
    /* 标志闹钟种类 */
    uint8    timer_flag;
    /* alarm 结构体 */
    alarm_alarm_t g_alarm;
    /* alarm_fm 结构体 */
    alarm_fm_t      g_aufm;
```

版 权 所 有 侵 权 必 究

```
    /* 保存 snooze 时间 */
    time_t   snooze_time;
}alarm_vars_t;

/*ALARM 结构体*/
typedef struct
{
    //uint16 magic;
    /* 闹钟使能标志 */
    uint8   enable;
    /* 闹钟 1 的时间设置 */
    time_t start_time;
    /*闹钟日期的设置标识：0-单次，1-每天，2-按星期*/
    uint8   cycle_flag;
    /*闹钟日期的设置方式*/
    cycle_t cycle;
    /*0-内置 1-flash 2-card*/
    uint8   ring_flag;
    /*响应闹钟时所播放音乐的路径*/
    file_path_info_t path;   /
    /*闹钟音乐播放的音量大小*/
    uint8   volume;
    /*Snooze 标志及次数*/
    uint8   alarm_count;
} alarm_alarm_t;

/*ALARMFM 结构体*/
typedef struct
{
    //uint16 magic;
    /* 闹钟使能标志 */
    uint8   enable;
    /*定时 FM 的开始时间设置*/
    time_t start_time;
    /*定时 FM 的结束时间设置*/
    time_t end_time;
    /*周期循环标识：0-单次，1-每天，2-按星期*/
    uint8   cycle_flag;
    /*闹钟日期的设置方式*/
```

版 权 所 有 侵 权 必 究

```
        cycle_t cycle;
        /*FM 频段*/
        uint16 frequency;
        /*FM 音量*/
        uint8   volume;
        /*录音使能标志*/
        uint8   fmrec_enable;
        //uint8   timer_flag;
    } alarm_fm_t;
```

23 ap_ebook 应用

23.1 需求概述

ebook 用于实现对 TXT 格式的文本文件的浏览阅读，同时具有分页显示电子书内容，设置书签，手动播放以及自动播放等功能

功能需求如下：

1. 支持.txt 格式文件浏览阅读
2. 支持手动和自动播放模式，自动播放时间间隔为 2－30S
3. 支持书签增加，删除功能
4. 支持选择页数进行跳转

性能需求

1. 打开大于 10M 的*.txt 文件时间小于 3 秒
2. 快速定位到书签的位置
3. 流畅浏览阅读电子书

23.2 总体架构设计

23.2.1 总体架构图

按界面划分，ebook 应用的界面包括列表场景，播放场景和设置菜单场景三个主要部分，

版 权 所 有 侵 权 必 究

其中设置菜单场景在不同的场景下有不同的菜单选项。 ebook 应用的结构图如下：

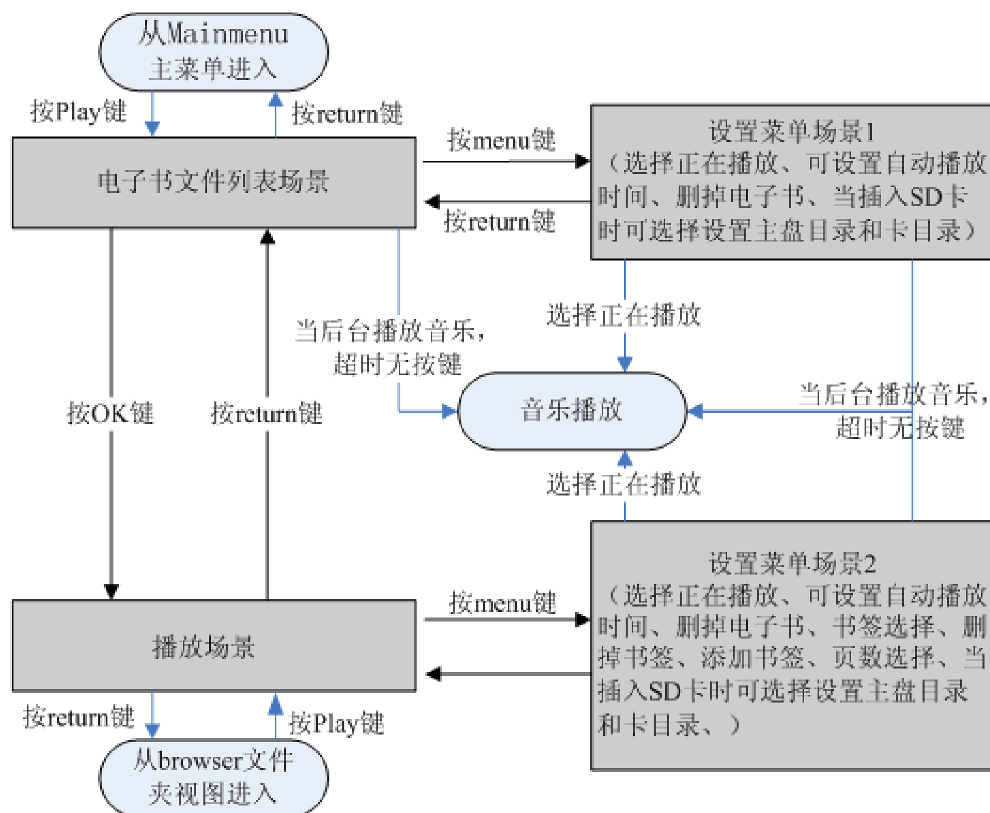


图 23-1ebook 结构体

23.2.2 功能模块划分

按功能划分，ebook 应用的主要模块可分为：主模块，文件列表模块，菜单模块，阅读模块，常驻模块，公用模块。

具体实现功能如下：

表 23-1 功能模块划分表

模块名称	功能简述	对应文件
主模块	主要是实现应用初始化，参数恢复，备份，资源开关 ，场景切换，引擎开关，进程创建或关闭	Ebook_main.c
文件列表模块	进行文件浏览，查看之前的初始化，以及从文件列表退出后对返回结果的处理	Ebook_filelist.c
菜单功能模块	主要是处理菜单选项实现和菜单配置功能	Ebook_menu.c

版 权 所 有 侵 权 必 究

		Ebook_menu2.c Ebook_menu_cfg.c
阅读功能模块	主要实现文件开关，文件信息读取，文本文件的解码，阅读查看电子书内容，书签添加，删除，页数选择等功能	Ebook_fileoperate.c Ebook_fileoperate2.c Ebook_reading.c Ebook_decode.c Ebook_bookmark.c Ebook_bmklist.c Ebook_bmklist_sub.c
常驻代码功能模块	主要是将使用频率比较高的函数存放在常驻空间里，减少 bank 切换，提高效率。	Ebook_residentcode.c
公用函数模块	主要是存放一些应用内公用的函数	Ebook_comfun.c Ebook_message.c

23.3 与其它应用的同步和交互

manager：

当有后台播放音乐时，如果选择切换盘符时，需要先发送关闭后台引擎的消息给 manager,然后再切换盘符。

23.4 应用依赖库及其接口

ebook 应用使用了如下库：

- 系统和 libc 的接口 api.a
- 应用运行时库 ctor.o
- Applib 的全部函数
- Common UI 的菜单，headbar，动画和其他公共提示模块
- Enhanced 库中的文件选择

版 权 所 有 侵 权 必 究

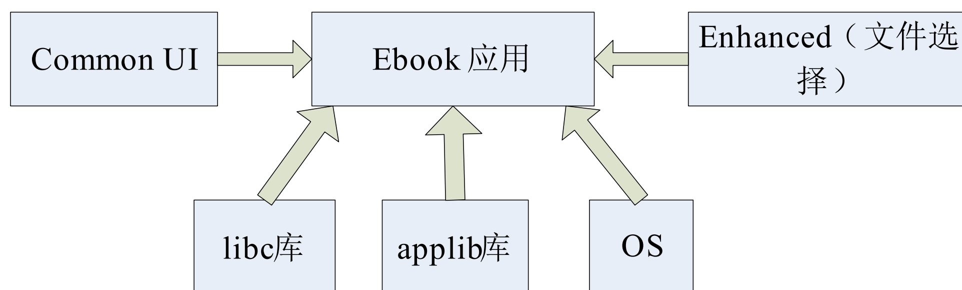


图 23-2 应用依赖关系图

23.5 应用的业务流程

23.5.1 应用的总流程和场景调度流程

main()函数所实现的功能主要是承载应用的启动，应用的退出，场景切换；本模块非常驻代码，对应的模块为 ebook_main.c。

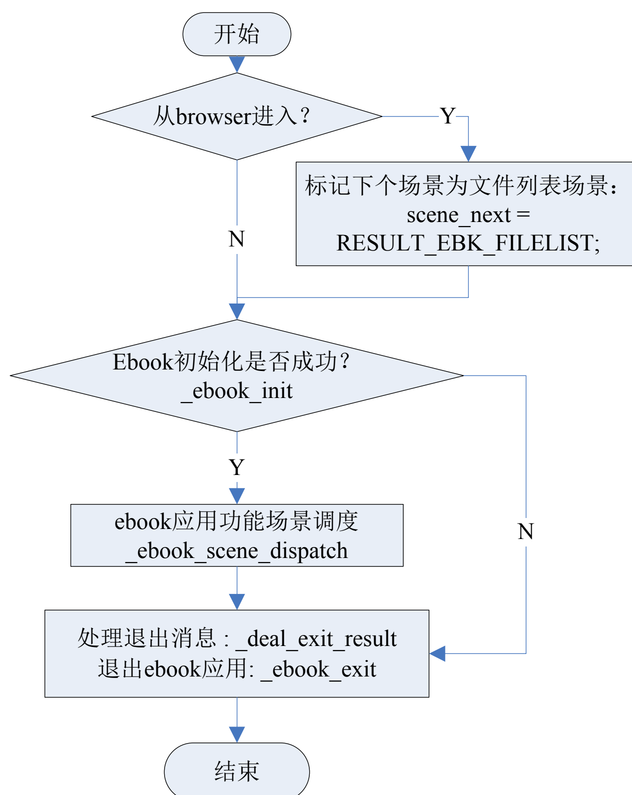


图 23-3Ebook 总流程图

场景调度由 _ebook_scene_dispatch 函数来实现，调度的场景有三个：文本播放场景、文件列表播放场景、菜单选项场景。

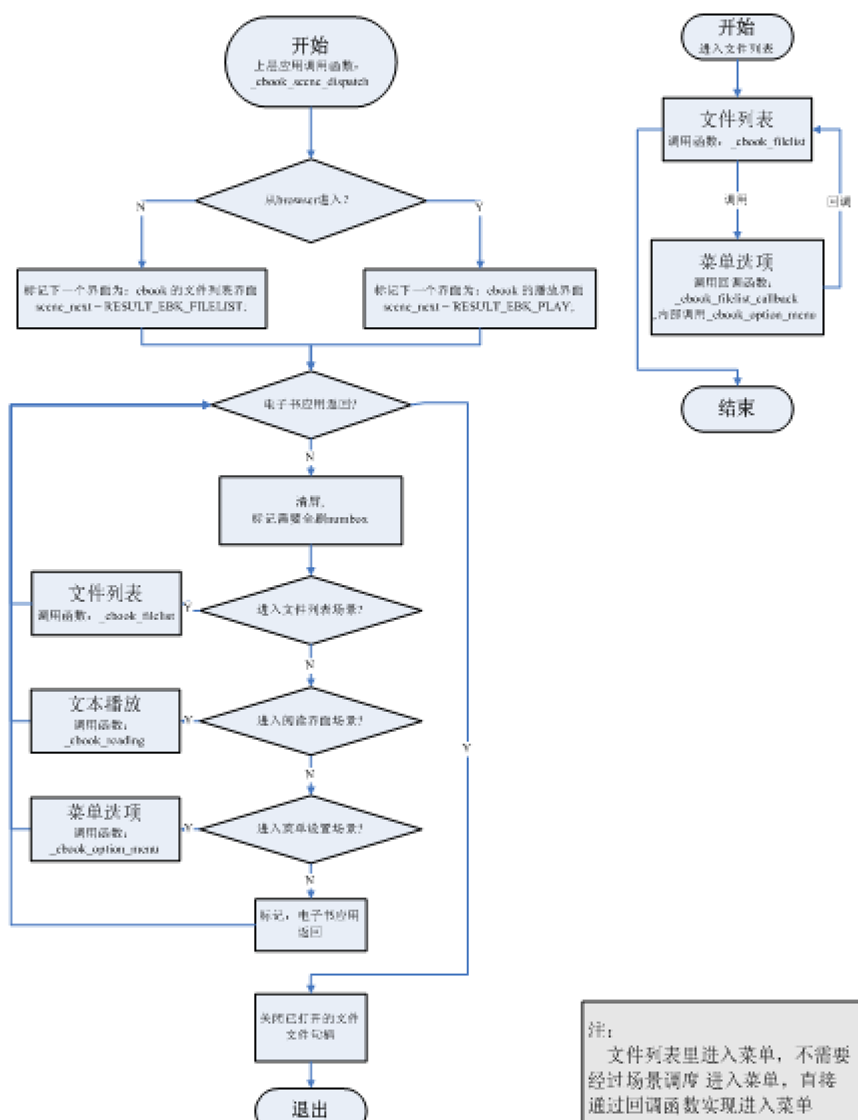


图 23-4 场景调度流程

注意一点，文件列表中进入菜单场景，是不退出的，而是经过回调方式来进入菜单。

23.5.2 文件列表场景流程图

该模块主要是所有*.txt 的文本文件的查看，浏览；对应的模块为 ebook_filelist.c。

该模块是通过调用 common ui 的接口实现文件的浏览，查看，列出该磁盘所有 ebook 应用可执行的文件。

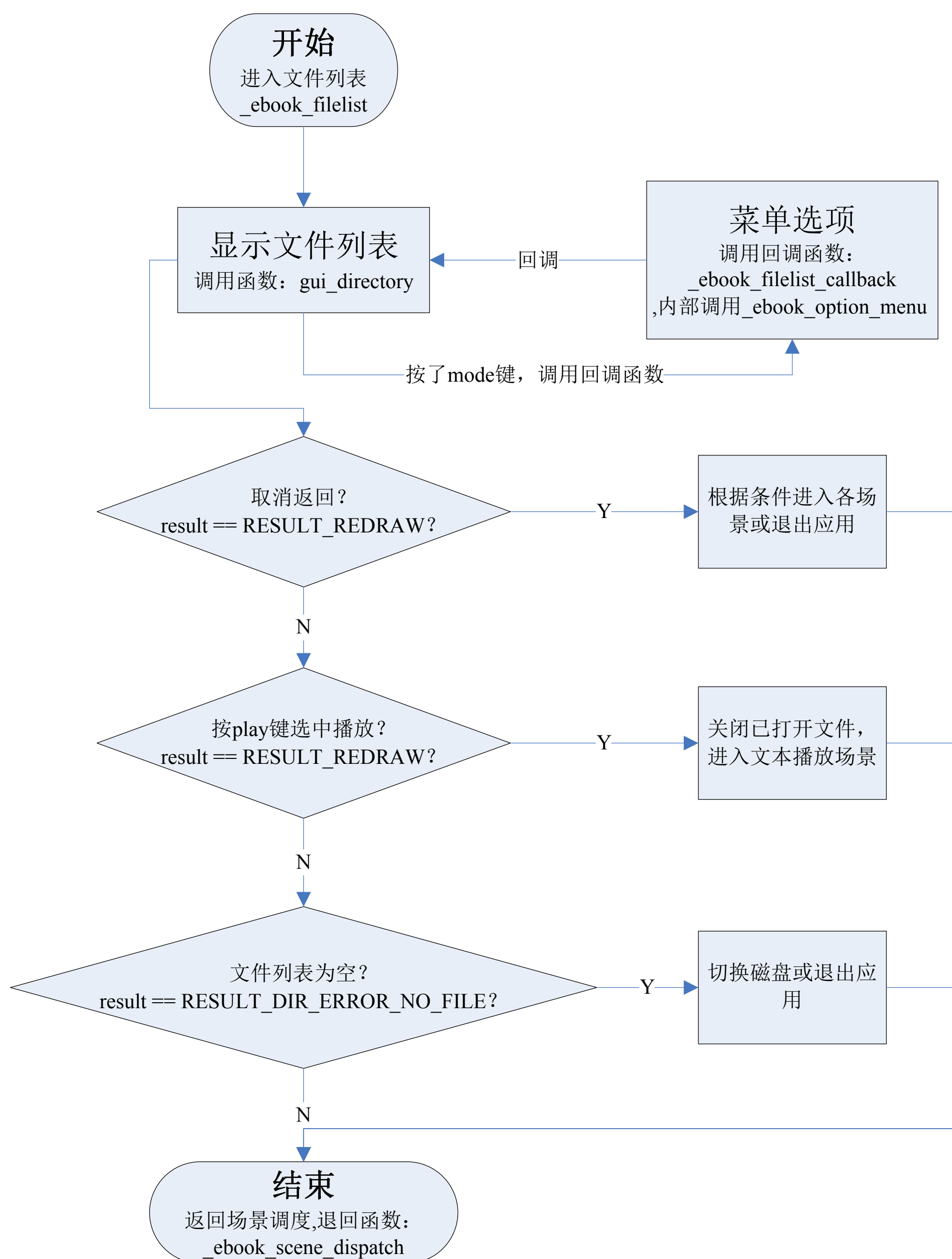


图 **23-5** 文件列表场景流程

23.5.3 阅读场景流程图

该模块主要是实现文本文件和书签文件的打开，关闭；解码文本文件，计算页数信息，将页数信息回写到书签文件中；添加书签，删除书签，选择页数直接定位阅读位置等功能。对应的模块为：ebook_fileoperate.c、ebook_fileoperate2.c、ebook_reading.c、ebook_decode.c、

版 权 所 有 侵 权 必 究

ebook_bookmark.c、ebook_bmklist.c、ebook_bmklist_sub.c。

需要注意的是：由于该模块用到比较多的缓存 buf，主要包括用于页数计算缓存文本内容，大概 560byte；用于显示一屏数据缓存的内容，大概 560byte；用于显示一屏的书签信息，大概 72byte；用于存储书签内容的 buf,大概 256bye; 这么多的缓存空间不可能占用常驻数据空间，所以暂时使用 enhanced bank 段的空间作为数据空间用，而在使用这些缓存 buf 时，是不能调用 enhanced 接口的，否则会将数据冲掉，导致程序运行错乱；所以程序在设计阅读模块时，是不会用到 enhanced 接口的。

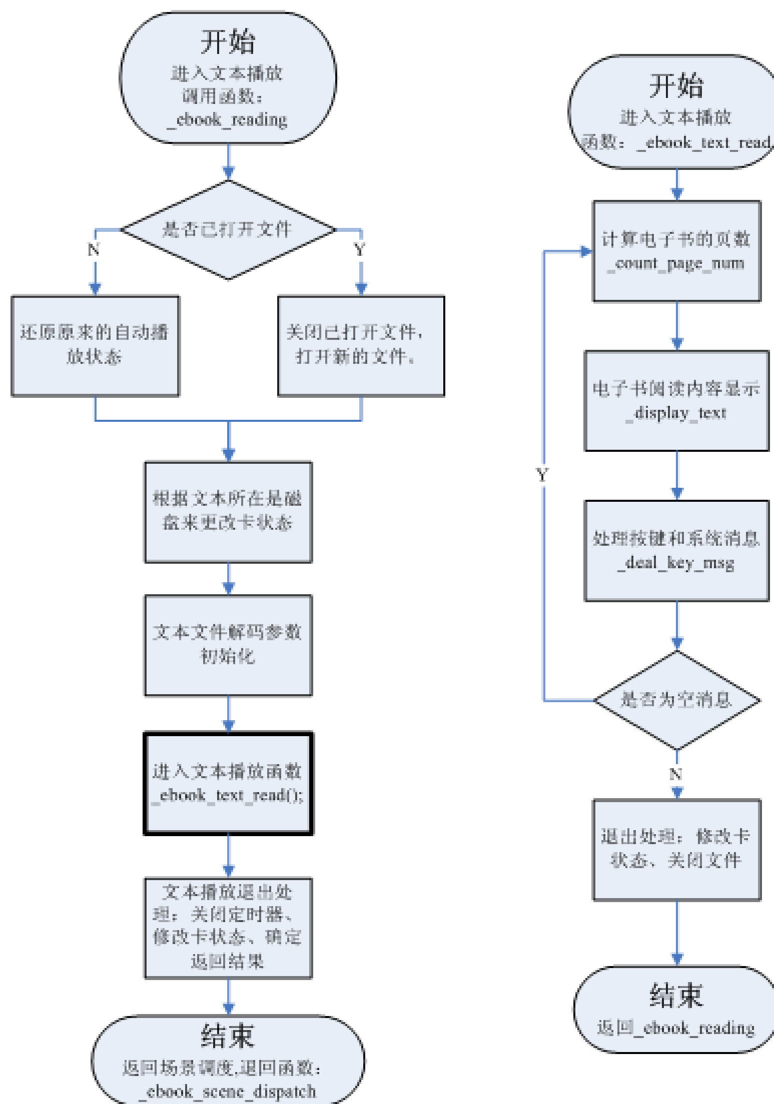


图 23-6 文本播放场景流程图

23.5.4 弹出菜单场景流程图

菜单模块主要的功能是实现动态菜单的显示，处理菜单选项激活以后的返回结果。

Ebook 应用设计了 12 个入口菜单，分别是：

M_NOPLAYNOREAD_NOCARD	无播放和无阅读菜单
M_NOPLAYNOREAD_CARDEXIST	无播放和无阅读菜单(card)
M_NOPLAYREAD_NOCARD	无播放和有阅读菜单
M_NOPLAYREAD_CARDEXIST	无播放和有阅读菜单(card)
M_NOWPLAYNOREAD_NOCARD	正在播放和无阅读菜单
M_NOWPLAYNOREAD_CARDEXIST	正在播放和无阅读菜单(card)
M_NOWPLAYREAD_NOCARD	正在播放和有阅读菜单
M_NOWPLAYREAD_CARDEXIST	正在播放和有阅读菜单(card)
M_LPLAYNOREAD_NOCARD	上次播放和无阅读菜单
M_LPLAYNOREAD_CARDEXIST	上次播放和无阅读菜单(card)
M_LPLAYREAD_NOCARD	上次播放和有阅读菜单
M_LPLAYREAD_CARDEXIST	上次播放和有阅读菜单(card)

从文本阅读进入菜单

在阅读场景下，当识别到按键为 MODE 时，返回结果为 RESULT_EBK_READINGMENU，退回场景调度函数_ebook_scene_dispatch，再执行_ebook_option_menu 进入菜单选项。

版权所有 侵权必究

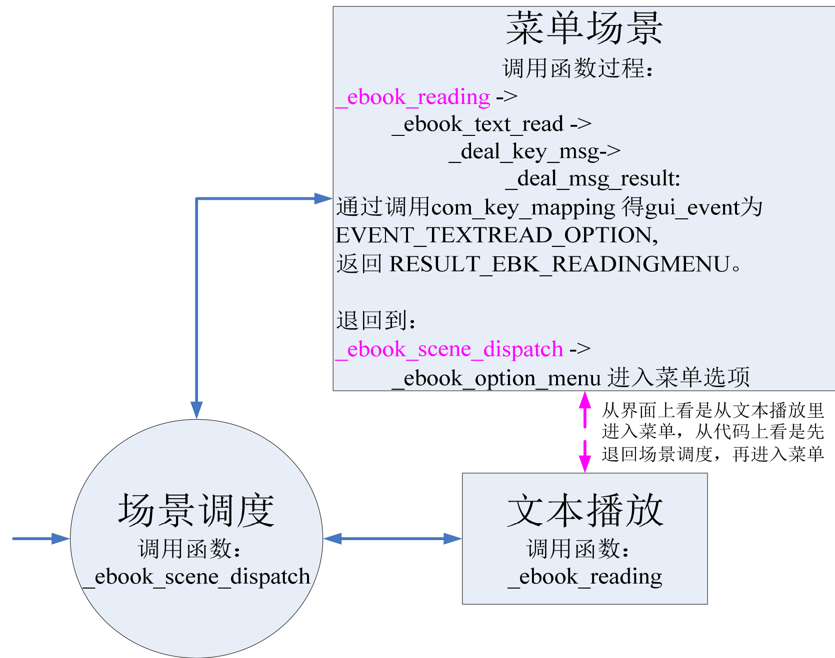


图 23-7 文本播放关键逻辑流程图

从文件列表进入菜单

在文件列表里进入菜单，则是通过回调函数的方式实现，详细请看 23.5.2。

23.6 如何修改阅读场景每页的行和列的数量

在阅读场景里，显示文本是每行每行来显示，每页的行数和每行可显示的宽度在 Ebook.h 里定义：

```
#define ONE_PAGE_ROW_NUM          9          //一屏显示文本的实际行数

/*!
 * \brief
 * ROW_MAX_WIDTH  文本解码一行最大宽度，每行最多显示 128 个像素点宽度
 */
#define ROW_MAX_WIDTH              128
```

考虑到不同的语言文字，每个字符的长度是不确定的，因而每列可显示的字符数是没法统一，只能设置可显示的像素点数目。

版 权 所 有 侵 权 必 究

定义了每页的行数和每行的宽度后，在文本解码参数初始化时赋值给系统变量：

```
/*********************************************************************
 * Description :文本文件解码参数初始化
 *
 * Arguments   :
 *              无
 * Returns     :
 *              无
 * Notes       :
 *
*********************************************************************/
void _init_decode_param(void)
{
    view_decode.valid = FALSE;
    view_decode.remain = 0;
    view_decode.param.line_count_one_page = ONE_PAGE_ROW_NUM;
    view_decode.param.max_width_one_line = ROW_MAX_WIDTH;
    view_decode.param.mode = 0;
    view_decode.text_show_line = _show_text_line;
    view_decode.param.language = view_file.language;
    libc_memcpy(&page_count_decode, &view_decode, sizeof(view_decode_t));
}
```

view_decode.text_show_line，是定义每行的显示函数，从上面代码可看到是调用_show_text_line 函数来显示每一行的文字。

显示每一页内容时，调用如下函数：

```
_decode_one_page(&view_decode, ……, ……);    //解码文本，并显示出来
```

版权所有 侵权必究

炬力集成电路设计有限公司

地址: 珠海市高新区科技创新海岸科技四路 **1** 号
电话：**+86-756-3392353**
传真：**+86-756-3392251**
邮政编码：**519085**
网址：**http://www.actions-semi.com**
电子邮件 （业务）: **mp-sales@actions-semi.com**
（技术支持）: **mp-cs@actions-semi.com**

版 权 所 有 侵 权 必 究