移动开发系列

MEDIATEK 使用联发科芯片任意品牌手机适用

爱莫塔库论坛出品

走出山寨

MTK芯片开发指南

Revolution of Mobile Production

手机制造业的革命

陈智鹏 编著

第一本MTK芯片中文开发指南

以创意、架构、设计、实用为主线

图文结合，配合实例说明

真实案例，为您揭示MTK平台开发的核心技术

巴掌大小的手机背后隐藏着一条

极有可能代表中国制造业希望的产业链。

人民邮电出版社
POSTS & TELECOM PRESS

移动开发系列

Android 开发入门指南

书号：21394-5
定价：45元

Android SDK 开发范例大全

书号：20893-4
定价：79元

Android 开发入门与实战

书号：20930-6
定价：55元

AppStore掘金 —iPhone SDK 应用程序开发

书号：21823-0
估价：45元

移动开发系列

MEDIATEK 使用联发科芯片任意品牌手机通用

MTKoo 爱莫塔库论坛出品

走出山寨

MTK芯片开发指南

陈智鹏 编著

人民邮电出版社

北京

图书在版编目（CIP）数据

走出山寨：MTK芯片开发指南 / 陈智鹏编著. — 北京：人民邮电出版社，2010.1
ISBN 978-7-115-21668-7

Ⅰ. ①走… Ⅱ. ①陈… Ⅲ. ①移动电话机—芯片—开发—指南 Ⅳ. ①TN929.53-62

中国版本图书馆CIP数据核字(2009)第206621号

内 容 提 要

MTK 平台是联发科公司推出的一种新型的手机设计解决方案，到目前为止，它已经成为了国产手机生成厂商首选的平台，其高度的模块集成特性、快捷的出货速度和低廉的价格优势催生了中国大陆数量庞大的手机研发公司，它创造了这个时代最响亮的名词："山寨"，并引领了从 2006 年以来山寨机的潮流。

MTK 平台的特殊性使得初次接触的开发人员上手困难，而相关资料的缺乏让广大 MTK 爱好者也无法完整和准确地研究该平台。本书全面地介绍了 MTK 应用开发的基础知识，它不仅向读者阐述了 MTK 上层应用开发的方方面面，而且还阐述了底层的驱动原理和具体设计。本书旨在帮助读者在较短的时间里对 MTK 平台开发获得全面深刻的理解与认识，从而使读者更好地把握 MTK 平台开发的特点和重点，为读者以后的手机设计生涯打下坚实的基础。另外，本书的所有实例都经过实际的测试，跟随本书完成所有的实例能帮助读者在掌握理论的同时，获得更多的实际经验。

本书适合作为广大专科、本科院校，编程培训中心等的学习教材和辅导教材，也适合作为广大移动开发编程爱好者的自学参考资料。

走出山寨：MTK 芯片开发指南

◆ 编　　著　陈智鹏
　责任编辑　屈艳莲
　执行编辑　蒋　佳

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
　邮编　100061　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
　印张：20.75
　字数：499 千字　　2010 年 1 月第 1 版
　印数：1 – 4 000 册　　2010 年 1 月北京第 1 次印刷

ISBN 978-7-115-21668-7

定价：49.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

前 言

本书编写目的

当我们谈论山寨机的时候，总会提到 3 个字母："MTK"。以联发科公司名字命名的手机设计解决方案在短短的几年时间，凭借着其自身的特性，成就了无数的手机设计公司和光怪陆离的产品，采用 MTK 平台的手机占领了国内市场绝大部分的市场份额，创造了这个时代最响亮的一个名词："山寨机"。

但是，由于 MTK 平台的特殊性，使得大多数开发人员不能很快地上手。而且 MTK 爱好者也无法通过有效的途径来学习这一新兴的平台。能不能有一本启蒙性的图书来指导初学者跨过 MTK 应用开发的门槛呢?

以上介绍的问题是我编写本书的源动力。本书将 MTK 的应用开发分为若干章节，在每一部分都配有大量的实例，读者可以通过这些实例的实现过程来逐步明晰整个开发过程的点点滴滴。本书的目的是为了给广大读者对 MTK 开发有启蒙性的作用，因此我们将重点放在了如何完成每一个模块上面，同时在宏观上让读者掌握全局的框架和设计思路。

编写这本书的时候，MTK 又往后跨越了几个版本，新的功能不断涌现。但是，就应用开发特别是上层的 GUI 设计来说，各版本之间的代码变化不大。因此，我选择 MT6226 06B 这一经典版本进行讲解。同时，我始终认为，学习一个平台，应该重点放在其最本质和基础的部分，只有掌握该平台的核心才能在以后的设计工作中游刃有余。

本书的主要内容

本书共分为 16 章。

· 第 1 章主要讲述 MTK 的历史以及目前国内手机市场的特点。

· 第 2 章阐述了 MTK 的整体软件架构，并提前介绍了 MMI 的结构和手机启动的过程。

· 第 3 章详细介绍了 MTK 平台的编译过程，对每一步骤都作了详细地阐述。

· 第 4 章通过一个精练的实例，为读者讲解了 MMI 开发过程中的重要内容，例如菜单添加、各种资源的载入、事件的响应等，并对可能出现的问题和异常作了详细地解释。

· 第 5 章对 MTK 图形系统作了梳理，对 GUI、GDI 等容易混淆的知识点作了详细地说明。

· 第 6 章和第 7 章讲解了 MTK 平台上的基本绘图方法，高级的图层原理和控制。对于开发中常用到的图层知识，使用了详细的例子来帮助读者理解。

· 第 8 章重点讲解了 Task 的原理，并用详细的步骤阐述了如何创建和使用一个 Task。

· 第 9 章对 MTK 平台的文件操作作了归纳和分析。对常用的文件操作函数依次作出解释，并设计了一个具体的实例来辅助读者理解和掌握。

· 第 10～12 章阐述了更高级的 GUI 开发知识。阐述了控件的绘制原理，屏幕模板的设计以及高级屏幕模板的使用，并且通过了一个实际的例子讲述如何使用这些特性来绘制一个绚丽的屏幕界面。

· 第 13 章讲解了如何在 MTK 平台上实现多国语言，包括字库的设计以及在不同软件版本条件下的实现步骤。

· 第 14 章通过 MTK 平台软件方案中自带的 DEMO 例子，详细地讲解了 MTK 通过 SOCKET 来访问网络的过程。对每一步骤都作了详尽的解释，并在结尾实现了一个精练的 SOCKET 访问例子来帮助读者掌握。

· 第 15 章集前面所有章节的主要内容，为读者讲解了如何从一个原始版本高仿诺基亚的 N73 智能机界面。采用了逐步讲解逐步实现的方法，集中了我多年的实际经验，对高仿过程中的修改作了详细解释。

· 第 16 章着重分析了 MTK 平台底层的驱动开发原理。对每一个重要的部分都采用了大量的配图和实例来辅助讲解，并标明了每一个重点代码段所处的位置。

鸣谢

最后，我要感谢所有提供过帮助的那些人，没有他们的帮助与付出，是很难这么顺利地完成本书的。

· 首先，出版社的编辑蒋佳老师为本书的修订和出版做了大量的工作，他给本书提出了大量宝贵的意见，与他的合作是非常高效和愉快的！

· 书中，第 16 章 MTK 驱动开发由我的朋友杨鹏编写。感谢他的辛苦付出！

· 感谢我的女朋友关建华女士为书中大量的图片所做的设计修改工作！

· 感谢众多网友发布自己的经验总结以及提供的 MTK 资料，使得本书能够获得宝贵的参考资料。

尽管我在本书的写作过程中非常认真与努力，但由于编写水平有限，书中难免存在错误和不足之处，恳请广大读者批评指正。如果您对本书有什么意见、问题或想法，欢迎您通过邮件与我联系，将不胜感激。

E-mail：zolo1983@sohu.com

作　者

2009 年 10 月

目 录

第 1 章 MTK 的横空出世 …… 1

1.1 引子 …… 1

1.2 MTK 简介 …… 1

1.3 MTK 平台的生命力 …… 2

1.4 山寨来了 …… 3

1.5 走出山寨的强者 …… 5

1.6 来自 3G 的机遇和挑战 …… 6

第 2 章 MTK 软件方案 …… 9

2.1 MTK 软件方案组成 …… 9

2.2 MTK 总体软件方案架构 …… 10

2.2.1 Nucleus 实时操作系统 …… 11

2.2.2 L1 协议栈 …… 11

2.2.3 设备驱动 …… 12

2.2.4 L2/L3/L4 层协议栈 …… 13

2.2.5 MMI …… 14

2.3 MMI 架构 …… 14

2.3.1 Task …… 15

2.3.2 MMI 与 L4 通信 …… 15

2.3.3 框架层（Framework Layer）…… 16

2.3.4 应用层（Application Layer）…… 18

2.3.5 UI 层 …… 18

2.3.6 MMI 开发常用目录 …… 19

2.4 启动流程 …… 20

第 3 章 MTK 工程的编译 …… 22

3.1 编译环境的搭建 …… 23

3.1.1 工具安装与配置 …… 23

3.1.2 进行开发环境检查 …… 25

3.2 如何进行编译 …… 26

3.2.1 包含 Modis 仿真环境的工程 …… 27

3.2.2 不包含 Modis 仿真环境的工程 …… 27

3.2.3 生成文件及 Log 日志文件 …… 28

3.3 软件系统的目录结构 …… 28

3.4 构建原理 …… 29

3.5 配置编译选项 …… 35

3.5.1 包含源代码的第三方库 …… 36

3.5.2 无源代码的第三方库 …… 36

3.6 烧写版本 …… 37

第 4 章 MMI 开发起步 …… 40

4.1 MMI 初探 …… 40

4.1.1 MMI 概念 …… 40

4.1.2 MTK- MMI 模块概览 …… 42

4.2 如何浏览代码 …… 44

4.3 Hello MTK …… 47

4.4 屏幕起步 …… 49

4.4.1 什么是屏幕 …… 49

4.4.2 如何创建新的屏幕 …… 50

4.4.3 History 机制 …… 51

4.5 规范化 …… 55

4.5.1 确定目录结构及模块相关文件 …… 55

4.5.2 将相关文件路径加入 Plutommi 的编译配置文件 …… 56

4.5.3 为新增模块增加编译开关 …… 56

4.6 资源 …… 58

4.6.1 字符串 …… 61

4.6.2 菜单……64
4.6.3 图片……69
4.7 SDK 的制作和使用……70
4.7.1 生成真机版本的 SDK……71
4.7.2 生成模拟器版本的 SDK……72
4.7.3 使用真机版本 SDK……72
4.7.4 使用模拟器版本 SDK……73

第 5 章 MTK 的图形系统……75

5.1 概述……75
5.2 图形系统结构……75
5.2.1 模板子系统……76
5.2.2 绘画子系统……76
5.2.3 设备子系统……77

第 6 章 基本绘图……81

6.1 文本绘制……81
6.1.1 基本函数……81
6.1.2 字体……83
6.1.3 颜色……84
6.1.4 排版……85
6.2 图形绘制……86
6.3 图像……90
6.3.1 静态图像……90
6.3.2 动画……91
6.3.3 图像排版……93
6.4 区域裁剪……93
6.5 背景绘制……95
6.5.1 填充结构体……95
6.5.2 纯色背景……97
6.5.3 渐进色背景……98
6.5.4 纹理背景……100
6.5.5 图片背景……101
6.5.6 动画背景……101
6.5.7 系统默认背景……103

第 7 章 图层……104

7.1 主要作用……105
7.2 创建新层……105
7.3 激活新层……106
7.4 基础层……107
7.5 合并……108
7.6 通透……110
7.7 图层剪切……111
7.8 透明效果……112
7.9 释放图层……113
7.10 分配更多的内存……114
7.11 锁层……117
7.12 GDI 锁……119

第 8 章 Task……121

8.1 创建 Task……121
8.1.1 增加索引和 ID……122
8.1.2 增加映射……122
8.1.3 配置 Task……123
8.1.4 实现……124
8.2 Task 的消息循环过程……125
8.3 使用 Task……126
8.3.1 消息结构……126
8.3.2 填充消息结构体……126
8.3.3 投递到消息队列……127

第 9 章 文件操作……129

9.1 常用函数……130
9.1.1 FS_Open……130
9.1.2 FS_Close……131
9.1.3 FS_Read……132
9.1.4 FS_Write……132
9.1.5 FS_Seek……133
9.1.6 FS_Commit……133
9.1.7 FS_GetFileInfo……134

9.1.8 FS_GetAttributes ····· 135
9.1.9 FS_FindFirst ····· 135
9.1.10 FS_FindNext ····· 136
9.1.11 FS_FindClose ····· 136
9.1.12 FS_CreateDir ····· 137
9.1.13 FS_GetDrive ····· 137
9.2 文件读写 ····· 139

第 10 章 控件 ····· 146

10.1 GUI 基本控件 ····· 146
10.1.1 按钮（Button） ····· 147
10.1.2 文本按钮（TextButton） ····· 148
10.1.3 图标按钮（IconButton） ····· 149
10.1.4 单行输入框 ····· 150
10.1.5 提示框 ····· 151
10.1.6 文本图标列表控件 ····· 151
10.2 WGUI 高级控件 ····· 162
10.2.1 文本图标列表 ····· 163
10.2.2 动态列表控件 ····· 164
10.2.3 左右按键 ····· 165

第 11 章 屏幕模板 ····· 166

11.1 系统提供的屏幕模板 ····· 166
11.2 自定义屏幕模板 ····· 167
11.2.1 定义新的屏幕模板函数 ····· 167
11.2.2 提取变化的元素 ····· 168
11.2.3 提取重绘函数 ····· 171
11.2.4 模板历史管理 ····· 172
11.2.5 退出模板 ····· 174

第 12 章 高级屏幕模板 ····· 176

12.1 高级模板的绘制流程 ····· 178
12.2 模板 ID ····· 180
12.2.1 模板 ID 的传递 ····· 180
12.2.2 模板 ID 和窗体组件、组件的属性的关联和交互 ····· 181
12.3 模板组件 ····· 182
12.3.1 构成模板的组件定义 ····· 182
12.3.2 各组件的属性定义 ····· 183
12.4 模板组件类型 ····· 184
12.4.1 控制类型控件 ····· 184
12.4.2 标准显示类控件 ····· 184
12.4.3 扩展显示类控件 ····· 185
12.5 自定义模板 ····· 186
12.5.1 模板数据库 ····· 186
12.5.2 普通模板加入 DM ····· 187
12.5.3 自绘制控件 ····· 189
12.5.4 更加漂亮的界面 ····· 189

第 13 章 多国语言 ····· 196

13.1 添加新的语言 ····· 196
13.1.1 前期准备 ····· 196
13.1.2 MTK 支持该语言 ····· 196
13.1.3 MTK 不支持该语言 ····· 197
13.1.4 自己实现输入法的总结 ····· 198
13.2 移植多国语言 ····· 198
13.2.1 语言宏控的修改 ····· 199
13.2.2 字库文件的添加 ····· 199
13.2.3 修改默认显示语言 ····· 199
13.2.4 移植 Zi 输入法 ····· 199
13.2.5 移植手写汉王（hanwang）输入法 ····· 201
13.2.6 打开 MTK 自带输入法支持 ····· 202
13.3 虚拟键盘 ····· 202
13.4 输入法提示符的显示 ····· 202

第 14 章 SOCKET ····· 203

14.1 SOC 定义浅析 ····· 203
14.1.1 SOCKET 宏常量 ····· 204
14.1.2 套接字类型 ····· 204

14.1.3 套接字事件 …… 205
14.1.4 封装的套接字函数 …… 205
14.2 SOC 实例分析 …… 206
14.2.1 进入工程模式 …… 206
14.2.2 输入 URL 地址 …… 206
14.2.3 Socket 初始化 …… 210
14.2.4 请求准备工作 …… 211
14.2.5 发送请求 …… 212
14.2.6 连接 …… 214
14.2.7 等待服务器响应 …… 215
14.2.8 处理消息事件 …… 216
14.2.9 回调函数执行 …… 216
14.2.10 输出结果 …… 216
14.3 HTTP 连接方式编程总结 …… 217
14.3.1 创建 Socket 连接 …… 217
14.3.2 设置 Socket 非阻塞/异步模式 …… 217
14.3.3 链接准备 …… 217
14.3.4 连接 …… 218
14.3.5 发送 HTTP 请求 …… 218
14.3.6 接收数据 …… 218
14.3.7 关闭 Socket 连接 …… 218
14.3.8 启动/关闭定时器 …… 218
14.3.9 注册事件回调函数 …… 218
14.4 精简的代码 …… 219
14.5 WAP 浏览器 …… 221

第 15 章 高仿 NOKIA N73 …… 223

15.1 认识 N73 界面 …… 223
15.2 待机界面分析 …… 224
15.3 基本布局 …… 225
15.3.1 定义宏开关 …… 225
15.3.2 打开左右按键 …… 226
15.3.3 上移快捷菜单 …… 226
15.4 状态栏修改 …… 227
15.4.1 电量和信号状态图标 …… 227
15.4.2 修改状态栏重绘区域 …… 228
15.4.3 移动小图标绘制坐标 …… 229
15.4.4 时间显示修改 …… 230
15.4.5 状态栏刷新 …… 235
15.4.6 运营商名称 …… 237
15.4.7 修改快捷菜单 …… 238
15.4.8 注册快捷菜单事件 …… 240
15.4.9 添加快捷菜单安全区 …… 241
15.5 动态列表 …… 242
15.5.1 添加动态列表控件 …… 242
15.5.2 今天无日历项 …… 243
15.5.3 列表初始化 …… 244
15.5.4 音乐标题显示 …… 248
15.5.5 按键切换 …… 249
15.5.6 动态列表触摸事件 …… 255
15.6 主菜单初步 …… 258

第 16 章 MTK 驱动开发 …… 262

16.1 基本架构 …… 262
16.2 MTK 手机启动过程 …… 268
16.3 EMI 配置 …… 272
16.3.1 EMI 概述 …… 272
16.3.2 EMI 时间设置 …… 272
16.3.3 EMI 自动配置过程 …… 275
16.3.4 EMI 存储区域划分 …… 277
16.4 驱动定制工具的使用 …… 280
16.4.1 驱动定制工具简介 …… 280
16.4.2 工具使用说明 …… 281
16.4.3 GPIO/GPO 设置 …… 283
16.4.4 EINT 设置 …… 286
16.4.5 ADC 设置 …… 289
16.4.6 Keypad 设置 …… 292
16.5 设备默认值的设置 …… 294
16.5.1 PWM 设置 …… 294

16.5.2 LCD 对比度设置 …… 296
16.5.3 电池电量等级设置 …… 297
16.5.4 声音模块设置 …… 298
16.6 可插入设备的配置 …… 302
16.6.1 充电参数设置 …… 302
16.6.2 辅助设备检测设置 …… 304
16.6.3 USB 设置 …… 305
16.7 其他设备的配置 …… 307
16.7.1 MMC 设置 …… 307
16.7.2 触摸屏设置 …… 308
16.7.3 LCD 屏幕移植 …… 310
16.7.4 摄像头移植 …… 320

1 MTK 的横空出世

1.1 引子

可以这么说，看这本书的读者所买的第一部手机 80%都是国外品牌，应该都集中在诺基亚、摩托罗拉、三星、西门子等这些品牌上。

似乎从 2003 年以后，手机就逐渐成为了人们的生活必需品，拥有一部手机再也不是一种奢侈。但中国庞大的市场几乎全被外国厂商垄断，他们凭借强大的资本和技术优势，不断加快新产品的推出速度并进行渠道改革，产品已覆盖了高、中、低端的消费市场。

摩托罗拉当年的 V3 系列，在中国消费者中引起何等的震动；诺基亚凭借其 S60 系统的成功，一跃从一个边缘手机厂商成长为中国市场的老大；还有西门子、三星这些企业，也在手机研发上加大投入，引领手机技术的不断创新。

而在这些璀璨的星辰里，国产手机厂商显得如此暗淡。大家也许发现了，这个领域中的引领者都是跨国大公司，他们有资金、有技术、有大量的研发人员。企业雄厚的背景，使得他们可以用几年的时间来研发一款手机，达到慢工出细活的目的。而中国的手机公司很多才刚刚完成原始积累，或许才转行该领域不久，想涉及该领域，那些垄断巨头们会不会排挤你暂且不论，即便是给你充分的外在条件让你发挥，你也很难做到。因为很多的核心技术都在他们手中，考虑到相关的配件设备迥异，手机板上各种芯片的研发集成难度，短时间内你是很难完成整机设计和生产的。所以，时间就能把一个国内手机公司拖垮。在那个年代，国内很多企业都打过这方面的主意，比如曾经名声响亮的波导，曾经赞助英超的科健，现在都难以觅其踪影。

但是中国市场如此庞大，这块大蛋糕岂能让外国人独自分享？

时势造英雄，在 2000 年，台湾一家名为联发科（MTK）的公司进入了手机芯片设计领域。而联发科公司的进入，引发了随后手机产业中的世界大战！

1.2 MTK 简介

联发科技股份有限公司（MEDIATEK），简称为 MTK，创立于 1997 年 5 月 28 日，是世界顶尖的 IC 专业设计公司，位居全球消费性 IC 片组的领航地位。产品领域覆盖数码消费、数字电视、光储存、无

线通信等很多系列，是亚洲惟一连续 6 年蝉联全球前 10 大 IC 设计公司的华人企业，被美国《福布斯》杂志评为“亚洲企业 50 强”。联发科总部如图 1.1 所示。

图 1.1 联发科总部

2002 年后，联发科的手机芯片部门正式运营，当时市场上主流的芯片公司是高通、德州仪器、爱立信、飞利浦等，它们是诺基亚、摩托罗拉等一流公司的主要供应商，联发科是惟一的中国公司。

2004 年，联发科正式进入内地手机市场。其提供的“Turn-Key”一站式解决方案竞争力极强，并凭此获得内地市场近 50%的基带芯片份额，并拥有成熟的 GSM/GPRS/EDGE 方案。

2007 年 9 月 10 日，联发科与 ADI 签署协议，以现金约 3.5 亿美元取得 ADI 旗下 Othello 和 SoftFone 手机芯片产品线相关的有形及无形资产以及团队。通过此项交易案，MTK 的无线通信部门获得一支近 400 位具有丰富产品开发及客户服务经验的专业团队；扩大全球各地的客户群；增加新的手机基带和射频芯片产品包括 GSM、GPRS、EDGE、WCDMA 和 TD-SCDMA 芯片以丰富其现有的产品组合。

1.3 MTK 平台的生命力

MTK 为什么会流行，是什么带给他如此旺盛的生命力？

1. MTK 的商业模式（Turn-Key）

在 MTK 的（Turn-Key）手机解决方案中，将手机芯片和手机软件平台预先整合到一起。这种方案可以使终端厂商节约成本，加速产品上市周期。MTK 公司的产品因为集成较多的多媒体功能和较低的价格在大陆手机公司和手机设计公司中得到广泛的应用。（Turn-Key）直译过来就是“交钥匙”，言下之意，MTK 已经将这辆“车”的钥匙都给你了，你还能要求什么呢？这种商业模式最大限度地降低了开发商的开发周期。

2. MTK 的完工率较高

MTK 的完工率基本在 60%以上，其参考设计都已通过 CTA 测试，保证了系统的稳定性。这样手机厂商拿到手机平台基本上就是一个半成品，大大减少了手机的生产环节并降低了研发成本，只要稍微加工一下就可上架出货了。这个特性也就解决了 1.1 节中提到的国产手机公司遇到的难题。

3. 价格优势

MTK 现在的出货量很大，所以价格很透明，相比之前业界巨头垄断时候的方案价格，具有巨大的优势。

正是因为有着如此多的而且适合中国国情的特性，MTK 早在 2006 年就已经占有了手机芯片供应市场几乎一半的份额，并且让中国手机产业的发展在随后的几年里犹如超新星的爆发。2008 年基于 MTK 平台的手机市场占有率如图 1.2 所示。

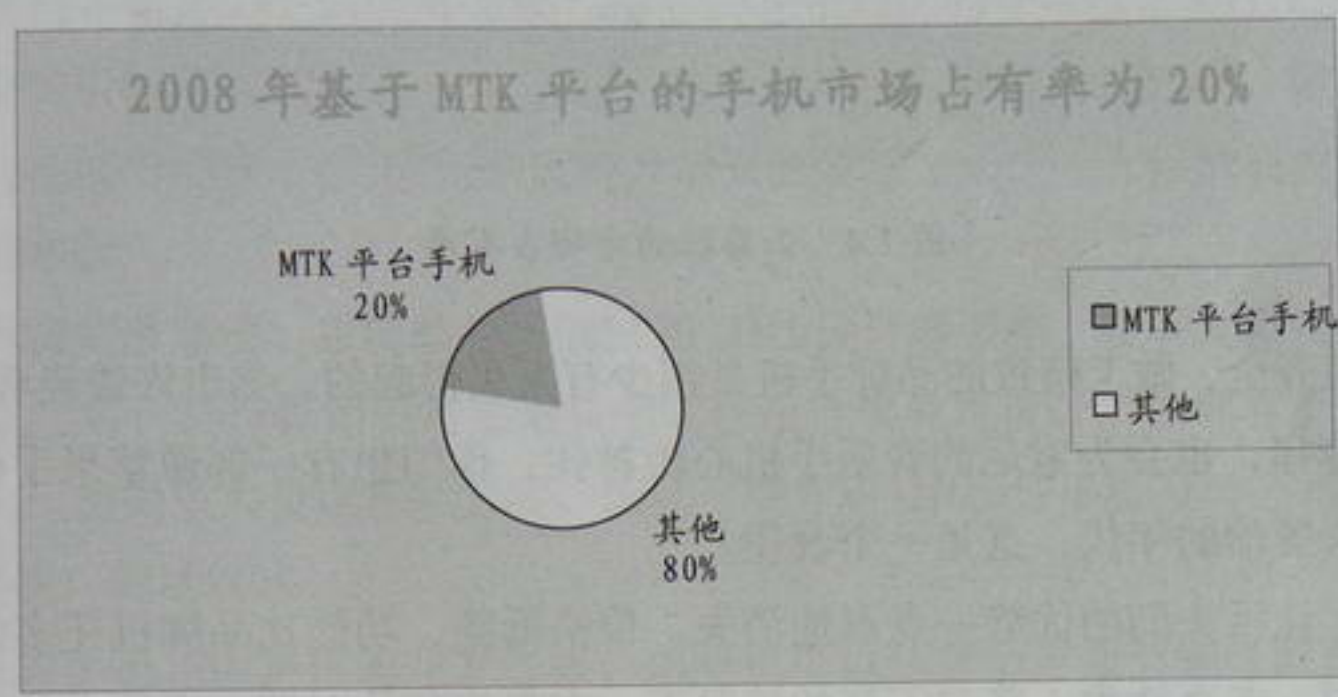


图 1.2 基于 MTK 平台的手机市场占有率

1.4 山寨来了

MTK 的出现，让渴望进入手机制造领域的企业眼前一亮，其出货速度快的特点被这些生产厂商充分地利用。在深圳，很多小公司只要招几名结构设计人员、硬件工程师和软件人员，对 MTK 方案稍加修改，就可以迅速地制造出一部功能齐全的手机。其实就是拼装手机，而拼装这些手机的厂商既不是地下加工厂，又不是正规军，所以它们的产品得了个别名叫“山寨机”。

讲得动听一点：山寨机就是一种初级创新产品，这类型的手机没有获得政府许可或没有获得入网许可而在国内大量地生产，其 3 种表现形式如图 1.3 所示。

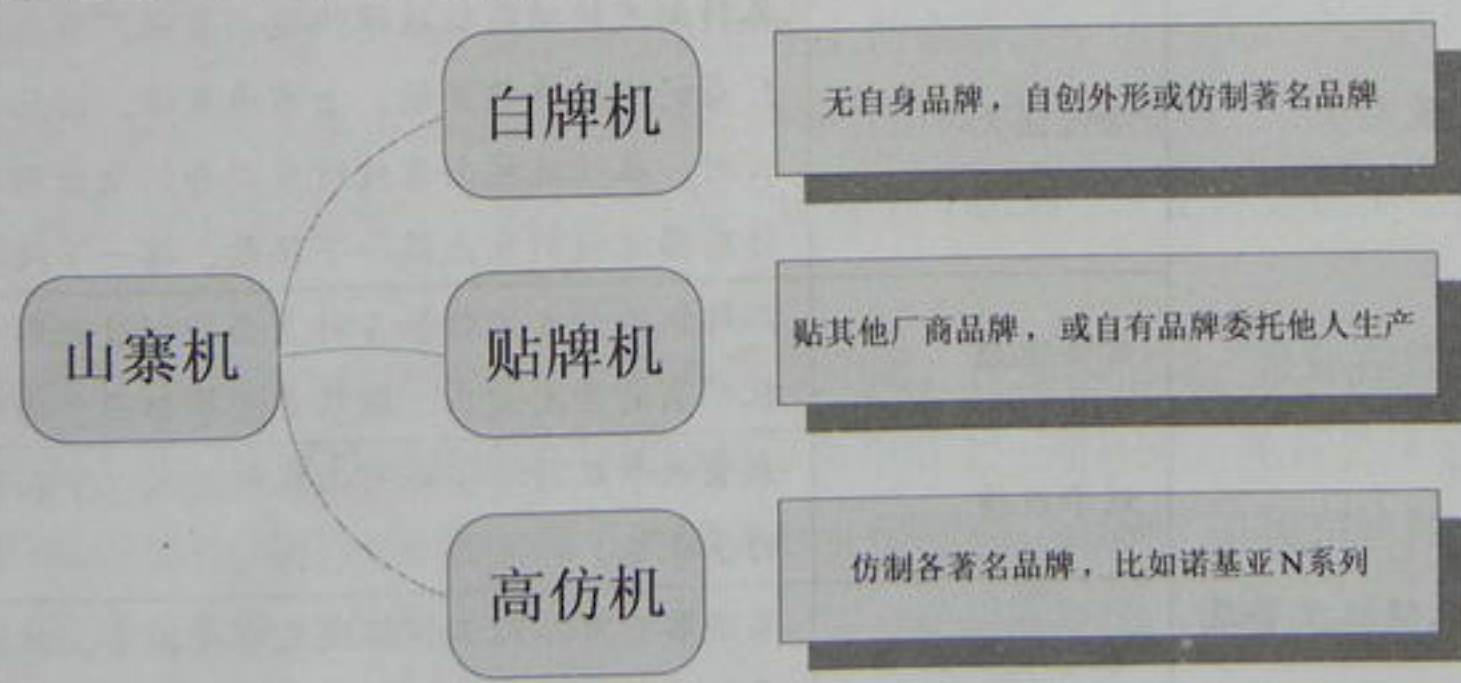


图 1.3 山寨机分类

仅仅从数量上讲，在 2008 年，山寨机就几乎占据了整个中国手机市场的半壁江山，如图 1.4 所示。

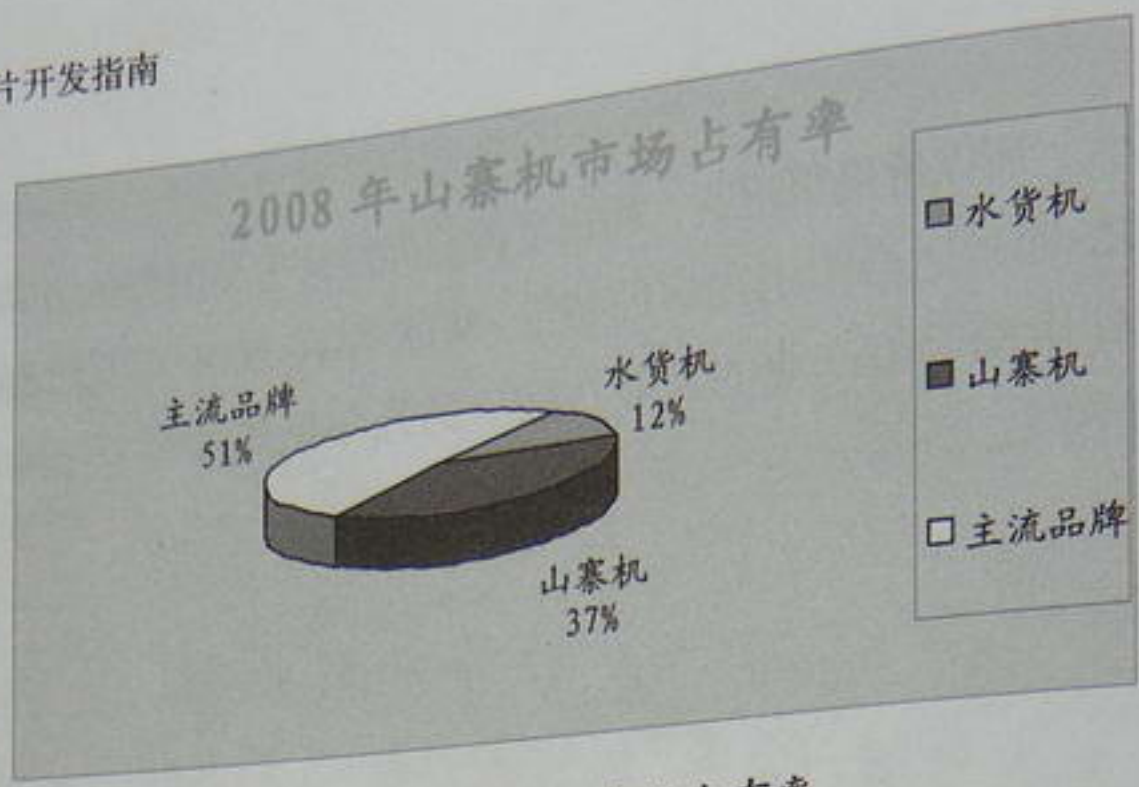


图 1.4　山寨机的市场占有率

过去，一些功能齐全、做工精致的品牌手机是很少有人买得起的。你也许曾经看到摩托罗拉 E680i 大大的屏幕而心里痒痒，也会为索尼的音乐手机心驰神往，你幻想有一部像苹果手机一样可以触摸的手机，但在那个巨头统治的年代，这是一个奢望。

山寨机的出现，让巨头们的优势一点点地消失。价格低廉，功能比品牌机还多的产品，为什么不买呢？特别是在中国，老百姓收入普遍不高的情况下。在最疯狂的时期，市面上的山寨机几乎将“山寨”发挥到极致。在深圳，你随处可以看到像卖菜一样的小贩向你推销各种新奇古怪的手机。比如有手表造型的手机，具有验钞功能的手机，香烟盒形状的手机，宝马、奔驰、宝时捷等车形状的手机，甚至还有一不留神就以为是诺基亚、三星等大牌的 NOKILA、Samsang 手机。这种高仿手机，有的名字与国外的名牌极其相似，界面和外观设计毫无二致。

山寨机独有的特性如表 1.1 所示。

表 1.1　山寨机独有特性

山寨机特征		说　明
品牌无法做到的特性	独特外观	独特外观的机型往往针对细分市场，量小，要求反应速度快，品牌机厂商很难生产
	功能强大	各种新式的功能往往很快被山寨机厂商采用。由于山寨机厂商的市场感觉灵敏，上市速度快，因此在该功能刚一问世时，往往就被山寨机所采用并衍生出新功能，比如加速传感器就被衍生出摇一下换歌、摇一下换墙纸等多种功能
	价格低廉	品牌机的开发流程和各种上面分析过的因素，导致了品牌机厂商的开发成本、销售费用等都高于山寨机
品牌机不愿意在低端机上普及的功能	双卡双待	未来双卡双待有望成为山寨机进入 3G 和智能手机领域时的杀手锏
	超大铃声	某些客户群体因工作环境或自身因素，往往喜欢大音量的手机
	大触摸屏	提供对视频、游戏等功能的完美支持
	手写	这可以说是山寨机将其发扬光大的功能

2008 年，当山寨产品在手机领域摧城拔寨，用各种各样无所不及的方式吸引着人们的眼球之时，人们已经意识到，一种新的山寨文化出现了。

山寨机所独有的快速出货销售模式，使其在市场上颇具竞争力。一款山寨机从设计、开模、采购、组装到出货只需 45 天。而正规厂商仅入网检测这一项就需要一个半月，出货最快也要在两个月以上。

联发科的做法缩短了产业链，直接压缩了大型设计公司的空间。将芯片、软件平台以及第三方应用软件捆绑，整套卖给手机厂商，这能够帮助手机厂商在很短的时间内做出可上市的产品。极端情况下，手机厂商只需要购买联发科方案的 PCB 套片，再购买屏幕、摄像头、外壳、键盘等简单零部件就可以出品手机。业内极端的形容是："应用联发科方案只需要 3 个人，一人接洽联发科，一人找代工工厂，一人负责销售和收款"。

原来设计公司负责完成驱动、系统整合、调试和 MMI 设计等工作，在联发科提供包括硬件和软件在内的全面解决方案之后，设计公司只需要做应用软件设计和外观设计。深圳有上千家中小型手机设计公司，能够给手机厂商提供大量成熟的方案，因此"联发科+中小型手机设计公司"的组合，已经足够满足 2G 时代山寨机厂商的需求。

比较一下 2G 品牌手机和山寨机的开发过程，山寨机可以从 3 个方面节省时间和费用。

（1）山寨机厂商紧贴市场，反应速度快，从方案设计到最终确定的速度快。

（2）深圳的手机产业链已经达到了极高效率，比许多手机企业的运作速度还要快。正所谓"船小好掉头"。

（3）山寨机不做入网检测，在"检测/返工"方面花费的时间也很少。

这几点为山寨机带来的好处是加快了上市速度、降低了成本，弊端则是质量无保证（部分山寨机厂商采用低价劣质元器件生产，导致质量问题的增多）。在中国销售的正规手机，必须要通过工信部属下泰尔实验室独家垄断的手机入网检测，每款手机在这个环节要花费 20 万～23 万元、1～3 个月的时间进行检测。在手机已经变成快速消费品的今天，能否提前 1～3 个月上市将决定一款手机的销量。

这一杀手锏，使很多国内有牌照的手机公司都遭遇了不小的打击，也给曾经的手机老大摩托罗拉的伤口上撒了一把盐，加速了摩托罗拉在中国手机市场上的衰落。

3 年来中国无数厂商杀入手机市场，并以高额返利等营销模式大量侵蚀二、三级城市市场。据有关方面的数据显示，2007 年，以国内这一模式出货的"山寨机"已达 1.5 亿部，还不包括出口到国外的产品。

山寨产品以极低的成本模仿主流品牌产品的外观或功能，并加以创新，最终在外观、功能、价格等方面全面超越这个产品。山寨产品在抄袭与超越的羊肠小道上一路狂奔，尤其是挣脱了牌照的束缚，握紧了低成本高回报的福祉之后，彻底颠覆了传统的行业潜规则，山寨手机深深地打上了草根创新、群众智慧的烙印，是当之无愧的中国式山寨。

1.5 走出山寨的强者

随着进入这一领域的公司越来越多，再按照之前那种简单修改和拼装模式生产的手机，已经不能

在激烈的市场竞争中站稳脚跟了。

更为重要的是，长久地照搬 MTK 方案，不针对手机核心部分进行创新和超越，长此以往，必将进入一个死胡同，最终的结果就是直接摧毁中国手机研发的核心竞争力。

近年来，中国的国产手机中多了两个响亮的名字，金立和天语。作为山寨手机研发的先驱者和引领者，这两家公司也是从很低的阶段起步的，但他们有着更长远的目标，他们注重自身品牌的打造。虽然用的方案仍然是 MTK，但在基础上做了大量的创新，在小厂商忽视的结构设计上费足了心思，设计更加人性化。

比如天语推出的超薄手机，800 万像素的手机甚至领先于诺基亚这些巨头更早地投放市场。如图 1.5 所示为天语 C280 手机。

图 1.5　天语 C280

为了提高电池的使用效率，中国的手机厂商还制造了更加耐用的电池，金立等公司推出的超强待机功能的手机，如图 1.6 所示。

这些国产手机厂商都是做大厂商没有做到的，想大厂商没有想到的，认清自身的优势和劣势，大打差异牌，这是值得消费者认真考虑的地方。他们的产品在界面友好性以及显示效果上大大超出了同级的老资格的国产品牌机，甚至和国外品牌相比，设计上更适合中国国情，这一点也为这些厂商添足了筹码。

在这些公司的带领下，国内的手机公司开始意识到产品创新和掌握核心技术的重要性，逐步放弃了之前粗犷型的仿制阶段，开始加大投入，对其产品功能进行大胆的扩展，开发出更具有竞争力的手机产品。虽然这些产品在一些方面上还很落后，新增的功能还很稚嫩，但我们可以看到希望，鼓励国内厂商做大做强，完成真正的技术积累，来抗衡国外品牌的冲击。

与此同时，国外一些手机厂商也开始重视 MTK 解决方案。著名的日本公司夏普也发布了一款采用 MTK 方案的手机 SH9130C，如图 1.7 所示。

图 1.6　金立 L6

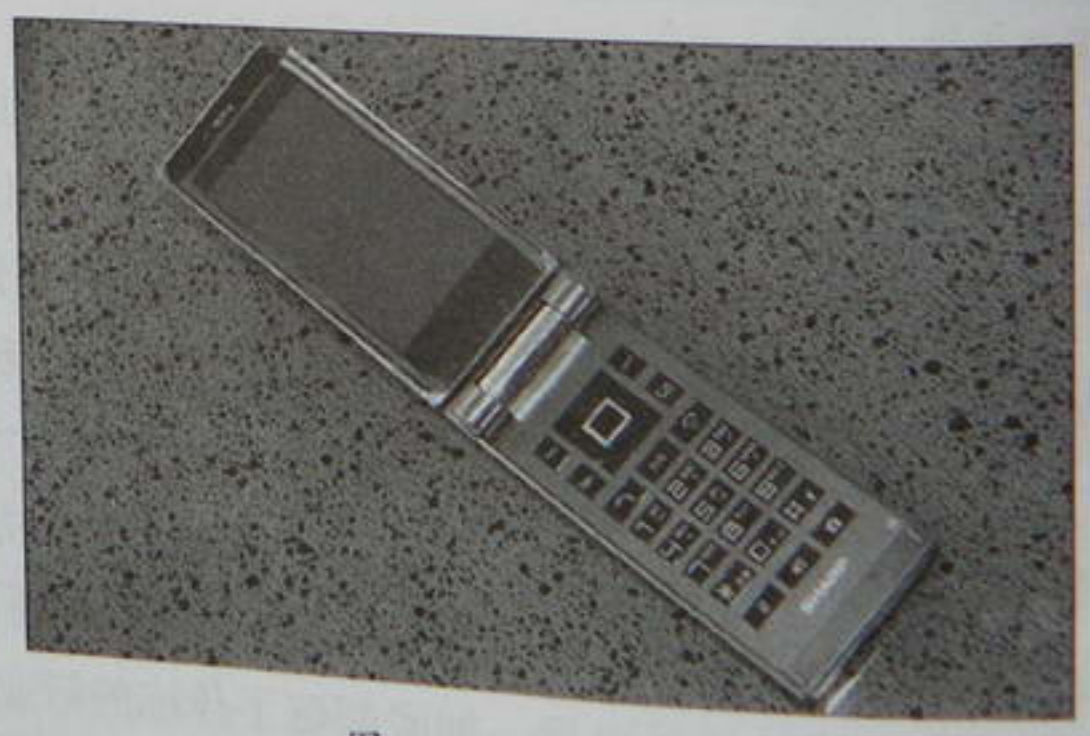


图 1.7　夏普 SH9130C

1.6　来自 3G 的机遇和挑战

对 MTK 来说，在市场不断扩大的同时，其单芯片集成解决方案也不断被其他厂商效仿，竞争不断

加剧，所以其向 3G 市场过渡已成为必然。

2009 年以前，因受限于芯片，所以山寨机厂商主要做 2G 的普通手机，更确切地说，是 GSM 的普通手机。未来预计山寨机将进入智能手机和 3G 手机领域，特别是其中的 WCDMA 手机领域。2009 年联发科推出了智能机芯片平台 MT5616，并宣称在 2009 年继续发布 WCDMA 芯片。

山寨机以及许多希望进入智能手机领域的二线品牌厂商的步骤也将和 MTK 一致，先进入 2G 智能手机领域，再进入 3G 智能手机和 3G 普通手机领域。从目前的行业动态来看，在市面上首先出现的山寨智能手机也是 2G 智能手机。对山寨机未来市场的预测如表 1.2 所示。

表 1.2　山寨机未来市场的预测

山寨机未来进入细分市场预测		
类型	2G	3G
普通手机	2G 普通手机 （山寨机已经进入）	2G 普通手机 （预计是山寨机的第 3 目标）
智能手机	2G 智能手机 （预计是山寨机的第 1 目标）	2G 智能手机 （预计是山寨机的第 2 目标）

前面提到 MTK 收购了美国 ADI 手机芯片产品线，就是为了进入 TD-SCDMA 市场。在 2008 年 2 月中国移动的首期 TD-SCDMA 终端招标过程中，MTK 通过 ADI 在招标产品中已占据逾 50%的芯片解决方案份额。

在 2008 年 10 月 21 日举行中国国际信息通信展览会上，MTK 展示出了他们的 TD-SCDMA 解决方案和采用这些方案的将近十款 TD 手机。与此同时，在中国移动和 TD-SCDMA 联盟的展台上，我们也看到了联发科解决方案以及终端的展台。

MTK 如果真正能够和 TD 绑在一起，这将是中国移动 3G 发展过程中很重要的步骤，中国移动在低端市场上不用再为终端而发愁。MTK 和中国移动的合作，也使其 MTK 方案能够平稳过渡到 3G 时代。由于方案提供者能够顺利过渡，众多 MTK 的跟随者又有了新的目标，他们将跟随 MTK 进入下一轮手机换代的革命中，老百姓也可以用上更便宜的 3G 手机了。在留住客户的同时，又开拓了新的利润增长点，MTK 将会在 3G 时代取得空前的发展。

芯片一体化可以给山寨智能手机的开发过程带来效率提升，由于智能手机整体开发时间较长，因此能节约的时间会更多。根据厂商资料，智能手机很可能将重现 2G 山寨机的开发模式，3～4 人，2～3 个月，就可开发出一款手机。但这还是一个未知数，前提是 MTK 能不能在 3G 网络中实现其一体化芯片的模式。如果一体化芯片最终实现，那么山寨机的开发又将领先品牌机，并冲击整个手机市场。

但作为一家手机芯片及软件方案提供商，其提供的平台仅属于低端层次，在和主流操作系统平台的横向比较中处于下风，无法像智能机一样进行任意扩展，这是 MTK 的一个硬伤，也是中国移动仅将其作为低端市场平台的原因。但是 MTK 显然意识到这个问题，MTK 定会走向与微软公司的 Windows Mobile 合作的道路，后者可能会为 MTK 平台定制简化版的 Mobile 操作系统。在 Symbian 和 Android 获得越来越多处理器厂商认同的同时，Windows Mobile 的队伍却越来越小。如果微软公司能与 MTK 合作，将会使前者获得重生的一个大好机会，而后者也可解决迈向智能手机的难题。目前，联发科已经正式

获得微软公司智能手机操作系统 Windows Mobile 的授权，向高端智能领域发起了冲击。但最大的困难还是：在 3G 网络中，MTK 很难实现其赖以成名的 Turn-Key 模式。

Turn-Key 模式在 GSM 上是成功的，因为 GSM 非常成熟，而在 3G 领域，包括 TD-SCDMA 在内，由于其本身还是新兴市场，具有复杂性和不成熟性，因此要实现 Turn-Key 模式还很难。TD 产业至少还要发展 2～3 年的时间，才有可能出现 Turn-Key 模式。到那时，很可能就不是 MTK 或展讯率先引入这一模式了，而是其他公司。因此，MTK 能不能在 3G 时代将 Turn-Key 模式的优势发挥出来，能不能实现 2G 时代的辉煌，将是 MTK 最大的挑战！

结语：

MTK 在中国内地的成功折射出国内企业在手机研发领域的薄弱，大陆的公司没有技术积累，只能够通过购买成型的方案来进行仿制或修改。而山寨机的空前繁荣，也可以隐约看到其中不乏创新精神，联发科不可能成为中国手机产业的救命稻草，只有加强投入，特别是对研发的投入，掌握核心技术，不断创新，开发出适合自己、适合本国消费者的产品，既有时尚的外形，又有一流的功能，价格又很便宜，这样国产手机才会有长足的发展。

一味地模仿和粗制滥造，不注重品牌经营将会让中国本来就不发达的手机产业进入又一个冬天。

2 MTK 软件方案

2.1 MTK 软件方案组成

在上一章我们已经提到，MTK 成功的秘诀在于其销售的不仅仅是芯片，而是包括软件平台、第三方应用软件在内的完整解决方案。因此，我们有必要对其软件方案做一个大致的梳理。如图 2.1 所示是 MTK 软件方案架构图。

图 2.1　MTK 软件方案架构图

从图 2.1 我们可以看到：MTK 提交给客户的定制版本中，已经集成了人机界面 MMI、GSM/GPRS 协议层，实现了 E-mail 客户端、短信、SIM 卡应用。这样的软件版本几乎是一个半成品了，客户可以在其原有基础上进行扩展。MTK 还为客户集成了第三方软件，比如 Java、中文输入、Wap 等。惟一需要强调的是，MTK 所集成的第三方软件，客户必须获得其原开发商的授权。在图 2.1 中，圆角矩形中的软件都需要获得原开发商的授权。

2.2 MTK 总体软件方案架构

在 MTK 提供的原始文档中，其架构图庞大而琐细，很多部分不属于本书的关注范围，因此，图 2.2 所示是一个精简的软件架构图。读者从图中可以看到整个架构实际上分为 5 个核心部分，分别是 MMI、L2/L3 协议栈、Drivers 设备驱动、WAP 应用和 L1 层协议栈。与此同时，MTK 还额外提供了开发辅助工具 Catcher 和 META。Catcher 用于真机调试时的抓包工具；META 是 MTK 手机参数调试工具，提供手机的射频测试、NVRAM 访问测试等功能。

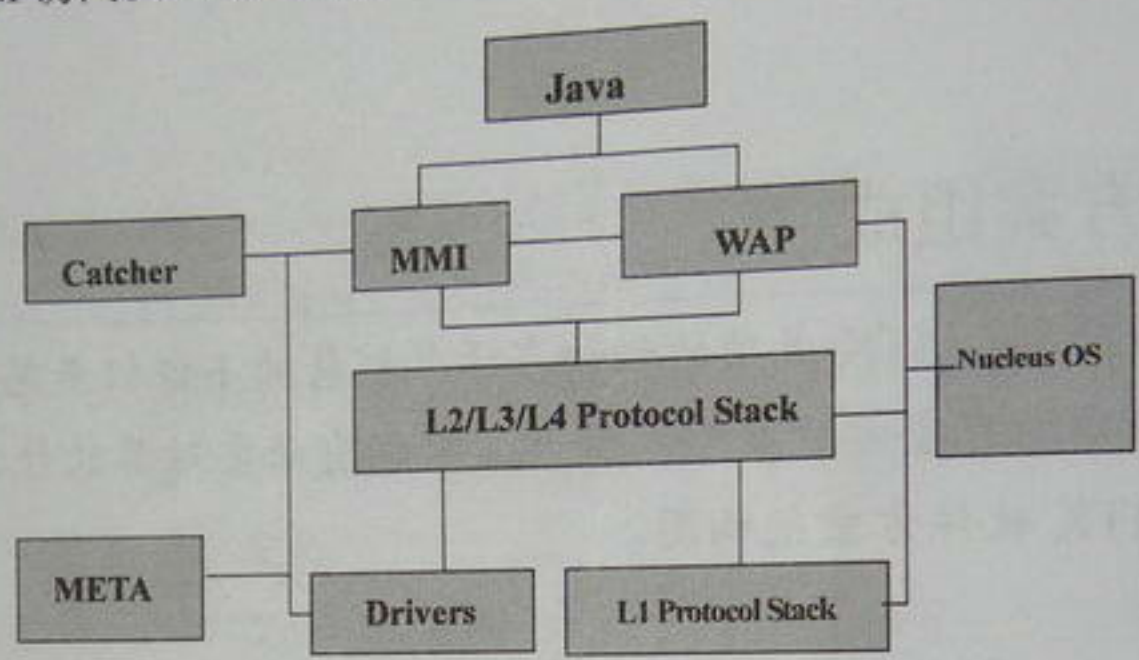


图 2.2 MTK 软件方案总体架构

其中，操作系统（Nucleus）、物理层协议栈（L1 Protocol Stack）、驱动程序（Drivers）、GSM 协议栈（L2/L3 Protocol Stack）和 MMI 运行于 MS（Mobile Station）。Mobile Station 由用于在一个无线网络中进行通信的全部用户设备和软件组成。比如在移动通信网络中用户所使用的手机。Mobile Station 是当前 2G 网络中的一个术语，但在 3G 网络中，MS 这一称呼将退出历史舞台，用户设备（User Equipment）缩写为 UE，将是 MS 的代名词。

在 GSM 网络中，Mobile station 由如下几部分构成。

· 移动通信终端（Mobile Terminal，MT）

提供一些被移动台服务利用的普通功能，等价于一个 ISDN 网络访问的终点，也可以说是无线接口的一个结束点。

· 终端设备（Terminal Equipment，TE）

是指任何 ISDN 兼容设备，其能够附加到这个网络上，例如一部电话、一台传真机或一台计算机。

· 终端适配器（Terminal Adapter，TA）

隐藏无线传送具体的实现细节，并且通过 TA，TE 发送 AT 指令可以控制移动台（Mobile Station，MS）的功能，与 GSM 网络业务进行交互。

· 客户识别模块（Subscriber Identity Module，SIM）

客户识别模块（Subscriber Identity Module，SIM）：也就是我们熟悉的 SIM 卡。GSM 数字移动电话机必须装上此卡方能使用。它在一块电脑芯片上存储了数字移动电话客户的信息和加密的密钥等内容，可对 GSM 网络客户身份进行鉴别，并对客户通话时的语音信息进行加密。

严格地讲，MS 就是指移动终端，UE=MS+SIM 卡，表示可以注册入网的终端。

说 明

ISDN（Integrated Services Digital Network，综合业务数字网）是一个数字电话网络国际标准，也是一种典型的电路交换网络系统。它通过普通的铜缆以更高的速率和质量传输语音和数据，ISDN 是欧洲普及的电话网络形式，GSM 移动电话标准也可以基于 ISDN 传输数据。

2.2.1 Nucleus 实时操作系统

Nucleus 是为实时嵌入式应用而设计的一个抢占式多任务操作系统内核，其 95%的代码是用 ANSI C 写成的，因此非常便于移植并能够支持大多数类型的处理器。从实现角度来看，Nucleus 是一组 C 函数库，应用程序代码与核心函数库连接在一起，生成一个目标代码，下载到目标板的 RAM 中或直接烧录到目标板的 ROM 中执行。在典型的目标环境中，Nucleus PLUS 核心代码区一般不超过 20KB 字节大小。Nucleus Plus 内核在典型的 CISC 体系结构上占据大约 20KB 空间，而在典型的 RISC 体系结构上占据空间为 40KB 左右，其内核数据结构占据 1.5KB 的空间。Nucleus Plus 以其实时响应、抢先、多任务以及源代码开放等特性获得在通信、国防、工业控制、航空/航天、铁路、网络、POS、自动化控制、智能家电等领域的广泛应用。

MTK 在上层应用和 Nucleus 之间增加了一个适配层（KAL，Kernel Abstraction Layer）。通过这种适配层的封装，增强了程序移植性、测试和调试的功能，并使代码更容易集成。

KAL 封装了一系列 API，这些 API 为其他软件提供服务，例如队列、消息、timer、内存管理等，如图 2.3 所示。

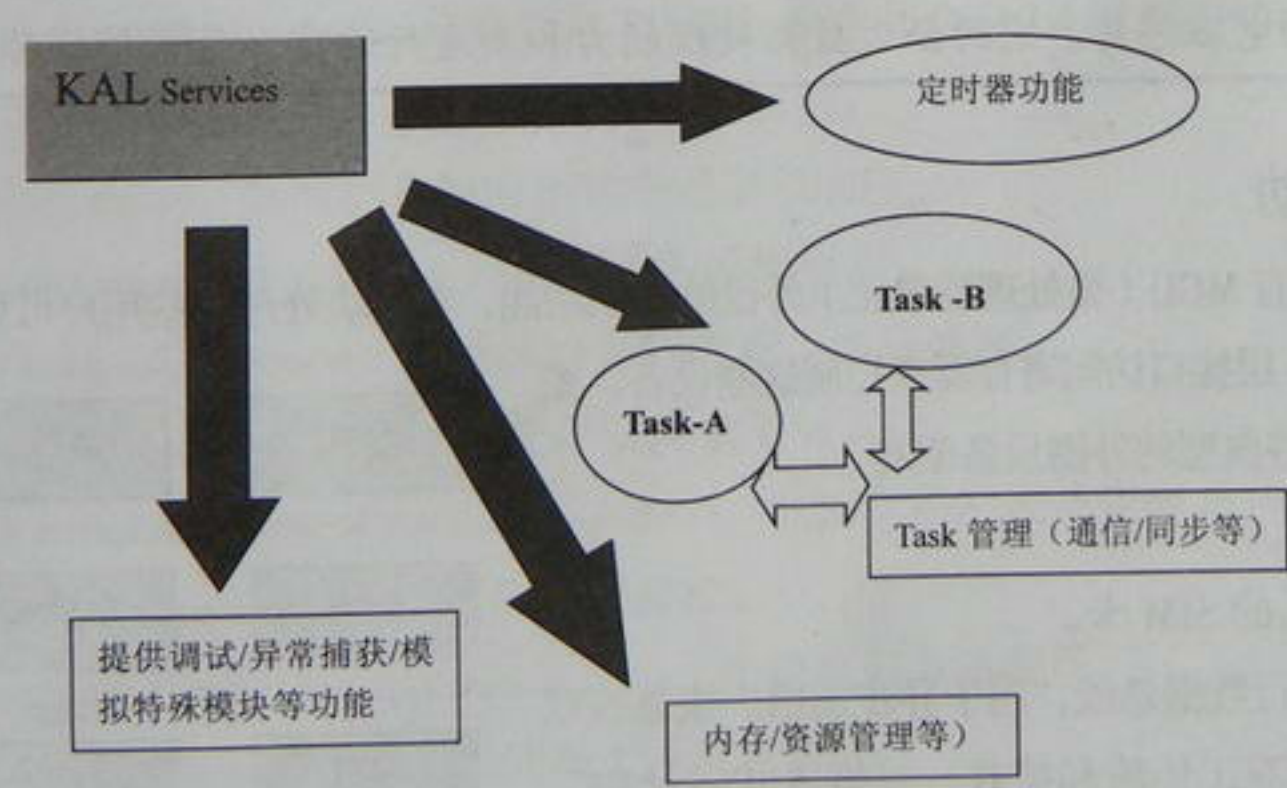


图 2.3 适配层（KAL）

更详细的介绍请参考对应的 Nucleus 文档，读者可从互联网上获得文档资料。

2.2.2 L1 协议栈

L1 层或者叫物理层，提供物理介质上的比特流传输，遵循 GSM 技术 05 系列规范，为上层软件提供服务，且控制逻辑信道到物理信道的映射和安排、无线控制以及 TDMA 桢。

基本模块如图 2.4 所示。

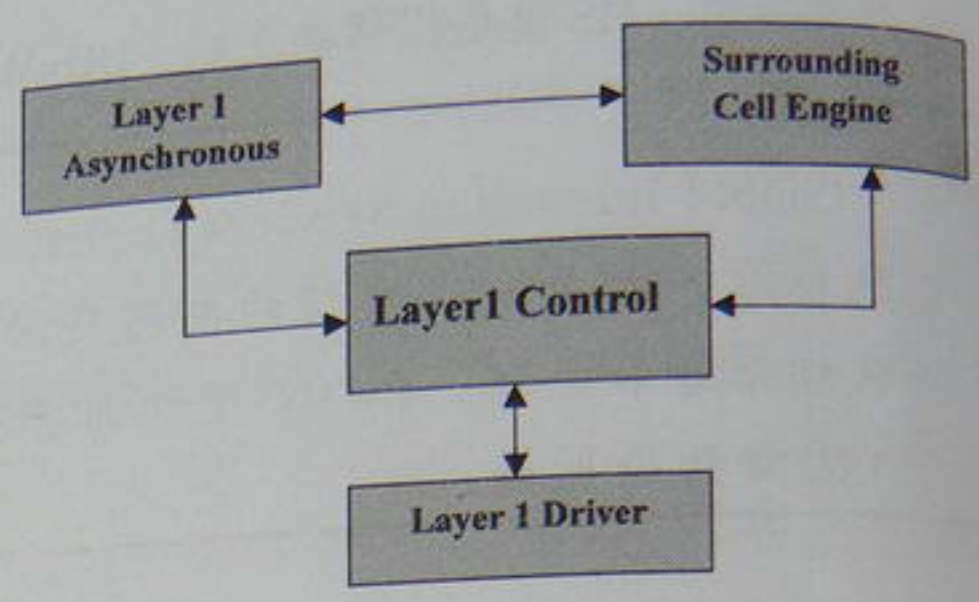


图 2.4　L1 协议栈内部关系

每个模块的功能如下。

· L1 异步逻辑（Layer1 Asynchronous）：处理上层软件的消息请求，经过 L1 处理后将结果返回给上层软件。

· Surrounding Cell Engine：处理相邻小区（Cell）的功率测量以及同步信息获取。

· L1 层控制（Layer1 Control）：处理无线环境中的 TDMA 时序安排，包括定时提前以及接受来自基站的功率控制。

· L1 驱动（Layer1 Driver）：处理 DSP 以及无线控制相关的内容。

这里只是一些理论知识的介绍，具体的细节请参考本书的驱动部分。

名词解释

Cell，直译为信元，也可以翻译为小区。在无线通信中，信元包括来自单一基站信号范围的地理区域（其坐标包括一个无线发收器和网络通信设备）。无线传输网络包含许多六角形的交叠处理的信元位置，使得在无线传输中能有效地使用无线频谱。Cell 就是“移动电话”的基础。

DSP（digital signal processor）是一种独特的微处理器，是以数字信号来处理大量信息的器件。其工作原理是接收模拟信号，将其转换为 0 或 1 的数字信号，再对数字信号进行修改、删除、强化操作，并在其他系统芯片中把数字数据解译回模拟数据或实际环境格式。它不仅具有可编程性，而且其实时运行速度可达每秒数以千万条复杂指令程序，远远超过通用微处理器，是数字化电子世界中日益重要的电脑芯片。它的强大数据处理能力和高运行速度，是最值得称道的两大特色。

2.2.3　设备驱动

设备驱动支持所有 MCU（微处理器单元）外设的控制功能，本模块处理一些用户可见操作的设备，如键盘、LCD 等。通过 L4 层接口访问寄存器来控制这些设备。图 2.5 所示是 MTK 的一些典型的外围设备驱动。

图 2.5　常用外围设备

缩略语解释如下。

· SIM：大家熟悉的 SIM 卡。

· UART：通用串行数据总线，用于异步通信。该总线双向通信，可以实现全双工传输和接收。在嵌入式设计中，UART 用来与 PC 进行通信，包括与监控调试器和其他器件。

· SPI：串行端口。

· LCD：液晶显示屏。手机设备中最重要的部分。

· GPIO：通用可编程输入输出口（General Purpose Input/Output）。输出可用于控制继电器、蜂鸣器、LED 等。输入可以获取传感器状态、高低电平、低速信号等。比如你做温度检测，在侦测完温度之后，可以通过一个设定为 Output 的 GPIO 发一个信号出去

以代表现在的温度是什么状态，比如是温度过高还是正常等。

· RTC：实时时钟（Real Time Clock），RTC 是主板上的晶振及相关电路组成的时钟电路的生成脉冲，它的主要作用就是提供稳定的时钟信号给后续电路用。主要功能有时钟、日历、闹钟及周期性中断输出。

· Keypad：键盘。提供手机的按键事件响应。

· PMIC：电源管理芯片（Power Management IC）。集成多个电源集成电路的功能来节省印刷电路板空间，这些电源集成电路包括低压差稳压器、开关稳压器、接口电路以及在某些情况下甚至包括电池充电器。

· BMT：电池充电管理。处理电池充电过程中的一些事件。

· USB：处理基于 USB 1.1 的协议及相关驱动。

· Camera：摄像头驱动。

· MSDC：记忆卡驱动、支持 SD 卡、MMC 卡和 Stick 卡。

限于篇幅，我们只列举了一部分的设备驱动列表，还有大量的设备驱动知识有待于在实际开发中进行了解和深入研究。

2.2.4 L2/L3/L4 层协议栈

这部分包含大量的 GSM/GPRS 需求以及提供一个优秀的 GSM/GPRS 平台给应用程序。本地 MMI 和远程 PC 通过 AT 命令可以方便地使用该软件平台，主要有如下几个部分。

· RMI：Remote MMI，PC 端通过 UART 口与协议栈进行通信。

· L4：MMI 通过 L4 与 GSM/GPRS 协议栈进行通信，包括以下子模块。

➢ ATCI：AT Command Interpreter，解释来自 PC 端的命令并命令 L4 做相应的动作。

➢ L4A：L4 adaptation Layer，MMI 与 L4A 通过消息通信。

➢ L4C：L4 Control entity，处理所有的应用程序请求和响应。

➢ UEM：User equipments adaptation，驱动相关的适配层。

➢ PHB：Phone book management，电话簿相关的管理，如分类等。

➢ SMU：SIM management Unit，安全性管理以及 STK 管理。

➢ CSM：Circuit switching protocol stack management，电路交换协议栈管理。

➢ RAC：Registration access control，访问控制授权。

➢ SMSAL：Short message service application layer，短消息服务适配层。

➢ TCM：Terminal context management，终端上下文管理。

· NVRAM：Nor-volatile RAM，是 MMI 到 Flash 的一个适配层，保存一些默认设置。

· SIM：SIM 卡控制。

· DATA：电路交换数据服务。

· CC：Circuit-switched call control，电路交换呼叫控制。

· SS：Supplementary service，附加服务。

· SMS：Short message service，短消息服务。

· SM：Session management，会话管理。

· MM/GMM：Mobility management，移动性能管理。

· SNDCP：Sub-network dependent convergence protocol 子网相关汇聚协议。

· LLC：Logical link control，逻辑连接控制。

这些术语在这里解释可能显得很枯燥，感兴趣的读者，可以参考通信相关的书籍，并可以在实际的项目中，慢慢体会这些模块之间的关系。这里只是给大家一个启蒙性的介绍，具体内容在第 16 章驱动部分有深入的讲解。

2.2.5 MMI

包括用户接口的架构、应用以及和协议栈的通信。MMI 是移动开发中最热的一部分，该部分也是本书重点关注的地方。

为了清晰地进行介绍，我们将 MMI 架构放在第 2.3 节讲解。

2.3 MMI 架构

首先，我们来看一下 MMI 整体的架构图，如图 2.6 所示。

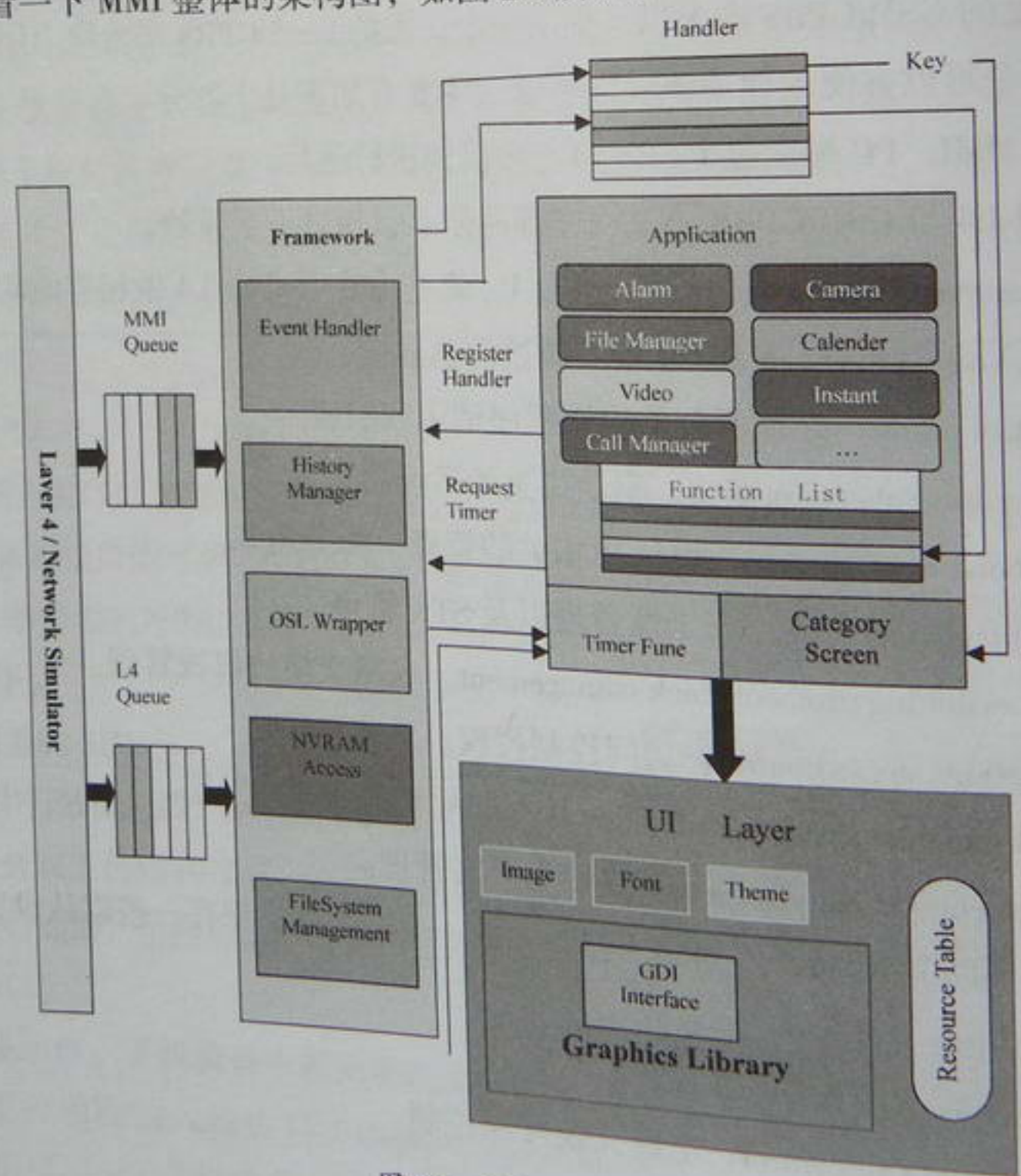


图 2.6 MMI 架构图

通过上面的结构图，我们可以看到 MMI 可以分为以下 4 个主要部分：

- （框架层）FrameWork。
- （应用层）Application。
- （UI 层）UI Layer。
- MMI 与 L4 层的通信。

2.3.1 Task

在分别讲述 4 个部分的具体内容之前，我们要先讲一下 MTK 系统中一个重要的属性：Task。Task 是一个具有一定封装性的软件模块，是操作系统层面的概念，当该模块被执行的时候，可以把它理解为一个独立的进程。在整个手机系统运行的过程中，有非常多的 Task 被执行。

MTK 系统有很多内置 Task（如负责处理音频、视频、图像等相关请求的多媒体 Task）。MMI 也是一个特殊的 Task（负责处理所有与用户交互相关的事件请求）。

MTK 也允许开发人员增加处理自己业务逻辑的自定义 Task。

系统中具体的 Task 信息，都保存在 sys_comp_config_tbl 变量中，我们看不到。但是 MTK 提供给客户的 custom_comp_config_tbl，客户是可以修改的，在这里用户可以定义自己的 Task。

关于 Task，还需要注意数据结构 comptask_handler_struct。

```
typedef struct {
  kal_task_func_ptr    comp_entry_func;
  task_init_func_ptr   comp_init_func;
  task_cfg_func_ptr    comp_cfg_func;
  task_reset_func_ptr  comp_reset_func;
  task_end_func_ptr    comp_end_func;
} comptask_handler_struct;
```

comptask_handler_struct 内部成员的执行顺序如下。

- comp_init_func

在系统还未进行调度时（即在 Application_Initialize 中）调用该函数用于初始化。

- comp_entry_func

任务被调度（Task schedule）后，执行 comp_entry_func，进入具体的 Task 入口函数。comp_cfg_func、comp_reset_func、comp_end_func 没有特别的意义。

本小节中，我们对 Task 做了一个初步探讨，在后面的（Task）章节中，我们会详细讲解如何自定义一个 Task。

2.3.2 MMI 与 L4 通信

在 2.2.4 小节中，我们已经讲过，MMI 通过 L4 与 GSM/GPRS 协议栈进行通信，以此获得包括短信、通话以及任何与通信相关的功能。L4C 负责处理所有的应用程序请求和响应，MMI 与 L4C 协议栈之间还封装了一个适配层 L4A，用于提供上层应用向底层的转换，如图 2.7 所示。

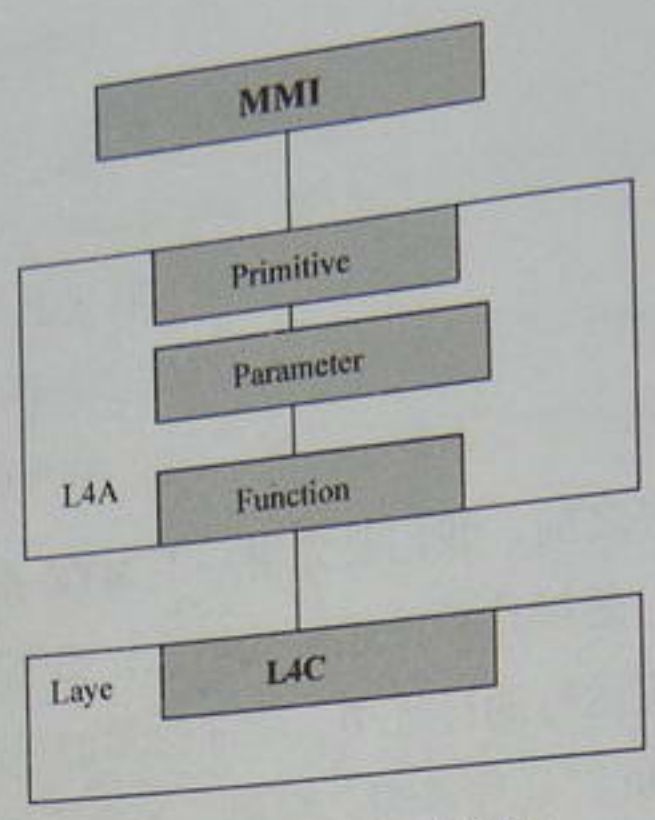


图 2.7　MMI 与 L4 通信

从图 2.7 中我们可以将整个软件方案大致分为两个层次：L4 层以下的部分是底层，以上的部分属于上层或者应用层。L4 层其实也可以看做一个 Task，其在手机的通信中属于绝对核心的模块，内部复杂程度可以单写一本书进行介绍，感兴趣的读者可以查看这方面的资料。

2.3.3　框架层（Framework Layer）

顾名思义，框架层就是 MTK 软件方案中已经设计好的部分，以后的客户逻辑将直接使用其已设计好的特性。就好像修房子，把整个屋子的地基、柱子、梁等重要部分搭建好，后期就在这个框架上进行后续工作。

框架层提供如下的核心功能。

1．消息队列

Task 之间的通信就是由队列来实现的。队列提供了一种传递多种消息的机制。一个发送请求将会把要发送的消息压入队列中，而一个接受请求将会把待接收的消息从队列里取出来。被封装的消息可能放在消息的前部也可以放在队列的末尾，每一个 Task 都能够向另外一个 Task 发送一个事件。一般情况下，事件是通过操作系统传输的，所以，OS 提供了相关的函数给开发人员。

目标 Task 通过调用接受相关的函数得到事件。MTK 提供了两种通信模式：一种是 Task 之间（Intra-task）通信，一种是 Task 内部（Inter-task）通信。

（1）Inter-task 队列

Inter-task 队列用于 Task 之间的通信。Task 能够接受所有来自其他 Task 的消息，而且一个正在等待事件的 Task 可以暂时挂起。这是一个非常重要的功能，因为处理器通过这种方式就能处理其他事件，而不必阻塞在当前的 Task 上。发送和接收队列提供了多种选项：无条件挂起、定时挂起和禁止挂起。

可能有多种原因导致 Task 在一个队列上挂起。比如一个 Task 试图接收一个来自空队列的消息，或者一个 Task 试图发送一个消息到已满的队列。当这个队列可以处理当前 Task 的请求时，一个被挂起的 Task 就可以恢复了。

（2）Intra-task 队列

有的时候，一些 Task 内部的事件消息可能要相互通知，这些事件只能在操作系统内核进行转移传递。因此，我们不必要求系统该如何做，也不必调用任何系统函数，只需要手工创建并维护一个队列，其余的就交给系统来完成。

（3）循环队列（MMI 独有）

2. 提供操作系统的抽象

对一些依赖于操作系统的应用程序提供一种封装，比如队列、管道、信号量和定时器，就相当于在操作系统和上层应用之间添加一个适配层，使得 MMI 代码更加具有适应性。

打开文件 Wrapper.c 可以清楚地看到，我们在 MMI 中经常使用的定时器函数 startTimer 其实是下层逐步封装上来的。

追根溯源：

StartTimer→StartMyTimer→StartMyTimerInt（经过封装的）→L4StartTimer→最后还是回归到操作系统内部的时钟。

3. 事件句柄

针对不同的消息事件，注册并执行应用程序的回调。

（1）协议层事件

基本事件，有独一无二的协议事件 ID 标识，如以下常见的函数。

```
void SetProtocolEventHandler(PsFuncPtr funcPtr, U16 eventID)
```

（2）按键事件

协议事件的一种，定义按键事件处理。

```
void SetKeyHandler(FuncPtr funcPtr, U16 keyCode, U16 keyType)
{
currKeyFuncPtrs[keyCode][keyType] = funcPtr;
}
```

执行按键事件。

```
void ExecuteCurrKeyHandler(S16 keyCode, S16 keyType)
{
(*currKeyFuncPtrs[keyCode][keyType])
}
```

（3）高亮事件

该事件在列表控件中特别常见，一般由用户上下按键切换列表菜单项，当单击高亮菜单项时就会触发高亮事件。

```
void register_fixed_list_highlight_handler(void (*f) (S32 item_index))
{
 MMI_fixed_list_menu.item_highlighted = f;
}
```

4. 历史机制

帮助应用程序维护屏幕的切换流程，存储切换过程中的重要数据。

5. NVRAM 访问

提供数据存储的封装和 NVRAM 数据的检索。

6. 文件系统管理

提供数据存储的封装和对来自文件系统数据的检索。文件系统用的是 FAT 格式，可供用户 MOUNT 存储设备，还提供了大量的文件系统读写接口。MTK 通过表格的形式让客户选择支持的 Flash，非常方便。

2.3.4 应用层（Application Layer）

包含用户定义的代码，用来执行业务逻辑。最基本的就是如下部分，这也是我们以后编写应用程序的基础，可以提供以下功能。

- 应用程序基本框架，包括各种现成的应用程序模板。
- 屏幕历史的保存，用于屏幕切换时的数据保存和状态维护。
- 进入一个新屏幕的入口函数，退出当前屏幕的处理函数。
- 按键事件的注册和执行。

2.3.5 UI 层

1. 为应用程序提供 UI 接口功能

在上一节中，我们已经提到 Application Layer 主要提供应用程序的后台逻辑实现，而要体现在屏幕上，就需要 UI 接口来实现了。这是一个关键点，在任何平台上都显得尤其重要，我们将在后面的章节着重分析其原理和实际使用。

2. 提供界面组件

（1）Category 屏幕

为屏幕绘制提供高效的封装。用户可以直接使用这种屏幕模板而不用考虑内部实现，接受来自应用程序的资源定义，如字符串和图片 ID 保持应用程序屏幕的布局、风格、外观的独立，能够独自实现加载而相互不受影响，并且提供历史管理的接口，使程序界面能够正确地实现跳转。

（2）界面元素

提供多种风格的控件，比如按钮、菜单、滚动条等。

（3）主题

负责实现一套具有固定的外观和感觉功能的用户界面元素。主题由 3 部分组成。

- 元素形状和颜色。
- 字体风格、样式和文本颜色。
- 相关的其他属性。

（4）字体

用于图形库在显示过程中进行字符渲染的数据，如常见的宋体、黑体等。显示在屏幕上的可见的字体样式其实在内存中都是以某种数据格式存在的。

（5）图片

设置独立的图片集，常见的使用方式如图标、墙纸和闪屏。

（6）图形库

提供对原始图形接口的支持，包含用于显示字体和图片的。

2.3.6 MMI 开发常用目录

1. 应用程序目录

待机界面(Idle Screen)： plutommi\mmi\IdleScreen；

主菜单(Main Menu)： plutommi\mmi\MainMenu；

电话本(Phone Book)： plutommi\mmi\PhoneBook；

短消息(Messages)： plutommi\mmi\Messages；

通话记录(Call History)： plutommi\mmi\CallManagement；

设置(Setting)： plutommi\mmi\Setting；

文件管理(File Manager)： plutommi\mtkapp\FileMgr；

娱乐(Fun &Games)： plutommi\mmi\FunAndGames；

情景模式(User Profiles)： plutommi\mmi\PROFILES；

工具箱(Organizer)： plutommi\mmi\Organizer；

服务(Services)： plutommi\mmi\SAT；

音频播放器(Audio Player)： plutommi\mtkapp\AudioPlayer；

照相机(Camera)： plutommi\mtkapp\Camera；

录音(Sound Recorder)： plutommi\mtkapp\SoundRecorder。

2. 资源目录

位置：\plutommi\Customer\ 该目录下有如下几个重要的文件夹。

（1）CustResource

针对具体客户的数据和资源目录。这些数据都会在构建整个系统的时候被复制到同级文件夹\CustomerInc 和\Res_MMI 中。因此，在实际项目开发中，特别要注意当前修改的是不是目录\plutommi\Customer\CustResource 中的文件，否则修改将没有效果。

（2）Images

开发中需要的图片文件全部放在该文件夹中，支持 BMP、PBM、GIF、JPG 等基本格式，PNG 格式的图片需要专门的库支持。

（3）Res_MMI

包含配置相关资源的文件。构建过程中会将位于目录。

\plutommi\Customer\CustResource\PLUTO_MMI\Res_MMI 中的内容复制到与文件夹 Customer 同级的文件夹 Res_MMI 中。

（4）ResGenerator

用于生成资源的相关工具。

（5）ResourceDLL

资源工具依赖的动态链接库。

2.4 启动流程

在前两节中，我们为大家罗列了很多术语名词，介绍了 MTK 整个软件方案的方方面面，但是这些孤立的知识点可能会使读者感觉难以理解。

接下来我们介绍基于 MTK 平台的手机启动过程，通过这个流程为大家把前面的知识串联起来，对手机软件如何相互联系，相互影响，运转进而构成一个通信系统做一个形象和整体的描述，加深大家对这一内容的理解。

（1）首先硬件上电给基带芯片供电，在经过一定时序条件后，给基带芯片复位信号，开始了 ARM 核的启动过程。谈到启动，我们必须熟悉 Scatter file（在第 16 章做详细讲解）。Scatter file 定义了 Load Region 和 Excecute Region，我们要关心系统运行时代码、数据的地址分布。

（2）Bootarm.s 是一个重要的文件，与启动过程息息相关，其中的 INT_Initialize 函数是 ARM 启动开始执行的代码。

（3）BSP 所做的工作如下。

① 配置 PLL，配置基带芯片的 EMI 参数，以让系统能够以最大的速度读取外部存储设备数据，让 CPU 以最大速度运行，从而缩短启动过程。

② 做好 runtime 代码及数据的准备，确保 Excecute Region 的代码及数据到位。

③ 配置好 ARM 7 种异常模式的堆栈，进入 RTOS Nucleus 操作系统的初始化。

④ Nucleus 留给客户的初始化函数 Application_Initialize，做平台该做的初始化工作，比如外部控制器的初始化等。

名词解释

BSP 是板级支持包，是介于主板硬件和操作系统之间的一层，应该说是属于操作系统的一部分，主要目的是为了支持操作系统，使之能够更好地运行于硬件主板。BSP 是相对于操作系统而言的，不同的操作系统对应于不同定义形式的 BSP，例如 VxWorks 的 BSP 和 Linux 的 BSP，相对于某一 CPU 来说尽管实现的功能一样，可是写法和接口定义是完全不同的，这样才能与上层 OS 保持正确的接口，良好地支持上层 OS。

上述的过程偏重于底层，在本书第 16 章的驱动部分有完整的讲解。下面，我们来看看软件部分在启动过程中的流程。

（1）函数 MMI_task 主要完成的是不停地读取 protocol task 中的消息，然后根据消息和参数进行相应的处理函数。

（2）在函数 MMI_task 里读取了消息后，系统会根据消息决定相应的处理方式。

（3）如果系统收到消息 MSG_ID_MMI_EQ_POWER_ON_IND，系统根据当前系统的状态来决定启动的方式，比如通过按键启动、充电器启动等。如下代码片段所示：

```
case MSG_ID_MMI_EQ_POWER_ON_IND:
{
…
 switch (p->poweron_mode)
 {
      case POWER_ON_KEYPAD:
         OslMemoryStart(TRUE);
         g_charbat_context.PowerOnCharger = 0;
          g_pwr_context.PowerOnMode = POWER_ON_KEYPAD;

         DTGetRTCTime(&StartUpTime);
         memset(&LastDuration, 0, sizeof(LastDuration));

        #if (defined(MMI_ON_HARDWARE_P) && !defined(__MTK_TARGET__))
        /* MoDIS only */
        mmi_netset_write_auto_test_result(MMI_FALSE);
        #endif

         /* disk check */
        #ifdef __FLIGHT_MODE_SUPPORT__
        g_phnset_cntx.curFlightMode = p->flightmode_state;
        #endif
           mmi_bootup_entry_disk_check();
        break;

        case POWER_ON_PRECHARGE:
        case POWER_ON_CHARGER_IN:
…
```

（4）正常启动（用户按键盘启动）中首先调用了函数 InitializeAll，它是系统启动的初始化函数。

（5）接着调用函数 InitNvramData，让一些需要读 NVRAM 的模块读出 NVRAM 中的数据，以此来初始化这些需要用到 NVRAM 中的数据来初始化的模块。

（6）初始化后就为进入主屏幕而作相关的准备。播放开机动画及声音，点亮显示屏，并且设置一个开机动画播放完毕后需要执行的回调函数。

（7）最后进入 EntryIdleScreen 函数，该函数就是程序的初始界面，同时在里面把左软键设置为进入主菜单，右软件设置为进入电话本。

至此，一个启动过程就结束了，以后的事情都交给注册好的事件响应函数和菜单响应函数去做。

结语：

本章我们为读者总结了 MTK 软件方案的架构。

在学习过程中，我们对操作系统、协议栈和设备驱动有了一个初步的概念，并对 MMI 的架构也做了一个前期的梳理，对 MMI 的 3 个层以及如何与 L4 通信也做了讲解，为读者在后面学习 MMI 开发打下基础。

大家知道，对于应用开发来说，MTK 的架构其实并不复杂，其 MMI 的架构也非常清晰。根据 MMI 的架构图大家就可以明白我们开发时所做的工作究竟处于哪个层次。

3 MTK 工程的编译

当开发人员初次得到 MTK 提供的软件工程时，首先要确定当前工程的版本，是 06B 还是 07B，带 Modis 仿真环境还是仅仅是 PC_Simulator 模拟器。然后将该原始版本按照 MTK 提供的编译规则在本机上编译一个可以烧制到硬件板上的 Bin 文件，以验证本机编译环境是否配置正确和当前代码的完整性。

作为一名开发人员，在开发之前，需准备好下面清单列出的内容。

1. 软件

（1）对应版本的源代码。源代码将由你的公司提供，或在网络上下载。

（2）烧写程序的下载工具 Flashtool（3.6 节会讲解）。

（3）编译工具和辅助工具（ADS、PERL 等，3.1 节讲解）。

（4）手机工程测试工具 META。

（5）抓包调试工具 Catcher（仅包含 Modis 的版本有效）。

（6）UI 制作工具 MCT。

2. 硬件

（1）开发用的手机板。

（2）数据烧录线。

（3）USB 线。

（4）充电器。

（5）锂电池。

（6）硬件原理图以及 MTK 向客户提供的开发文档。

如果看了这些清单很迷茫，不知道从哪儿下载的话，你可以访问http://www. 52rd.com和http://www.imtkoo.com/bbs/网址，获得大量的 MTK 培训文档和工具。

本书采用时下流行的 MT6226 06B 的软件版本进行讲解，其中 06B 表示软件的版本号。

为讲述方便，本书假定整个 MTK 工程文件夹放置于 E 盘根目录，工程文件夹名为 MTK6226。

说 明

MTK 平台软件版本的编译是一个很复杂的工作。开发人员必须认真地配置开发环境，正确地使用各种开发工具，使步骤准确。在版本编译过程中如果出现一些问题，需要耐心地分析原因。

3.1 编译环境的搭建

MTK 的编译环境都是搭建在 Windows 平台上的。本书中，下面的软件都安装在 Windows XP SP2 系统中。

3.1.1 工具安装与配置

（1）按照默认路径安装编译器：ADS1.2。

（2）安装 ADS1.2_update_848.exe 补丁包。

（3）按默认路径安装 ActivePerl-5.6.1.633。

（4）按照默认路径安装 MinGW-5.1.4。

安装的时候请注意如图 3.1、图 3.2 和图 3.3 所示关键步骤的选择。

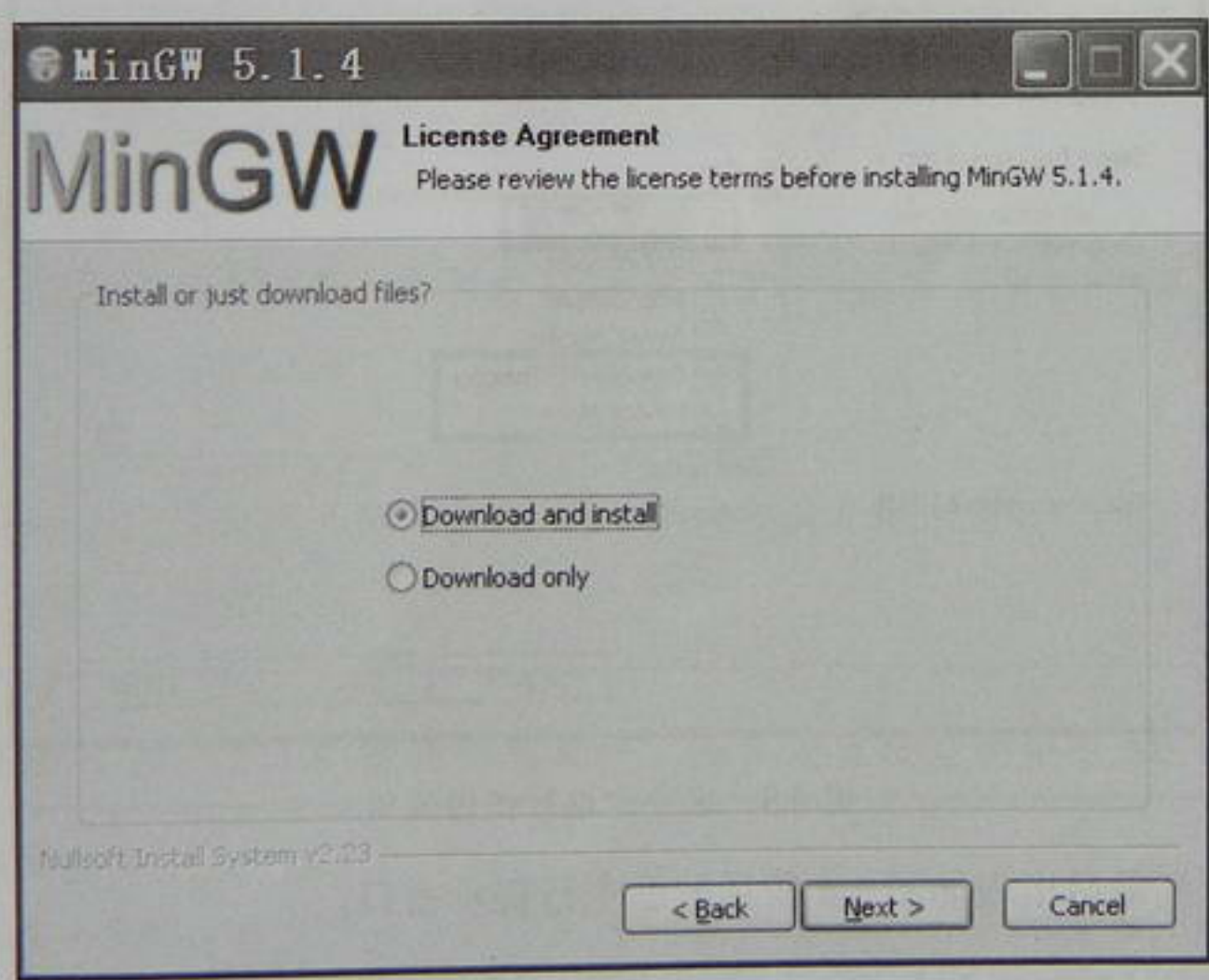


图 3.1 选择“Download and install”

（5）MSYS1.0.10。

因为 MTK 整个软件模块异常庞大繁杂，所以采用 GNU 的 make 命令来进行编译，使工程的编译更加灵活和便于控制。因此，有必要安装一种提供 GNU 环境的工具，而 MSYS 提供了这种功能，它是 Windows 环境上的一个出色的 UNIX 工作环境。

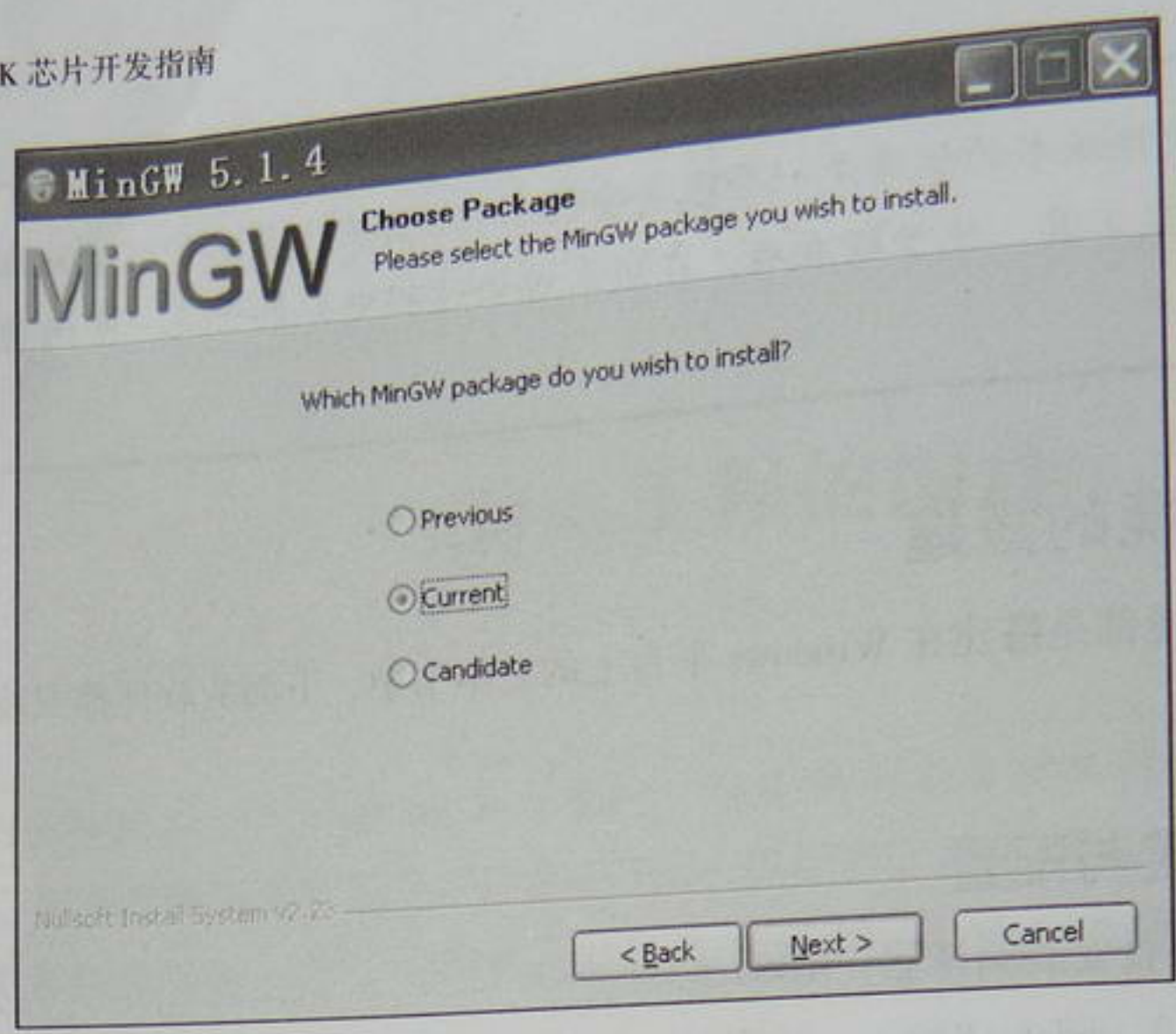


图 3.2 选择“Current”

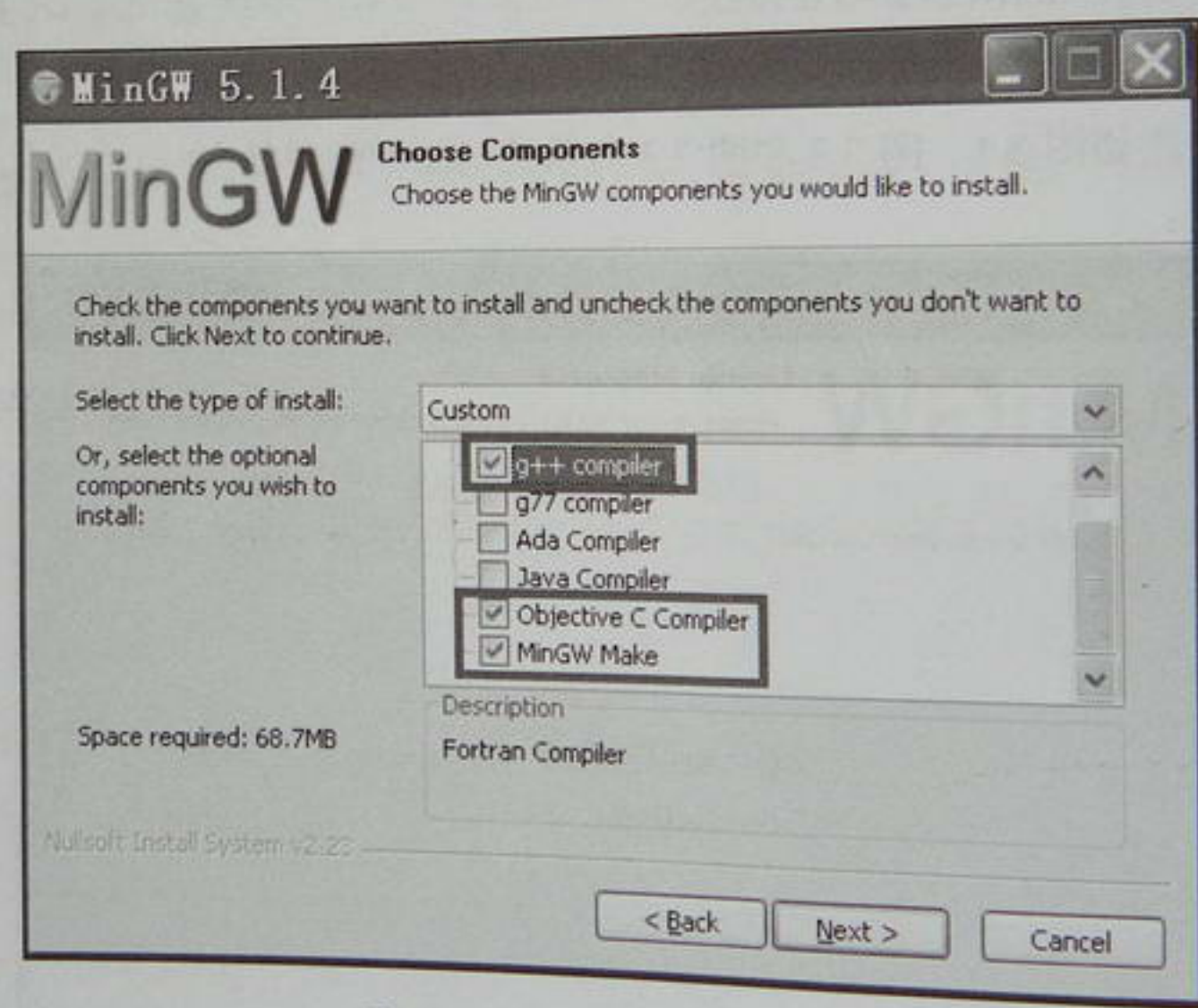


图 3.3 选中方框标识的选项

按默认安装，直到最后出现如图 3.4 所示的命令行提示窗口。

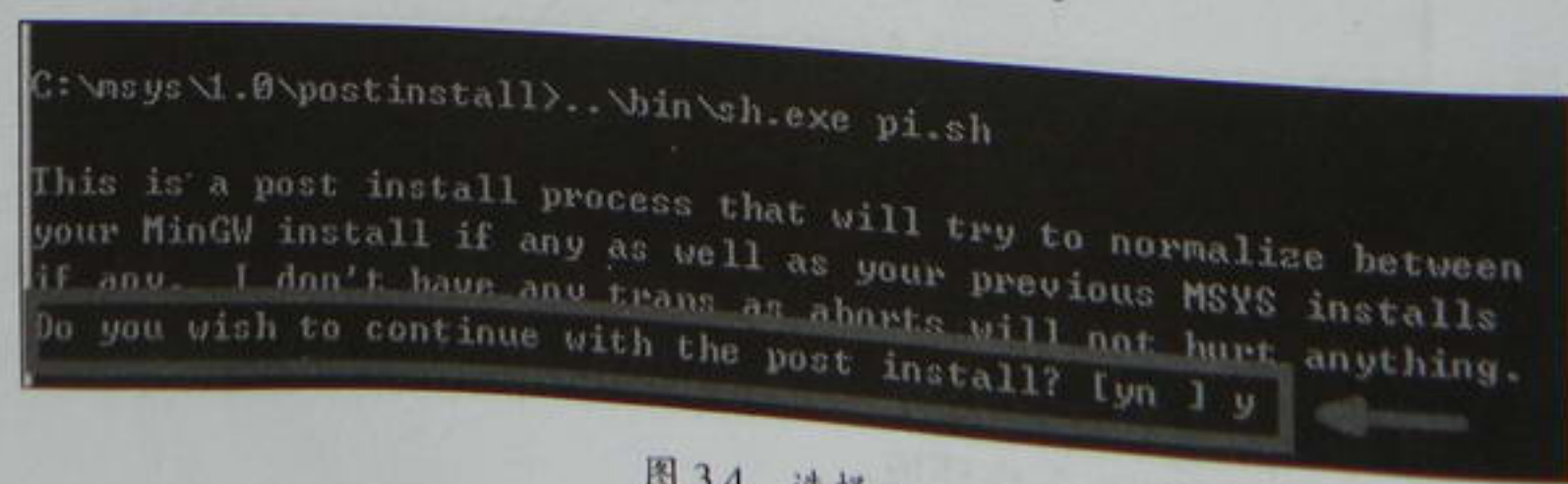


图 3.4 选择 y

选择 y 输入回车，确定继续安装如图 3.5 所示。

```
Do you have MinGW installed? [yn ] y
```

图 3.5　回车确认

输入 y 即确认 MinGW 已经安装，之后安装程序会继续询问安装路径。如图 3.6 所示。

```
Please answer the following in the form of c:/foo/bar.
Where is your MinGW installation? c:/MinGW
```

图 3.6　询问安装路径

这时，只需要输入 C:/MinGW 然后按回车即可。

> **注　意**
>
> 请确保 MinGW 先于 MSYS 安装。

（6）ImageMagick 6.3.6。

按默认路径安装，进行到提示 Select Additional Tasks 时，所有的选择全部取消，后面再按默认设置安装。

（7）7-zip 版本 3.13。

按默认路径安装，完成后复制..\7_Zip\7z.exe 至目录..\plutommi\Customer\Res Generator，并改名为 7za.exe。

（8）Visual C++ 6.0。

按照默认路径安装。最好装在 C 盘，避免后期不必要的路径修改。

（9）复制 MinGW。

在工程目录 ..\Tools\下建立文件夹，命名为 MinGW。进入先前安装的 MinGW，全选所有文件，复制到新的 MinGW 中，即..\Tools\MinGW。

（10）复制 MSYS。

在工程目录..\Tools\下建立文件夹，命名为 MSYS。进入先前安装的 MSYS\1.0\目录，全选所有文件，复制到新的 MSYS 中，即..\Tools\MSYS。

> **注　意**
>
> 请一定注意是进入 MSYS\1.0\复制所有文件，而不是直接复制整个 MSYS 文件夹！

（11）复制 ImageMagick。

复制..\ImageMagick-6.3.5-Q16 文件夹下全部文件到目录..\plutommi\Customer\ ResGenerator 中。

（12）设置 make.exe。

改名 mingw32-make.exe（..\Tools\MinGW）为 make.exe，并放在..\Tools\目录下。

3.1.2　进行开发环境检查

上面的小节中，我们已经安装了软件，并对环境做了初步修改。在开始编译之前，我们有必要对编译环境进行检查。

> **注　意**
>
> 如果之前的软件全部是按照默认要求进行安装的，一般不需要再修改代码，最好是全部装在 C 盘下。

将命令行切换到工程目录，输入如下命令：

```
E:\> CD MT6226
E:\MT6226 >CD  Tools
E:\ MT6226 \Tools>chk_env.exe.
```

输入检查环境变量的文件名后，如果一切正常，每个已经安装过的软件就会显示 OK，并会有如图 3.7 所示的提示成功的界面。如果某一个工具没有装好，就会出现 FAIL 的提示，开发人员可以快速定位并重装或配置。

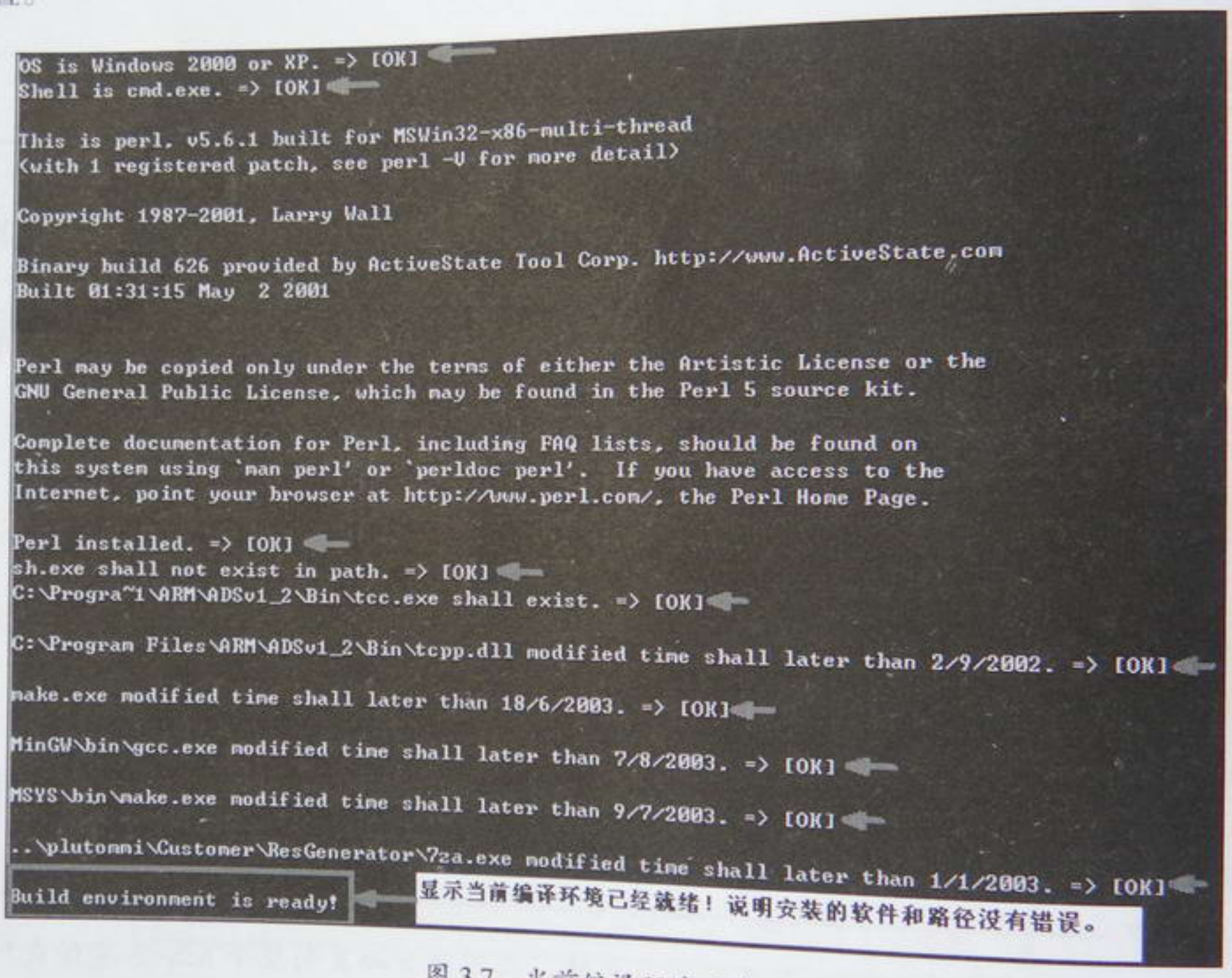


图 3.7　当前编译环境正常

3.2　如何进行编译

在各种工程的发布版本中，位于..\make\下都有一个×××_GPRS.MAK 文件，这个文件就是编译的核心，因为命令行的编译命令都要调用这一文件。

×××是 MTK 为具体的手机厂商定制的项目名称。比如：CompanyA_GPRS.MAK 和 COMB_GPRS.MAK。

在本书中，我们演示的工程项目名称为 MCUTEL26_06B_W07，那么，对应的 MAK 文件就是

MCUTEL26_06B_W07_GPRS.MAK。

> **提 示**
>
> 由上面的文件命名就可以看到，编译的时候实际上是根据 MAK 文件内容，将其拆分为两部分，即项目名（MCUTEL26_06B_W07）+工程名（GPRS）。

3.2.1 包含 Modis 仿真环境的工程

在软件版本进化到 07B 以后，MTK 为开发人员提供了一种更加优秀的仿真环境 Modis，使用它能极大地提高开发效率。因此，在 07B 版本中也需要通过编译生成 Modis。

假设.mak 文件名为：RELEASE_07B。

进入 E 盘根目录 E:\:

```
E:\> CD MTK6226
E:\ MTK6226
```

首先编译 Modis 模拟器，输入如下命令：

```
make  RELEASE_07B  GPRS  NEW_MODIS
```

> **提 示**
>
> 本书着重讲解的软件工程版本是 MT6226 06B。所以，上面的命令必须保证是 07B 的版本。

当 Modis 编译完毕后，将会在工程中产生一个 Modis.dsw 文件，这就是一个 VC 工程。开发人员以后可以在 PC 上通过 Modis 来模拟手机环境，加快调试和开发效率。Modis 的使用很简单，只是配置有一点复杂，具体请参考 MTK 提供的文档。

现在，我们继续编译一个用于真机的版本，即生成一个 Bin 文件。

输入命令：

```
make  RELEASE_07B  GPRS  NEW
```

3.2.2 不包含 Modis 仿真环境的工程

该工程相比 Modis，要简单一些，因为不必单独对仿真环境进行一次生成模拟器操作，这种模拟器将会在版本编译过程中自动生成。

进入 E 盘根目录 E:\:

```
E:\> CD MTK6226
```

在 E:\ MTK6226 目录下输入命令：

```
在E:>\MTK6226\make  MCUTEL26_06B_W07  GPRS  NEW
```

当编译完成后，会生成一个 Bin 文件，位于路径 MTK6226\Build*.BIN 中。

与此同时，用于 PC 上进行仿真的模拟器也生成出来。

进入 MTK6226\PlutoMMI\MMI\目录，双击 PC_Simulator.DSW，即可进入模拟器。

3.2.3 生成文件及 Log 日志文件

Log 文件与.bin 文件一样，也是位于 build 目录下，如果编译出错，可以在命令行中看到出错的模块，然后到 build 目录下找对应的 Log 文件。

（1）编译的 Log 文件位于目录 E:\MTK6226\build\MCUTEL26_06B_W07\log 下。

（2）编译生成的 Bin 文件位于 E:\MTK6226\build\MCUTEL26_06B_W07\g_prs\MT6226o\ 目录下。

（3）编译生成的各个模块的目标文件位于 E:\MTK6226\build\MCUTEL26_06B_W07\

3.3 软件系统的目录结构

进入 MTK 工程目录，可以看到至少有 40 多个文件夹，目录树非常庞大。在进行编译之前，应该对整个目录结构有一个清晰的认识。限于篇幅，我们只为读者讲解在一般开发过程中常用的文件目录。

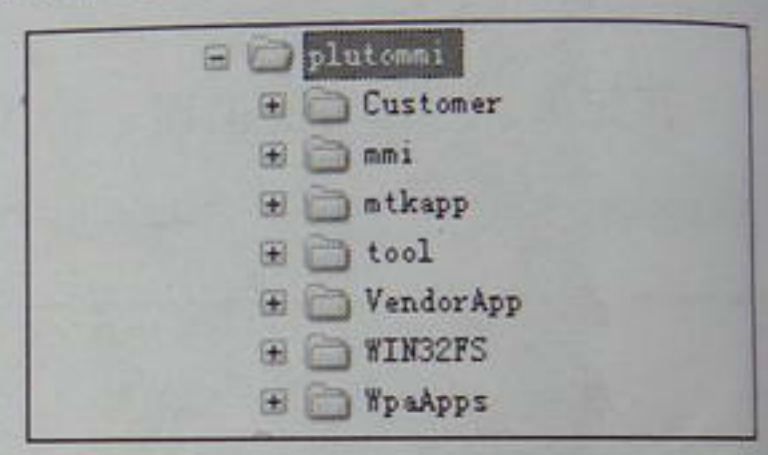


图 3.8 plutommi 目录结构

（1）PlutoMMI 是 MTK 提供的整套 MMI 应用所在的目录，只要是定制和修改的 MMI 应用，都会进入该目录。该目录下有如下几个文件夹，如图 3.8 所示。

· Customer 是图片、字符串等资源的定制目录，如果在项目开发过程中，修改了图片，资源或者是声音，都会进入该目录中运行资源生成的批处理文件，或者调用相关的编译语句，如 Update 等。

· mmi 是 MTK 设计的 MMI 应用所在的目录，是应用层开发人员修改最频繁的目录。

· mtkapp 中是 MTK 后来编写的 MMI 应用。读者在后期开发的一些模块，也可以放入该目录中。

（2）applib、bootloader、init、media、bmt 等目录为系统中相对独立的模块，可以单独进行编译。这些目录包含了这些模块的实现文件和头文件，每个目录（模块）都在接下来介绍的 make 目录中有相对应的目录，保存构建时的编译链接配置文件。

（3）make 目录是工程构建过程中最重要的一个目录，同时也是很多刚接触 MTK 的开发人员最容易出错的地方。在每个模块对应的目录下，都有 4 个文件控制了编译链接时的过程和配置，分别如下所示。

<module_name>.def 文件中是该模块构建过程中用到的条件定义。

<module_name>.inc 文件中是该模块所有用到的头文件所在目录的路径，是相对整个工程根目录的路径。

<module_name>.lis 文件中列出了该模块的所有 C 文件，也就是实现文件。其路径也是相对整个工程根目录的。

<module_name>.pth 文件中是该模块的所有 C 文件所在目录的路径，是相对整个工程根目录的路径。

来看下面的例子。

进入目录 E:\MTK6226\make\plutommi 中，将看到 4 个文件，这就是前面所讲的模块构建配置文件，该目录存放的是 plutommi 的构建配置文件，分别是 plutommi.def、plutommi.inc、plutommi.lis 和 plutommi.pth。读者可以结合前面讲述的知识进入这些文件查看内容，加深理解。

> **提 示**
>
> 在为 MMI 添加新的应用模块后，都必须按照上述规则将模块的各个文件路径加入进去，否则会出现很多问题，详细情况将在后面的章节中进行深入讲解。

（4）Fast_DL 目录是开发时下载二进制映像和资源等的相关文件的目录。其中的 CMM 文件配置了对应的二进制映像文件的路径和名称。读者可以查看该目录下的.cmm 文件的第 99 行，此处标识了下载映像文件的具体路径。

（5）kal 是内核抽象层的相关文件的目录。

（6）nvram 是存取 NV 中的内容的相关代码的目录。

（7）nucleus 是 nucleus 实时操作系统的目录，在给客户的发布版中只有头文件。在编译的时候根据其头文件找到对应的 Lib 库，找出需要的函数。

（8）drv 是驱动相关的代码的目录。

（9）fs 是文件系统相关的目录。

（10）tools 是工程相关工具的目录，包括 MinGW、MSYS 等 GNU 编译工具。

（11）interface 是系统各层的接口目录，还包含重要模块的接口，如 bluetooth、WIFI 等。

（12）config 是一些系统和任务（task）相关头文件的目录，用于配置具体模块。

（13）inc 是中断控制和寄存器地址相关头文件的目录。

（14）mtk_lib 中是已经编译链接好的 ARM 版的库。由于 MTK 整个工程十分庞大，完整地全新构建一次需要很长的时间。为了减少构建的时间，将一些已经完全调试稳定且基本不再改动的库和 MTK 不开放源代码的库放到这个目录，这样每次构建时只需要将这些库和其他编译好的库链接起来就可以了。这些库一般是 ARM 版的，若有 thumb 版的，则再增加一个和 ARM 版相对的 thumb 版的库，一般叫第二个库，如 media_sec.lib 就是 thumb 版的多媒体库。

（15）custom 是所有客户化项目的驱动程序及系统和 MMI 应用定制相关的文件的目录。如果手机主板的 RAM、Flash 等存储器的尺寸和地址空间有改动，MMI 应用的特性有不同，蓝牙、WAP 的配置有变化，或是有项目相关的新应用，其相关文件都是放在这个目录中和特定项目对应的目录下，这点在前面已经讲过。我们的项目名称为 MCUTEL26_06B_W07，所以，就放在 MCUTEL26_06B _BB 目录下。要注意的是 custom 也作为一个模块存在，这使得其可以通过 make 目录中的 custom 模块的 4 个文件进行一些客户化的定制。

3.4 构建原理

在上面的例子中，我们已经初步了解了 MTK 中进行工程编译的命令。在掌握了其用法之后，有必要对于其进行深入了解。下面我们就为读者仔细分析一下其内部原理。

MTK 手机软件系统的构建使用了 GNU 的 make 命令，使得整个工程的构建可以自动进行，且可以灵活控制。整个构建过程由 Make.bat、make3.pl、MoDIS.dsw、Gsm3.mak、Option.mak、<customer>_ <project >.mak 等文件控制。构建 PC 模拟版是通过 msdev 和 VC 的工程文件 MoDIS.dsw 来完成的；Gsm3.mak 是构建 ARM 版的核心 Makefile 文件，整个构建过程由其控制；其他文件是一些启动、选项配置、子过程的文件，其关系

图如图 3.9 所示。

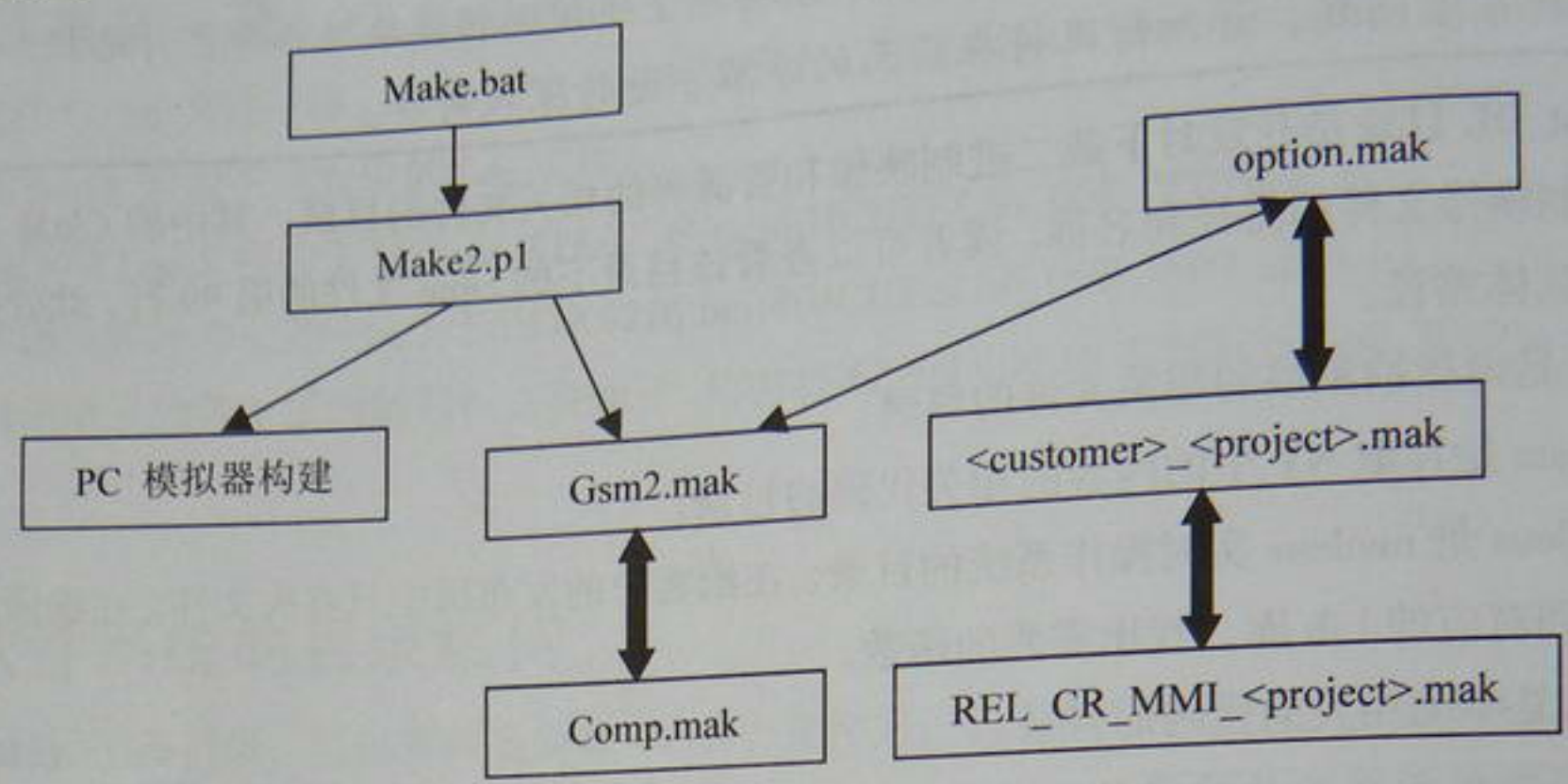


图 3.9 构建关系

在命令行输入的语句都是以 Make 开头的，这个命令很重要，它对应了工程中的一个批处理文件 Make.bat。

1. Make.bat

这个批处理文件启动了整个工程的构建过程。该文件根据命令行输入的语句解析出工程名、平台和将要执行的编译操作。与此同时，检查 MTK6226\make\路径是否存在或者检查命令行语句是否合法。在上述操作正确无误后，即可启动工程的构建。

该批处理文件的使用方法如下所示。

语法：

```
make [customer_name] [project] [action] [modules]
```

解释：

customer_name 即为具体客户定制的项目名称。

Customer_name 可以是任意的，也可以替换为如下值：

mtk （默认）

MCUTEL26_demo （MCUTEL26_demo 项目）

或者具体手机公司要求的名称

project 需要构建哪种类型的工程。比如客户可能只需要一个能通话的系统，界面有没有无关紧要，或者客户要求

GSM 或者要求 GPRS。

project 可以替换为如下值：

l1s （只需要 Layer 1 层）

gsm （仅 GSM）

gprs （仅 GPRS）

basic （只需要基本框架）

action 构建工程时需要执行的具体操作。

action 可以替换为如下值：

new （完全构建工程）

update 或简写为 u （更新已存在的工程，涉及资源）

remake 或简写为 r （重新编译工程逻辑代码，不涉及资源）

clean 或简写为 c （清除已生产的目标文件）

module 工程中的各个模块。

可替换为相应的模块名（plutommi,Custom, applib……）

当你只改动了某一个模块，而且只修改了一小部分逻辑上的代码时，你就可以将 module 换为你改动过的模块。这样就只会编译一个模块，能大大地节约时间。举个例子，如果你在编译系统过程中，已经编译到了 PlutoMMI 模块，但这个时候出错了，当你修改完代码准备重新编译的时候，就只需要从 PlutoMMI 模块接着编译即可，因为编译器会自动链接需要的目标文件，代码如下：

```
make MCUTEL26_06B_W07 GPRS remake PlutoMMI
```

其中较常用的 action 有 new、update、remake 和 new_modis。

“new”是全新开始构建整个工程的 ARM 版，包括图片、声音、字符串等资源都要重做，依赖的其他动作最多，是最彻底也是耗时最长的一个动作，一般在拿到一个新的 MTK 发布版本后要做一次。还有，根据笔者经验，在一些情况下编译出现奇怪的问题而又能很好地解决时，可以尝试使用 new 操作。

“update”是重新更新整个工程的 ARM 版，该动作会扫描工程中文件和库的依赖关系，若依赖关系有变化会建立新的依赖关系，随后根据新的依赖关系重新编译链接有改动的部分，如果对资源文件有修改，如修改了图片、字符串等，或者增加、删除一些驱动或应用的情况下需要用 update 操作。

“remake”是重新编译整个工程的 ARM 版，该动作只是简单地重新编译链接有改动的部分，不检查依赖关系，是耗时最短的一个动作，也是最常用的动作。一般在修改了一些逻辑上的代码，不涉及资源文件的情况下，remake 操作是最佳选择。

“new_modis”是全新构建整个工程的 PC 模拟版，其调用 VC 的编译器和链接器得到一个可以在 Windows 系统上运行的 PC 模拟版。MMI 应用软件工程师可以在没有硬件板的情况下在 PC 上检查和调试自己写好的应用程序。

提 示

需要注意的是，在 07B 版本及其以后的版本中才有 MODIS 仿真环境。本书主要讲解 6226_06B，所以，将不会使用命令 new_modis。

现在，让我们按照上述规则来编译 MTK6226 工程，可以用如下命令。

（1）构建全新的 GPRS 工程，需要选用 new 命令。用于清空所有旧的对象、库和 Log 日志文件等，并同时创建出必要的文件路径，生成新的资源配置文件。任何新的工程交给开发人员时，都应该使用 new 操作重新编译一次软件版本。

```
E:>\MTK6226\Make  MCUTEL26_06B_W07  GPRS  new
```

（2）在需要更新工程依赖关系，重新编译修改过的模块的时候，特别是资源文件有修改的情况下，应该使用 update 命令。值得注意的是 update 不会删除临时文件，如 object 文件和 Log 记录。编译的日志信息会追加在上次编译完成后对应的模块 Log 文件中。

```
E:>\MTK6226\Make  MCUTEL26_06B_W07  GPRS  update
```

（3）remake 命令只针对程序逻辑代码有修改的情况进行编译连接。若有资源文件修改，不应该使用 remake 命令。

```
E:>\MTK6226\Make  MCUTEL26_06B_W07  GPRS  remake
```

（4）clean 不仅清除所有对象 object、临时文件、库和可执行的二进制文件，同时 Log 文件也会被清除。

```
E:>\MTK6226\Make MCUTEL26_06B_W07 GPRS clean
```

（5）如果需要 clean 对应的某个模块，请在 clean 后添加相应的模块名，如 clean plutommi。

```
E:>\MTK6226\Make MCUTEL26_06B_W07 GPRS clean plutommi
```

Make.bat 文件实际上只起引导作用，其只有一行批处理语句 perl make3.pl %*，于是运行该批处理文件后控制权转给了 Perl 脚本。

2．make3.pl

在这个 Perl 脚本中解析了用户输入的命令行参数，设置变量，准备 make 时需要的临时配置文件，随后根据生成的可运行映像是 PC 模拟版还是 ARM 版来分别调用不同的构建过程。

通过给 make 命令指定 ARM 版的核心 Makefile 文件 Gsm3.mak，开始 ARM 版的构建过程。

3．Gsm3.mak

该文件中包含了 Option.mak 这个用来进行配置的 Makefile 文件，另外还包含了一些以.tmp 和.bld 为后缀的由 Perl 脚本 make3.pl 生成的临时配置文件。这些临时配置文件主要是一些 action 如 clean、remake 等所需的变量设置及客户化和版本号等一些信息。

Gsm3.mak 文件控制了 new、update、remake 等动作的过程。在 Gsm3.mak 文件中还将各个模块编译链接得到的库和 mtk_lib 目录下的库链接起来得到一个映像文件，然后使用 ADS 的工具 fromelf 将映像文件生成以项目名为前缀名的二进制文件，该文件可以下载到硬件板上运行。

那么，这些 Lib 库是如何构建和链接的呢？我们打开 Gsm3.mak 文件，找到图 3.10 所示的位置。

注意上图中的代码 libs: cleanlib startbuildlibs $(COMPLIBLIST)。

libs 要生成的库由变量 COMPLIBLIST 列出，在 ARM 版中，变量 COMPLIBLIST 从变量 COMPLIST 获得。

而变量 COMPLIST 是在 Option.mak 及其包含的 Makefile 文件中赋值的。因为有很多库需要编译链接，变量 COMPLIBLIST 展开后包含多个步骤，而这些步骤都是重复不变的，所以在定义步骤$(COMPLIBLIST)的构建过程时使用%.lib 代替。%.lib 这个步骤先清除了之前的一些依赖关系文件，将一些变量的设置写

入~compbld.tmp 这个临时文件中，然后给 make 命令指定 Makefile 文件 comp.mak，完成库的编译和链接，如下所示：

```
# ***************************************************************
# Library Targets
# ***************************************************************
libs: cleanlib startbuildlibs $(COMPLIBLIST)

startbuildlibs:
    @echo Start to build $(COMPLIBLIST)

cleanlib:
   # can be moved to cleanlog
    -@if exist $(strip $(TARGDIR))\$(NEWTARGNAME).bin (del $(stri
    -@if exist $(strip $(TARGDIR))\$(TARGNAME).elf (del $(strip $
    -@if exist $(strip $(TARGDIR))\$(TARGNAME).lis (del $(strip $
    -@if exist $(strip $(TARGDIR))\$(TARGNAME).sym (del $(strip $
    -@if exist $(strip $(TARGDIR))\$(TARGNAME).htm (del $(strip $

    @echo Clean $(COMPLIBLIST) under $(strip $(COMPLIBDIR))
    -@for /d %%i in ($(COMPLIBLIST)) do \
        (@if exist $(strip $(COMPLIBDIR))\%%i (del $(strip $(COMP

%.lib:
    @echo Beginning $* component build process... >> $(LOG)
    @perl tools\time.pl
```

图 3.10　Gsm3.mak 文件的代码截图

```
%.lib:
  @if /I %OS% EQU WINDOWS_NT \
        (if /I $(BM_NEW) EQU TRUE \
            (tools\make.exe -fmake\comp.mak -k -r -R $(strip $(CMD_ARGU)) COMPONENT=$* >
$(strip $(COMPLOGDIR))\$*.log 2>&1)
   \
        else \
            (tools\make.exe -fmake\comp.mak -r -R $(strip $(CMD_ARGU)) COMPONENT=$* > $(strip
$(COMPLOGDIR))\$*.log 2>&1) \
        ) \
    else \
        (if /I $(BM_NEW) EQU TRUE \
            (tools\make.exe -fmake\comp.mak -k -r -R $(strip $(CMD_ARGU)) COMPONENT=$* >
$(strip $(COMPLOGDIR))\$*.log) \
        else \
            (tools\make.exe -fmake\comp.mak -r -R $(strip $(CMD_ARGU)) COMPONENT=$* > $(strip
$(COMPLOGDIR))\$*.log) \
        )
```

4. comp.mak

这个 Makefile 文件控制了模块的编译链接过程。

在这个文件中，首先将当前模块要生成的库（由变量 COMPONENT 传入）设置给了变量 TARGLIB。之后从<module_name>.lis 文件中得到 SRC_LIST 和 CPPSRC_LIST 两个源文件列表，设定要编译的 C 文件、C++文件、汇编文件等的列表和要链接的中间目标文件的列表。将<module_name>.inc、<module_name>.def、<module_name>.pth 文件中的头文件路径、C 文件路径、编译链接参数等赋值给相应的变量。将与平台相关（如 6225、6226）的编译参数加上，最后进入库的编译链接过程。目标库的依赖关系如

图 3.11 所示。

```
# ************************************************************************
# Construct $(COMPONENT) specific varible
# ************************************************************************
TARGLIB        =  $(subst \,/,$(OBJSDIR))/lib/$(COMPONENT).lib
COMPOBJS_DIR   =  $(subst \,/,$(OBJSDIR))/$(COMPONENT)
COMPBUILD_DIR  =  $(FIXPATH)\make\$(COMPONENT)

# ************************************************************************
# Construct the list of object dependencies
# ************************************************************************
SRC_LIST       =  $(subst .S,.s,$(subst .C,.c,$(shell type $(COMPBUILD_DIR)\
CPPSRC_LIST    =  $(subst .S,.s,$(subst .CPP,.cpp,$(shell type $(COMPBUILD_D
CSRCS          =  $(filter %.c, $(SRC_LIST))
CPPSRCS        =  $(filter %.cpp, $(CPPSRC_LIST))
ASRCS          =  $(filter %.s, $(SRC_LIST))
ARMSRCS        =  $(filter %.arm, $(SRC_LIST))
COBJS          =  $(patsubst %.c,%.obj, $(notdir $(subst \,/,$(CSRCS))))
CPPOBJS        =  $(patsubst %.cpp,%.obj, $(notdir $(subst \,/,$(CPPSRCS))))
AOBJS          =  $(patsubst %.s,%.obj, $(notdir $(subst \,/,$(ASRCS))))
ARMOBJS        =  $(patsubst %.arm,%.obj, $(notdir $(subst \,/,$(ARMSRCS))))

INCDIRS        += $(shell type $(COMPBUILD_DIR)\$(COMPONENT).inc)
COMP_DEFS      =  $(shell type $(COMPBUILD_DIR)\$(COMPONENT).def)
DEFINES        += $(COMP_DEFS)
SRCPATH        =  $(shell type $(COMPBUILD_DIR)\$(COMPONENT).pth)
```

图 3.11　目标库的依赖关系

库的编译链接由 update_lib 步骤完成，这个步骤直接依赖了$(TARGLIB)。

目标$(TARGLIB)是由所有按照.c.obj、.s.obj、%.obj: %.cpp 等规则编译得到的中间目标文件链接获得的。其主要过程如以下代码和图 3.12 所示：

```
   $(TARGLIB):
   @if   exist          $(FIXPATH)\$(CUS_MTK_LIB)\$(COMPONENT).lib
\ (copy /z $(FIXPATH)\$(CUS_MTK_LIB)\$(COMPONENT).lib $(subst /,\,$(TARGLIB))) & \
    ($(LIB)   -r   $(TARGLIB)     $(COMPOBJS_DIR)/*.obj)
\else
\ ($(LIB) -create $(TARGLIB) $(COMPOBJS_DIR)/*.obj)
...
```

```
# ************************************************************************
# Library Targets
# ************************************************************************
update_lib: $(TARGLIB)

ifeq ($(strip $(RVCT_MULTI_FILE)),NONE)

$(TARGLIB) : $(COBJS) $(CPPOBJS) $(AOBJS) $(ARMOBJS)

   # If library for customer release exists.
   # Copy and update it or create a new one
    @if exist $(FIXPATH)\$(CUS_MTK_LIB)\$(COMPONENT).lib
        (copy /z $(FIXPATH)\$(CUS_MTK_LIB)\$(COMPONENT).
        ($(LIB) -r $(TARGLIB) $(COMPOBJS_DIR)/*.obj)
    else
        ($(LIB) -create $(TARGLIB) $(COMPOBJS_DIR)/*.obj

    @echo $(TARGLIB) is updated
```

图 3.12　update_lib

5. Option.mak

这是整个工程构建过程中的总控配置文件，在这个文件中包含两个 MAK 文件。

（1）<customer>_<project>.mak

定义了需要编译的模块列表，编译中包含文件和生成文件的路径，定义 trace 的头文件路径，全局的编译开关，用户的一些编译选项，最后定义了编译路径和目录。用户可以在这个文件中定义配置。

（2）REL_CR_MMI_<project>.mak

客户要定制自己的项目需要对 REL_CR_MMI_<project>.mak 中的两个变量 CUS_REL_SRC_COMP 和 CUS_REL_PAR_SRC_COMP 进行赋值。

这两个项目包括相关的配置文件，用户自定义的配置文件 USER_SPECIFIC.mak，另外还包含了一些临时生成的配置文件，这些配置文件一起设置了工程构建过程中用到的编译器、链接器、库和二进制的工具，设置了编译链接时的参数，公共的头文件路径，设置了需要包含 mtk_lib 目录中的哪些已有的库，设置了需要生成的库等一系列相关的配置。定义了编译环境中用到的工具及相关的目录，根据大的编译开关定义了小的编译和链接选项。

3.5 配置编译选项

MTK 手机软件系统的大部分配置都已经确定，基本上不需要再改变，一般是 MTK 发布新的版本时会做些改变，客户不需要关注。最主要的配置文件是 Option.mak，而客户化定制需要修改的主要是 <customer>_<project>.mak、REL_CR_MMI_<project>.mak 这两个文件。

Option.mak 文件中设置了工程构建时用到的编译器、链接器、库管理、二进制文件生成等工具的路径和可执行程序名，设置了基本的编译链接参数，指定 CPU 类型（ARM7EJ-S），确定最终使用的库列表（COMPLIST）。

一个项目是由工程中的多个基本库和新编译链接的库组成的，最终这些库链接到一起得到一个完整的可执行映像文件。由哪些库来组成一个项目是由 4 个关键的变量决定的，其中一个是前面提到的 COMPLIST，另外 3 个分别是 CUS_REL_SRC_COMP、CUS_REL_PAR_SRC_COMP 和 CUS_REL_MTK_COMP。这几个变量的关系如下所示：

```
# ************************************************************************
# Custom Release Component Configuration
# ************************************************************************
# Be sure the following:
# 1. CUS_REL_SRC_COMP +CUS_REL_PAR_SRC_COMP =
COMPLIST(CUSTOM_RELEASE = True)
# 3. CUS_REL_SRC_COMP + CUS_REL_PAR_SRC_COMP + CUS_REL_MTK_COMP =
COMPLIST(CUSTOM_RELEASE = False)
```

COMPLIST 确定了该项目最终是由哪些库（组件）组成。

MTK 的内部版本构建时，则包括 CUS_REL_SRC_COMP、CUS_REL_PAR_SRC_COMP 和 CUS_REL_MTK_COMP 这些库。客户版本构建时，则只包括 CUS_REL_SRC_COMP、CUS_REL_PAR_SRC_COMP 这些库。

手机设计公司构建的版本都是客户版本，因此只要改变 CUS_REL_SRC_COMP、CUS_REL_PAR_

SRC_COMP 这两个变量的值即可以定制自己的项目了。

Option.mak 和 REL_CR_MMI_GPRS.mak 这两个文件都可以对这两个变量 CUS_REL_SRC_COMP、CUS_REL_PAR_SRC_COMP 进行赋值，但 Option.mak 文件只是设定了整个工程需要的基本库，如手写、图形解码等，和特定项目没有关系。

因此客户要定制自己的项目只需要修改 REL_CR_MMI_<project>.mak 这个文件对这两个变量的赋值即可。

<costomer>_<project>.mak 文件也可以对 COMPLIST 赋值，但这只是对 MTK 的内部版本有用，对客户版本没有影响。值得注意的是，该项目的绝大部分功能特性（feature）是在这个文件中确定的，如使用的 sensor、支持哪些音视频格式、电话本大小、LCD 屏幕大小等。以下代码是该文件中的部分内容。

```
...
WEBCAM_SUPPORT        = TRUE      # WEB CAMERA support
R99_SUPPORT           = TRUE  # Release 99 support
EDGE_SUPPORT     = FALSE     # EDGE support
CHE_SUPPORT           = FALSE    # CHE support
TV_OUT_SUPPORT        = FALSE      # TRUE, FALSE  only MT6228 and
MT6229 support TV OUT
BACKGROUND_SOUND      = FALSE      # TRUE/FALSE
AUDIO_POST_PROCESS    = NONE
...
```

开发人员也可以在该文件中，声明自己的 XXX_SUPPORT。比如我们要增加一个 PNG 解码模块，就可以仿照上面的格式声明一个 PNG_SUPPORT，这样在以后的 PNG 模块开发中，就可以使用这个宏标志，方便控制代码。

下面以如何增加一个新的模块的实例来结束本章。

3.5.1 包含源代码的第三方库

假如我们需要加入一个 TTS 模块（TTS 包含了跟语音识别相关的功能），我们可以按如下步骤进行。

（1）把 TTS 的源代码包复制到你的 MTK 软件系统根目录下，这样所有的 TTS 代码都在 TTS 目录中。

（2）在 make 目录下新增一个 tts 目录，增加 4 个文件，分别是 tts.def、tts.inc、tts.lis 和 tts.pth。tts.def 文件的内容需要根据内容来修改，其他 3 个文件中加上源文件及其目录、头文件目录即可。

（3）在 REL_CR_MMI_<project>.mak 这个文件的合适位置加上如下语句。

```
CUS_REL_SRC_COMP += tts
```

（4）把这个工程 remake 一下，若没有错误，tts 模块就成功加上去了。

开发人员如果是在 plutommi 或者其他已经加入系统编译的模块中添加了应用，只需要在 make\Plutommi 中找到对应的 3 个文件，将新应用程序的相关目录加入到对应的文件末尾就可以了。

3.5.2 无源代码的第三方库

若我们需要加入一个没有源代码的第三方库，比如加入一个语音功能模块 ESD，我们可以按如下步骤进行。

（1）把 ESD 的所有第三方库文件复制到一个新建的 ESD 目录下。

（2）在 Option.mak 这个文件的合适位置加上如下语句。

```
ifeq ($(strip $(ESD_SUPPORT)),NONE)
   COMPOBJS  +=  plutommi\mmi\ ESD \Lib\ ESRCore..a
   COMPOBJS  +=  plutommi\mmi\ ESD \Lib\ESRGramma.a
   CUS_REL_OBJ_LIST  +=  plutommi\mmi\ ESD \Lib\ESRCore.a
   CUS_REL_OBJ_LIST  +=  plutommi\mmi\ ESD \Lib\ESRGramma.a
endif
```

后缀名为.a 的文件就是库提供方提供的文件。

ESD_SUPPORT 就添加在我们前面提到的<costomer>_<project>.mak 文件中，并初始化为 TRUE。

```
ESD_SUPPORT = TRUE;
```

修改完成后，remake 整个工程，就可以将该模块添加进工程。

在下一章中，我们将用一个例子来细化如何制作和使用 Lib 库。

3.6 烧写版本

烧写版本最常用的工具是 Flash_tool，该软件可以从网络下载获得。

该工具的主要设置很简单，分别为 COM 口和目标文件位置。

Download Agent 和 scatterfile 用自带的就可以了，选中这两项后，会出现 ROM 的选择项，单击后可选择.bin 文件。

（1）打开 Flash tool 程序，首先单击菜单“Download Agent”，出现弹出对话框后选择工具自带的 MTK_AllInOne_DA.bin 文件，如图 3.13 所示。

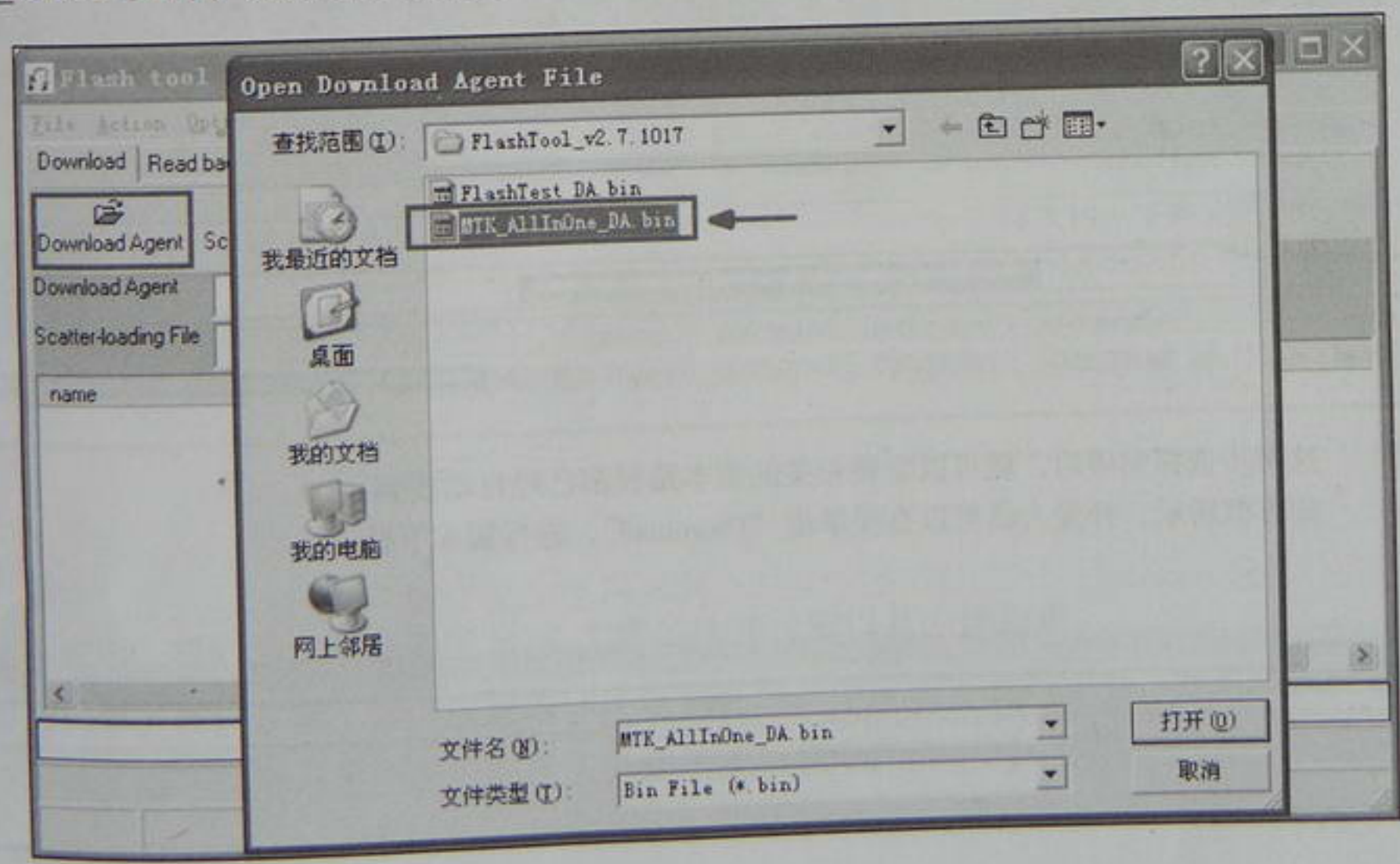


图 3.13　打开 Download Agent

（2）在 Flash_tool 目录中选择 scat。如图 3.14 所示。

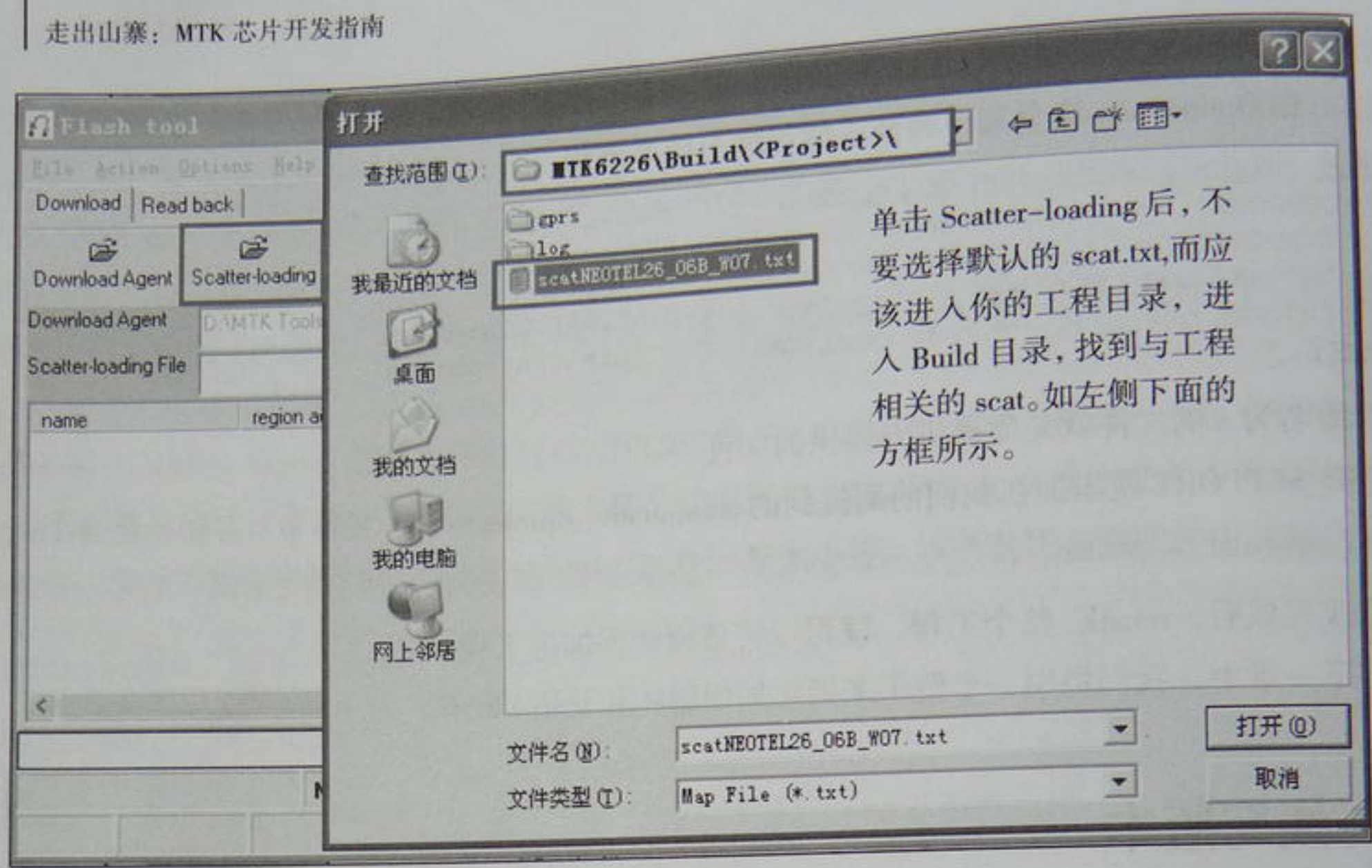


图 3.14　选择 scat

很多朋友可能会说，我直接选择默认的 scat 也可以。但是，建议大家这里就选择 Build 目录里面的 scat，因为这里选好后，单击打开，版本文件的路径就自动显示出来，如下图 3.15 所示。不用再进入目录去找版本，同时，也可以避免忘记选择版本文件而带来的编译错误。

（3）版本路径已就绪，方框中表示工程的具体路径。如图 3.15 所示。

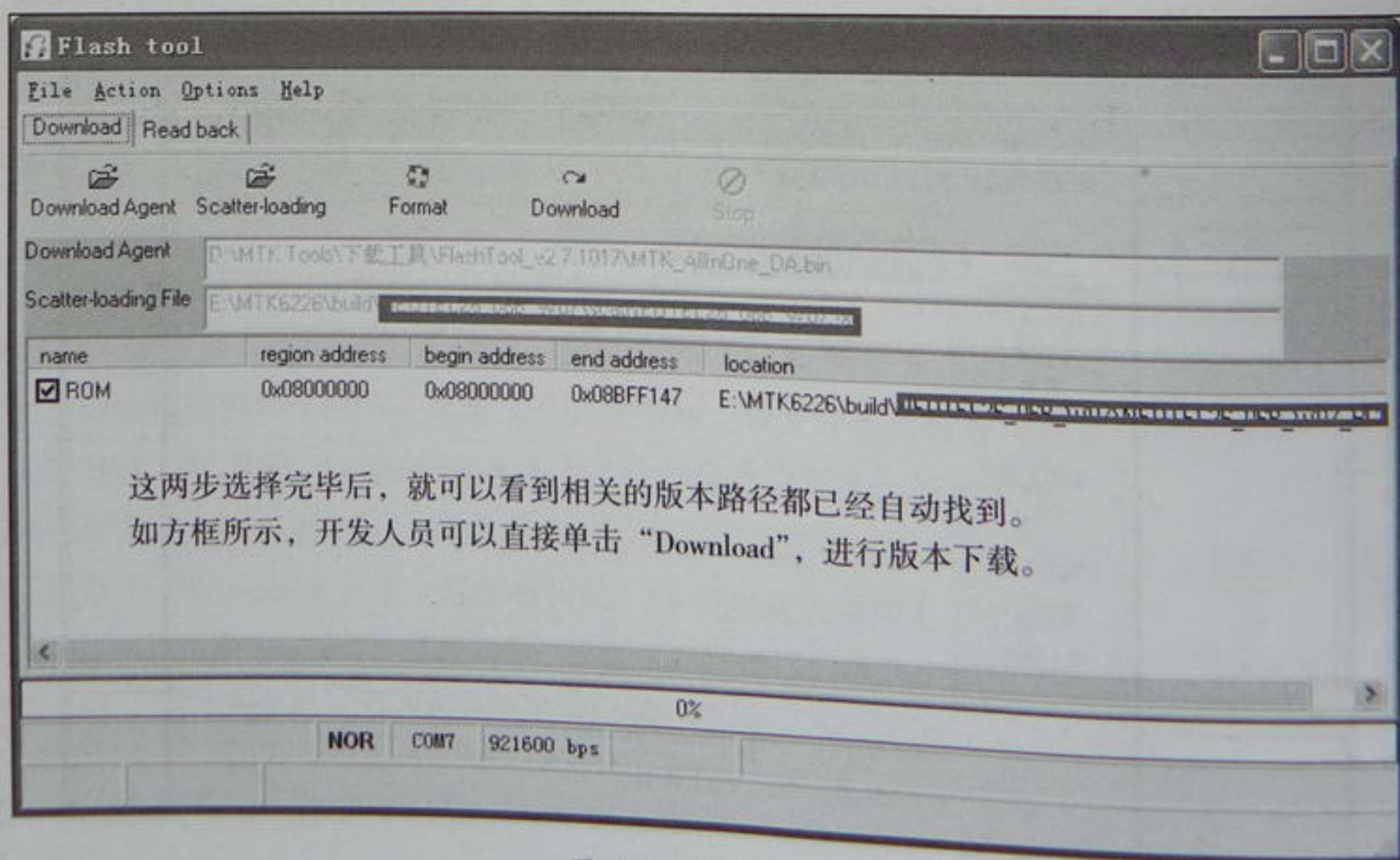


图 3.15　版本就绪

（4）设置 COM 口。如图 3.16 所示。

（5）单击 Download 下载，并长按手机的挂机键直到出现下载进度条。如图 3.17 和图 3.18 所示。

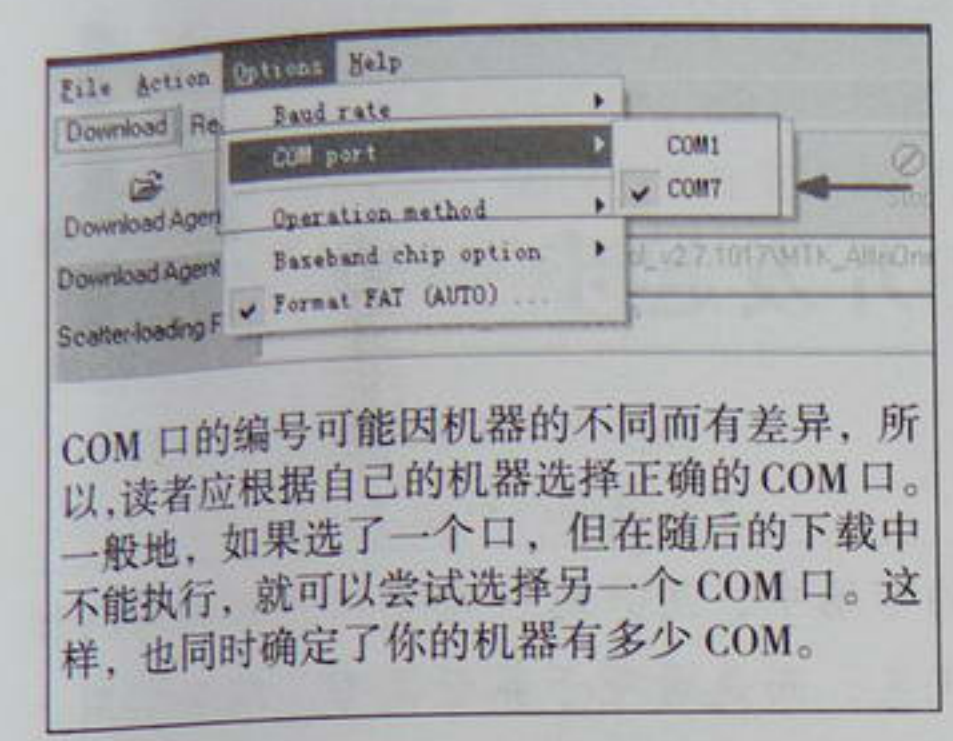


图 3.16　选择 COM 端口

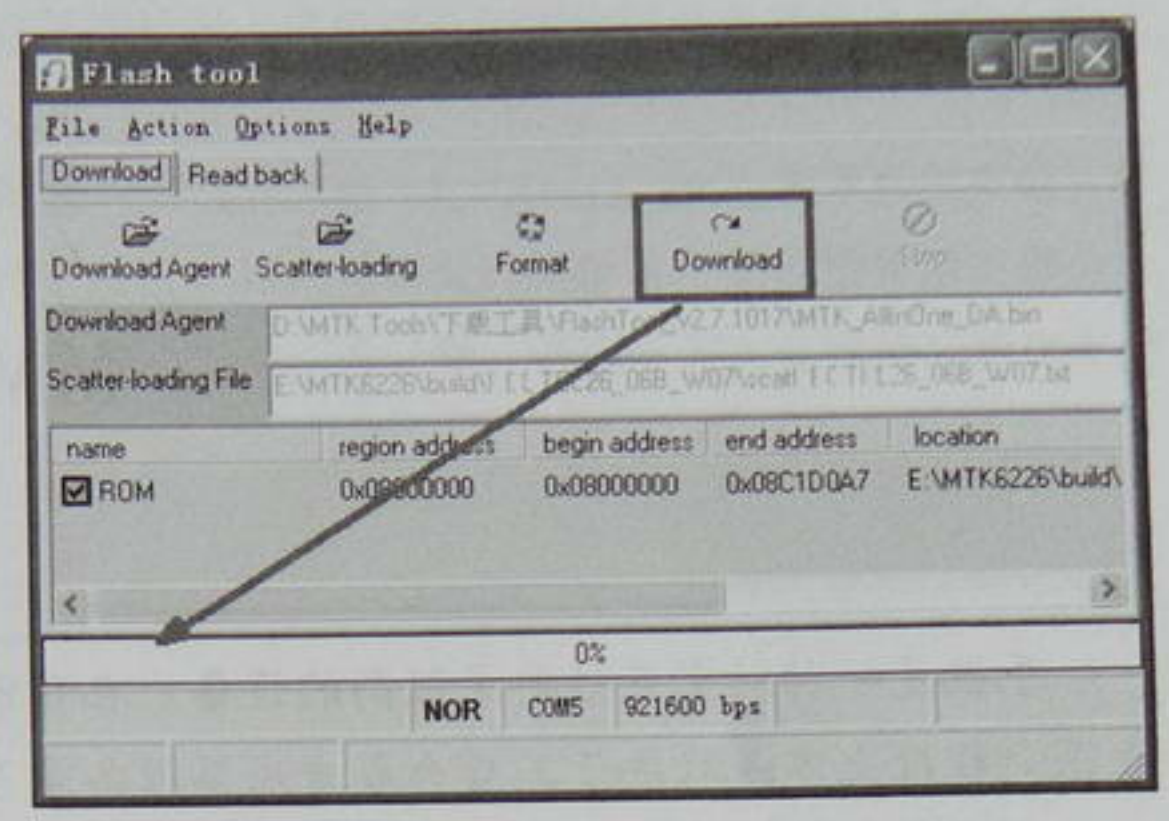


图 3.17　准备下载

结束时，会提示 OK。这时候就可以拔掉下载线，打开真机了。如图 3.19 所示。

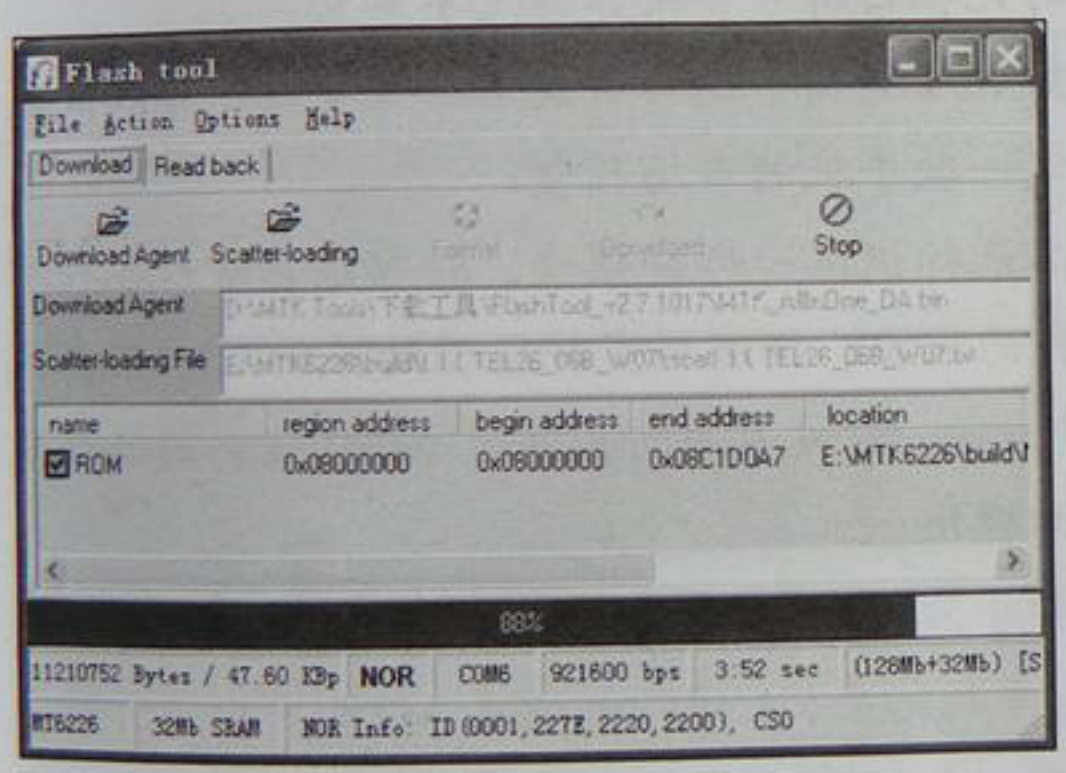


图 3.18　下载过程截图

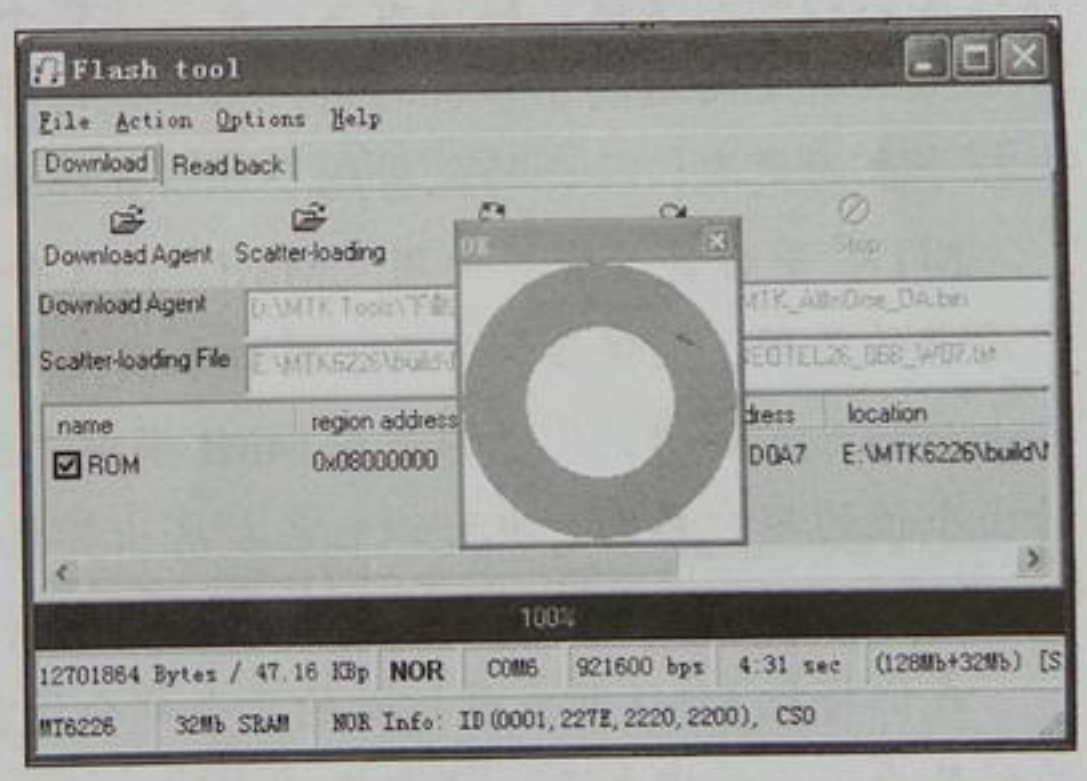


图 3.19　下载完成

提　示

在进行下载操作之前，必须要安装下载线驱动，否则将无法完成上述操作。

结语：

在这一章中，我们为大家分析了 MTK 工程的构建过程以及内部原理。

学习过程中，我们掌握了如何搭建编译环境和在编译中应该注意的细节，并对添加配置第三方库做了一个总结，最后，学习了使用专门的工具烧制真机版本。其中的一些理论还需要在实际的项目开发中深入体会。

4 MMI 开发起步

手机是一种对交互性要求极高的设备。在目前的市场中，用户对一个手机产品的评判标准很大程度上不是取决于其后台或者底层设备是否优良，而是看重于界面是否美观和实用、处理用户请求（交互性）是否敏捷和高效。

试想，用户购买了这样一款手机，用了几天，就发现按键经常无响应，打开一个功能要经过冗长的界面切换，短信丢失，甚至还时不时死机白屏……用户肯定不满意。

上面所述问题属于手机的用户接口设计（MMI）范畴。这部分是非常重要的，可以说是一款手机的灵魂，设计的优劣几乎可以直接决定一款手机的市场销量。

MTK 平台甚至直接使用“MMI”的文件名来标识这一部分模块，该模块负责对各种用户请求的处理，提供各种界面、窗口来满足用户的需要，拥有极富扩展性的界面 UI。只要是做上层的开发应用，几乎都是在 MMI 模块中进行的。开发人员可以基于 MMI 提供的接口设计出个性的用户界面，也可以自己扩展出其他的模块。

在这一章中，我们将通过一个精简的实例初步为读者介绍 MMI。

我们将把 MMI 开发最核心的几个部分提前串联起来，来说明如何在 MTK 平台进行简单的扩展应用。学完本章，读者将对 MMI 开发有一个初步的认识，将明白如何添加资源，如何使用基本的屏幕函数以及如何增加自己的应用模块，并能够对其中重要模块的内部机制有初步的了解。

说　明

MMI（Man Machine Interface，人机接口），是用户和机器交流的通道。分为两种：一种为 UNIX 或 DOS 的命令行界面，一种就是大家熟知的 Windows 界面及类 Windows 界面。

4.1 MMI 初探

4.1.1 MMI 概念

MMI（Man Machine Interface，人机接口），就手机领域来说，就是进行移动通信的人与提供移动通

信的手机之间交互的界面，包括硬件和软件。打个比方，当你在手机的主界面中点了一个日历图标时，系统就会马上转移到日历的界面中去；你单击音乐播放器，手机就会调用底层的音频解码函数，播放音频文件。这个过程，就是由 MMI 来完成的。

在人和机器的互动过程中，有一个层面，即我们所说的界面（Interface）。从心理学意义来分，界面可分为感觉（视觉、触觉、听觉等）和情感两个层次。用户界面设计是屏幕产品的重要组成部分，界面设计是一个复杂的有不同学科参与的工程，认知心理学、设计学、语言学等在此都扮演着重要的角色。

用户界面设计的 3 大原则是：

（1）置界面于用户的控制之下；

（2）减少用户的记忆负担；

（3）保持界面风格的一致性。

用户界面设计在工作流程上分为结构设计、交互设计、视觉设计 3 个部分，在本书中主要涉及后两部分。

1. 交互设计（Interactive Design）

在前面提到，交互设计是手机产品中最重要的部分。交互设计的目的是使产品让用户能简单使用，任何产品功能的实现都是通过人和机器的交互来完成的，因此，人的因素应作为设计的核心被体现出来，交互设计的原则如下。

· 有清楚的错误提示。误操作后，系统提供有针对性的提示。

· 让用户控制界面。“下一步”、“完成”，面对不同层次提供多种选择，给不同层次的用户提供多种可能性。

· 允许工作中断。例如用手机写新短信的时候，收到短信或电话，完成后回来仍能够找到刚才正写的新短信，这就是我们后期要提到的 History 机制。

· 使用用户的语言，而非技术的语言。比如提示的时候尽量“平民化”。

· 提供快速反馈。给用户心理上的暗示，避免用户焦急。

· 方便退出。如手机的退出，是按一个键完全退出，还是一层一层地退出，提供两种可能性。

· 导航功能。随时转移功能，很容易从一个功能跳到另外一个功能。

· 让用户知道自己当前的位置，使其作出下一步行动的决定。

2. 视觉设计（Visual Design）

视觉设计要达到用户愉悦使用的目的，视觉设计的原则如下。

· 界面清晰明了。允许用户定制界面。

· 减少短期记忆的负担。让计算机帮助记忆，例如 User Name、Password、IE 浏览器进入界面地址等可以让机器记住。

· 依赖认知而非记忆。如打印图标的记忆、下拉菜单列表中的选择。

· 提供视觉线索。图形符号的视觉的刺激；GUI（图形界面设计）：Where、What、Next Step。

· 提供默认（default）、撤销（undo）、恢复（redo）的功能。

- 提供界面的快捷方式。
- 尽量使用真实世界的比喻方式。如电话、打印机的图标设计，尊重用户以往的使用经验。
- 完善视觉的清晰度。条理清晰；图片、文字的布局和隐喻不要让用户去猜。
- 界面的协调一致。如手机界面按钮排放，左键肯定；右键否定；按内容摆放。
- 同样功能用同样的图形。
- 色彩与内容。整体软件不超过 5 个色系，尽量少用红色、绿色，近似的颜色表示相近的意思。

提　示

上面提到的 MMI 的总体特征都将体现在 MTK 平台的 MMI 模块中。上述内容也是我们为 MTK 平台的人机接口部分进行的预热和熟悉。

4.1.2　MTK- MMI 模块概览

如果读者已经按照前几章的介绍，成功地编译了工程，那么，对应的仿真环境（即模拟器）也就已经正确地生成。下面我们首先打开模拟器。

（1）进入目录 E:\MTK6226\plutommi\mmi。

（2）找到模拟器工程文件图标：PC_Simulator.dsw。

（3）双击该图标，启动 Visual C++ 6.0，紧接着模拟器的 VC 工程被打开。

（4）单击 Build 菜单，这时弹出其子菜单。按下<F7>快捷键开始构建，如图 4.1 所示。

构建完成并没有错误后，单击上面菜单中的 Execute PC_Simulator.exe 命令运行模拟器。如果一切正常，会出现如图 4.2 所示的待机界面，这个界面也是整个手机的最初界面，通过它才能跳转到其他的窗口。

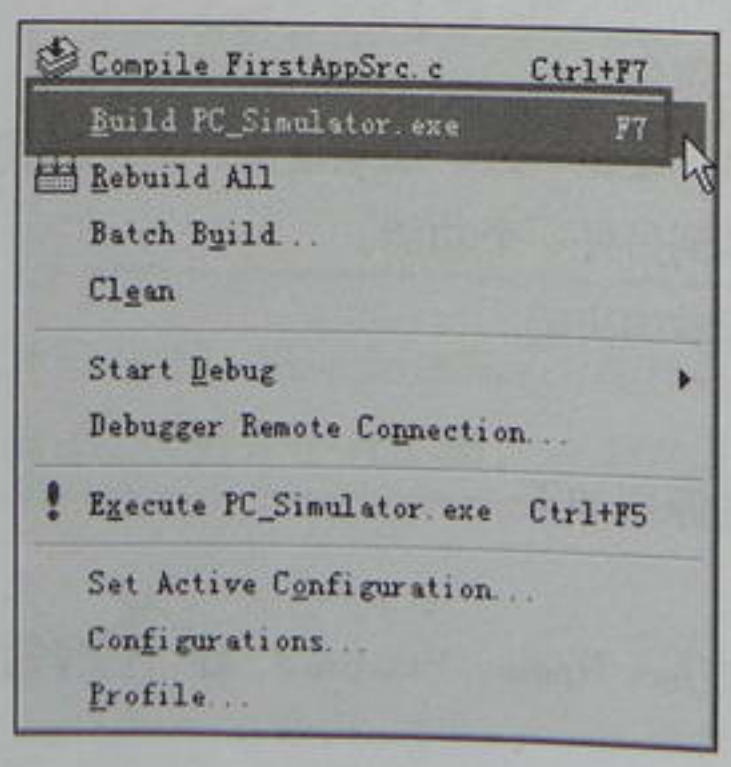


图 4.1　Build 菜单项

图 4.2　原始待机画面

现在，当我们单击上面各个图标时，都会执行图标对应的功能。

如果你之前已经按照第 3 章的介绍，将版本烧制到设备上。那么现在你可以进行如下操作

打开手机，进入待机界面，选择待机界面下方 4 个快捷图标中的第一个图标进入主界面，如图 4.3 所示。

现在，当我们单击 Audio Player 图标时，界面将会跳转到播放器界面，如图 4.4 所示。与此同时，MMI 将会调用底层音频解码函数并驱动喇叭发声（如果设备上有音乐文件）。

图 4.3　原始菜单风格

图 4.4　原始音乐播放器界面

同理，我们单击 Video Player 图标，界面将会跳转到视频播放器界面，同时，MMI 将会调用底层的视频解码函数，并最终渲染在屏幕上。

同时，也可以装上 SIM 卡，进入短消息模块，我们就能够接受/发送短消息。我们进入拨号框界面，就能够拨电话并接听对方来电。这期间 MMI 将会与 L4 等协议栈进行交流。

可以看到，上面的动作不仅仅是界面的绘制和更替，还涉及上层应用如何与底层设备进行通信、模块与模块之间的交流。这一系列的动作组成了 MMI 的架构体系。

为了简单起见，我们可以对其进行简化，如图 4.5 所示。

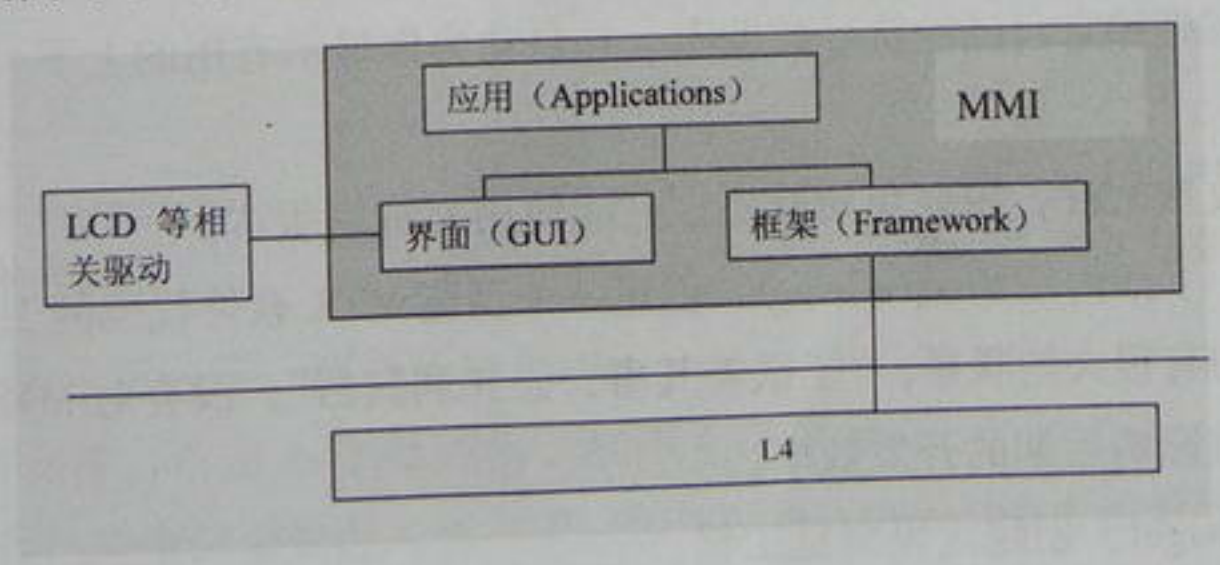


图 4.5　MMI 架构示意图

从图中我们可以看到，MMI 所做的工作不仅仅是 UI 方面，还包括其他的诸如任务结构、L4 通信等方面的内容。

在上图中，我们可以看到 MMI 中 3 个最重要的组成部分，该部分在第 2 章已经有过讲解。

1. Applications

包括系统自带的各种应用程序，例如 Alarm、canlender。也包括开发人员后期开发的应用程序逻辑

结构。

2. Framwork

OLS：操作系统适配置层，对操作系统进行封装。

Task：MMI 任务，与 L4 task 进行通信。

File system：文件系统，与存储设备通信，进行文件存取。

3. GUI

Theme：主题风格，主要是菜单等颜色设置，以及背景图片等。

UI component：UI 组件，即大量的图形控件。

Category Screen：模板屏幕，为系统提供大量的可重用的屏幕模板。

Font：字体风格。

InputMethod：输入法。

MCT：MMI 资源定制工具。

和大多数平台开发一样，图形界面的开发设计将会是开发人员最先了解和掌握的部分。因此，本书也将从 MMI 的 GUI 部分开始，逐步过渡到 MMI 的后台高级课程。

4.2 如何浏览代码

MMI 平台和其他开发平台诸如 Symbian、VC 不同，其开发过程混合着其平台本身的源代码，数量庞大，头绪繁杂，必须使用有效的代码编辑工具来管理。在这里，推荐大家使用 Source Insight 3.5 版，如图 4.6 所示。

图 4.6 Source Insight

Source Insight 是一个面向项目开发的程序编辑器和代码浏览器，它拥有内置的对 C/C++、C#和 Java 等程序的分析。Source Insight 能分析用户的工程源代码并在工作的同时动态维护它自己的符号数据库，并自动为你显示有用的上下文信息。

不过，该软件是收费软件，想要免费使用，只能使用其试用版本。

下面就给大家讲一下如何用 Source Insight 创建一个浏览 MTK 软件代码的工程。这一节看似与 MMI 的讲解无关，实际上却有很大的联系，“工欲善其事，必先利其器”。没有对 MMI 代码进行合理和高效的整理和归纳，将严重影响后期的开发效率。

使用 SI（Source Insight）创建代码浏览工程。

（1）打开 Source Insight。如图 4.7 所示。

（2）选择 Project 选项卡，选择 New Project 命令。如图 4.8 所示。

（3）给工程取名，并选择工程信息文件的存储位置。如图 4.9 所示。

小技巧

SI 为 MTK 代码创建的工程会产生一个.pr 的文件，推荐将其放置在整个 MTK 工程文件夹下面，方便管理。

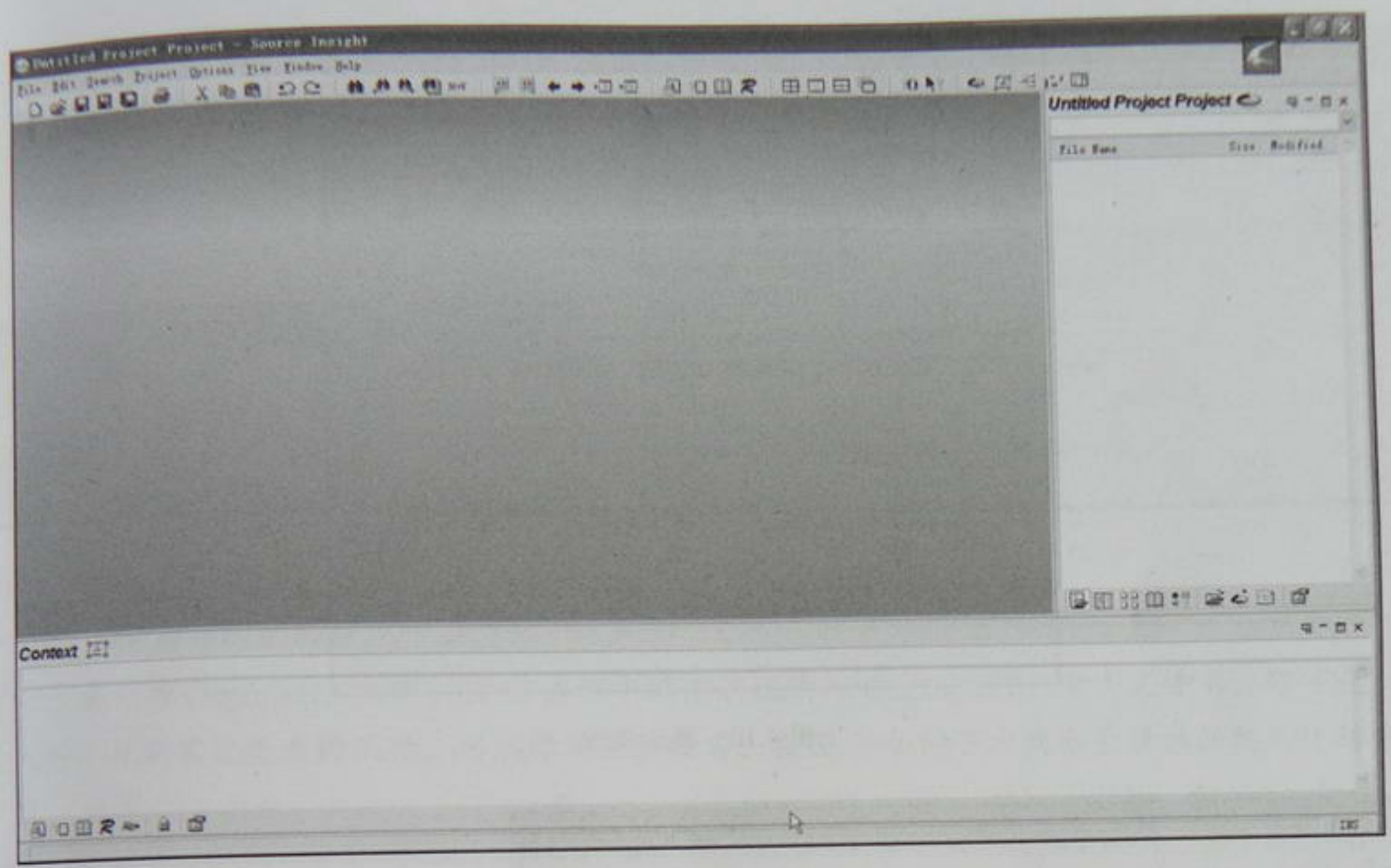


图 4.7 Source Insight 启动界面

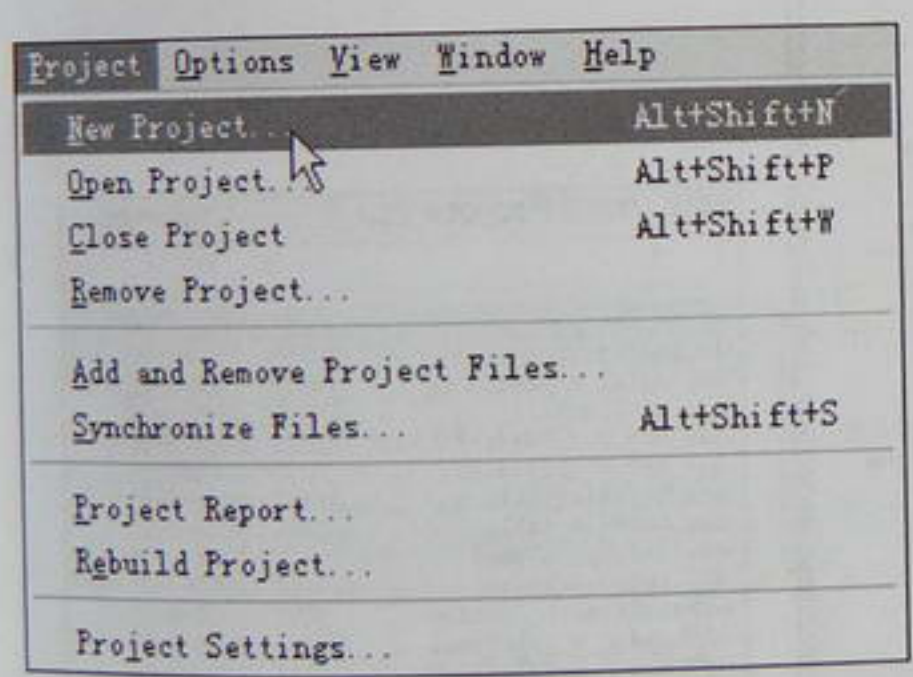


图 4.8 添加新工程

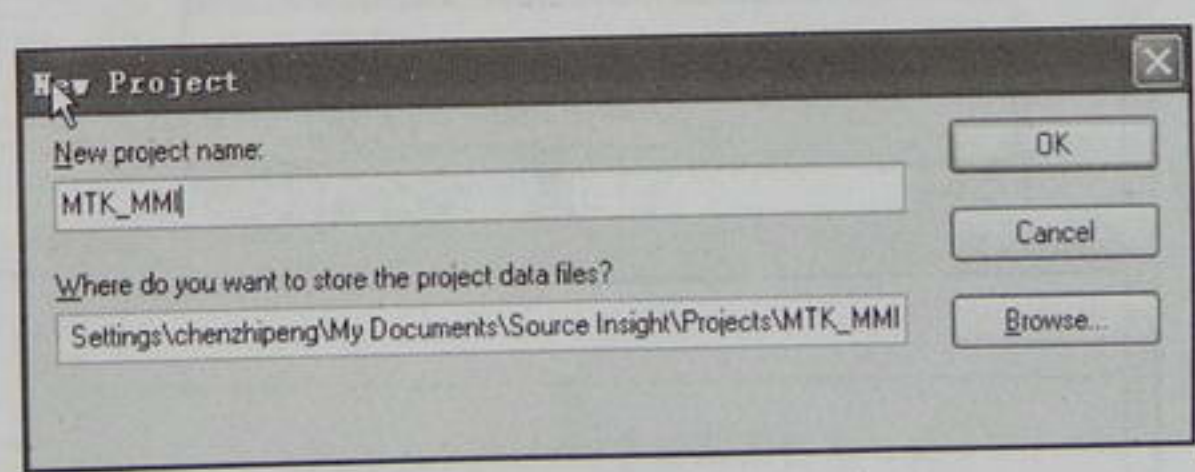


图 4.9 给工程取名

（4）选择 OK 默认进入下一步。如图 4.10 所示。

（5）选中你的项目文件，单击 Add all 按钮。如图 4.11 所示。

（6）这样，工程文件已经全部添加进入 SI 工程，以后就可以方便地查看编辑代码了。如图 4.12 所示。

（7）刚添加进来的文件需要进行一个同步操作，使得 SI 能够正确地扫描各个函数的上下文，为将来的浏览做准备。如图 4.13 所示。

说　明

MTK 项目文件很多，而且各个文件之间的联系很复杂。必须进行同步操作以后，才能够快速地定位一个函数或者一个变量所定义和使用的地方。否则，所有函数都将是灰色，表明 SI 还没有找到任何跟此函数相关的上下文。

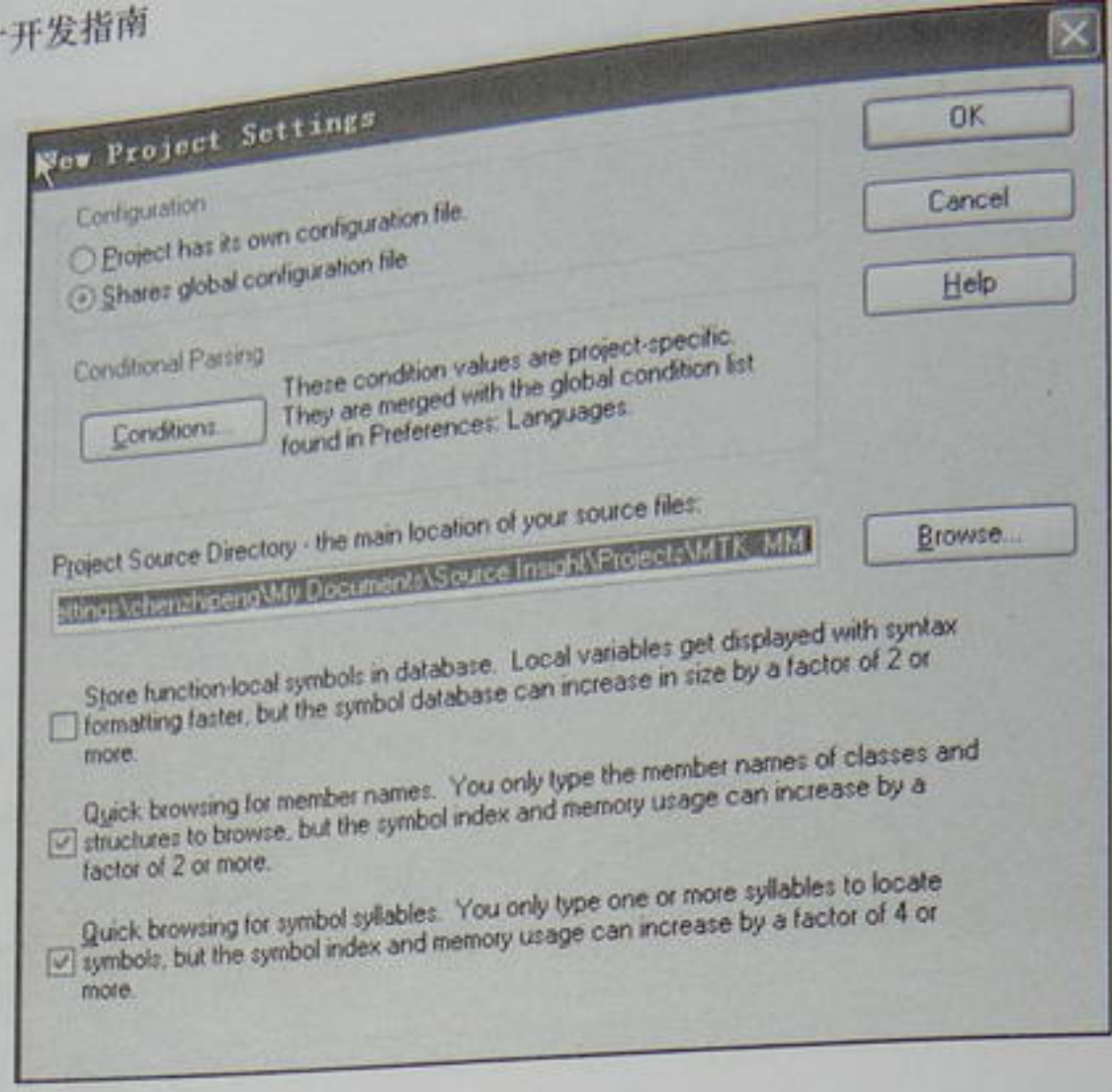


图 4.10　默认操作

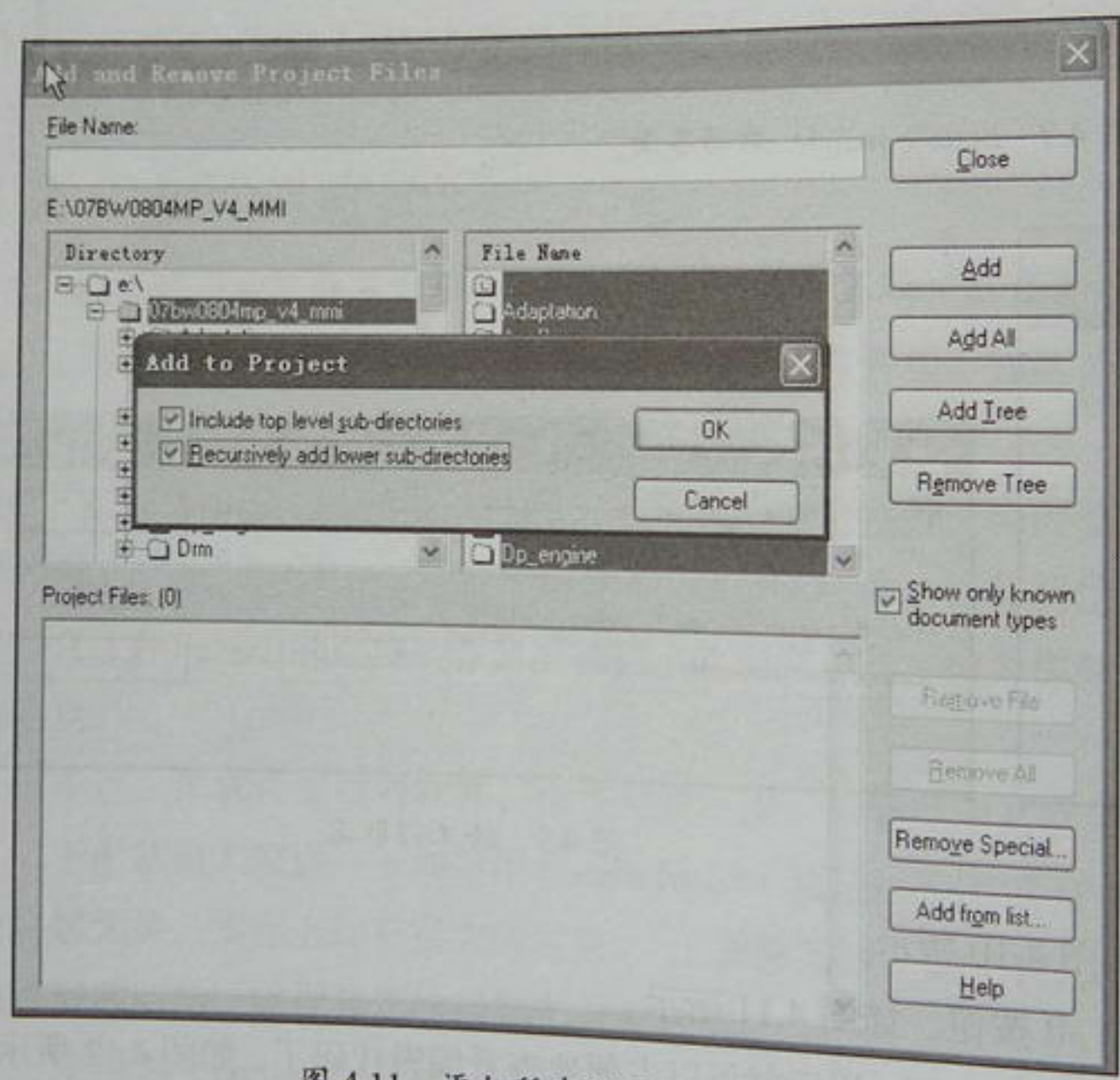


图 4.11　添加所有的项目文件

图 4.12　文件浏览窗口

（8）要查找一个变量所出现的位置或者其他，Search 功能太重要了。如图 4.14 所示。

（9）搜索完毕，会出现一个 result 结果窗口，当你需要重新 search 的时候，如果当前 result 窗口已有搜索内容，会有如图 4.15 所示的窗口出现。根据你的情况而定，你可以选择 replace，用新的搜索内容替换原有的，也可以选择 Append 将当前搜索结果附加于旧的内容之后。

到此为止，建立工程的基本步骤就讲完了。所有的项目文件都已加入工程，下一节开始就是我们开始实践的时候了。

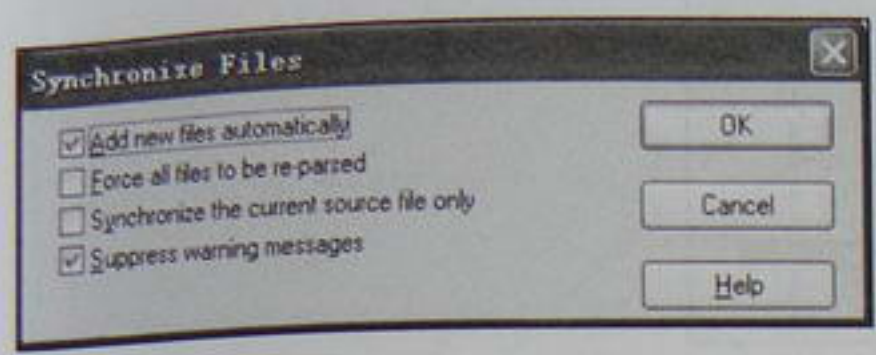


图 4.13　同步文件

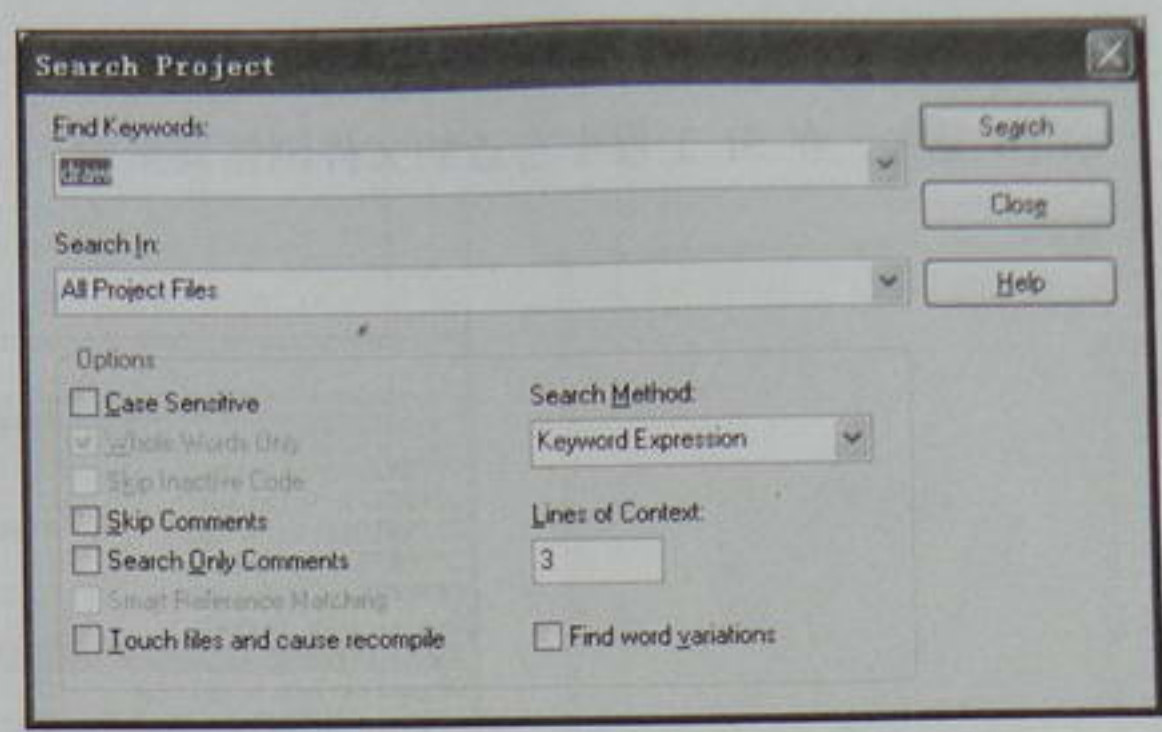


图 4.14　搜索界面

如何查看函数信息

我们以 Organizer.c 为例，在 SI 工程中的右边栏输入 Organizer.c，单击文件名，即可进入文件。找到需要查看的函数，用鼠标单击函数名，在 SI 界面的下方就会出现该函数的详细信息。如图 4.16 所示。

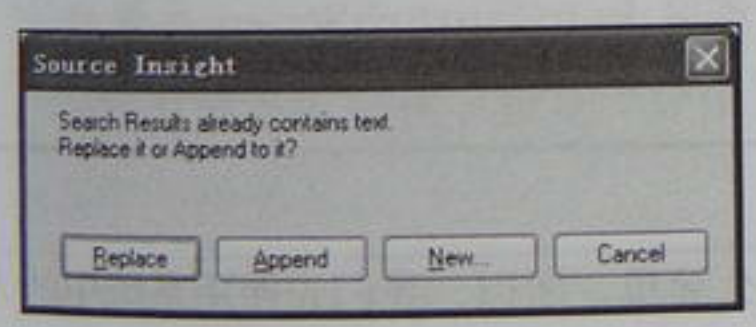


图 4.15　对搜索结果的处理

图 4.16　如何查看函数信息

4.3 Hello MTK

和其他新的语言开发一样，我们的程序也从“Hello MTK”开始。

当打开整个工程的软件目录时，是不是很茫然？这一大片代码，该怎样入手呢？

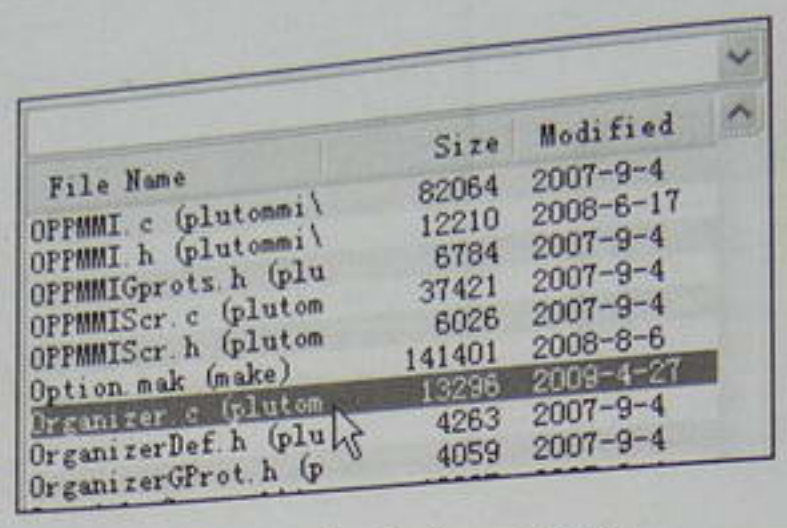
前面已经讲过，MTK 是一个耦合性很高的平台，很难做到使自己的程序与平台完全无关，很多时候程序都是和整个系统固定在一起的。因此，我们要做的就类似于在一个大房子里搞装修，需要在这个大程序中添加一个新入口，以便让代码能够融入这个“大家庭”。

在添加和修改代码之前，我们首先要确定在哪个地方进行试验。

在 SI 工程的文件浏览器中，打开文件 Organizer.c。该文件是主菜单中的工具箱实现代码。我们选择上述文件，主要是因为其内部模块相对独立，任何改动对整个系统的影响较小，再者笔者当初做开

发的时候，也是从该文件开始的。

（1）首先，在 SI 工程中右边的文件浏览器里输入 Organizer.c，然后双击文件名打开。如图 4.17 所示。



File Name	Size	Modified
OPPMMI.c (plutommi\	82064	2007-9-4
OPPMMI.h (plutommi\	12210	2008-6-17
OPPMMIGprots.h (plu	6784	2007-9-4
OPPMMIScr.c (plutom	37421	2007-9-4
OPPMMIScr.h (plutom	6026	2007-9-4
Option.mak (make)	141401	2008-8-6
Organizer.c (plutom	13296	2008-4-27
OrganizerDef.h (plu	4263	2007-9-4
OrganizerGProt.h (p	4059	2007-9-4

图 4.17　打开 Organizer.c

（2）增加第一个函数，取名为 EntryFirstAppScreen，该函数主要用来显示一个字符串。

```
void EntryFirstAppScreen(void)
{
entry_full_screen();//全屏幕
  clear_screen();//将屏幕清成空白
  gui_set_text_color(gui_color(255,0,0));//设置文本颜色为红色
  gui_move_text_cursor(100,100);//设置文本绘制的起始坐标
  gui_print_text(L"Hello MTK!");//打印字符串
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
}
```

注　意

如果 EntryFirstAppScreen 在文件头部没有做前置声明，那么 EntryFirstAppScreen 函数应该写在调用此函数之前，否则会提示该函数找不到。

（3）有了自己的函数，还需要系统的调用。

C/C++程序都有一个 main 函数，但是，在 MTK 平台下没有这么简单。所以，我们先使用系统中的入口函数，等到我们自己的程序入口做好，再改回来。

找到函数 void EntryOrganizer(void)，屏蔽其内部代码，如图 4.18 所示。

```
void EntryOrganizer(void)
{
  EntryFirstAppScreen();
#if 0
   …系统本身代码
#endif
}
```

保存所做修改，在 VC 工程中再次选择 Build，构建完成后，选择<F5>或单击“!”号菜单执行模拟器。

进入主菜单页面后，当你单击 Organizer 图标的时候，系统就会进入我们刚才写的程序中，输出一个白色的屏幕，上面写有“Hello MTK!”。

运行结果如图 4.19 所示。

```
void EntryFirstAppScreen(void)
{
  entry_full_screen();
  clear_screen();//将屏幕清成空白
  gui_set_text_color(gui_color(255,0,0));//设置文本颜色为红色
  gui_move_text_cursor(100,100);//设置字符串显示的坐标
  gui_print_text(L"Hello MTK! ");//打印字符串
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕

}

/*****************************************************************
 * FUNCTION
 *  EntryOrganizer
 * DESCRIPTION
 *  Entry Organizer menu screen
 * PARAMETERS
 *  void
 * RETURNS
 *  void
 *****************************************************************/
void EntryOrganizer(void)
{
  EntryFirstAppScreen();
#if 0
  /*----------------------------------------------------------------*/
  /* Local Variables                                                */
  /*----------------------------------------------------------------*/
```

现在让Organizer的入口函数调用我们新增的函数EntryFirstAppScreen()。EntryOrganizer内部代码暂时屏蔽。

图 4.18　调用新增的函数

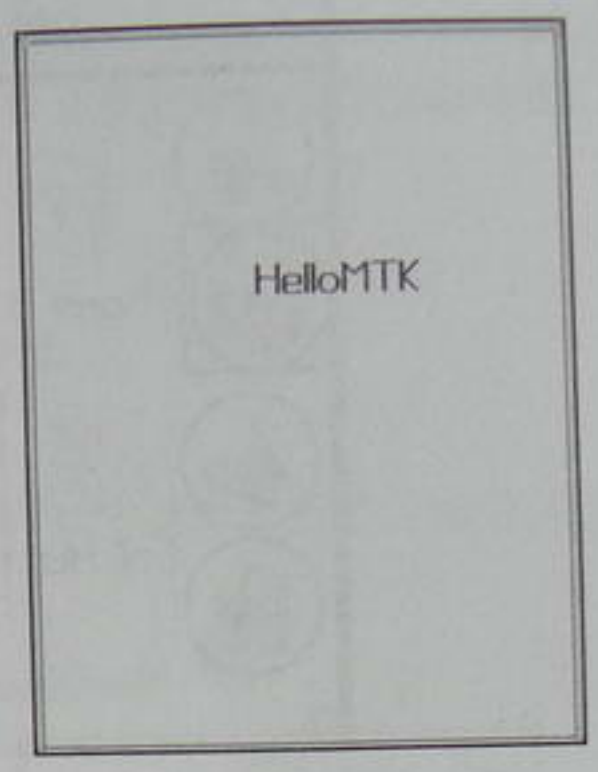


图 4.19　输出结果

这就是我们要输出的内容。

> **说　明**
>
> 虽然我们可以输出一个字符串，但是缺点很明显。读者可以耐心地等几秒钟，你会发现屏幕上多了一些小图标，白色的背景被打乱。或者，你可以用鼠标在刚刚生成的屏幕上乱点，还可能出现离开当前界面跳转到其他屏幕去的情况。这是由于我们的程序是直接在当前的主菜单屏幕中绘制的，是和其他的一些系统程序共用一个屏幕，所以，当其他的程序有变化时就会刷新当前屏幕，从而影响到自己程序的绘制。
>
> 这当然令人沮丧，但是“方法总比问题多”。接下来的一节就来讲解如何为新增的程序添加其专用的屏幕。

4.4　屏幕起步

4.4.1　什么是屏幕

做过 UI 界面开发的朋友都清楚，只要是与用户打交道的功能模块，都需要一个新的窗口用来放置各种交互控件。MMI 开发也不例外。新的应用程序只要不是在后台运行，都需要创建一个新的屏幕。

> **说　明**
>
> 在 MMI 中，界面基本上都是独占的，就是说其布满整个手机显示屏，我们将其称为“Screen”，即屏幕。

现在让我们进入工具箱（Organizer）的列表屏幕。

首先选择主界面左边纵列的第 2 个图标，如图 4.20 所示。

然后单击 Organizer 图标，就跳转到其列表屏幕，如图 4.21 所示。

图 4.20　选中 Organizer 菜单

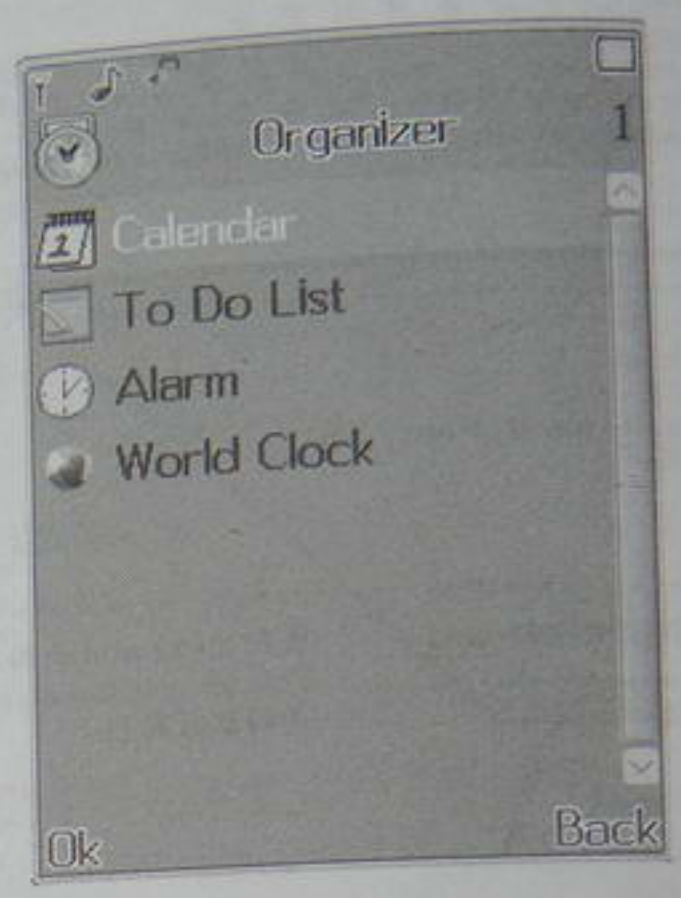


图 4.21　进入列表界面

这些界面就是一个单独的屏幕，整个手机系统就是通过不同的屏幕来响应用户的请求。

4.4.2　如何创建新的屏幕

浏览 MMI 的代码，你会发现有很多名字叫“EntryNewScreen”的函数。顾名思义，这就是我们这一节要重点讲的新屏幕函数。这个函数用途很广，可以说是 MMI 的基石函数之一。

EntryNewScreen 函数原型如下：

```
U8 EntryNewScreen (U16 newscrnID, FuncPtr newExitHandler, FuncPtr newEntryHandler, void *flag)
```

参数 newscrnID：屏幕 ID 的惟一标识。每一个屏幕对应一个 ID，方便系统管理。

参数 newExitHandler：屏幕的退出函数。系统在退出当前屏幕的时候会自动调用该函数，可以在里面做一些资源的释放工作，类似于 C++中类的析构函数。

参数 newEntryHandler：屏幕的入口函数。只有传入此参数，系统才会将屏幕加入历史，历史机制见下一节。

参数 flag：一般都设置为 NULL。目前暂时没有使用。

到此，了解了这个函数名，我们的程序就先借用 Organizer 的屏幕 ID。

```
void EntryFirstAppScreen (void)
{
  EntryNewScreen( ORGANIZER_SCREENID, NULL,NULL,NULL);
  entry_full_screen();//全屏幕
  clear_screen();//将屏幕清成空白
  gui_set_text_color(gui_color(255,0,0));//设置文本颜色为红色
  gui_move_text_cursor(100,100);//设置文本绘制的起始坐标
  gui_print_text(L"Hello MTK!");//打印字符串
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
}
```

> **提 示**
>
> 暂时借用 Organizer 的屏幕 IDORGANIZER_SCREENID，我们将在随后介绍如何为新程序模块增加屏幕 ID，到时我们再修改回来。

保存修改，运行模拟器。现在可以放心地使用程序界面了，再也不会有其他的屏幕刷新来影响我们的程序界面。

这样是不是就没有问题了？让我们来做一个试验。

在 Network Simulator 中单击 Incoming Call Indiacation 按钮，如图 4.22 所示。

接着，系统将会跳转到来电显示界面，如图 4.23 所示。

这时选择 Reject，你会发现我们原来输出字符串的屏幕消失，而是跳转到主菜单界面了，如图 4.24 所示。

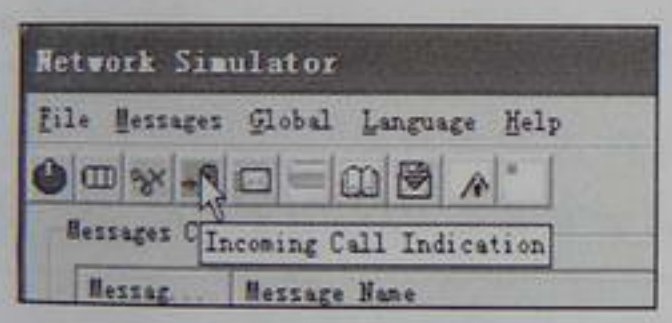


图 4.22 模拟来电事件

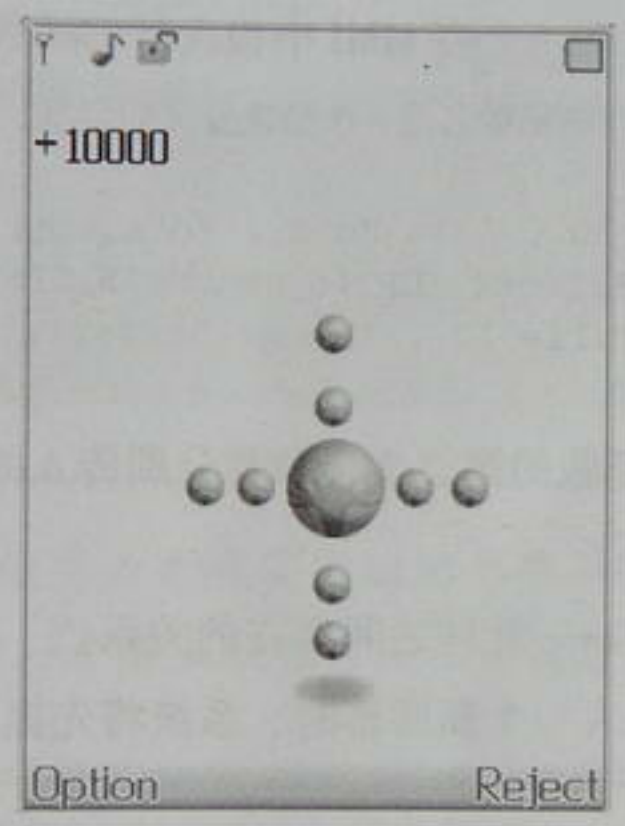


图 4.23 来电显示界面

图 4.24 回到主界面

结果令人沮丧，难道系统不知道之前做的事情吗？MMI 有没有一种机制来进行管理？

答案是肯定的。接下来将介绍很重要的 MMI 屏幕管理机制，即 History 机制。

4.4.3 History 机制

手机中的屏幕流程都是顺序的，用户都是通过按键在屏幕之间切换。在向前切换的过程中，在绘制新的屏幕之前，先前屏幕的内容存储在 History“数据库”中。如果当前的屏幕用于接收用户的输入，而此时一个异步事件发生例如来电，那么当前用户输入内容的缓冲区将会被存在 History 中，当前屏幕就会显示当前事件屏幕，直到该事件完成才恢复先前的屏幕。

正因为有了上述机制，我们才可以放心地在新屏幕上进行操作而不用担心自己的屏幕出现异常。有 3 种类型的屏幕切换，如下所示。

（1）向前移动：向下一个屏幕的切换作为用户按键输入的结果。

（2）向后移动：向前一个屏幕的切换作为用户按键输入的结果。

（3）异步事件的出现：屏幕向前或者向后切换依赖于异步事件，例如协议事件。

History 的实现类似一个栈，一个新的 History 节点添加在这个栈的顶端。

每一个 History 节点的结构如下所示：

```
typedef struct _history
{
    U16 scrn /* 将要存储的屏幕 ID*/
    FuncPtr entryFuncPtr; /* 入口函数指针，用于恢复屏幕时的重绘*/
    U8 inputBuffer[MAX_INPUT_BUFFER]; /* 用于存储当前的 input 缓冲区*/
    U8 guiBuffer[MAX_GUI_BUFFER]; /* 存储 GUI，一些图片数据*/
}
```

应用步骤如下。

（1）保存窗体的数据到历史中去，这个数据的内容将在下面提到。

（2）返回到 N 级历史，并重画当时的窗体。

（3）从历史的栈顶删除 N 个节点。

为了表述得更加直观，我们来看一个在 MMI 中很流行的函数，如下所示：

```
void EntryFunc()        //该函数名是一个抽象意义的名字，只做演示用
{
    EntryNewScreen( Screen_ID , Exit_Func , EntryFunc , NULL );
    guiBuffer = GetCurrGuiBuffer( SCR_ID_WORDMAIN_LIST );
    ShowCategroyXXScreen( Title_ID , … , guiBuffer);
}
```

我们注意到 EntryNewScreen 函数的第 2、3 个参数分别传入的是一个退出函数指针和入口函数指针。那么，两个参数究竟起什么作用呢?

现在来讨论以上的代码与 History 管理之间存在的关系。

在 MTK 环境中，每当我们进入一个新屏幕时，系统将先提取前一个屏幕需保留的数据，这些数据包括：

（1）屏幕 ID；

（2）进入屏幕时调用的函数和退出调用的函数，Exit_Func 和 Entry_Func ；

（3）组成窗体的控件的属性（如列表控件当前高亮显示的条目、当前屏的首末条目等）。通过跟踪调试程序可以看到这些动作是在每次调用 EntryNewScreen 函数的时候实现的。

具体的实现过程如下。

（1）EntryNewScreen 函数执行时，系统会先将前面所记录的 currExitScrnID、currEntryFuncPtr 以 History 结构为载体记录入栈。

（2）然后调用 EntryNewScreen 函数执行记录中的 currExitFuncPtr（就是前一个屏幕退出时需要做的工作）操作；

（3）最后将本窗口的 scrnID、exitFuncPtr、entryFuncPtr 分别记录入全局变量 currExitScrnID、currExitFuncPtr 和 currEntryFuncPtr，留待下次调用 EntryNewScreen 函数时使用。

上面的过程周而复始，只要有屏幕更替，就离不开这 3 个步骤。

提　示

上面提到的 currExitScrnID、currEntryFuncPtr 等变量都是函数指针，它们都将在屏幕切换过程中正确地指向某个函数。关于函数指针更深层的内容，请参考相关书籍。

我们再通过一个图例来说明这些数据在实际中是如何被使用的。

假设存在 A、B 两个窗口，A 窗口需要保留的数据为 data_A，我们先从 A 窗口进入到 B 窗口，data_A 将在 B 窗口调用 EntryNewScreen 函数的时候，被压入一个结构类似于栈的数据存储区域，如图 4.25 所示。

当从 B 调用 GoBackHistory 函数返回 A 时，data_A 从栈顶被弹出，然后 A 利用 data_A 将自身还原到其进入 B 之前的状态。如图 4.26 所示。

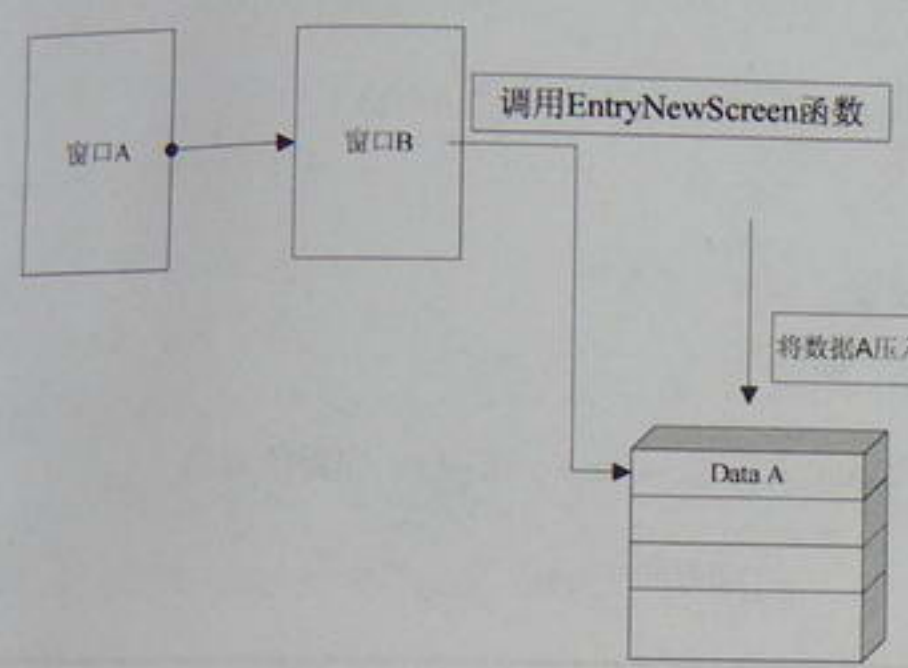


图 4.25 屏幕数据入栈

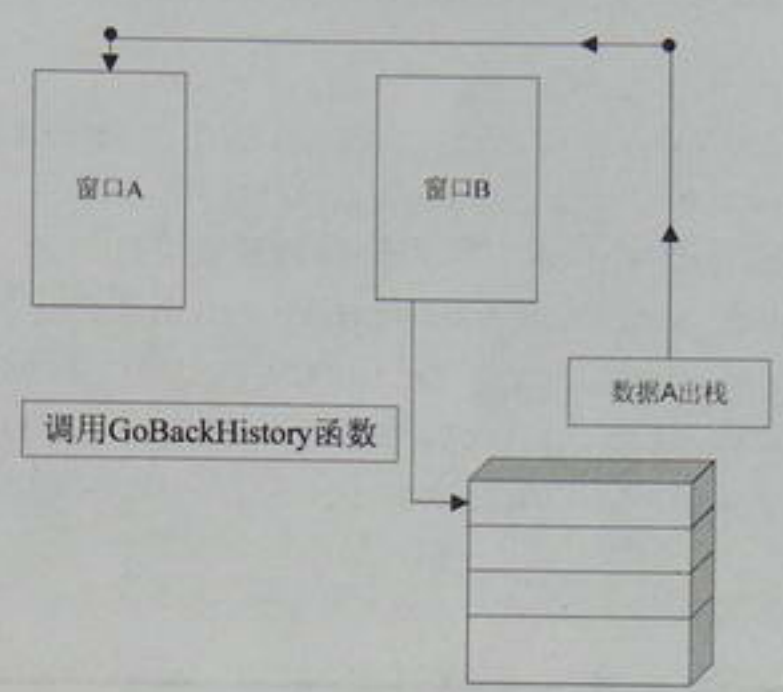


图 4.26 屏幕数据出栈

提 示

调用函数 GoBackHistory，将使得历史栈最顶端的屏幕出栈，而这个顶端屏幕正是之前 EntryNewScreen 函数中压入栈的前一个屏幕数据，所以，通过 GoBackHistory 函数就能成功地返回到本屏幕的前一个屏幕。

这就是 History 管理的作用。现在将我们的屏幕加入历史管理，代码如下：

```
void EntryFirstAppScreen (void)
{
  //请注意，自动加入历史时，第 2 个参数为 NULL
EntryNewScreen( ORGANIZER_SCREENID, NULL, EntryFirstAppScreen,NULL);
  entry_full_screen();//全屏幕
  clear_screen();//将屏幕清成空白
  gui_set_text_color(gui_color(255,0,0));//设置文本颜色为红色
  gui_move_text_cursor(100,100);//设置文本绘制的起始坐标
  gui_print_text(L"Hello MTK!");//打印字符串
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
}
```

前一节已经讲过，只有第 3 个数是屏幕入口函数，系统才会将该屏幕加入历史。所以将 EntryFirstAppScreen 作为参数传入 EntryNewScreen，标识了从其他屏幕返回时，正确进入 EntryFirstAppScreen，而不会出现前面的直接返回待机界面的尴尬。

如果不想让系统自动将屏幕加入历史，也可以自己手动实现，代码如下：

```
Void ExitFirstAppScreen()
{
history currHistory;
```

```
currHistory.scrnID = ORGANIZER_SCREENID;
currHistory.entryFuncPtr = EntryFirstAppScreen;
GetCategoryHistory(currHistory.guiBuffer);
pfnUnicodeStrcpy((S8*) currHistory.inputBuffer, (S8*) L"");
AddHistory(currHistory);
}
```

同时 EntryNewScreen 的参数也有变化：

```
void ExitFirstAppScreen (void)
{
  EntryNewScreen( ORGANIZER_SCREENID, ExitFirstAppScreen, NULL,NULL);
  entry_full_screen();//全屏幕
  clear_screen();//将屏幕清成空白
  gui_set_text_color(gui_color(255,0,0));//设置文本颜色为红色
  gui_move_text_cursor(100,100);//设置文本绘制的起始坐标
  gui_print_text(L"Hello MTK!");//打印字符串
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕

}
```

注 意

如果函数 EntryNewScreen 的第 2 个参数是你自己定义的退出函数，并且已经传参，那么第 3 个参数请务必为 NULL，请一定注意该细节。

以上面的代码为例，因为自定义了退出函数 ExitFirstAppScreen，就表示你将选择手动将屏幕加入历史，而这个时候，你又将第 3 个参数设置为 EntryNewScreen，又表明了你选择系统自动将屏幕加入历史。那么当屏幕进行切换的时候，系统仍将执行自动加入历史的动作。

与此同时，如果开发人员在进入新屏幕的时候动态分配了内存，那么最好创建自己的退出函数（也就是手动将屏幕加入历史管理），并在退出函数中进行内存释放操作。

至此，屏幕有了历史机制的管理，基本上是可以正常运行的程序了。现在，我们进入 FirstAppScreen 屏幕就是我们的 Hello MTK 界面，查看完毕后，单击右软键，发现无法退出。经过思考发现，右软键没有绑定退出函数，所以，我们马上为 EntryFirstApp 添加退出响应函数。

```
void EntryFirstAppScreen (void)
{
  EntryNewScreen( ORGANIZER_SCREENID, ExitFirstAppScreen, EntryFirstAppScreen,NULL);
  entry_full_screen();//全屏幕
  clear_screen();//将屏幕清成空白
  gui_set_text_color(gui_color(255,0,0));//设置文本颜色为红色
  gui_move_text_cursor(100,100);//设置文本绘制的起始坐标
  gui_print_text(L"Hello MTK!");//打印字符串
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP)注册了右键（KEY_RSK）的响应函数。从函数的名字就可以看到，只要单击右键，程序就会返回到前一个屏幕。

现在，我们再次进入新增的屏幕，按照上一节的方法，在模拟器上出现来电界面时，仍然选择 Reject。这时，屏幕成功地回到了刚刚我们运行的“HelloMTK!”界面。程序可以流畅地运行了。

4.5 规范化

浏览 MTK 提供的软件代码，你会发现整个平台软件代码都是由许许多多独立的模块组合起来的。他们有着大同小异的代码组织结构。

我们以 PhoneBook 为例，其代码组织结构和经典的 C 语言相同，分为 PhoneBookInc 和 PhoneBookSrc。

PhoneBookInc 目录下有图 4.27 所示的文件。

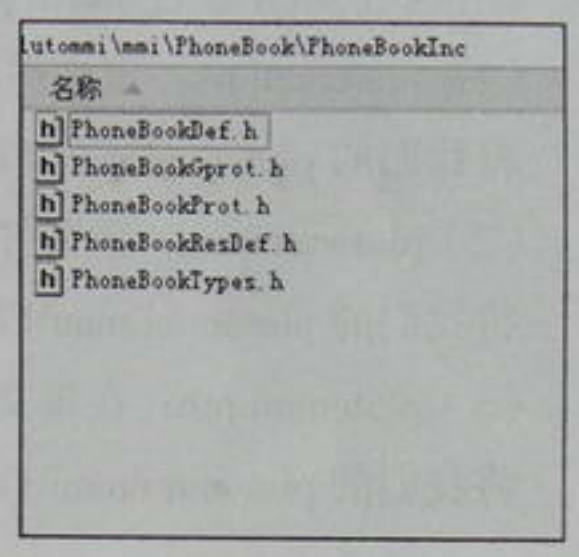


图 4.27 PhoneBook 头文件内容

PhoneBookDef.h 用于存放本程序模块的资源 ID，例如字符串、图片等，将会在下一节详细介绍。

PhoneBookGprot.h 提供函数声明，其内部很多函数都有 extern 关键字，说明这里面的函数全是提供给外部使用的。

PhoneBookProt.h 提供函数声明，但是只能提供给本程序的代码调用。

PhoneBookTypes.h 提供结构体，常量的定义。

PhoneBookSrc 目录下就是模块的实现文件。

以上的代码组织结构在 MTK 整个软件代码中是最常见的，所以，我们写的程序模块也需要遵循其所制定的规则。

4.5.1 确定目录结构及模块相关文件

既然是在为 MMI 添砖加瓦，那么我们的程序模块也就放在/plutommi/mmi 目录下，取名为 FirstApp。按照上述规则，我们模块的目录结构如下：

/plutommi/FirstApp：模块总目录；

/plutommi/mmi/ FirstApp / FirstAppInc：模块头文件目录；

/plutommi/mmi/ FirstApp / FirstAppSrc：模块源文件目录。

如图 4.28 所示。

图 4.28 FirstApp 的目录结构

其中 Inc 目录下我们需要根据上述代码结构新建 4 个头文件，分别是 FirstAppDefs.h、FirstAppProt.h、

FirstAppTypes.h 和 FirstAppGprot.h。

在 Src 目录下建立 FirstAppSrc.c 文件，将来程序的实现就完全在该文件中进行描述。

4.5.2 将相关文件路径加入 Plutommi 的编译配置文件

有了目录，还不完善，MMI 编译的时候如何才能将这些文件加入编译呢？这就需要我们将文件加入项目。

下面我们就来关注 make\plutommi\目录下的 3 个文件。

（1）plutommi.lis：该文件用来指明 MMI 所要编译的具体的源文件。

将我们的 plutommi\mmi\FirstApp\FirstAppSrc\FirstApp.c.附加在文件末尾。

（2）plutommi.inc：该文件用来指明源文件所需要的头文件的目录。

将我们的 plutommi\mmi\FirstApp\FirstAppInc 目录附加在文件末尾。

（3）plutommi.pth：在该文件中指明要编译的源文件目录。

将我们的 plutommi\mmi\FirstApp\FirstAppSrc 目录附加在文件末尾。

4.5.3 为新增模块增加编译开关

在这里，我们推荐开发人员为自己的模块增加一个宏开关，这样在不需要自己的模块的时候，可以关闭它，减少编译后的 Bin 文件大小，以节省空间。打开文件\make\Option.mak，在末尾添加宏__MMI_FIRSTAPP__，如图 4.29 所示。

图 4.29 在 Option.mak 文件中增加开关

同时，我们也可以在其他位置添加开关。

打开文件 plutommi\Customer\CustResource\PLUTO_MMI\MMI_featuresPLUTO.h，在其末尾添加宏：

```
#define __MMI_FIRSTAPP__
```

注 意

上述几个步骤的添加修改很重要，开发人员必须按照要求进行修改。如果有一步没有操作正确，编译将不会成功，在模拟器上也将提示找不到相应的模块文件。

到此为止，我们新增的程序的骨架已经搭建好。接下来把刚才写在 Organizer.c 中的函数全部转移到自己新增的模块中去。也就是说，FirstAppSrc.c 文件中将包含我们新增模块的实现。到目前为止，代码内容如下：

```
#include "MMI_features.h"
#include "PixtelDataTypes.h"
#include "Unicodexdcl.h"
#include "gdi_include.h"
#include "wgui.h"
#include "gui.h"
#include "MMI_features.h"
#include "PixtelDataTypes.h"
#include "Globaldefs.h"
#include "CustDataRes.h"
#include "wgui_categories.h"
```

```
#include "wgui_categories_util.h"
#include "wgui_softkeys.h"
#include "HistoryGprot.h"
#include "OrganizerDef.h"
void EntryFirstAppScreen(void)
{
 EntryNewScreen( ORGANIZER_SCREENID,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();//将屏幕清成空白
 gui_set_text_color(gui_color(255,0,0));
 gui_move_text_cursor(100,100);//设置字符串显示的坐标
 gui_print_text(L"Hello MTK!");//输出字符串
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
void ExitFirstAppScreen()
{
   history currHistory;
   currHistory.scrnID = ORGANIZER_SCREENID;
   currHistory.entryFuncPtr = EntryFirstAppScreen;
   GetCategoryHistory(currHistory.guiBuffer);
   pfnUnicodeStrcpy((S8*) currHistory.inputBuffer, (S8*) L"");
   AddHistory(currHistory);
}
```

由于目前我们的代码需要由 Organizer.c 文件来加载，所以，在 Organizer.c 文件中的头部增加如下语句：

```
#ifdef__MMI_FIRSTAPP__
#include "FirstAppGProt.h"
#endif
```

这样，Organizer 中的 EntryOrganizer 函数就可以正确调用我们的代码了，如图 4.30 所示。

至此，准备工作告一段落，现在，让我们在命令行输入：Make Yourproject Gprs Update，重新编译一次。由于之前增加了一个新模块开关，所以，需要使用 Update 命令。

编译成功之后，进入目录 plutommi\mmi，单击“PC_Simulator.dsw”打开 VC 模拟器工程。因为刚才的 make 操作已经重新修改了模拟器工程文件的关联，所以我们需要对模拟器重新构建。选择菜单[Build]→[Build PC_Simulator.exe]对整个模拟器工程进行重新构建，如图 4.31 所示。

图 4.30 函数细节

图 4.31 重新构建模拟器

> **提　示**
>
> 在 Visual C++编译模拟器过程中，有可能出现“找不到某些函数实现”的错误提示。就本例来说，有可能提示找不到函数 EntryFirstAppScreen 的实现。这时，需要在 Visual C++左边的 MMI 目录树中新建一个文件夹，命名为“FirstAppSrc”，并将我们新增的文件 FirstAppSrc.c 导入文件夹，即可解决该问题，如图 4.32 所示。

一般只要重新生成模拟器即可

make ~ gprs gen_modis

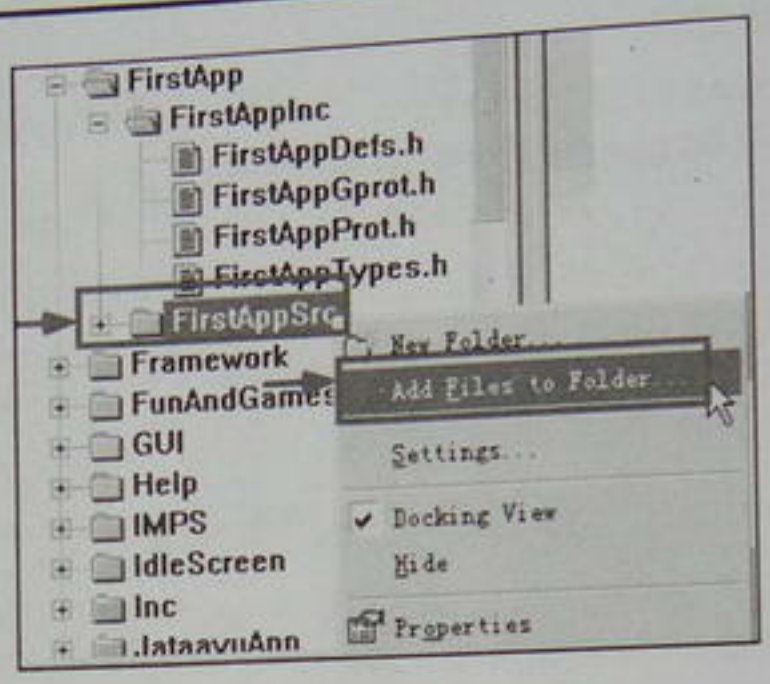


图 4.32　重新导入实现文件

Visual C++的重建完成后，进入 Organizer 界面，单击工具箱图标，成功地进入了本章刚刚开始的“Hello MTK”界面，这就意味着我们的程序模块已经融入了 MMI 的文件结构当中。

4.6　资源

如果大家开发过基于 Windows 平台的程序，使用过 Visual C++、Visual Basic 或者 Delphi 等开发环境，都会对其快捷的界面布局操作印象深刻。开发人员只需要用鼠标拖动各种控件，添加菜单或插入图片，不用添加任何代码，就可以编译链接生成一个 UI 界面。

而上面提到的控件、菜单或者是字符串都属于资源的范畴。

程序中使用的数据一般分为动态数据与静态数据两种。动态数据在程序运行的时候创建，例如 Visual C++中的 new 操作，是在程序需要内存空间的时候才进行创建。静态数据是在程序运行之前都已经确定的，在编译时将其转换为二进制数据存进最终的烧录文件中。

在 MTK 平台中常见的资源包括字符串、图像、菜单、字库、声音、主题等。在开发过程中，最受关注的有 3 种：字符串、图像和菜单。

在 plutommi\Customer\custResource\PLUTO_MMI\Res_MMI 下你会看到很多 Res_前缀的文件，这些都是各个模块的资源文件定义，包括了其所使用的字符串、菜单和图片等。

在讲解如何装载字符串、菜单、图片等资源之前，我们首先要弄明白资源是由谁加载的，弄清楚与哪些文件有关。

进入路径 Plutommi\Customer\ResGenerator\中找到 mtk_resgenerator.exe 文件，这个可执行文件就是每次编译目标烧录文件之前临时编译生成的。下面所讲的步骤都是与 mtk_resgenerator.exe 配合，并最终完成资源的构建工作的。

第 1 步：添加文件。

需要为我们的模块添加一个资源文件，目录如图 4.33 所示，其内容将会在接下来的章节中进行添加和完善。将其命名为 Res_FirstApp.c，并初步添加如下内容：

```
#include "MMI_features.h"
#include "StdC.h"
#include "CustomCfg.h"
#ifdef FIRST_PASS
#include "BuildCfg.h"
#endif
#include "ComposeRingToneDefs.h"
#include "FirstAppDefs.h"
#ifdef DEVELOPER_BUILD_FIRST_PASS
#include "PopulateRes.h"

void PopulateFirstAppRes(void)
{
}
#endif
```

第 2 步：修改 makefile 文件。该文件用于资源装载的预编译。添加刚才我们新增的头文件目录，目录里面包括了与资源文件定义有关的 FirstAppDefs.h 文件。

（1）打开文件\plutommi\Customer\ResGenerator\Makefile，增加图 4.34 所示的头文件目录路径，即将刚才我们定义的 FirstAppInc 目录加入，如图 4.34 所示。

图 4.33　增加文件

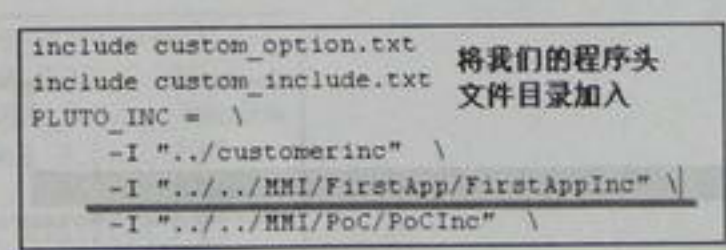


图 4.34　增加 FirstApp 相关目录

（2）打开文件 \plutommi\Customer\ResGenerator\readexcel.c，将新增的定义文件加入，如图 4.35 所示，当前文件是 FirstAppDefs.h。

怎么我的工程下没有此文件!? 如果找不到此文件，此资源可有

提　示

该文件会在工程编译中生成可执行文件 readexcel.exe，仅用于装载字符串资源。

第 3 步：修改 PopulateRes.c 文件。

进入路径 Plutommi\mmi\Resource\，打开文件 PopulateRes.c，添加刚才我们定义的资源装载函数 PopulateFirstAppRes，如图 4.36 所示。

```
#ifdef __MMI_FIRSTAPP__
#include "FirstAppDefs.h"
#endif
#include "enum_list.h"

#ifdef __MMI_RESOURCE_ENFB_SUPPORT__
```

图 4.35　包含新增的头文件

```
#ifdef __MMI_CCA_SUPPORT__
extern void PopulateCCARes(void);
#endif

#ifdef __MMI_FIRSTAPP__
extern void PopulateFirstAppRes(void);
#endif
```

图 4.36　资源加载函数

找到函数 void PopulateResData(void)，添加如下代码，如图 4.37 所示。

```
PRINT_INFORMATION(("Populating Alarm Resources\n"));
PopulateAlarmRes();

#ifdef __MMI_FIRSTAPP__
PRINT_INFORMATION(("Populating FirstApp Resources\n"));
PopulateFirstAppRes();
#endif

PRINT_INFORMATION(("Populating SPOF Resources\n"));
PopulateSpofRes();
```

图 4.37 调用资源加载函数

添加完毕后，工程进行编译的时候，上面提到的 mtk_resgenerator.exe 就会调用上述函数 PopulateResData。

至此，我们已经为新增的程序模块做好了基本的资源框架，接下来是不是就应该添加字符串 ID，或者图片 ID，并加入装载函数？还不行。

在进行模块的内部资源 ID 定义之前，我们还要做一件重要的事情，那就是定义程序的基础 ID。在 MMI 软件系统中，所有程序的资源 ID 都是独一无二的，必须保证这些 ID 不能相互冲突。因此，我们需要对 ID 进行一下包装。

（1）打开文件 plutommi\mmi\inc\MMIDataType.h，该文件用于基础 ID 的统一定义和包装。

将程序模块 ID 命名为 FIRSTAPP，这里的“50”表示程序模块的 ID 定义不会超过 50 个，如图 4.38 所示。

```
    RESOURCE_BASE_RANGE(HEALTH_MENSTRUAL,  50),
    RESOURCE_BASE_RANGE(HEALTH_BMI,                50),
#ifdef __MMI_FIRSTAPP__
    RESOURCE_BASE_RANGE(FIRSTAPP,      50),
#endif
    /* Message */
    RESOURCE_BASE_RANGE(MESSAGES,                  1500),
    RESOURCE_BASE_RANGE(CHAT,                      200),
    /* Unified Message */
```

图 4.38 定义模块 ID 范围

（2）按照源代码的格式，添加如下宏定义，如图 4.39 所示。

```
#ifdef __MMI_FIRSTAPP__
#define  FIRSTAPP_BASE                    ((U16) RESOURCE_BASE_FIRSTAPP)
#define  FIRSTAPP                         FIRSTAPP_BASE /* Be backward compatible */
#define  FIRSTAPPT_BASE_MAX               ((U16) RESOURCE_BASE_FIRSTAPP_END)
RESOURCE_BASE_TABLE_ITEM(FIRSTAPP)
#endif

/*
* Message
*****************************************************************************
```

图 4.39 定义相关宏

现在，程序模块有了自己独立的 ID，接下来我们就可以安心地进行我们模块内部资源的添加工作。

在前面几节的例子中，由于我们没有自己的屏幕 ID 号，只好临时借用 Organizer 中的屏幕 IDORGANIZER_SCREENID，现在，我们就可以在 FirstAppDefs.h 文件中添加新的屏幕 ID，命名为 SCR_FIRSTAPP_MAIN。需要注意的是，该 ID 应该属于一个枚举，添加完毕，文件内容如图 4.40 所示。

```
#ifndef _FIRSTAPPDEFS_H
#define _FIRSTAPPDEFS_H
#include "MMI_features.h"
#include "PixtelDataTypes.h"

typedef enum
{
    SCR_FIRSTAPP_MAIN = FIRSTAPP_BASE + 1
}SCREENID_LIST_FIRSTAPP;


#endif
```

图 4.40　添加新的枚举变量

这样，以后只要是程序模块新增的资源定义文件，都将在此文件中添加。

现在，我们就可以用 SCR_FIRSTAPP_MAIN 替换原来我们借用的屏幕 ID，修改如下函数：

```
void EntryFirstAppScreen(void)
{

  EntryNewScreen( SCR_FIRSTAPP_MAIN, ExitFirstAppScreen, NULL, NULL);
  entry_full_screen();
  clear_screen();//将屏幕清成空白
  gui_set_text_color(gui_color(255,0,0));
  gui_move_text_cursor(100,100);//设置字符串显示的坐标
  gui_print_text(L"Hello MTK!");//输出字符串
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

现在，资源添加的前期准备工作就完成了，接下来的几个小节中，我们将详细为读者讲解各种资源的添加和修改操作。

4.6.1　字符串

字符串资源主要是为了让程序转换成其他语言或者要实现多国语言的时候更为方便。如果程序使用字符串资源而不是将字符串直接放入原始代码中，那么程序所使用的所有文字将在资源描述文件中。如果转换了资源描述文件中的文字，那么建立程序的另一种语言版本所需做的一切就是重新连接程序而已，而不需要重新编译代码，明显提高了开发效率。图 4.41 是 Visual C++ 6.0 中添加字符串资源的截图。

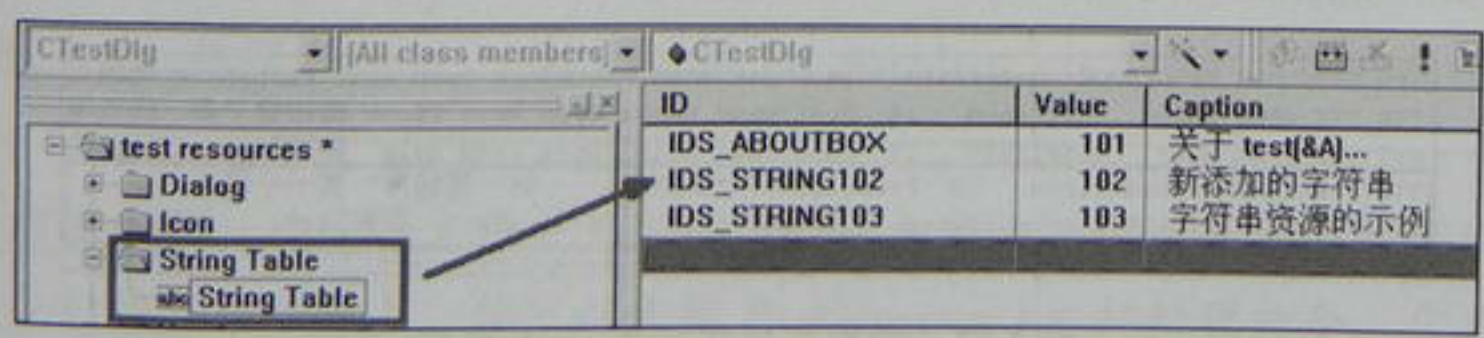


图 4.41　VC 的字符串资源

上图的字符串资源添加分为 3 项：ID 名称、值、字符串标题。这种字符串描述方式和 MTK 平台中的方式差不多，读者可以打开文件进行比较。

进入路径：plutommi\Customer\CustResource\PLUTO_MMI\，打开文件 ref_list.txt，可以看到字符串描

述文件的具体格式，如图 4.42 所示，是不是和 Visual C++6.0 中的方式有异曲同工之妙？

```
STR_ID_SWFLASH_STORAGE_CARD      SWFLASH 20      Card    Card    記憶卡  记忆卡
STR_ID_CALFWD_TO_OPTION Undefined        20      Divert option   Divert to
STR_ID_CALFWD_TO_NEW_NUMBER      Undefined       20      Divert to new number
STR_ID_CALFWD_TO_VM      Undefined       20      divert to voice mail    To Voi
```

图 4.42　MTK 的字符串资源定义

第 1 列是字符串 ID（比如当前是 STR_ID_SWFLASH_STORAGE_CARD），字符串 ID 必须要与程序模块定义头文件中的 ID 一样。比如你的模块 XXXDefs.h 文件中的 ID 是 STR_ID_NEW，那么此文件中的 ID 也要保证是 STR_ID_NEW。

> **提　示**
>
> ID 值保持一致很重要，开发人员必须特别小心此处，否则可能使程序无法编译或者出现离奇错误等。

第 2 列是字符串所属的程序模块名字，（当前是 SWFLASH）不必太关注，开发人员甚至可以将其写为 Undefined。

第 3 列式指本字符串的最大长度。

> **提　示**
>
> 此处的最大长度要根据字符串的所有语言格式来确定，不能因为中文只有 3 个字符就写 3，有可能英文有 16 个字符。

第 4 列是字符串具体内容。开发人员可以根据自己程序模块的需要而添加，这就是字符串定义过程中的重点了。

第 5 列以后，就是字符串对应的各国语言版本了。如果你需要进行多国语言移植，这里就是你应该关注的地方。

在 4.3 节的“Hello MTK”演示中，我们已经使其在一个白色的屏幕上输出了简单的字符串。如本节开头所讲，这种方法显然不合适，最好的方法是将其做成资源，在需要使用的时候载入，而且做成资源还可以实现多国语言化。

现在就让我们根据上述规则来添加字符串。

（1）打开上述字符串描述文件 ref_list.txt，按照上面提到的格式新添两行数据，如图 4.43 所示。

```
STR_TTS_OPEN_PHONEBOOK  undefined   14  Open Read PhoneBook Open Read phoneBook 閱讀電
STR_TTS_OPEN_EBOOK  undefined   14  Open Read EBook Open Read EBook 閱讀電子書  阅读电
STR_TTS_OPEN_HELP   undefined   14  Open Read Help  Open Read Help  閱讀幫助    阅读帮
STR_FIRSTAPP    undefined   20  FirstApp    FirstApp    第一個程序  第一个程序
STR_FIRSTAPP_HELLOMTK   undefined   20  HelloMTK    HelloMTK    你好,MTK!   你好，MTK
##number    8992
```

图 4.43　增加新的字符串 ID 和内容

> **提　示**
>
> 每一列填写完毕后，建议使用<Tab>键进入下一列的编辑。当 STR_FIRSTAPP 写完后，按<Tab>键，继续写 undefined，依此类推。

（2）确定字符串 ID。在文件 FirstAppDefs.h 中添加字符串 ID，如图 4.44 所示。

（3）字符串装载。在 PopulateFistAppRes 函数中添加图 4.45 所示代码。

```
#ifndef _FIRSTAPPDEFS_H
#define _FIRSTAPPDEFS_H
#include "MMI_features.h"
#include "PixtelDataTypes.h"

typedef enum
{
   SCR_FIRSTAPP_MAIN = FIRSTAPP_BASE + 1
}SCREENID_LIST_FIRSTAPP;

typedef enum
{
   STR_FIRSTAPP = FIRSTAPP_BASE + 1,
   STR_FIRSTAPP_HELLOMTK,
   STR_FIRSTAPP_TOTAL
}STR_ID_FIRSTAPP_ENUM;

#endif
```

图 4.44　确定字符串 ID

```
void PopulateFirstAppRes(void)
{
   ADD_APPLICATION_STRING2(STR_FIRSTAPP, "FirstApp", "Name of FirstApp");
   ADD_APPLICATION_STRING2(STR_FIRSTAPP_HELLOMTK, "HelloMTk", "A simple string");
}
#endif
```

图 4.45　字符串加载

提　示

宏 ADD_APPLICATION_STRING2 的第 1 个参数为字符串 ID；第 2 个参数为字符串默认显示的内容，如果开发人员在 ref_list 中没有添加正确的内容，那么程序就用默认的内容显示。第 3 个参数为字符串的描述信息，可以按需填写。

修改 FirstAppSrc.c 文件中的函数，现在可以使用系统提供的 API 读取字符串 ID，如图 4.46 所示。

```
void EntryFirstAppScreen(void)
{

 EntryNewScreen( SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();//将屏幕清成空白
 gui_set_text_color(gui_color(255,0,0));
 gui_move_text_cursor(100,100);//设置字符串显示的坐标
 //输出从资源载入的字符串
 gui_print_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));

 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

图 4.46　打印字符串

由于我们添加了字符串资源，并且修改了 Res_FirstApp.c 文件，所以，按照规则，这时我们应该在命令行执行 update 操作。使得资源修改能够在模拟器上得以体现。输入“Make Yourproject Gprs Update”命令，编译完成后，进入 VC 模拟器工程，选择［Build］→［Build PC_Simulator.exe］命令，重新构建模拟器。构建成功后，选择［Build］→［Execute PC_Simulator.exe］命令，运行模拟器。

提　示

如果当前添加了一些资源文件并且只是想在模拟器上测试效果和程序逻辑，不想花费大量时间在命令行的真机编译上，你可以使用 ResGenerator.bat 批处理命令。它只执行资源的生成操作，而不需要重新编译所有的文件。文件路径为：plutommi\Customer\ResGenerator.bat。

双击该文件图标运行，待.bat 文件执行完毕且没有错误后，进入 VC 模拟器工程，按照正常的步骤构建一次模拟器即可。

一切正常的话，进入 Organizer 界面，会发现原来英文的字符串已经变成了“你好，MTK!”字符串，

如图 4.47 所示。

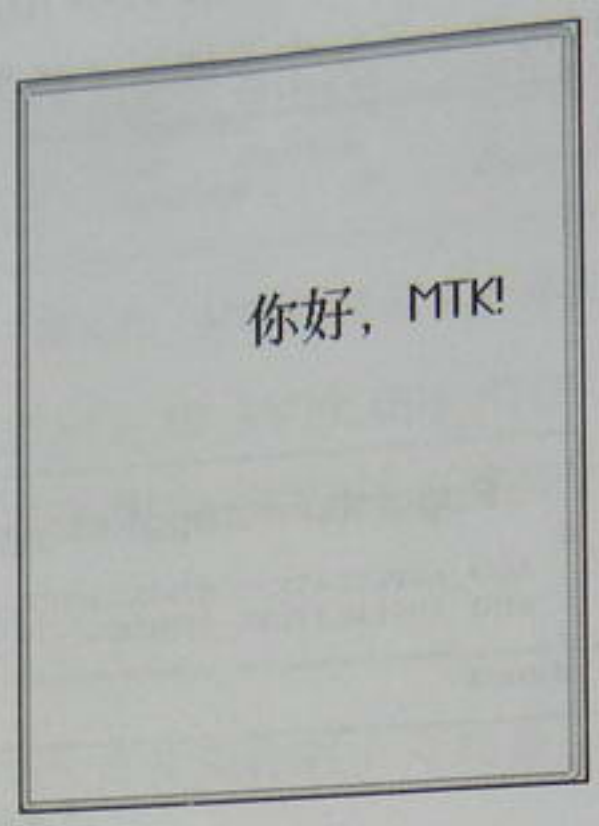


图 4.47 字符串显示

> **提 示**
>
> 如果读者是基于原始版本的开发，你会发现屏幕上的文字不是中文的。这是由于当前版本还没有增加中文字库的缘故，这里我们只是为了演示方便而提前加入了。

4.6.2 菜单

无论你使用的是手机，电脑还是其他的 PDA 设备，当你需要某一个功能的时候，你总会找到一个对应的图标，单击后，就可以跳转到另外一个界面而进入某个模块，而这个图标就是菜单。菜单本身注册了其将要执行的行为，当用户单击这个菜单的时候，就会响应其注册函数，从而实现了人机交互。

可以说，菜单是一个可视化系统的基石，无论这个系统是基于 Windows、Symbian 还是其他平台。

菜单也是我们添加新的应用程序的一个典型入口。在前面的章节中，由于我们没有自己的入口函数，只能使用系统已有的。所以，在本小节中，我们就将先给我们新增的模块提供一个入口，也就是菜单，并一步一步地深入下去。

1. 要制作菜单，首先要确定其 ID

(查了一下，QQ的入口菜单也是加在此)

整个 MMI 中用到的菜单 ID 都位于 plutommi\mmi\inc\GlobalMenuItems.h 文件中。在枚举中添加 MENU_ID_FIRSTAPP，注意该 ID 必须放在 MAX_MENU_ITEMS_VALUE 之前。

代码添加位置如下：

```
enum GLOBALMENUITEMSID
{
IDLE_SCREEN_MENU_ID = 1,
    ORGANIZER_CALENDER_MENU,
    ORGANIZER_TODOLIST_MENU,
    ORGANIZER_WORLDCLOCK_MENU,
/*和 organizer 中的 ID 放在一起便于归纳*/
#ifdef__MMI_FIRSTAPP__
    ORGANIZER_FIRSTAPP_MENU,
#endif
}
```

2. 将新 ID 整合在 Organizer 的菜单 ID 之中

打开文件 Res_MainMenu.c（plutommi\customer\custresource\pluto_mmi\res_mmi）。由于我们需要将 ID 整合在 Organizer 的子菜单中，所以，我们要前往 Res_MainMenu.c 中修改 Organizer 的 ID。

```
typedef enum
{
#if defined(__MMI_CALENDAR__)
ORG_ENUM_CALRNDAR,
#endif

#if defined(__MMI_TODOLIST__)
ORG_ENUM_TODOLIST,
#endif
ORG_ENUM_ALARM,
#if defined
(__MMI_WORLD_CLOCK__) && !defined(__MMI_WORLD_CLOCK_TO_PHONESETUP__)
    ORG_ENUM_WORLDCLOCK,
#endif
#ifdef__MMI_MESSAGES_CLUB__
     ORG_ENUM_SERVICE,
#endif

//添加我们程序的菜单枚举值，以便让系统原来的 Organizer 加载我们的菜单
#ifdef__MMI_FIRSTAPP__
   ORG_ENUM_FIRSTAPP,
#endif

#ifdef__MMI_HEALTH__
   MENU_ENUM_HEALTH_MANAGEMENT,
#endif
    ORG_ENUM_TOTAL
} OrganizerMenu;
…
/* oganizer */

//下面的 ADD_APPLICATION_MENUITEM 宏位于函数 PopulateMainMenuRes()
ADD_APPLICATION_MENUITEM((MAIN_MENU_ORGANIZER_MENUID,MAIN_MENU_PIM_MENU_ID,ORG_ENUM_TOTAL,
                #if defined(__MMI_CALENDAR__)
                ORGANIZER_CALENDER_MENU,
            #endif
            #if defined(__MMI_TODOLIST__)
                ORGANIZER_TODOLIST_MENU,
            #endif
                ORGANIZER_ALARM_MENU,

            #if defined (__MMI_WORLD_CLOCK__)
                ORGANIZER_WORLDCLOCK_MENU,
            #endif
        #if defined(__MMI_FIRSTAPP__)
                ORGANIZER_FIRSTAPP_MENU,
            #endif
                SHOW,MOVEABLEACROSSPARENT|INSERTABLE,DISP_LIST,
                MAIN_MENU_ORGANIZER_TEXT,MAIN_MENU_ORGANIZER_ICON));
```

> **提 示**
>
> 在我们修改文件 Res_Mainmenu.c 的时候，一定要注意修改的是哪一个宏下面的 PopulateMainMenu Res 函数。因为宏__MMI_BI_DEGREE_MAIN_MENU_STYLE__和__MMI_VERSION_2__都有上述函数，开发人员一定要确定当前是哪一个宏起作用。否则，修改可能看不到效果，如图 4.48 所示。

经过上面的操作后，Organizer 中就多了一个 FirstApp 菜单。现在，我们要着重讲解宏 ADD_APPLICATION_MENUITEM 的原理和使用。

```
#if defined (__MMI_BI_DEGREE_MAIN_MENU_STYLE__)
    PopulateMainMenuRes
#else
    #if defined(__MMI_VERSION_2__)
        PopulateMainMenuRes
    #else
        PopulateMainMenuRes
    #endif
#endif
```

图 4.48 宏__MMI_VERSION_2__

宏 ADD_APPLICATION_MENUITEM 用来装载菜单资源，其参数解释如下。

参数 1：新加菜单的 ID。

参数 2：新菜单项的上一级菜单 ID，即父菜单。

参数 3：此菜单的下一级菜单项的总个数，在上述代码示例中，用 ORG_ENUM_TOTAL 表示，这里我们假设有 X 个。

参数 4 和参数 4+X：分别为每一个子菜单的 ID。

参数 4+X+1：隐藏属性，一般设置为可见，即 SHOW。

参数 4+X+2：表示菜单项目的转移属性。可以选择 NONMOVEABLE， MOVEABLEWITHINPARENT, MOVEABLEACROSSPARENT，INSERTABLE，SHORTCUTABLE。

常用到的是 NONMOVEABLE，表示关闭转移属性，目前大部分菜单用这个属性，还有 SHORTCUTABLE，表示菜单项将添加到快捷菜单列表中。

参数 4+X+3：下一级菜单的显示风格。有如下风格可选。

DISP_LIST：列表显示。大部分菜单用此风格。

DISP_MATRIX：矩阵显示。如 NOKIA 主菜单显示的九宫格。

DISP_CIRCULAR_3D：循环 3D 显示。不常用。

DISP_PAGE：翻页风格。每个菜单项占一页。

DISP_FIXED_GRID：不常用。

参数 4+X+4：此菜单项的显示字符串 ID。

最后一个参数：菜单项的小图标 ICON 的 ID。

因为先前的操作是将新菜单加入 Organizer 的下属菜单，所以，现在让我们继续加载菜单本身。打开文件 Res_FirstApp.c，修改函数，如图 4.49 所示。

> **提 示**
>
> 修改了文件 Res_FirstApp.c 后一定要进行资源编译操作，即执行 update 命令。否则，将不会在模拟器中看到效果。

当用户进入一个列表项上下进行选择的时候，当前只有一个菜单项会被高亮显示。如果用户这时候按左键或者 OK 键，系统就会进行界面跳转，进入菜单所指向的模块。

```
void PopulateFirstAppRes(void)
{
  ADD_APPLICATION_STRING2(STR_FIRSTAPP, "FirstApp", "Name of FirstApp");
  ADD_APPLICATION_STRING2(STR_FIRSTAPP_HELLOMTK, "HelloMTk", "A simple string");
  ADD_APPLICATION_MENUITEM((
    ORGANIZER_FIRSTAPP_MENU,//菜单 ID
    MAIN_MENU_ORGANIZER_MENUID,//父菜单 ID，这里是 ORGANIZER的菜单 ID
    0,//因为没有下属菜单，所以为 0
    SHOW,
    SHORTCUTABLE,
    DISP_LIST,//列表显示
    STR_FIRSTAPP,
    0));
}
```

特别要注意该宏的内部，所有参数必须外加一个括号包裹起来，也就是双层括号：
ADD_APPLICATION_MENUITEM((arg1,arg2,arg3))
这一点很重要，否则执行update操作的时候不能通过。

图 4.49　菜单加载

这是如何实现的呢?

系统将会在我们上下选择菜单的时候，对当前高亮项发一个通知。这样，用户就通过单击当前高亮菜单，执行其注册的功能，这其实隐含了两件事情。

1. 由谁来接受高亮通知

每个菜单项都要在开机的时候告诉系统将由哪个函数来接受通知，SetHiliteHandler 函数就是用来实现此功能的原型为 void SetHiliteHandler(U16 itemid, FuncPtr hiliteFuncPtr)。

第一个参数是菜单项的 ID。

第二个参数是用来接受通知的函数指针。

2. 接受通知后的处理

在菜单项接到高亮通知后，其实只需要做一件事情，就是更改左右按键的响应函数，其中最重要的就是左软键，如前面所讲，我们要将菜单项：ORGANIZER_FIRSTAPP_MENU 作为程序的入口菜单，那么就应该将做菜单的响应函数设为 EntryFirstAppScreen，并将此函数作为实参传入 SetHiliteHandler 函数。

完成这两步以后，我们新增程序的菜单项就可以正确地接收和处理通知。

修改 FirstAppProt.h 文件，增加如下代码：

```
extern void  mmi_firstapp_hilite(void);
extern void  mmi_firstapp_init(void);
```

将前面提到的 SetHiliteHandler 封装到我们的初始化函数之中，修改 FirstAppSrc.c 文件，增加如下代码：

```
void mmi_firstapp_hilite(void)
{
    SetLeftSoftKeyFunction(EntryFirstAppScreen,KEY_EVENT_UP);
}
void mmi_firstapp_init(void)
{
    SetHiliteHandler(ORGANIZER_FIRSTAPP_MENU,mmi_firstapp_hilite);
}
```

到目前为止，FirstAppSrc.c 文件中代码如下：

```
#include "MMI_features.h"
#include "PixtelDataTypes.h"
#include "Unicodexdcl.h"
#include "gdi_include.h"
#include "wgui.h"
```

```
#include "gui.h"
#include "MMI_features.h"
#include "PixtelDataTypes.h"
#include "Globaldefs.h"
#include "CustDataRes.h"
#include "wgui_categories.h"
#include "wgui_categories_util.h"
#include "wgui_softkeys.h"
#include "HistoryGprot.h"
#include "OrganizerDef.h"
#include "FirstAppGprot.h"
#include "FirstAppTypes.h"
#include "FirstAppDefs.h"


void mmi_firstapp_hilite(void)
{
    //将左按键的响应函数设置为新增的入口函数 EntryFirstAppScreen
    SetLeftSoftkeyFunction(EntryFirstAppScreen,KEY_EVENT_UP);
}
void  mmi_firstapp_init(void)
{
    //将高亮响应处理绑定给新增的菜单 ID
    SetHiliteHandler(ORGANIZER_FIRSTAPP_MENU,mmi_firstapp_hilite);
}
void EntryFirstAppScreen(void)
{
  EntryNewScreen( SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();//将屏幕清成空白
  gui_set_text_color(gui_color(255,0,0));
  gui_move_text_cursor(100,100);//设置字符串显示的坐标
  //输出从资源载入的字符串
  gui_print_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}

void ExitFirstAppScreen()
{
    history currHistory;
    currHistory.scrnID = SCR_FIRSTAPP_MAIN;
    currHistory.entryFuncPtr = EntryFirstAppScreen;
    GetCategoryHistory(currHistory.guiBuffer);
    pfnUnicodeStrcpy((S8*) currHistory.inputBuffer, (S8*) L"");
    AddHistory(currHistory);
}
```

因为我们增加的菜单位于 Organizer 之中，所以，将初始化函数放在 Organizer 的初始化函数中进行初始化调用。

```
void OrgInit(void)
{
 /* calendar */
#if defined(__MMI_CALENDAR__)
   ClndrInit();
#endif
 /* alarm */
```

```
 AlmInit();
#ifdef __MMI_FIRSTAPP__    //在函数 OrgInit 中调用新加程序的初始化函数
   mmi_firstapp_init();
#endif
}
```

现在，只需要在 VC 工程中重新构建一次。进入模拟器，单击 Organizer 图标，进入其列表，就可以看到新增了一个 FirstApp 列表项。点击该列表项，成功进入我们自定义的屏幕。

4.6.3 图片

做 UI 不能没有图片资源，图片资源的加载方式和字符串类似，也分为定义资源 ID，在代码中通过 API 读取等基本步骤。

在本小节中，我们通过为刚才新增的菜单添加图标的操作，来讲解如何正确地装载图片资源。

（1）定义图片 ID。

打开头文件 FirstAppDefs.h，添加如下代码：

```
typedef enum
{
  IMG_FIRSTAPP=FIRSTAPP_BASE+1
}IMAGEID_LIST_FIRSTAPP;
```

（2）打开图 4.50 所示目录，并在 MainLCD 目录下新建文件夹 FirstApp。

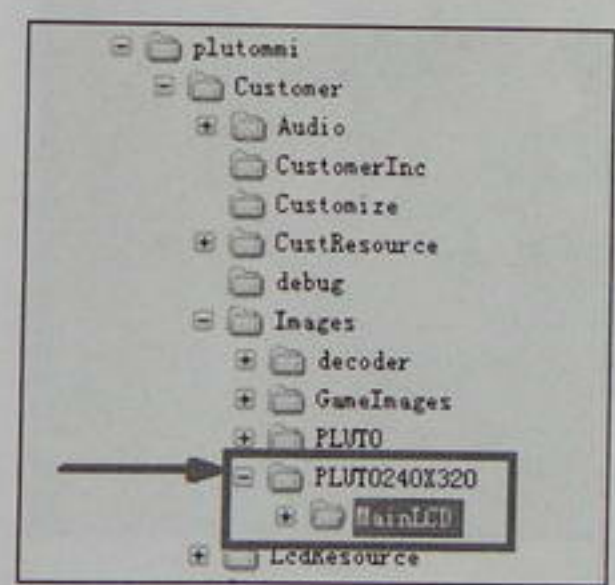


图 4.50　MainLCD

（3）在 FirstApp 文件夹中新加一幅图片 ICON.bmp。

图片添加完毕之后，将整个 MainLCD 文件夹打包为 image.zip。

> **提　示**
>
> 每次添加了新的图片之后，必须将 MainLCD 文件夹打为 ZIP 压缩包。否则，在 make 操作的时候，文件夹内的文件将会被上次的压缩包文件替换。

（4）打开文件 Res_FirstApp.c，在函数 PopulateFirstAppRes 中添加图片加载代码。

```
  ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP, CUST_IMG_PATH"\\\\MainLCD\\\\FirstApp\\\\ICON.BMP
","FirtAPP ICON");
```

宏 ADD_APPLICATION_IMAGE2 用来加载图片资源。

参数 1：图片 ID。即在 FirstAppDefs.h 文件中定义的图片 ID。

参数 2：图片存放路径。宏 CUST_IMG_PATH 在运行的时候会自动转化为相应的图像根目录。就当前例子来说，转换后的路径为 plutommi\Customer\Images\PLUTO240X320。

参数 3：图片的描述字符串。仅做参考作用。

（5） 为菜单添加图标。

继续修改 PopulateFirstAppRes 函数，由于前面步骤已经将图片资源载入，所以我们可以使用图片来作为菜单的图标。

```
ADD_APPLICATION_MENUITEM((
    ORGANIZER_FIRSTAPP_MENU,
    MAIN_MENU_ORGANIZER_MENUID,
    0,
    SHOW,
    SHORTCUTABLE,
    DISP_LIST,
    STR_FIRSTAPP,
    IMG_FIRSTAPP));
```

现在，在命令行执行 update 操作以后，重新编译一次 VC 工程，最终效果如图 4.51 所示。

图 4.51 新菜单项

4.7 SDK 的制作和使用

如果先前已经对 MTK 平台有所了解或者移植过第三方模块的读者，可能注意到，第三方模块的提供者会向开发人员提供若干个 Lib 库文件。出于对代码的保护，许多实现文件都被封装在 Lib 库中，提供者提供给开发人员一个 Lib 库和相关的头文件，就可以在开发人员看不到源码细节的前提下，移植相应的模块到当前的平台。而这个 Lib 库其实就是一个简单意义上的 SDK。作为项目开发人员，现在你做的可能是为公司单独开发一个产品，也有可能你会接到一个为客户公司提供用于二次开发的开发包的项目，你会被要求将一些模块文件制作成若干个 Lib，提供给客户。所以，有必要对 SDK 的制作有一个了解。

在继续讲解之前，我们还是要了解 MTK 的一些特性。MTK 平台是一个耦合度很高的平台，我们很

难做到将自己的应用模块化，即使在这方面下了很大的功夫，并且自信做得还不错，但如果你的应用依赖了 MTK 平台，当你想把模块从一个工程移到另个工程时，将付出很多的时间和精力。因为你有太多与平台相关的实现细节需要去关注，所以，要在这样的一个平台实现 SDK 的开发就需要保证我们的模块是跨平台的，当然也不能依赖 MTK 平台（除了它的编译器）。

大家常见的图片文件解码器模块、语音识别模块、GPS 模块等，其核心的内容都遵守业界规范，不依赖于平台，基本上都是使用纯 C 语言实现的。

在生成 SDK 前请先假定你的模块已经实现了跨平台，接下来就让我们去关注一下实现的细节。

首先，MTK 分为两个版本：模拟器和真机版本。当然，向客户提供的 SDK 也应该对应有模拟器和真机两个版本。下面分别讨论如何生成这两个版本对应的 SDK。

4.7.1 生成真机版本的 SDK

我们用一个最简单的例子来阐述如何生成 SDK。

首先在工程根目录下新增一个目录 TestSDK，再在.\TestSDK\inc 目录下放置一个头文件 testSDK.h。其文件内容如下：

```
#define x 0
…
extern void print();
…
```

然后在目录.\TestSDK \TestSDK\src 中放置一个 C 源文件 testSDK.c。其文件内容如下：

```
#include "testSDK.h"
…
void print()
{

}
```

紧接着，我们需要在.\make 目录中增加一个 TestSDK 目录，该目录下存在如下 4 个文件。

TestSDK.def 文件内容：

```
NODEBUG
```

TestSDK.inc 文件内容：

```
TestSDK\inc
```

TestSDK.lis 文件内容：

```
TestSDK\src\testSDK.c
```

TestSDK.pth 文件内容：

```
TestSDK\src
```

TestSDK.inc 是头文件路径，TestSDK.lis 文件中列出了所有将要参与编译的 C 文件路径，TestSDK.pth 中者列出了 C 文件存在的路径。

接下来，我们需要在.\make\REL_MMI_GPRS.mak 文件中附加语句：

```
CUS_REL_SRC_COMP += TestSDK   //将我们新增的测试 SDK 加入编译
```

最后，在命令行输入：

```
make NEOTEL26_06B_W07 gprs remake
```

待编译完成之后，进入目录 build\NEOTEL26_06B_W07\gprs\MT6226o\lib，即可找到我们刚刚编译出来的 Lib 文件。

最后，将生成的 TestSDK.lib 和.\TestSDK\inc 目录下的头文件复制出来，就可以打包发布了。

4.7.2 生成模拟器版本的 SDK

→完全无实际价值，若使用到了MTK中的函数

相对于真机版本，模拟器版本的 SDK 更加简单。你如何新建一个工程!!!

首先，在 VC 中新建一个静态库工程（建立一个空工程），如图 4.52 所示。

单击 OK 按钮，下一步选择默认，然后单击 Finish 按钮，完成工程的创建。

进入工程，会发现左边工程的目录树包含了两个空的文件夹：Source Files 和 Header Files。右键单击上面两个文件夹，将之前的 testSDK.c 和 testSDK.h 文件分别加入对应的文件夹，如图 4.53 所示。

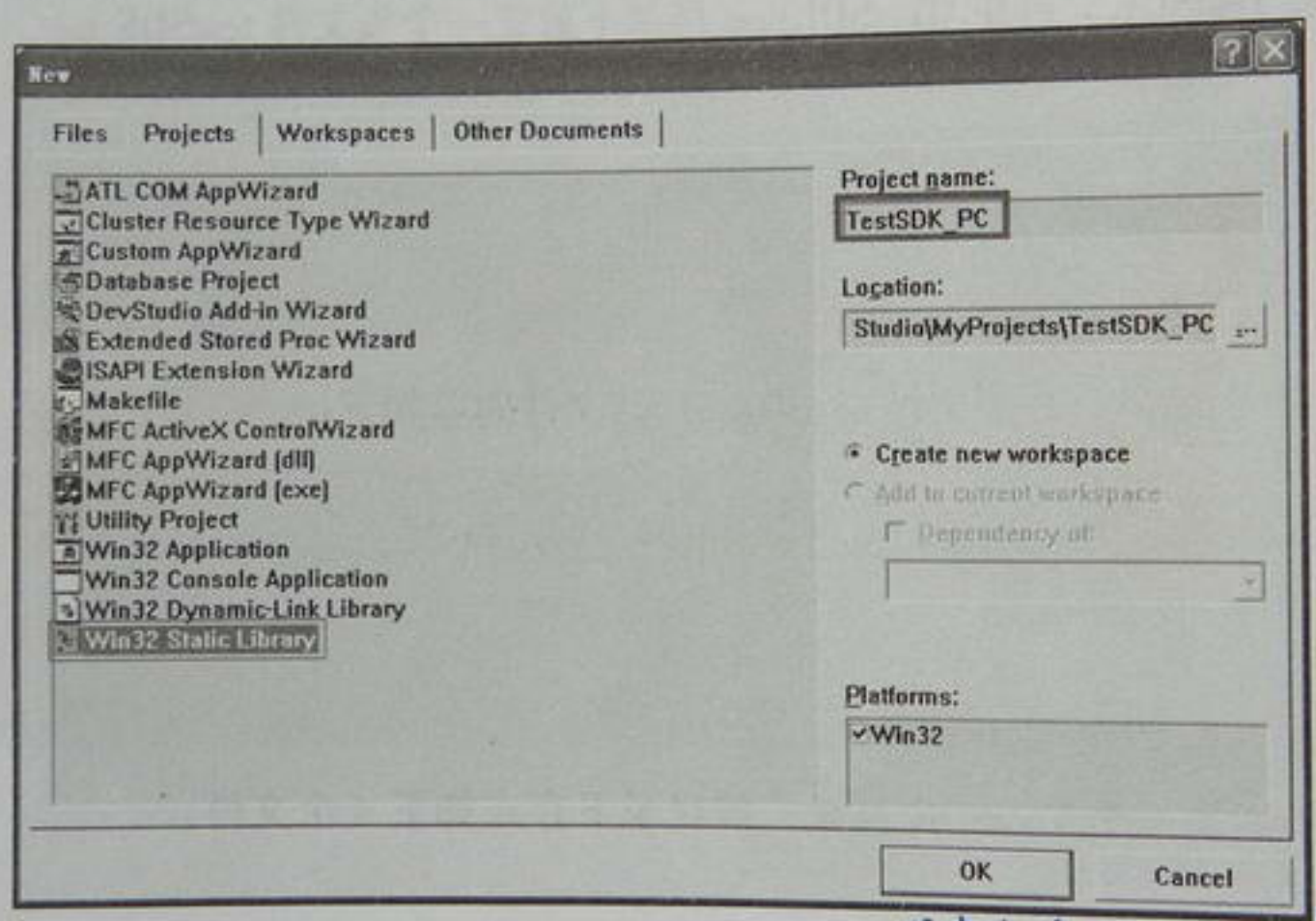


图 4.52 建立静态库工程

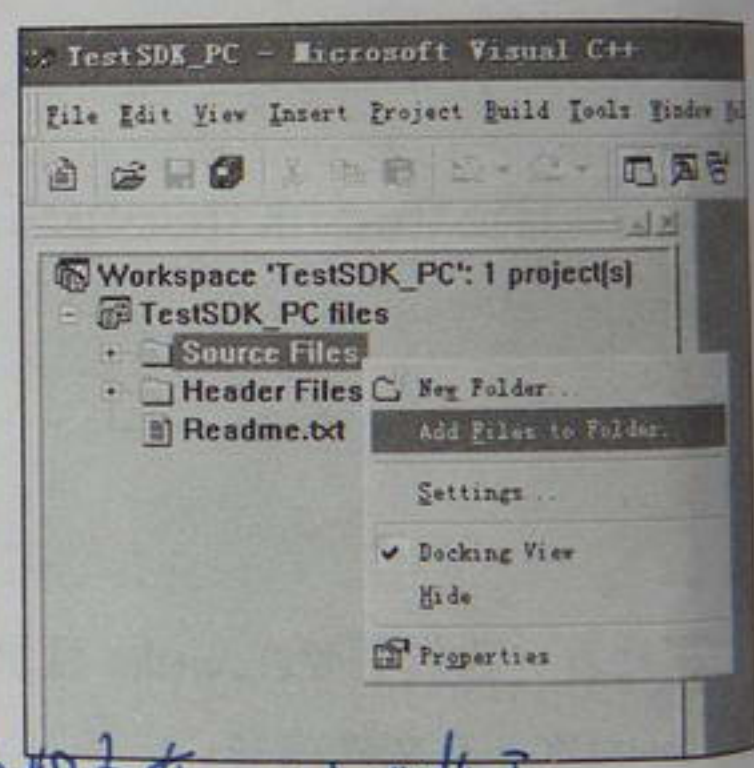


图 4.53 添加新文件

解决办法：每个模块都会在 ...\modis下生成它的模块，其中又各个函数文件的.obj 然后用此的命令行自己打包

接着，选择菜单，按〈F7〉快捷键，构建整个工程。

> **提 示**
>
> 如果 VC 在编译过程中提示找不到 TestSDK 对应的头文件，请在 option 中将头文件路径包含进来。
>
> link lib /out:...\...\...lib ...\...*.obj

最后，把编译出来的文件 TestSDK_PC.lib 复制出来和上面的真机版本一起发布，这样就完成了两个版本的 SDK 制作。客户公司只需要加入我的 Lib 库和头文件，就能够使用我们的 API。

4.7.3 使用真机版本 SDK

首先，需要确定在哪个模块使用 SDK。

假如 plutommi 模块将使用我们先前的 SDK，就需要在路径.\plutommi\mmi 中增加一个 TestSDK 目录，

将先前生成的 TestSDK.lib 复制到上述目录中。然后，需要向.\make\Option.mak 文件的 COMPOBJS 项中追加 Lib 路径。

```
COMPOBJS += \plutommi\mmi \TestSDK\TestSDK.lib
```

最后，将命令行切换到当前工程目录，运行 make NEOTEL26_06B_W07 gprs remake 命令。

编译无错后，TestSDK 即加入到我们的工程中。

上面所讲都是使用我们自定义的 SDK。作为开发人员，如果接到了移植第三方模块的任务，该如何操作？其实方法和前面一样。

假如我们要移植语音识别模块 tts。SDK 提供者给了你一个 tts 目录，里面包含 inc、Lib 和 src3 个文件夹，前面已经讲过各个文件的作用，这里不再赘述。你现在要做的如下所述。

（1）将 tts 文件夹复制到 plutommi\ mtkapp \目录下。

（2）在 option.mak 文件中增加如下语句：

```
COM_DEFS += __TTS__                                  #模块编译开关
CUSTOM_COMMINC += plutommi\mtkapp\tts\Inc            #将头文件路径加入
COMPOBJS += plutommi\mtkapp\tts\Lib\tts.lib          #Lib 库的位置
```

（3）将命令行切换到当前工程目录，运行 make NEOTEL26_06B_W07 gprs remake 命令。编译无误即成功添加 tts 模块。

4.7.4 使用模拟器版本 SDK

首先，在 VC 工程的 Settings 中加入 Lib 库（TestSDK_PC.lib），如图 4.54 所示。

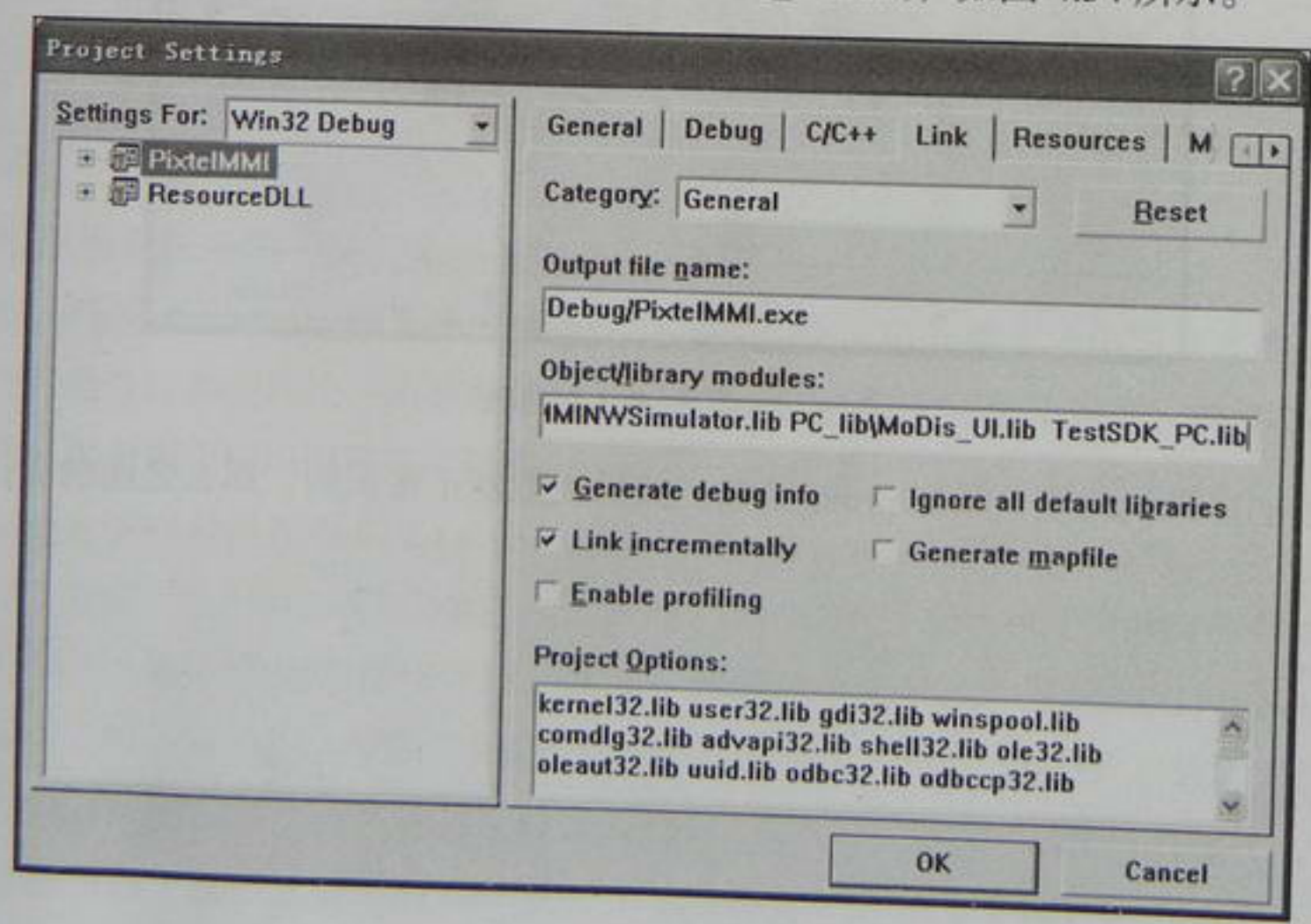


图 4.54 加入 Lib 库

其次，在 VC 的［Tools］→［Options］对话框中加入头文件路径和 Lib 库路径，如图 4.55 和图 4.56 所示。

最后，编译 VC 工程。

现在，让我们验证 Lib 库是否添加成功。

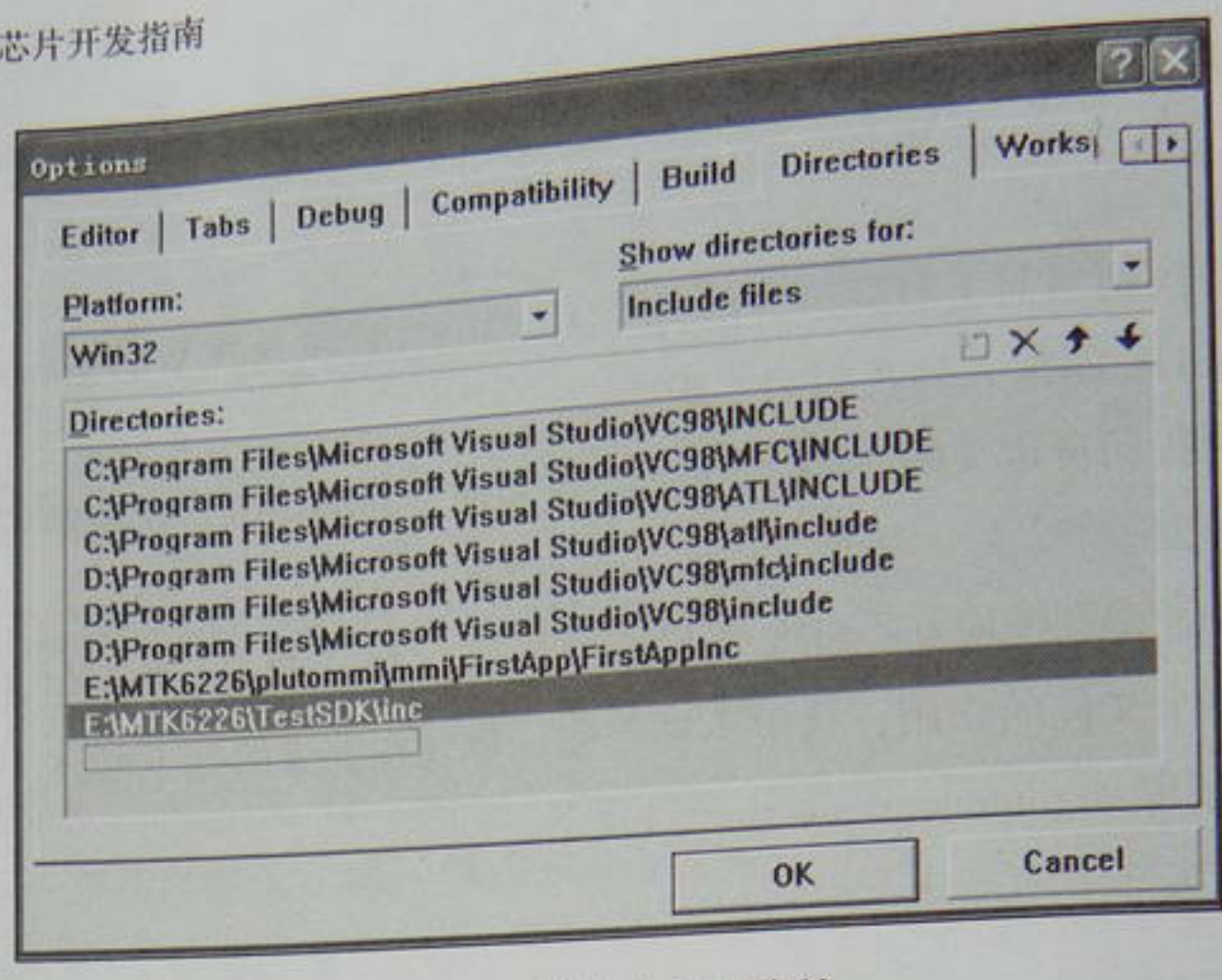


图 4.55 增加头文件路径

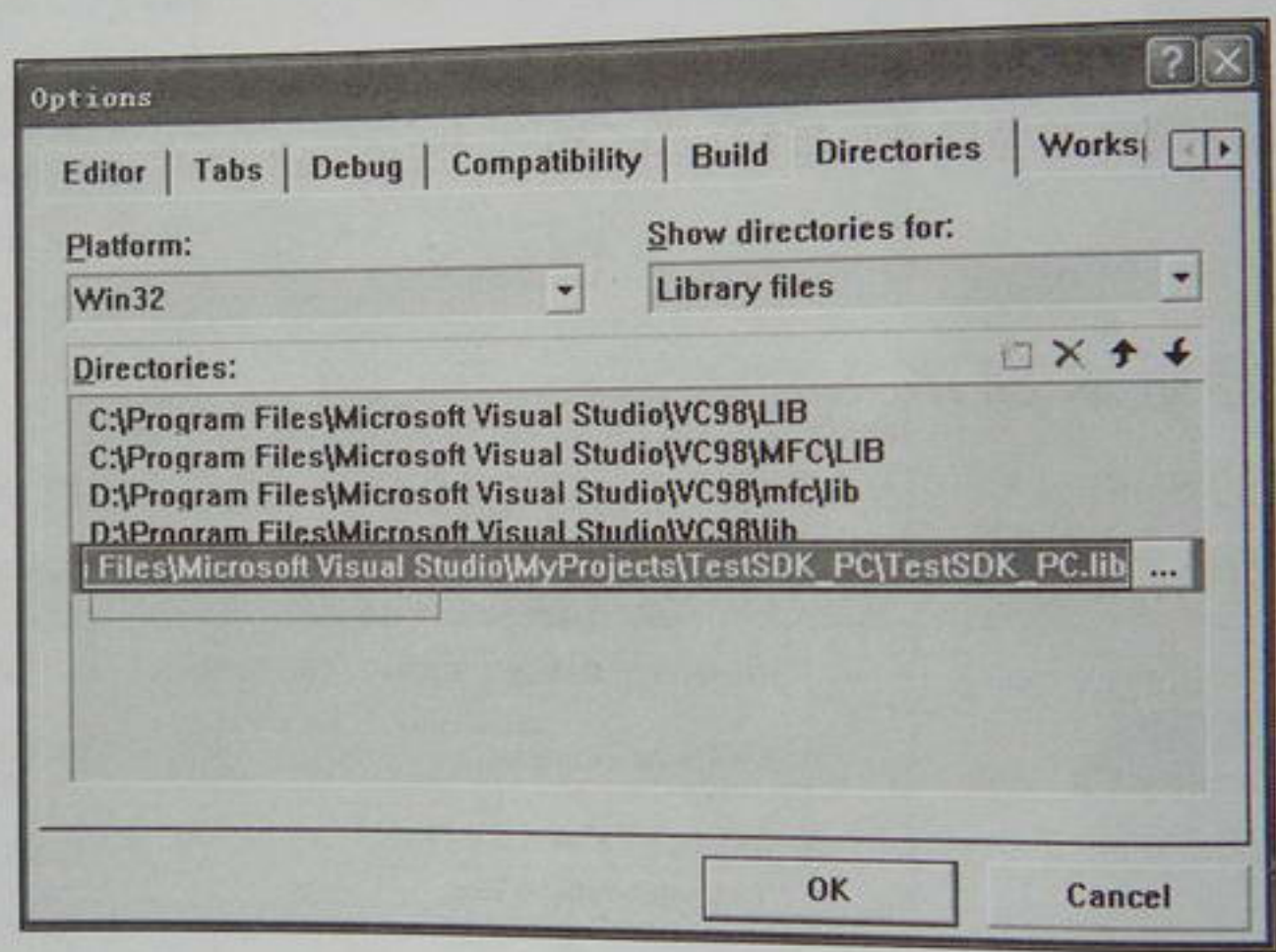


图 4.56 增加 Lib 路径

在我们先前的屏幕代码里，添加 print()函数。如果模拟器正常识别，那么之前的操作完全正确。

```
 …
#include "testSDK.h"
  void EntryFirstAppScreen (void)
{
 …
 print();
…
}
```

结语：

在这一章中，我们通过连续的实例，向读者呈现了 MTK 应用开发的所有重要知识点。学完本章，读者可以掌握 MTK 开发中最常见部分的结构和逻辑，为后面更加深入的学习打下基础。

5 MTK 的图形系统

在第 4 章中，我们通过一个浓缩的实例，讲解了 MMI 开发中的重要元素，掌握了 MMI 开发最基础的框架。在后面的学习过程中，大家会发现其实 MMI 开发都是以第 4 章的内容为核心，要做的就是在这个基础上增加更多的细节。因此，笔者建议，如果对第 4 章还有疑惑或没有完全理解，请务必认真阅读和研究。

从这一章开始，我们就将在“Hello MTK”实例的基础上，逐步深入地讲解 MMI 开发的各个知识点。而在任何上层开发中，图形系统都是首要的部分，毕竟，使用机器的用户面对的是图形化的界面。

本章将对图形系统及其相关的内容做一个详细梳理，理清内部各个子系统的功能，为后期的应用开发打下基础。

5.1 概述

在继续讲解图形系统之前，我们先来了解一下 GUI。

GUI（Graphical User Interface，图形用户界面或图形用户接口），是指采用图形方式显示的设备操作界面。与早期使用的命令行界面相比，图形界面对于用户来说在视觉上更易于接受。如 Windows 是以图形界面方式操作的，因为用户可以用单击按钮来进行操作，很直观，而 DOS 就不具备 GUI，所以只能输入命令进行操作。DOS 的这种界面叫 CUI（Command line User Interface），即命令行模式的人机接口。

准确来说 GUI 就是屏幕产品的视觉体验和互动操作部分。因此，原始意义的 GUI 是针对最终用户来说的，也就是一种供用户使用的图形操作界面。而从开发者的角度来说，这里的用户应该是基于应用层（Application Layer）开发的程序员，因为应用程序最终是要调用 GUI 的接口进行屏幕绘制工作，GUI 也变成了操作系统提供给应用层的一整套绘画函数接口。因此从开发意义上来讲，GUI 与图形系统是同一个概念。

这里，读者要明白一点：这里讲的 GUI 是一个综合概念，区别于下面具体的 GUI 函数接口。

5.2 图形系统结构

浏览源代码，大家会发现大量带有诸如 WGUI、GUI、GDI 字串的函数或者变量，这些都是 MTK 图

形系统的各个子系统中的函数。整个图形系统分为以下 3 个部分。

5.2.1 模板子系统

熟悉 Visual C++的读者对模板一定印象深刻：把那些功能相近的函数或类，取其原型进行归类就形成了模板。使用模板的好处就是可以简化大量的重复劳动。

在 MMI 的图形系统中，也使用了模板机制。例如平台中的 Category Screen、WGUI Controls 等都属于模板，开发人员只需要调用其提供的创建函数，就可以创建出基于系统的标准的控件和屏幕，无需耗费大量的时间自行设计。

WGUI，全称是 WrapperGUI，即包装后的 GUI，可以说 WGUI 最能够体现模板子系统的特性。其包含以下 4 个部分。

· Category Screen：屏幕模板集，与 Windows 系统中的窗口模板类似。开发人员可以直接调用屏幕模板函数，只需要传入自己的参数就可以快速产生一个屏幕。如常见的 ShowCategoryXScreen 函数。X 为数字，如待机界面使用的 ShowCategory33Screen 函数。

· WGUI Controls：控件模板，MTK 系统中控件的概念与 Windows 系统中控件的概念一致。MTK 为开发人员封装了大量的控件（如按钮、列表、输入框等）。

· Touch Screen：触摸屏应该算得上是 MTK 手机最重要的部分了，因此可以认为 Touch Screen 是图形系统中最核心的控制模块，主要负责触摸屏事件。

· Draw Manager：伴随 Touch Screen 一起出现，因 Touch Screen 模块为方便控制会记录 Screen 中所有元件的属性及状态，为减轻图形系统代码冗余，将 Category Screen 中的元件也统一交由 Draw Manager 绘制，DrawManager 会依据每个元件的属性及状态依次将其绘制出来。其详细过程我们将会单开辟一章来讲解。

5.2.2 绘画子系统

该部分只负责绘画，不负责与用户交流，通常系统提供的模板组件都有其相应的绘画类接口，MTK 把它划归为 GUI 层（GUI_Layer）。

在源代码中，如果是 GUI_XXX 之类的函数，一般都是绘图模块的接口，包含以下部分。

图形：即通过程序计算，以画点为基础绘制出各种几何图形。

图像：以外部的图像资源为基础，通过各种图像解码器绘制出相应的点阵图，例如 BMP、GIF 格式的图片。

文本：即文字或文本串的输出。

填充：将图形与图像整合到一起，绘制出相应的填充区域，通常用来作为各种元件的背景。

控件：每种控件都会有多个相应的绘画类接口以控制其在不同交互模式下的显示状态，这里的接口只负责绘制各种状态，不负责与用户交互，一般此类接口都是给 WGUI Controls 调用的。

说　明

以上所说的几种类型中控件、填充及文本是专属 GUI 的，图形与图像虽然在 GUI 中也有接口，但我们在应用层开发过程中更多的是直接使用 GDI 中的接口，因为 GUI 毕竟是对下层的 GDI 的封装，而且功能不是太完善。

有一点要注意，在很早期的版本中（主要是 2006 年以前的），比如 MT6225 的某个版本中，GUI 接口名称都是以 pixtel_UI_为前缀的，后来都改成以 gui_为前缀了，所以如果您的代码使用的是旧版本，要记得将实例中的代码做相应的改动。

另外我们还在 GUI Layer 中加了一个 Theme 模块，主要用来控制系统的显示风格。

5.2.3 设备子系统

也即 GDI（Graphical Device Interface），图形设备接口，GDI 将是我们应用开发重点使用的接口。

此类接口负责与设备协调，并对上层形成一套简洁统一的接口。一般情况下此类接口只能被图形系统内部使用，我们把其划归为 GDI 层。GDI 是更深入底层的接口，完全包含所有的图形图形功能点，我们推荐开发人员尽量使用 GDI 接口进行 UI 绘制处理工作。

GDI 使用句柄模式如下：

```
gdi_handle
GDI_HANDLE
```

使用句柄的好处就是可以方便地操纵 GDI 对象。GDI 对象包括一个图层、一张动画 GIF 图片、一张解码显示的 JPEG 图片等。

GDI 函数都会返回一个 GDI_RESULT 变量。开发人员可以通过这个变量判断函数内部的请求操作是否成功。

```
成功：GDI_RESULT>0
失败：GDI_RESULT<0
```

GDI 是线程安全的。通过 MUTEX 来保证某个线程执行时不会受到其他线程的影响。

GDI 的数据类型：

```
typedef U16 gdi_color;
typedef S32 gdi_handle;
typedef S32 gdi_result;
#define GDI_COLOR gdi_color
#define GDI_HANDLE gdi_handle
#define GDI_RESULT gdi_result
```

GDI 层分为以下几个部分。

1. 图形类（Graphic）接口

GDI 中的图形绘制一般会有相应的硬件加速，并提供如下基本的绘制功能。

（1）颜色格式转换

相关的宏：

```
GDI_RGB_TO_BUFFER_FORMAT(R,G,B)
GDI_BUFFER_FORMAT_TO_RGB_R(VALUE)
GDI_BUFFER_FORMAT_TO_RGB_G(VALUE)
GDI_BUFFER_FORMAT_TO_RGB_B(VALUE)
```

来看一个代码片段，如图 5.1 所示。

```
static void mmi_camera_preview_clear_background(void)
{
  /*----------------------------------------------------------------------*/
  /* Local Variables                                                      */
  /*----------------------------------------------------------------------*/
  gdi_color bg_color;
  /*----------------------------------------------------------------------*/
  /* Code Body                                                            */
  /*----------------------------------------------------------------------*/
  gdi_layer_push_and_set_active(g_camera_cntx.base_layer_handle);
  if (g_camera_cntx.active_camera_osd_ptr->bg.is_draw_bg_color)
  {
    bg_color = GDI_RGB_TO_BUFFER_FORMAT(
            g_camera_cntx.active_camera_osd_ptr->bg.r,
            g_camera_cntx.active_camera_osd_ptr->bg.g,
            g_camera_cntx.active_camera_osd_ptr->bg.b);
```

图 5.1 颜色格式转换的宏定义

我们可以按照规则，编写我们需要的 RGB 值转换的代码：

```
gdi_color     blue_color;
blue_color = GDI_RGB_TO_BUFFER_FORMAT(0,0,255);//blue_color 将代表蓝色
```

（2）绘制像素点

代表函数：

```
gdi_draw_point
```

（3）绘制直线及各种类型的线条

代表函数：

```
gdi_draw_line
 gdi_draw_line_style
```

（4）绘制矩形及各种类型的框

代表函数：

```
gdi_draw_rect
 gdi_draw_solid_rect
  gdi_draw_frame_rect
  gdi_draw_round_rect
  gdi_draw_button_rect
gdi_draw_shadow_rect
gdi_draw_gradient_rect
```

2. 图像（Image）文件的解码和绘制

图像分为静态图片与动画两种，每种格式图像在 GDI 中都有其相应解码器，GDI 会依据图像格式自动寻找相应解码器。

（1）提供如下绘制功能：

绘制静态 GIF 图像和 GIF 动画；

绘制 BMP 图像；

绘制 WBMP 图像；

绘制 PBM 图像；

绘制 JPEG 图像；
获得图像文件的尺寸；
获取动画文件的帧数；
能够读取来自资源的图片 ID 和来自文件的图片内容。
（2）相关的 API 概览（在下一章详解）：
gdi_image_draw（_id 或者_file）；
gdi_image_draw_resized；
gdi_image_draw_animation；
gdi_image_draw_animation_once；
gdi_image_draw_animation_frames；
gdi_image_draw_animation_single_frame；
gdi_image_get_dimension；
gdi_image_stop_animation；
gdi_image_stop_animation_all。
（3）解码器支持的范围：
支持 1bit、4bit、8bit、16bit、24bit、32bit 的 BMP 格式文件；
支持 GIF87、GIF89a 的 GIF 格式文件，支持静态或动态的 GIF 文件；
支持非渐进模式的 JPEG 格式文件；
只支持黑白色的 WBMP 格式文件；
支持 MTK 自创的 PBM 格式文件。
（4）解码器相关的 API：
gdi_image_decoder_create；
gdi_image_decoder_free；
gdi_image_decoder_decode_file；
gdi_image_decoder_decode_file_with_output_clipping；
gdi_image_decoder_get_dimension_file；
gdi_image_decoder_stop；
gdi_image_decoder_stop_all。

3. Font 即字库与字体管理

用以绘制单个文字。不过，在实际开发应用中，GDI 中的 Font 很少使用，而且其提供的 API 很少，应该优先使用 GUI 封装的 Font 函数。

4. LCD&图层（Layer）

控制每个硬件屏幕（目前主要是主屏及副屏）以及与屏幕相辅的缓冲区 Layer 管理。MTK 芯片组最多支持 4 个图层，使用多图层能够使开发人员设计出细腻的操作界面和华丽的特效，以此提升整个 UI 界面的用户欢迎度。关于 Layer 管理，我们会在后面的章节中重点讲解。

如果 GDI 与 GUI 的接口有功能重叠，建议尽量使用 GUI 的，因为 GDI 接口随硬件变化可能会有改变。

通过上面的分析，我们可以看到，整个图形系统具有一定的层次关系：最上层是模板子系统，接着是绘画子系统、设备子系统，这是一个层层调用的关系。

结语：

本章对 MTK 图形系统做了一个大致的梳理，可以作为开始 UI 设计的启蒙。读者学完本章后应该对我们的应用开发所处的位置有一个清晰的认识，了解了应用开发过程和图形系统之间的联系，清楚了 GUI、WGUI、GDI 之间的关系，并认识到 GDI 在上层界面开发过程中的重要位置。

6 基本绘图

在这一章中，我们将通过若干实例来为读者讲解如何绘制各种基本元素，应当调用哪些API，包含哪些基本元素以及绘图过程中需要注意的细节等。

通过本章的学习，读者可以掌握文本、图形、图像、背景等核心元素的绘制和原理。

6.1 文本绘制

前面我们已经通过一个实例，在屏幕上绘制了一个“Hello MTK!”字符串。

```
void EntryFirstAppScreen(void)
{
 clear_screen();                                                    //将屏幕清成空白
 gui_move_text_cursor(100,100);
 gui_print_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));  //打印字符串
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);   //重绘屏幕
}
```

这些函数都属于文本绘制 API 的范围，上面所用到的仅是众多文本绘制函数中的冰山一角。这些函数都位于文件 GUI.c 之中，大家可以打开该文件，查看各个函数的原型。

限于篇幅，这里只讲解最常用的一些函数。

6.1.1 基本函数

- gui_move_text_cursor

函数原型：void (*gui_move_text_cursor) (S32 x, S32 y) = NULL;

用来设定绘制的文本所处的屏幕坐标，坐标原点是屏幕的左上角。

- gui_print_text

函数原型：void (*gui_print_text) (UI_string_type _text) = NULL;

这个函数是 MMI 开发中最常用的文本绘制函数，其参数可以有多种形式，示例如下。

➢ gui_print_text(L"Hello MTK");

使用 L 宏的参数，将只会在模拟器调试的时候有效，在真机上将不会支持这种方式。如 L"Hello MTK!" 表示将 ANSI 字符串转换成 Unicode 的字符串，防止乱码产生。在模拟器上，还可以用 L 宏来显

示中文（前提是软件版本已经包含中文字库）。如：

gui_print_text(L"世界，你好！");

➢ gui_print_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));

这种参数形式是我们推荐的。因为将需要显示的字符串写入资源，在需要的情况下通过 GetString 再读取，可以很方便地实现多国语言转换并且在真机上也可以正确显示。

➢ gui_print_text ((UI_string_type)Str);

可以传递一个 S8(char)类型字符数组指针。当你在开发过程中，如果需要将实时获得的字符串打印出来，就可以通过上面的方式来定义一个 S8 的字符数组，并将你的字符串赋值给 Str 数组，然后将 Str 传入 gui_print_text。

如下示例所示：

```
S8 Str[16];
memset(Str,0,16);
S8 tmpStr[16] = 从文件获取的内容
//用于真机显示的转换函数
mmi_chset_text_to_ucs2((U8*) Str, 16, (U8*)tmpStr );
//模拟器上可以用下面的函数进行转换
AnsiiToUnicodeString(Str,, tmpStr);
gui_print_text ((UI_string_type)Str);
```

编译成功后，烧录到真机，即可看见正确的字符串显示。

说　明

mmi_chset_text_to_ucs2 函数将中文字符串转换为 UCS2 编码格式，这样保证在真机上的正确显示，不会出现乱码。AnsiiToUnicodeString 也具有同样的功能，具体细节参考文件：Conversions.c。

• gui_print_character

函数原型：void (*gui_print_character) (UI_character_type _c) = NULL;

该函数将会在屏幕上输出一个字符。

例：

gui_print_character((UI_character_type)'[');

在屏幕上输出一个字符“[”。

• gui_print_bordered_text

函数原型：void (*gui_print_bordered_text) (UI_string_type _text) = NULL;

输出一个带边框的文本。

gui_print_bordered_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));

• gui_measure_string

函数原型：

void (*gui_measure_string) (UI_string_type text, S32 *width, S32 *height) = NULL;

用于测量一个字符串在屏幕上占多少个像素长度和高度（注意不是字符串的字节数）。

示例如下：

```
int width,height = 0;
char* str = "This is a string!";
gui_measure_string((UI_string_type) str &swidth, &height);
```

通过代码调试，可以看到“width=163，height=20.”该函数在进行文本开发排版的时候特别有用，可以通过这种方式，精确地算出屏幕上的位置。

· gui_itoa

函数原型：UI_string_type(*gui_itoa) (S32 value, UI_string_type s, S32 radix) = NULL;

用于将整形数据转换为字符串，其中 radix 指的是将当前的数字转换为多少进制。

例如要想输出数字 25，如以下代码所示：

```
S16 num = 25;
S8 outPut[8];
gui_itoa(num,(UI_string_type) outPut,10);
gui_print_text(outPut);
```

与之对应的函数是 gui_atoi，其作用和 gui_itoa 相反，是将字符串转换为整形数据，读者可以自己查看。

6.1.2 字体

在文本开发中，字体的设置是最为常见的工作。

在 MTK 系统中，字体属性都被封装在一个名为 stFontAttribute 的结构体之中。

```
typedef struct stFontAttribute
{
   U8 bold;
   U8 italic;
   U8 underline;
   U8 size;
   U8 color;
   U8 type;
   U8 oblique;
   U8 smallCaps;
}
```

成员解释如下。

Bold：粗体（0：正常，1：粗体）。

italic 斜体（0：正常，1：粗体）。

underline 下划线（0：正常，1：下划线）。

size：字号大小。

常用字号有以下几种。

SMALL_FONT：小号。

MEDIUM_FONT：中号。

LARGE_FONT：大号。

SUBLCD_FONT：副屏字体。

DIALER_FONT：拨号字体。

VIRTUAL_KEYBOARD_FONT：虚拟键盘字体。

Color：颜色，该属性暂时无效，颜色设置由函数 gui_set_text_color 实现。

Type：字体类型，暂时无效。

Oblique：倾斜，比 italic 的倾斜角度小。

Smallcaps：小写锁定，暂时无效。

例如，我们设置一个无下划线的正常状态的中号字体：

```
stFontAttribute newfont = {0, 0, 0, MEDIUM_FONT, 0,0};
stFontAttribute f = newfont;
gui_set_font(&newfont);
```

说　明

上述字体设置只对英文字符有效，汉字及其他象形文字将不受影响，汉字的大小是由字库文件决定的。

6.1.3 颜色

做 UI 开发时，文本的颜色肯定不能是千篇一律的黑色，MTK 系统为开发人员提供了多种默认的颜色。

在 GUI 层中，为开发人员预定了以下常用颜色：

```
CONST_HARDWARE color UI_COLOR_LIGHT_GREY = {120, 120, 120, 120};
CONST_HARDWARE color UI_COLOR_WHITE = {255, 255, 255, 100};
CONST_HARDWARE color UI_COLOR_DARK_GREY = {64, 64, 64, 100};
CONST_HARDWARE color UI_COLOR_BLACK = {0, 0, 0, 100};
CONST_HARDWARE color UI_COLOR_NONE = {0, 0, 0, 0};
CONST_HARDWARE color UI_COLOR_RED = {255, 0, 0, 100};
CONST_HARDWARE color UI_COLOR_3D_FILLER = {180, 180, 200, 100};
CONST_HARDWARE color UI_COLOR_GREY = {192, 192, 192, 100};
CONST_HARDWARE color UI_COLOR_GREEN = {0, 255, 0, 100};
```

在更下一层的 GDI 中，也有预定义的颜色：

```
#define GDI_COLOR_WHITE        gdi_act_color_from_rgb(255, 255, 255, 255)
#define GDI_COLOR_BLACK        gdi_act_color_from_rgb(255, 0, 0, 0)
#define GDI_COLOR_GRAY         gdi_act_color_from_rgb(255, 127, 127, 127)
#define GDI_COLOR_RED          gdi_act_color_from_rgb(255, 255,   0,   0)
#define GDI_COLOR_BLUE         gdi_act_color_from_rgb(255,   0,   0, 255)
#define GDI_COLOR_GREEN        gdi_act_color_from_rgb(255,   0, 255,   0)
```

开发人员可以在需要的时候直接使用这些预定义的颜色值。如：

```
gui_set_text_color(UI_COLOR_BLACK);
gdi_color bg_color = GDI_COLOR_BLACK;
```

在调用这些颜色值的时候，需要注意当前函数的参数是支持 GUI 还是 GDI。

同时可以看到，不同的颜色可以直接通过 RGB 值组合而来，例如“RED=255, BLUE = 0, GREEN = 0”，组合起来的颜色就是红色。

所以，可以用以下方式来设定我们需要的颜色：

```
GUI:
color my_color = {20, 20, 60, 100};                    //my_color 将是深蓝色
GDI:
gdi_color my_gdi_color = gdi_act_color_from_rgb(255, 255, 103, 102);
```

说 明

my_color 一行中的最后一个参数“100”与 my_gdi_color 一行中的第一个参数“255”表示 Alpha 值（透明度），此参数目前用处不大，可忽略。

现在，我们用前面定义的字体来绘制一个带边框的文本，边框颜色为绿色，文本为红色。

```
void EntryFirstAppScreen(void)
{
  stFontAttribute newfont = {0, 0, 0, MEDIUM_FONT, 0,0};
  color text_color = {255, 0, 0, 0};
  color border_color = {0,255,0,0};
  stFontAttribute f = newfont;
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();
  gui_move_text_cursor(100,100);                          //设置绘制坐标
  gui_set_text_color(text_color);                         //设置文本颜色

  gui_set_font(&newfont);                                 //使用自定义的字体
  gui_set_text_border_color(border_color);                //设定边框颜色
  gui_print_bordered_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));

  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);    //重绘屏幕

  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

保存代码，编译运行模拟器，即可看到文字周围布满了带有色彩的边框，效果如图 6.1 所示。

图 6.1 输出结果

6.1.4 排版

想要精确排版要先测量好字串的宽与高，我们将利用前面讲到的排版函数 gui_measure_string 来演示如何将字串左右居中。

```
void EntryFirstAppScreen(void)
{
  int width,height,x_pos,y_pos= 0;
  EntryNewScreen( SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();                                  //将屏幕清成空白
  clear_screen();
  gui_set_text_color(gui_color(255,0,0));
  gui_measure_string((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK),
  &width,&height);                                      //这两句即是进行居中对齐的核心语句
  x_pos = (UI_device_width - width) / 2;
  y_pos = (UI_device_height - height) / 4;
  gui_move_text_cursor(x_pos, y_pos);
  gui_print_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

编译运行模拟器，可以看见先前的字符串“HelloMTK”已经居中对齐了，如图 6.2 所示。

通过下图，我们来认识一下在进行排版过程中需要了解的屏幕区域。在上面的代码中我们经常看到绘图函数用的一些常量：

```
gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
```

UI_device_width 和 UI_device_height 分别表示当前屏幕的宽度和高度。

这些常量在程序开发中很常见，其他的还有诸如标题栏、状态栏、菜单项的高度和长度等常量，如图 6.3 所示。

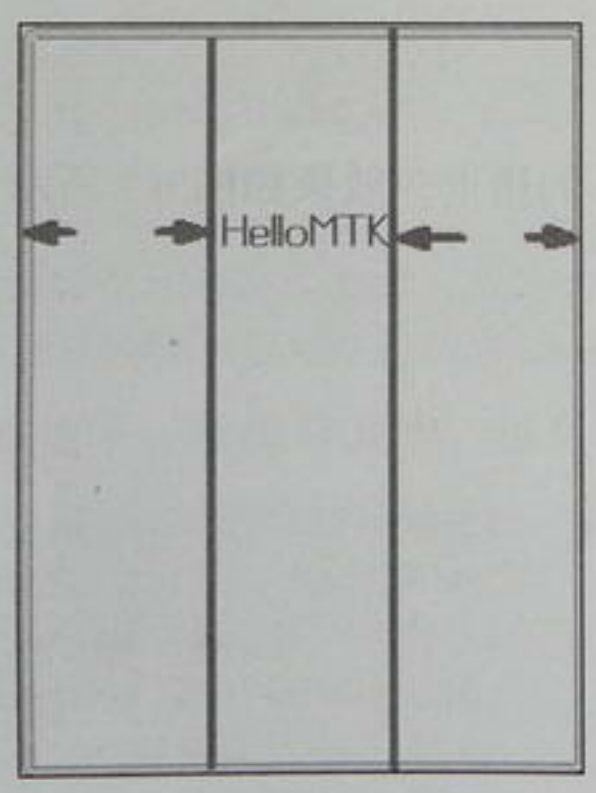


图 6.2 居中对齐

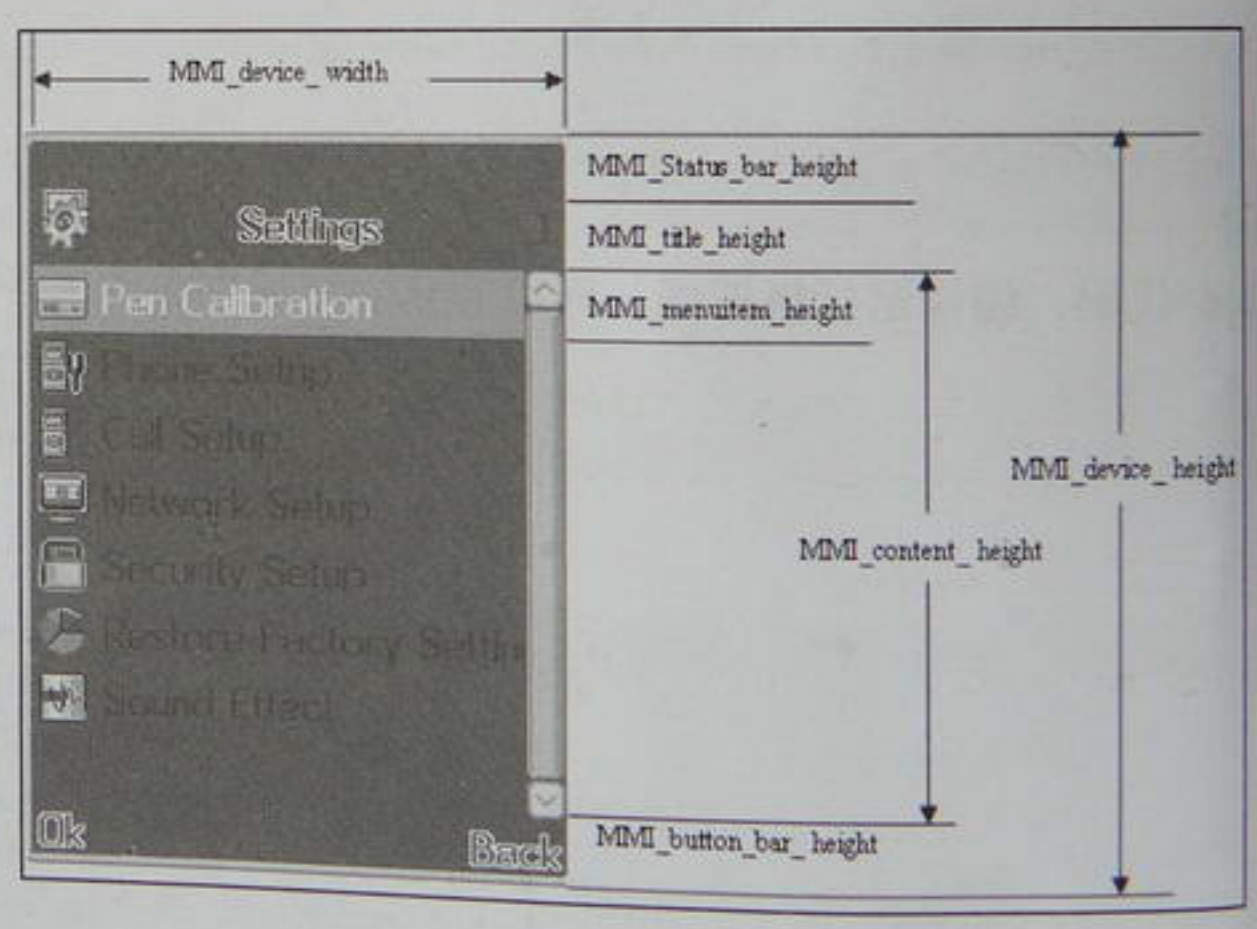


图 6.3 排版常量

请牢记这些常量所表示的含义，因为系统源代码大量使用了这些常量。

虽然你仍然可以通过写入实际的变量来替代这些常量，但是这样做不便于后期代码维护。因此，推荐大家用常量的方式来编写代码，这也是一个良好的编程习惯。

6.2 图形绘制

在上一章已经讲过，图形都是通过程序的计算，用一个个像素点画出来的，因此，图形的基础就

是像素点，绘制图形的函数就从绘制像素点开始。

· gui_putpixel

```
gui_putpixel(UI_device_width / 2, UI_device_height / 2, UI_COLOR_BLACK);
```

函数原型：void (*gui_putpixel) (S32 x, S32 y, color c) = NULL;

参数 x，y 分别表示在屏幕的何处绘制像素点，c 表示一个 GUI 颜色。点的聚集又可以组成一个最简单的图形——线。

➢ gui_line

函数原型：void (*gui_line) (S32 x1, S32 y1, S32 x2, S32 y2, color c) = NULL;

画线函数还可以细分为两个功能更具体的函数。

➢ gui_draw_horizontal_line

函数原型：void (*gui_draw_horizontal_line) (S32 x1, S32 x2, S32 y, color c) = NULL;

该函数与 gui_line 相比，省略了一个参数，用于绘制水平直线。所以，只需要一个 *y* 轴坐标即可。

➢ gui_draw_vertical_line

函数原型：void (*gui_draw_vertical_line) (S32 y1, S32 y2, S32 x, color c) = NULL;

该函数与 gui_line 相比，省略了一个参数，用于绘制垂直直线。所以，只需要一个 *x* 轴坐标即可。

· gui_draw_rectangle

函数原型：void (*gui_draw_rectangle) (S32 x1, S32 y1, S32 x2, S32 y2, color c) = NULL;

该函数将在屏幕上绘制一个矩形。

· gui_fill_rectangle

函数原型：void (*gui_fill_rectangle) (S32 x1, S32 y1, S32 x2, S32 y2, color c) = NULL;

该函数将在屏幕上绘制一个填充了 c 颜色的实心矩形。

以该函数为基础，下面还有许多衍生的矩形绘制函数，诸如百叶窗效果、十字纹效果的填充矩形。但这些很少用到，感兴趣的读者可以自己去尝试。

· gdi_draw_frame_rect

函数原型：void gdi_draw_frame_rect(S32 x1, S32 y1, S32 x2, S32 y2,
gdi_color rect_color,
gdi_color frame_color,
S32 frame_width)

该函数将会在屏幕上绘制一个带边框的填充矩形。

下面我们来做一个带框的填充矩形，并且该矩形包含“HelloMTK”文本。

```
void EntryFirstAppScreen(void)
{

  stFontAttribute newfont = {0, 0, 0, MEDIUM_FONT, 0,0};
  color text_color = {255, 0, 0, 0};
  color border_color = {0,255,0,0};
  stFontAttribute f = newfont;
  S32 width = 0;
```

```
    S32 height = 0;
    S32 x_pos = 0;
    S32 y_pos= 0;
    EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
    entry_full_screen();
    clear_screen();

    gui_set_text_color(text_color);                    //设置文本颜色

    gui_set_font(&newfont);                            //使用自定义的字体
    gui_set_text_border_color(border_color);           //设定文本边框颜色

    x_pos = (UI_device_width - width) / 2;             //这两句即是进行居中对齐的核心语句
    y_pos = (UI_device_height - height) / 4;
    gui_move_text_cursor(x_pos,y_pos);                 //设置绘制坐标
    //测量字符串宽度和高度
    gui_measure_string((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK),
                       &width,&height);
    //绘制带边框的填充矩形。边框色为 GDI_COLOR_RED，填充色为亮绿色
    //边框的宽度为 3 个像素
    gdi_draw_frame_rect(x_pos-4, y_pos-4, x_pos+width+4, y_pos+height+4,
                        gdi_act_color_from_rgb(255, 0, 255, 255), GDI_COLOR_RED, 3);


    gui_print_bordered_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));

    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕

    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);

}
```

说　明

在上面的函数 gdi_draw_frame_rect 中，大家可能会问：为什么频繁地减去 4 和增加 4？

其实，这是调整坐标的方法。因为矩形边框不可能紧贴文字，必须要有一定的范围预留。通过对文字坐标的像素校准，就可以让矩形框很自然地包含文字而不会贴在一起。读者可以将上述的−4 和+4 去掉，然后在模拟器上看看效果，就明白了。

同时，要注意的是，这种直接在函数里面写像素具体值的编码风格不好，因为，如果一个代码里面需要修改像素，调整坐标的地方很多。假如有 100 个，每个变量可能都需要集体增加和减少若干像素。如果我们直接编写：var1+4，var2−4，var3*4……那可以想象，在后期，如果发现 4 个像素太少了，要移动 7 个像素的话，岂不是要修改 100 个地方！

因此，我们建议大家在正式项目中，将变量定义为一个宏，这样更加方便修改。我们将上述代码修改为：

```
//在文件头部定义像素校准值
#define PIXTEL_ADJUST 4

...
void EntryFirstAppScreen(void)
{
```

```
    stFontAttribute newfont = {0, 0, 0, MEDIUM_FONT, 0,0};
    color text_color = {255, 0, 0, 0};
    color border_color = {0,255,0,0};
    stFontAttribute f = newfont;
    S32 width = 0;
    S32 height = 0;
    S32 x_pos = 0;
    S32 y_pos= 0;
    EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,
                   NULL,NULL);
                   entry_full_screen();
                   clear_screen();
    gui_set_text_color(text_color);                //设置文本颜色

    gui_set_font(&newfont);                        //使用自定义的字体
    gui_set_text_border_color(border_color);       //设定文本边框颜色

    x_pos = (UI_device_width - width) / 2;         //这两句即是进行居中对齐的核心语句
    y_pos = (UI_device_height - height) / 4;

    gui_move_text_cursor(x_pos,y_pos);             //设置绘制坐标
    //测量字符串宽度和高度

    gui_measure_string((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK),
                       &width,&height);

    gdi_draw_frame_rect(x_pos-PIXTEL_ADJUST, y_pos-PIXTEL_ADJUST,
                        x_pos+width+PIXTEL_ADJUST, y_pos+height+PIXTEL_ADJUST,
                         gdi_act_color_from_rgb(255, 0, 255, 255), GDI_COLOR_RED, 3);

    gui_print_bordered_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));

    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

保存上述代码，编译并运行模拟器，即可在屏幕上看到一个带有底色的带框矩形，如图 6.4 所示。

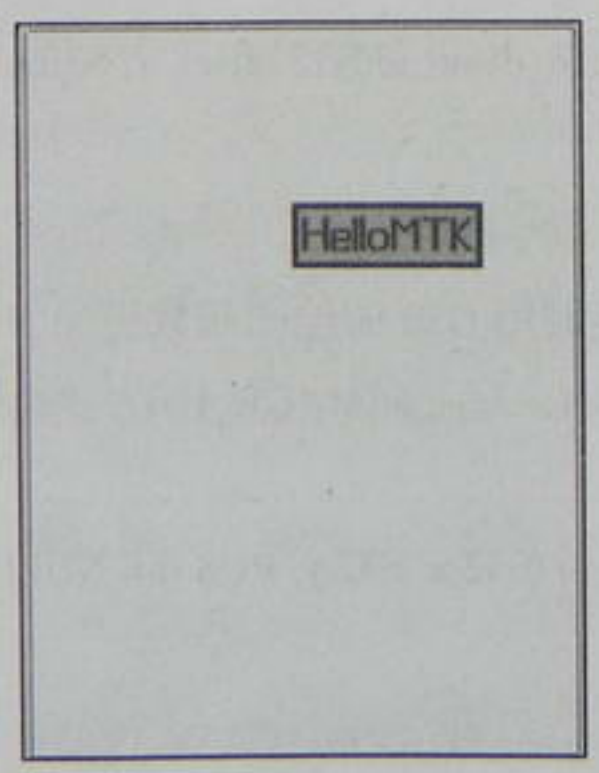


图 6.4　带框的文字

除此之外，系统还提供多种填充矩形 API，其使用方式都和上面的代码类似。

6.3 图像

我们再一次强调：在计算机科学中，图形和图像这两个概念是有区别的：

图形一般指用计算机绘制的画面（在 6.2 节中已经讲过），如直线、圆、圆弧、曲线和图表等；

图像则是指由输入设备捕捉的实际场景画面或以数字化形式存储的任意画面。

图像是由一些排列的像素组成的，在计算机中的存储格式有 BMP、PCX、TIF、GIFD 等，一般数据量比较大。它除了可以表达真实的照片外，也可以表现复杂绘画的某些细节，并具有灵活和富有创造力等特点。

图像按显示方式可分为静态图像与动画两种。

在开始后面的学习之前，我们先增加两张 GIF 图片。

一张具有动画效果的 GIF：IMG_FIRSTAPP_ANIMATE；

一张静态的动画 GIF：IMG_FIRSTAPP_PICTURE。

图像的加入请参考第 4 章的讲解。

6.3.1 静态图像

按图像格式可分为 BMP、JPG、GIF、PNG 等。

按存储方式可分为以下两种。

1. 资源

第 4 章的图像资源是我们平台内部用的最多的一种存储方式，也是我们建议开发人员使用的一种方式。

资源又分两种使用方式。

（1）读入资源 ID。图片通过一个 ID 对应实际的文件，对应函数如下。

· gdi_image_draw_id

函数原型：GDI_RESULT gdi_image_draw_id(S32 offset_x, S32 offset_y, U16 image_id);

示例如下：

```
gdi_image_draw_id(10, 10, IMG_ID_NEW_ICON);
```

对比 GUI 的绘制方式，这种直接调用 GDI 的绘图函数更加方便，参数传入图片 ID 即可。

（2）资源 Buffe。资源 Buffer 即以 GetImage(IMAGE_ID)方式由资源 ID 转换过来。对应函数如下。

· gui_show_image

函数原型：void (*gui_show_image) (S32 x, S32 y, PU8 i) = NULL;

示例如下：

```
gui_show_image(10,10,(PU8) GetImage (IMG_ID_NEW_ICON));
```

该函数将会在屏幕坐标（10,10）处，绘制 ICON 图标。

· gdi_image_draw

函数原型：GDI_RESULT gdi_image_draw(S32 offset_x, S32 offset_y, U8 *image_ptr);

示例如下：

```
gdi_image_draw(10, 10, get_image(IMG_ID_NEW_ICON));
```

该函数将会在屏幕坐标（10,10）处绘制 ICON 图标。

> *说　明*
>
> 在代码中，有两个转换资源 ID 的函数。一个是原始的 GetImage 函数，另一个则是最前面进行封装的 get_image 函数。区别不大，开发人员可以按需要使用这两个函数。

2. 文件

即在系统运行时从文件系统中动态获取的图像。

示例如下：

```
gdi_image_draw_file(10, 10, (S8*)L"E:\\MTK6226\Plutommi\mmi\Firstapp\New_Icon.gif");
```

该函数将会在屏幕坐标（10,10）处绘制一个 GIF 图片文件，该文件来源于一个文件目录。

> *注　意*
>
> gdi_image_draw_file 的文件存储路径参数必须转换成 Unicode 编码格式，否则，将不能正确绘制图片。

6.3.2　动画

动画，其实是多张静态图片的组合。当程序连续播放每张图片时，由于每张图片给人的视觉暂留效果，就形成了一组动态的画面。

· gdi_anim_draw_id

该函数调用动画图片文件的 ID，进行绘制。

示例如下：

```
gdi_handle my_anim;
void EntryFirstAppScreen(void)
{
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();
 gdi_anim_draw_id(50,100,IMG_FIRSTAPP_ANIMATE, &my_anim);
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

通过上面的代码，在屏幕坐标（50,100）上绘制一个动画，如图 6.5 所示。

在 MT6226_06B 的软件版本中，为函数 gdi_anim_draw_id 增加了一个宏替换。

```
#define gdi_image_draw_animation_id      gdi_anim_draw_id
```

所以，如果你是用的该版本，最好遵循其原则，使用 gdi_image_draw_animation_id。

图 6.5 动画效果

• gdi_anim_stop

该函数将暂停动画的播放。

示例如下：

```
gdi_handle my_anim;
void stop_animate(void)
{
 gdi_anim_stop(my_anim);
}
void EntryFirstAppScreen(void)
{
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();
 gdi_anim_draw_id(50,100,IMG_FIRSTAPP_ANIMATE, &my_anim);
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
 SetKeyHandler(stop_animate,KEY_LSK,KEY_EVENT_UP);
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

说 明

可能有读者会问：为什么 gdi_anim_stop 不直接使用，而一定要用函数 stop_animate 来封装一次呢？这是由于 SetKeyHandler 中需要传入一个函数地址，而 gdi_anim_stop 中会包含参数。所以，通过一个无参数的函数封装，就可以顺利地传入 SetKeyHandler。这种做法在 MTK 软件代码中很常见。

可以看到，动画的起始播放和停止，都需要一个 gdi_handle 句柄。

这个 gdi_handle 实际上是一个 layer（图层）句柄。一个图层都跟一个句柄相关，句柄实际上就表示它掌控着这个图层的资源的代理和使用权，当程序需要调用某个图层的时候，只需要找到句柄，就可以获得图层的使用权，我们可以把句柄看成是一个图层的索引。关于句柄的深入知识，可以参考 Windows 相关书籍。

6.3.3 图像排版

在前面的文字排版中，我们学习了字符串测量函数 gui_measure_string。在图像处理中，也有函数实现类似的功能。

· gdi_image_get_dimension_id

函数原型：

```
GDI_RESULT gdi_image_get_dimension_id(U16 image_id, S32 *width, S32 *height)
```

示例如下：

```
void EntryFirstAppScreen(void)
{
  int x_pos,y_pos,width,height = 0;
  EntryNewScreen( SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();//将屏幕清成空白
  //读取图片的尺寸信息，将宽度和长度值分别赋值给 width 和 height
  gdi_image_get_dimension_id(IMG_ICON 改成新图片, & width, height);
  x_pos = (UI_device_width - width) / 2;//进行居中处理
  y_pos = (UI_device_height - height) / 2;
  gdi_image_draw_id(x, y, IMG_ICON);//绘制图片
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

按照上述代码进行修改、保存、编译、运行模拟器，即可在屏幕上看到，绘制的图片已经居中对齐。

6.4 区域裁剪

在这里说的裁剪并不是一般意义上的对字符串或者图片内容进行截断，而是指在屏幕显示上，通过一些遮挡让字符串显示需要的部分，而字符串实际内容不变。

比如：本来的字符串是“This is a first application”，当由于一些原因，只能显示前面两个单词时，即“This is”，那么，裁剪显示就派上了用场。

看一下下面这对函数。

· gui_push_clip

函数原型：void (*gui_push_clip) (void) = NULL;

· gui_pop_clip

函数原型：void (*gui_pop_clip) (void) = NULL;

这对函数是用来进入和恢复图像剪切区的。当我们需要对某个图片或者字符串进行裁剪显示，一般遵循如下 3 个步骤。

（1）调用函数 gui_push_clip 来进入剪切区域。

（2）调用函数 gui_set_clip 来设置剪切区域的范围。

（3）调用函数 gui_pop_clip 恢复本来的剪切区域。

例如刚才显示在屏幕上的字符串“HelloMTK!”，如果现在我只想显示 Hello 这个单词，操作如下：

```
void EntryFirstAppScreen(void)
{
  EntryNewScreen( SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();                                  //将屏幕清成空白
  gui_push_clip();                                 //进入剪切区域
  gui_set_text_color(gui_color(255,0,0));
  gui_move_text_cursor(50,100);
  gui_set_clip(50,100,89,120);                     //设置要剪切区域的范围
  gui_print_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));
  gui_pop_clip();                                  //退出刚才的剪切区域
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

同样的方法，可以使用 gui_set_clip 来裁剪图像。

说　明

在这里，我们来说一下 gui_set_clip 的参数。

函数原型：void (*gui_set_clip) (S32 x1, S32 y1, S32 x2, S32 y2) = NULL;

需要开发人员传入 4 个顶点坐标。以上面的例子为例。

我们需要显示左边的 Hello 字符串。所以，我们设置 x1，y1 为原字符串的起始坐标，即（50,100）。然后，根据前面我们通过字符串测量函数得到的字符宽度，酌情设置右下角的坐标。

我们最终确定了 x2，y2 为（89,120）。字符串的剪切区域就形成了。

按照上述代码修改后，编译、运行模拟器，进入我们新增的屏幕，你就会发现，当前屏幕上只显示了字符串“HelloMTK”的前半段即“Hello”，如图 6.6 所示。

现在，我们来讲一下对图像的剪切。

现有一幅 GIF 静态图片，如图 6.7 所示。

图 6.6　裁剪后的文字

图 6.7　测试图片

该图片尺寸为 150 × 150，现在，我们需要显示图片的下半部分即房子。

```
void EntryFirstAppScreen(void)
{
  EntryNewScreen( SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();//将屏幕清成空白
  gui_push_clip();
  gui_set_clip(50,175,200,275);
  gdi_image_draw_id(50,100,IMG_FIRSTAPP_PICTURE);
  gui_pop_clip();
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

按照上述代码修改后，编译、运行模拟器。进入我们新增的屏幕，可以看到，图片只显示下方有房子的部分，如图 6.8 所示。

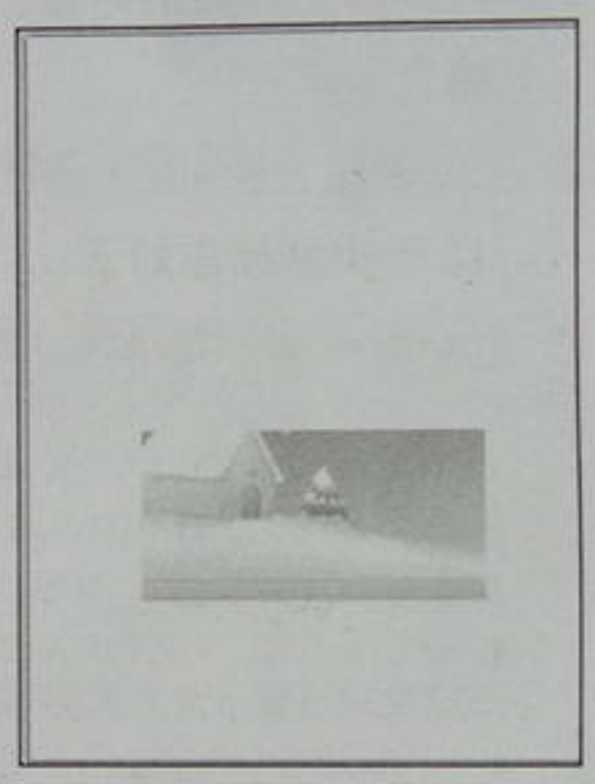


图 6.8 “裁剪”后的图片

同时，关注下面两个函数。

· gui_get_clip

函数原型：void (*gui_get_clip) (S32 *x1, S32 *y1, S32 *x2, S32 *y2) = NULL;

通过该函数的调用，获得当前激活的裁剪区的大小。即在准备产生新的剪切区之前，检查一下系统上次设置的剪切区。

· gui_reset_clip

函数原型：void (*gui_reset_clip) (void) = NULL;

该函数用来重置图像剪切区到默认范围，即屏幕大小。

6.5 背景绘制

在上面的例子中，我们都是在一个纯白色的界面上绘制图片和字符串，这自然无法体现出界面的美观性。

在界面设计中，有一个很简单但很有效的增加美观性的方法，那就是为元件绘制不同的背景。图形系统中每种可视的元件都可以为其加上背景，背景是元件显示风格的最大衬托，也是整个界面效果的重要组成。

如果将所有元件的背景统一修改，则很容易使整个系统的显示风格发生变化，所以通常的图形系统会将所有元件的背景集中起来，再加一些别的容易控制又明显的显示元素，就形成了整个系统的主题（Theme）。

MTK 系统中的主题大部分是以背景为基础的，所以掌握背景的使用方法，将为我们掌握主题的修改方法打下基础，特别是后面要讲解的高仿界面的修改。

6.5.1 填充结构体

每个背景实际是一个图形或图像填充起来的矩形区域，很显然，这个区域的属性需要由某个数据结构来控制。MTK 提供了一个填充结构体 UI_filled_area，其定义如下：

```
typedef struct _UI_filled_area
{
    U32 flags;                                          //控制背景具体类型的标志
    PU8 b;                                              //背景图片
    gradient_color *gc;                                 //渐进颜色
    color c;                                            //背景色
    color ac;                                           //用于绘制交叉填充图形的替换色
    color border_color;                                 //边框颜色
    color shadow_color;                                 //阴影颜色
    UI_transparent_color_type transparent_color;        //透明色
} UI_filled_area;
```

在这个结构体中，flags 是最重要的一个标志，其控制了整个背景填充的风格。图形系统默认定义了多种标志，如类型标志、边框标志、阴影标志。

组合公式：

```
flags = 类型标志 | 边框标志 | 阴影标志
```

12 个类型标志如下：

```
#define UI_FILLED_AREA_TYPE_COLOR                       //颜色
#define UI_FILLED_AREA_TYPE_GRADIENT_COLOR              //渐进色
#define UI_FILLED_AREA_TYPE_TEXTURE                     //纹理
#define UI_FILLED_AREA_TYPE_BITMAP                      //位图
//百叶窗（背景色 c 与原始底色交替）
#define UI_FILLED_AREA_TYPE_HATCH_COLOR
//交叉百页窗（背景色 c 与替换色 ac 交替）
#define UI_FILLED_AREA_TYPE_ALTERNATE_HATCH_COLOR
//十字图纹（背景色 c 与原始底色交替）
#define UI_FILLED_AREA_TYPE_CROSS_HATCH_COLOR
//交叉十字纹（背景色 c 与替换色 ac 交替）
#define UI_FILLED_AREA_TYPE_ALTERNATE_CROSS_HATCH_COLOR
#define UI_FILLED_AREA_TYPE_NO_BACKGROUND               //无背景
#define UI_FILLED_AREA_TYPE_CUSTOM_FILL_TYPE1           //自定义背景 1
#define UI_FILLED_AREA_TYPE_CUSTOM_FILL_TYPE2           //自定义背景 2
#define UI_FILLED_AREA_TYPE_3D_BORDER                   //3D 边框
```

8 个边框标志如下：

```
#define UI_FILLED_AREA_BORDER                           //单边框（一个像素宽）
#define UI_FILLED_AREA_SINGLE_BORDER                    //单边框，同上
#define UI_FILLED_AREA_DOUBLE_BORDER                    //双边框（两个像素宽）
#define UI_FILLED_AREA_ROUNDED_BORDER                   //圆角边框
#define UI_FILLED_AREA_3D_DEPRESSED_BORDER              //下陷边框（类似按钮下陷效果）
#define UI_FILLED_AREA_DEPRESSED_BORDER                 //凸起边框（类似按钮弹起效果）
#define UI_FILLED_AREA_LEFT_ROUNDED_BORDER              //左圆角边框
#define UI_FILLED_AREA_RIGHT_ROUNDED_BORDER             //右圆角边框
```

2 个阴影标志如下：

```
#define UI_FILLED_AREA_SHADOW                           //单阴影（一个像素宽）
#define UI_FILLED_AREA_SHADOW_DOUBLE_LINE               //双阴影（两个像素宽）
```

6.5.2 纯色背景

我们首先要初始化 UI_filled_area 结构体，然后分别对结构体内部的每个变量赋值。比如将上面的标志位按照前面提到的组合公式进行组合，然后赋值给 flags，将需要的背景色赋值给变量 c 等。然后，将结构体对象传递给函数 gui_draw_filled_area 绘制出来即可。

假如我们要绘制一个绿色背景色、黑色边框、带有 2 个像素宽的红色阴影的背景，并且背景上面还有一段带边框的文本（这里，我们将结合前面讲到的一些图形绘制函数来绘制）。

```
#define PIXTEL_ADJUST 4
void EntryFirstAppScreen(void)
{
  stFontAttribute newfont = {0, 0, 0, MEDIUM_FONT, 0,0};
  color text_color = {255, 0, 0, 0};
  color border_color = {0,255,0,0};
  stFontAttribute f = newfont;
  S32 width = 0;
  S32 height = 0;
  S32 x_pos = 0;
  S32 y_pos= 0;
  UI_filled_area filler = {0};
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();
  /*请注意标识位的组合，通常情况下如果你要使用双倍的阴影，那么边框也需要
使用双倍的边界*/
  filler.flags = UI_FILLED_AREA_TYPE_COLOR  |
                  UI_FILLED_AREA_DOUBLE_BORDER |
                  UI_FILLED_AREA_SHADOW |
                  UI_FILLED_AREA_SHADOW_DOUBLE_LINE;
  filler.c = UI_COLOR_GREEN;                    //绿色的背景色
  filler.border_color = UI_COLOR_BLACK;         //黑色的边框色
  filler.shadow_color = UI_COLOR_RED;           //红色的阴影颜色
  //测量文本的高度和宽度
  gui_measure_string((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK),
                     &width,&height);

  gui_set_text_color(text_color);               //设置文本颜色

  gui_set_font(&newfont);                       //使用自定义的字体
  gui_set_text_border_color(border_color);      //设定文本边框颜色

  x_pos = (UI_device_width - width) / 2;        //这两句即是进行居中对齐的核心语句
  y_pos = (UI_device_height - height) / 4;
  //绘制纯色背景
  gui_draw_filled_area(x_pos-10, y_pos-10, x_pos+100, y_pos+100, &filler);
  gui_move_text_cursor(x_pos,y_pos);            //设置绘制坐标

  //绘制带框的填充矩形
  gdi_draw_frame_rect(x_pos-PIXTEL_ADJUST, y_pos-PIXTEL_ADJUST,
       x_pos+width+PIXTEL_ADJUST, y_pos+height+PIXTEL_ADJUST,
                  gdi_act_color_from_rgb(255, 0, 255, 255), GDI_COLOR_RED, 3);
```

```
    //绘制带框的文本
    gui_print_bordered_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕

    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

在上面的代码中，你也可以自己定义颜色，如：

```
…
color backgrouond = {255, 234, 182,255};
…
filler.c = background;
…
```

保存上面代码，编译、运行模拟器，进入我们的屏幕，效果如图 6.9 所示。

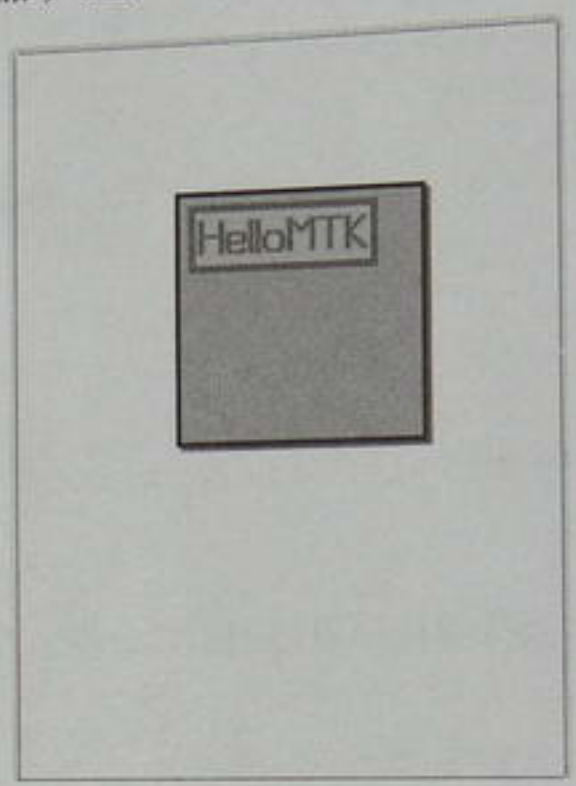


图 6.9 纯色背景

6.5.3 渐进色背景

所谓渐进色，就是在一个区域内从一种颜色过渡到另一种颜色的特效。从上面的定义中，我们就可以大致分析出需要的语句和变量。

首先，需要确定多少个颜色用于渐进，我们用 3 个颜色，红色、绿色和粉色，代码如下：

```
color g_colors[3] = {{255,0,0},{0,255,0},{255,0,255}};
```

确定每个颜色在渐进过程中占用的比例，即红色占多少比例，绿色和粉色分别占多少比例。我们的例子将会是从红色向绿色渐进，绿色向粉色渐进，所以 3 个颜色之间只会出现两次交替，我们声明一个数组来保存比例：

```
U8 perc[2] = {40,60};
```

渐进色和纯色背景一样，也有一个结构体来进行相应的封装。

```
typedef struct _gradient_color
{
    color *c;     /*总共多少种颜色进行渐进*/
    U8 *p;        /* 每个颜色占用的比例*/
    U8 n;         /* 总共的颜色 */
} gradient_color;
```

将我们定义的颜色及比例等变量传入结构体实例 gc。

```
gradient_color gc = { g_colors, perc, 3 };
```

现在，对代码进行修改，如下所示：

```
void EntryFirstAppScreen(void)
{
 stFontAttribute newfont = {0, 0, 0, MEDIUM_FONT, 0,0};
 color text_color = {255, 0, 0, 0};
 color border_color = {0,255,0,0};

 color g_colors[3] = {{255,0,0},{0,255,0},{255,0,255}};
 U8 perc[2] = {40,60};
 gradient_color gc = { g_colors, perc, 3 };

 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();
 /*将之前的纯色背景标识替换为UI_FILLED_AREA_TYPE_GRADIENT_COLOR,
 其他的标识不用更改*/
 filler.flags = UI_FILLED_AREA_TYPE_GRADIENT_COLOR |
                UI_FILLED_AREA_DOUBLE_BORDER |
                UI_FILLED_AREA_SHADOW |
                UI_FILLED_AREA_SHADOW_DOUBLE_LINE;
 //将结构体对象地址传入filler
 filler.gc = &gc;
 //后面代码不变
 …
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

保存代码，编译并运行模拟器，效果如图 6.10 所示。

系统定义了 3 种渐进效果的标志：

```
UI_FILLED_AREA_HORIZONTAL_FILL      //水平方式渐进显示，此为默认方式
UI_FILLED_AREA_VERTICAL_FILL        //垂直方式渐进显示，从上到下递进显示
UI_FILLED_AREA_FLIP_FILL            //反转渐进色显示
```

水平方式的渐进是默认方式，不用显式地声明，效果如图 6.1 所示。

将渐进标志替换为垂直显示：

```
filler.flags = UI_FILLED_AREA_TYPE_GRADIENT_COLOR
               | UI_FILLED_AREA_VERTICAL_FILL |
               UI_FILLED_AREA_DOUBLE_BORDER |
               UI_FILLED_AREA_SHADOW |
               UI_FILLED_AREA_SHADOW_DOUBLE_LINE;
```

效果如图 6.11 所示。

将渐进标识替换为反转显示：

```
filler.flags = UI_FILLED_AREA_TYPE_GRADIENT_COLOR
               | UI_FILLED_AREA_FLIP_FILL|
               UI_FILLED_AREA_DOUBLE_BORDER |
               UI_FILLED_AREA_SHADOW |
               UI_FILLED_AREA_SHADOW_DOUBLE_LINE;
```

效果如图 6.12 所示。

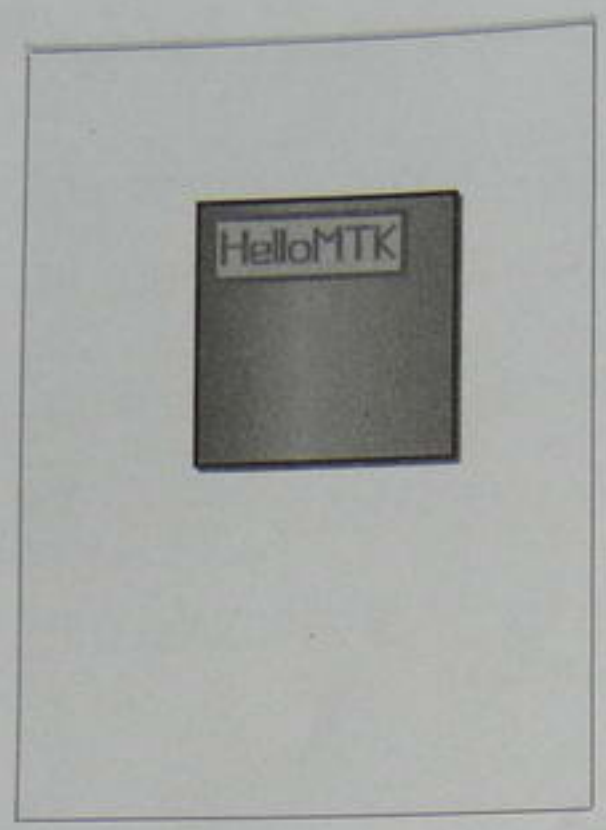


图 6.10　渐进色背景

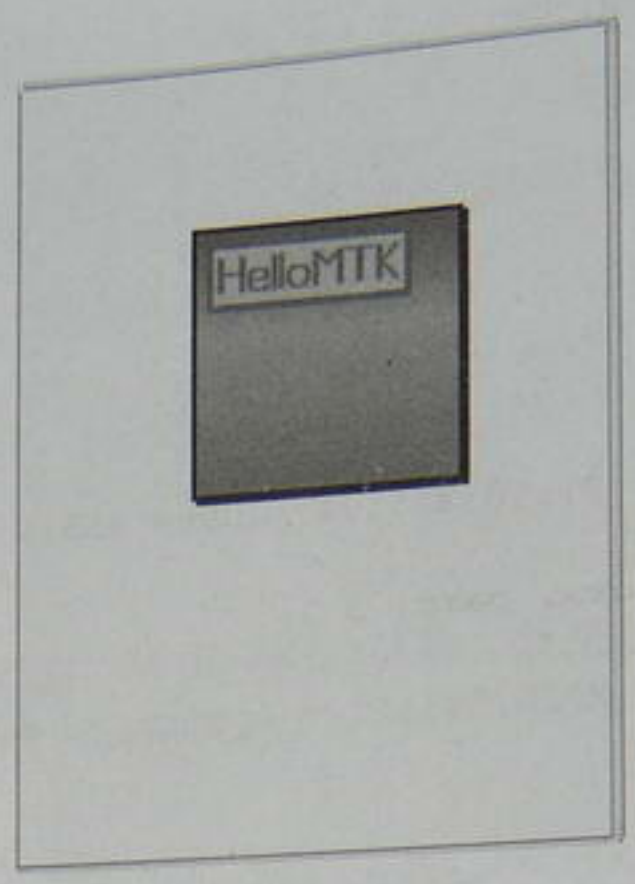


图 6.11　垂直渐进背景

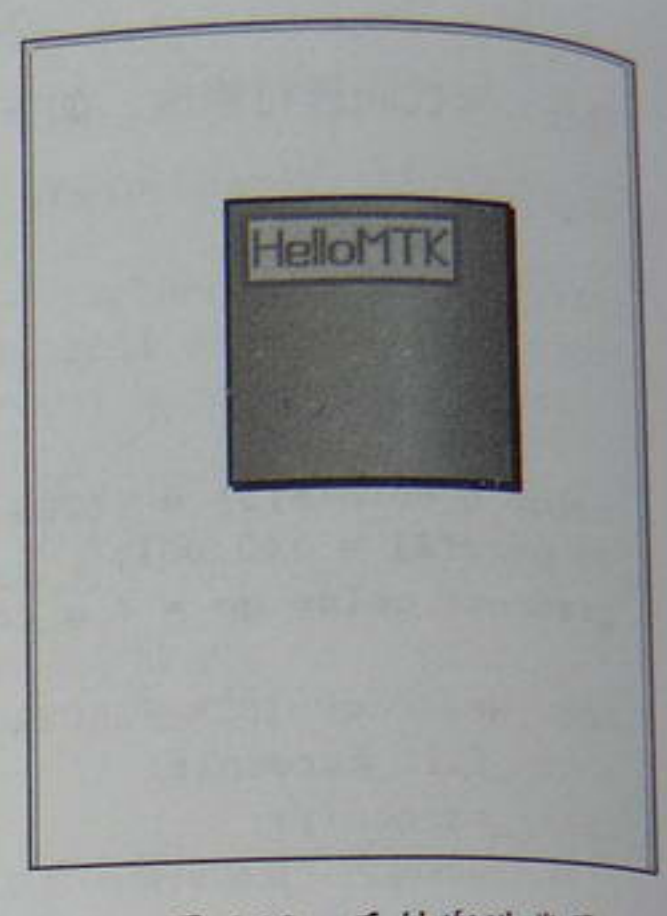


图 6.12　反转渐进背景

6.5.4　纹理背景

大家肯定知道木纹，木头的纹理其实都是一些不断重复的不规则的条纹。在图形系统中，我们也可以将一些图片进行重复绘制，从而形成纹理背景。

首先，我们要确定绘制纹理的图片文件，我们使用如图 6.13 所示图片，并定义 ID：

```
IMG_FIRSTAPP_TEXTURE。
```

图 6.13　测试图片

> *说　明*
>
> 新增的 ID 一定要附加在先前已有的 ID 之后，否则有时候会出现一些莫名的异常。

然后将标识位由先前的渐进色替换为纹理。

```
filler.flags = UI_FILLED_AREA_TYPE_TEXTURE |
               UI_FILLED_AREA_DOUBLE_BORDER |
               UI_FILLED_AREA_SHADOW |
               UI_FILLED_AREA_SHADOW_DOUBLE_LINE;
```

现在，将图片的地址赋值给 filler 结构体实例的变量 b：

```
filler.b = GetImage(IMG_ID_PHOART_ADJUST_BAR);
```

为了突出效果，我们将背景的显示区域增大：

```
gui_draw_filled_area(x_pos-10, y_pos-10, x_pos+100, y_pos+200, &filler); //绘制
```

最后，将上面新增的语句加入我们之前的代码，保存并编译、运行模拟器，效果如图 6.14 所示。

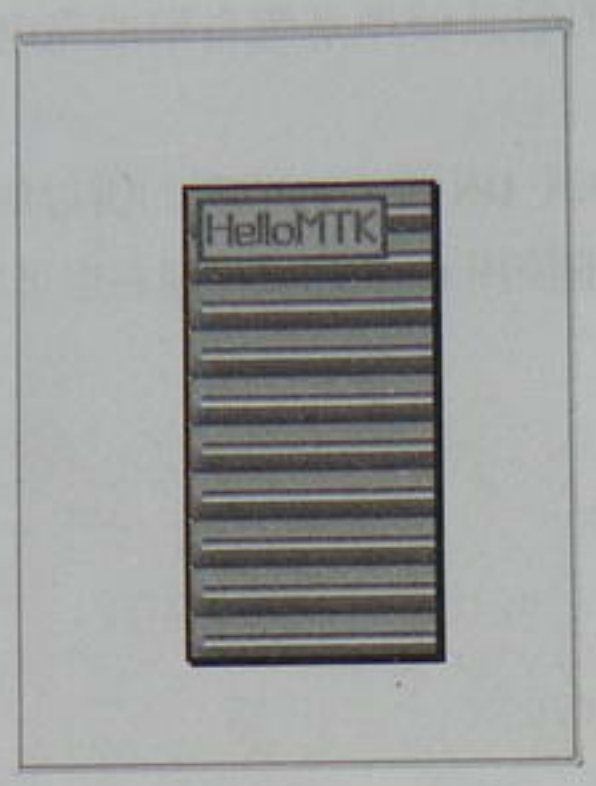


图 6.14 纹理背景

6.5.5 图片背景

图片背景的绘制其实与纹理背景差不多。首先还是要将图片背景的标识加入 flags。

```
filler.flags = UI_FILLED_AREA_TYPE_BITMAP |
               UI_FILLED_AREA_DOUBLE_BORDER |
               UI_FILLED_AREA_SHADOW |
               UI_FILLED_AREA_SHADOW_DOUBLE_LINE;
```

然后，传入我们的图片 ID：

```
filler.b = GetImage(IMG_FIRSTAPP_BACKGROUND);
```

将上面的语句加入先前的代码，保存并编译、运行模拟器，效果如图 6.15 所示。

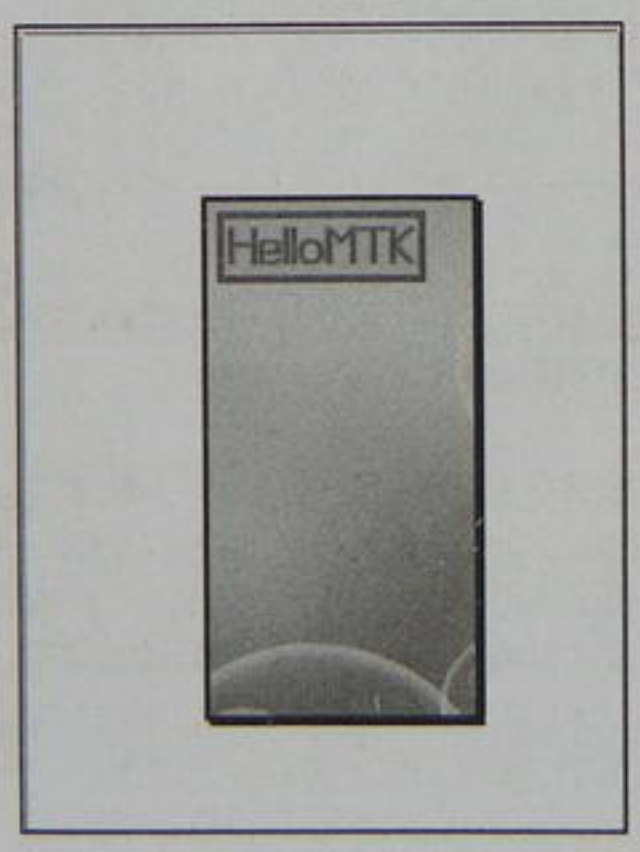


图 6.15 图片背景

6.5.6 动画背景

动画背景是界面特效的重要组成部分。几乎所有效果超炫的界面，都离不开优良的动画背景。其绘制的方法很简单，惟一需要注意的就是图层的运用（在下一章讲到）。

图 6.16 测试图片

首先，找到一张动画图片，我们暂且使用系统自带的充电动画图片，如下图 6.16 所示。

将动画 ID IMG_CHARGER_POWER_ON 传入之前我们讲过的动画绘制函数 gdi_anim_draw_id，绘制即可。先前使用 flags 标识位现在也可以不再使用。

示例代码如下：

```
#define PIXTEL_ADJUST 4
void EntryFirstAppScreen(void)
{
 stFontAttribute newfont = {0, 0, 0, MEDIUM_FONT, 0,0};
 color text_color = {255, 0, 0, 0};
 color border_color = {0,255,0,0};

 stFontAttribute f = newfont;
 S32 width = 0;
 S32 height = 0;
 S32 x_pos = 0;
 S32 y_pos= 0;
 UI_filled_area filler = {0};
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();
 gui_measure_string((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK),
                    &width,&height);

 gui_set_text_color(text_color);                    //设置文本颜色

 gui_set_font(&newfont);                            //使用自定义的字体
 gui_set_text_border_color(border_color);           //设定文本边框颜色

 x_pos = (UI_device_width - width) / 2;
 y_pos = (UI_device_height - height) / 4;

 gui_move_text_cursor(x_pos,y_pos);                 //设置绘制坐标
  // 绘制动画背景
 gdi_anim_draw_id(x_pos-PIXTEL_ADJUST, y_pos-PIXTEL_ADJUST,
                  IMG_CHARGER_POWER_ON,&my_anim);
 //绘制带框的填充文本
 gdi_draw_frame_rect(x_pos-PIXTEL_ADJUST, y_pos-PIXTEL_ADJUST,
       x_pos+width+PIXTEL_ADJUST, y_pos+height+PIXTEL_ADJUST,
                    gdi_act_color_from_rgb(255, 0, 255, 255), GDI_COLOR_RED, 3);
 gui_print_bordered_text((UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

保存代码，编译、运行模拟器，效果如图 6.17 所示。

现在可以看到我们的文本后面有一个不断运动的背景，很醒目。但是，如果稍等片刻，就会发现一个严重问题：文本“HelloMTK”连同外面的边框消失了，如图 6.18 所示。

这是由于背景动画是由一帧一帧的图片组成的，每一次切换帧都会重新刷新整个屏幕，而字符串由于一直恒定不变，所以，它就被新绘制的动画帧给覆盖了。要解决这个问题，就必须使用多图层，

我们将在下一章重点讲解。

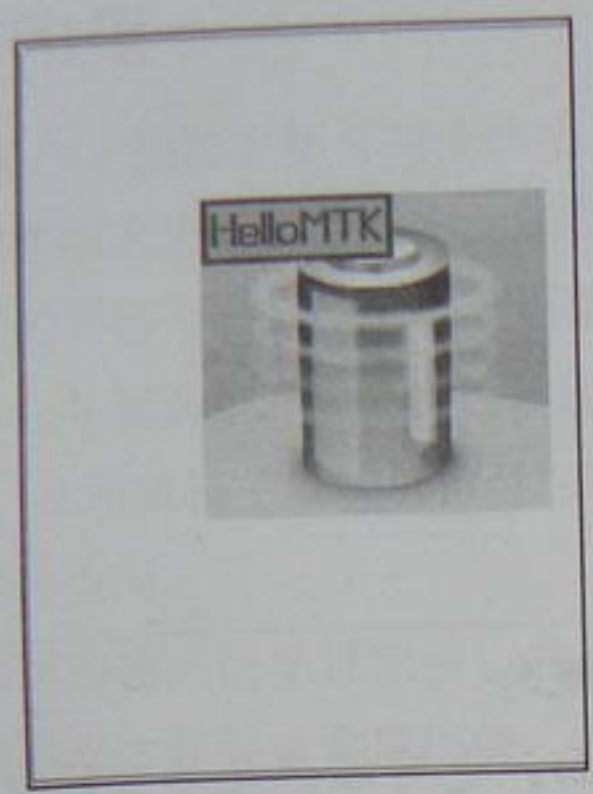


图 6.17　动画背景效果

图 6.18　文字消失

6.5.7　系统默认背景

除了我们自定义的 UI_filled_area 结构体外，系统还预定义了大量的用于主题的 UI_filled_area 结构体对象，在以后编辑主题的过程中，可以直接调用这些对象，如图 6.19 所示。

```
/* System UI fillers */

extern CONST_HARDWARE UI_filled_area MMI_down_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_up_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_disabled_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_focussed_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_clicked_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_multitap_filler;
extern CONST_HARDWARE UI_filled_area MMI_inputbox_normal_filler;
extern CONST_HARDWARE UI_filled_area MMI_inputbox_selected_filler;
extern CONST_HARDWARE UI_filled_area MMI_inputbox_disabled_filler;
extern CONST_HARDWARE UI_filled_area MMI_scrollcontrol_up_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_scrollcontrol_down_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_scrollcontrol_disabled_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_scrollcontrol_focussed_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_scroll_up_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_scroll_down_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_scroll_disabled_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_scroll_focussed_button_filler;
extern CONST_HARDWARE UI_filled_area MMI_scrollbar_normal_filler;
extern CONST_HARDWARE UI_filled_area MMI_scrollbar_disabled_filler;
extern CONST_HARDWARE UI_filled_area MMI_scrollbar_focussed_filler;
```

图 6.19　预定义的填充背景

从上面的代码中，我们看到，一般界面元素都已经有了默认的背景。比如按钮按下状态时的 MMI_down_button_filler，关闭输入框的选中状态 MMI_inputbox_selected_filler 等。

我们在开发的时候，多留意系统自带的背景，合理和高效地使用它们。

结语：

通过本章的学习，我们了解了 MTK 图形系统中重要的组成部分，学习了基本的绘制函数，了解了背景、图形、图像等重要知识点，可以作为开始 UI 设计的启蒙。读者学完本章应该对我们的应用开发所处的位置有一个清晰的认识，明白我们的开发过程和图形系统之间的联系。

7 图层

任何具有可视化界面的平台，都离不开图层这个概念。

通过图层的组合，可以绘制出细腻的用户操作界面，也可以产生华丽的特效。

使用图层，可以提高界面开发的效率。比如开发人员可以通过若干个图层来绘制多个元素，这些元素分别位于各个图层上，当每个图层都绘制完毕后，将图层重合在一起，就像是在一张图上完成的一样。也许有读者会问：为什么不把元素直接放在一个图层上呢？你应该考虑到这种情况，如果你需要绘制 5 个元素，当你绘制了 4 个元素时，突然发现之前有一个元素有问题，你必须重画。而带来的负面影响是：当前的 4 个元素都必须重画。使用图层的好处就是可以编辑任意一个图层的内容而不会影响到其他图层，进而提高开发效率。

通过图层，我们还可以弥补重绘界面时屏幕闪烁的缺陷。因为屏幕界面在响应重绘消息的时候要进行复杂的图形处理，所以屏幕在重绘时会由于过频的刷新而引起闪烁现象，解决这一问题的有效方法就是双缓冲技术。因为窗体在刷新时，总要有一个擦除原来图像的过程，它利用背景色填充屏幕绘图区，然后再调用新的绘图代码进行重绘，这样一擦一写造成了图像颜色的反差。当重绘消息的响应很频繁的时候，这种反差也就越发明显，于是我们就看到了闪烁现象。

我们会很自然地想到，避免背景色的填充是最直接的办法。但是这样做的话，窗体上会变得一团糟，因为每次绘制图像时都没有将原来的图像清除，导致图像残留，于是屏幕重绘时，画面往往会变得很杂乱。所以单纯地禁止背景重绘是不够的，我们还要进行重新绘图，但要求速度快，这一点该如何解决？

我们可以先在内存中做图，然后用函数将做好的图复制到前台，同时禁止背景刷新，这样就消除了闪烁。以上也就是双缓冲绘图的基本思路。

当启用双缓冲时，所有绘制操作首先呈现到内存缓冲区，而不是屏幕上的绘图图面。所有绘制操作完成后，内存缓冲区直接将其复制到与其关联的绘图图面上。因为在屏幕上只执行一个图形操作，所以消除了由复杂绘制操作导致的图像闪烁。

7.1 主要作用

在 MTK 系统中，图层主要有两个作用。

缓冲：在某些频繁更新的界面中，如果某些显示元素一直不变化，我们就可以将这些元素提取出来画到一个模拟的屏幕中，当界面需要更新时，只需将要刷新的元素更新到另外一个模拟屏幕，而后将两个模拟屏幕合并到真正屏幕上即可。这样做就省掉了不变元素的重画时间（某些元素重画很耗时，如图像显示等），从而减轻了系统负担及加速了画面更新。我们把这样的模拟屏幕就叫做层，也可以说层就是屏幕的缓冲空间。

特效：因为层的格式简单且统一，并且一般的图形系统中都会用硬件来加速层合并，所以在层合并时加上些特效会很方便并且实现极快。系统中的特效有通透、半透明、剪切等。

在第 6 章的 6.5.6 小节中，我们讲到了动画作背景的缺陷，即动画背景重绘的时候，遮盖了背景前的文字。

有了层以后，我们就可以将不变的文本及图像放到新建的一个层中，将动画放到背景层中，每当有动画换帧时，只需将新的帧画到背景层中，然后合并两个层到屏幕（动画刷新时自动会合并），这样做上层的文本等就不会被动画盖住了。

7.2 创建新层

创建层需要先建一个层句柄（句柄在第 6 章的 6.3 节中已有讲解），我们都是通过层句柄来控制层的。

为了方便，系统还预定义了主屏幕和副屏幕的层句柄。

主屏幕的层句柄：

```
GDI_LAYER_MAIN_BASE_LAYER_HANDLE
```

副屏幕（如果支持的话）的层句柄：

```
GDI_LAYER_SUB_BASE_LAYER_HANDLE
```

函数 gdi_layer_create 用来创建一个新层。

其函数原型是对更下层函数 gdi_layer_create_cf 的一个宏定义：

```
#define gdi_layer_create(OFFSET_X,OFFSET_Y,WIDTH,HEIGHT,HANDLE_PTR)\
gdi_layer_create_cf(GDI_LCD->cf,OFFSET_X,OFFSET_Y,WIDTH,HEIGHT,HANDLE_PTR)
```

OFFSET_X：新层的起始 *x* 坐标。

OFFSET_Y：新层的起始 *y* 坐标。

WIDTH：　新层的宽度。

HEIGHT：　新层的高度。

HANDLE_PTR：新层的句柄地址。

其前 4 个参数指出层的位置及大小（位置是以实际屏幕左上角为坐标原点的）。

同时，需要注意的是，在创建新图层之前，应该调用函数 gdi_layer_multi_layer_enable()开启多层支

持。下面代码中将在坐标（10,10）处创建一个宽 200，高 200 的层。

```
gdi_handle  new_layer;
void EntryFirstAppScreen(void)
{
  gdi_layer_multi_layer_disable();
  //如果层创建成功，就将层句柄赋值给 new_layer
  gdi_layer_create(10, 10, 200, 200, & new_layer);
…
}
```

在退出屏幕时，还需要在退出函数中调用 gdi_layer_multi_layer_disable()关闭多层。

```
void ExitFirstAppScreen()
{
  …
  gdi_layer_multi_layer_disable();
}
```

7.3 激活新层

打个比方，现在刚买了一台电冰箱，但是不能马上就使用它，你首先要做的是打开电源，激活冰箱。只有完成了这一步，你才能使用冰箱。

层的使用也一样，在对其进行操作之前，也必须先激活它。在我们的图形系统中，任何时刻有且只能有一个层处于激活状态，所有的绘画函数都是默认画到已经被激活的层中。

激活层的函数：void gdi_layer_set_active(gdi_handle handle)。

从函数的形参我们就可以看到，需要传入待激活的层的句柄。

下面代码的作用是激活刚才创建的新层：

```
gdi_handle  new_layer;
void EntryFirstAppScreen(void)
{
  …
  gdi_layer_create(10, 10, 200, 200, & new_layer);
  gdi_layer_set_active(new_layer);
  …
}
```

系统还提供了另外一组激活层的函数：

```
gdi_layer_push_and_set_active(gdi_handle handle)
gdi_layer_push_and_set_active()
```

这对函数必须成对使用，否则会发生异常。从函数名字上就可以看出，这是一个压栈出栈的过程。

· gdi_layer_push_and_set_active(gdi_handle handle)函数的应用。

内部代码片段如图 7.1 所示。

可以看到，系统会将当前的层压入激活层栈，gdi_layer_stack 的值增加 1 个单位（图 7.2），并马上对传入的层进行激活。

```
void gdi_layer_push_and_set_active(gdi_handle handle)
{
  /*----------------------------------------------------------------*/
  /* Local Variables                                                */
  /*----------------------------------------------------------------*/

  /*----------------------------------------------------------------*/
  /* Code Body                                                      */
  /*----------------------------------------------------------------*/
  GDI_LOCK;
  /* no more stack entry available */
  GDI_ASSERT(gdi_layer_stack_index < GDI_LAYER_ACT_LAYER_STACK_COUNT - 1);
  gdi_layer_stack[gdi_layer_stack_index++] = (gdi_handle) gdi_act_layer;
  gdi_layer_set_active(handle);
}
```

图 7.1　函数内部实现（1）

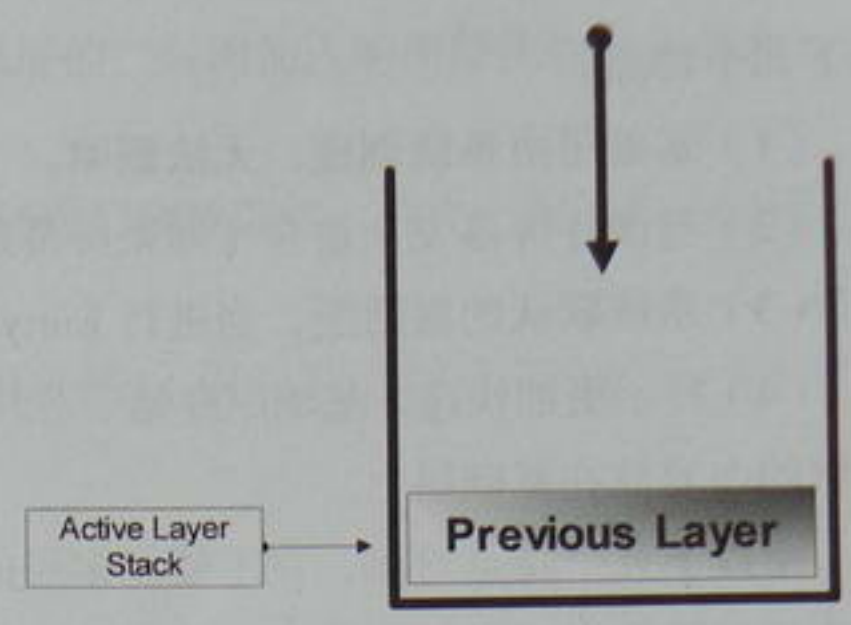


图 7.2　层入栈

· gdi_layer_pop_and_restore_active()函数的应用。

内部代码片段如图 7.3 所示。

当我们新建的层显示完毕，就需要将之前的层恢复显示。这时，函数内部调用会指向激活栈的栈顶，取出栈顶值，如图 7.4 所示。

```
void gdi_layer_pop_and_restore_active(void)
{
  /*----------------------------------------------------------------*/
  /* Local Variables                                                */
  /*----------------------------------------------------------------*/

  /*----------------------------------------------------------------*/
  /* Code Body                                                      */
  /*----------------------------------------------------------------*/
  GDI_ASSERT(gdi_layer_stack_index > 0);
  gdi_layer_set_active(gdi_layer_stack[--gdi_layer_stack_index]);

  GDI_UNLOCK;
}
```

图 7.3　函数内部实现（2）

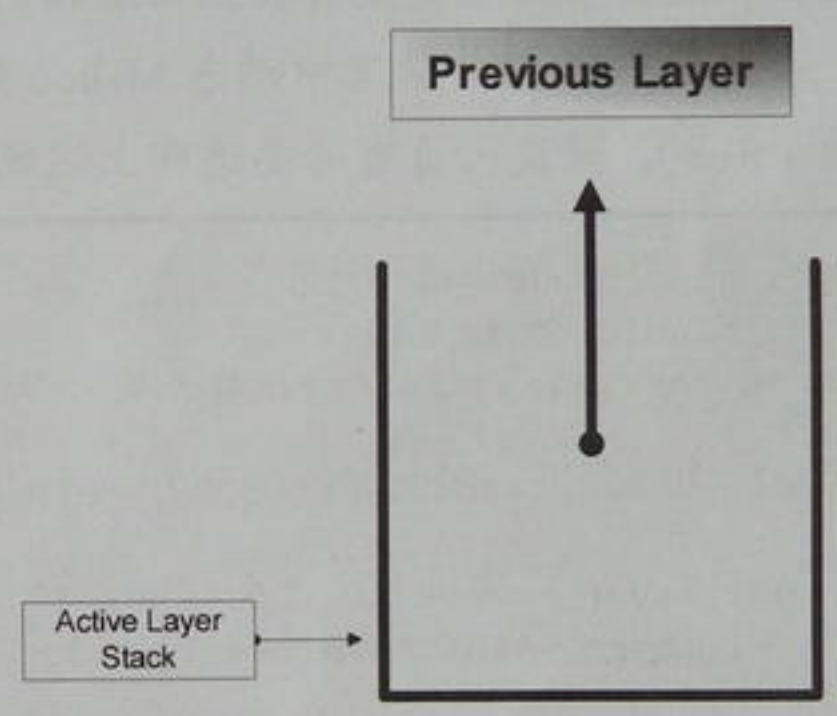


图 7.4　层出栈

这个栈顶值就是之前压入激活栈的层的句柄，接着，把取出的句柄传入函数 gdi_layer_set_active 中。这样，之前的层也顺利地恢复。

代码修改如下：

```
gdi_handle  new_layer;
void EntryFirstAppScreen(void)
{
  …
  gdi_layer_create(10, 10, 200, 200, & new_layer);
  gdi_layer_push_and_set_active(new_layer);
  …
  gdi_layer_pop_and_restore_active();
  …
}
```

7.4　基础层

正是因为层在图形系统中的特性，系统开机的时候会为每个硬件屏幕创建一个基础层，基础层有

以下几个特点。

（1）基础层由系统创建，无法删除。

（2）与硬件屏幕完全重合（如果屏幕是 240 × 320，那么基础层也将是这个尺寸）。

（3）系统默认的激活层，当进行 EntryNewScreen 操作时系统会自动将基础层激活。

（4）显示更加快速，基础层存储于芯片内的 Flash 中，所以在其上面绘画极快，一般我们会将刷新频繁的内容放在基础层上。

由以上几个特点看出，在不使用多层的情况下，我们可以将基础层当做硬件屏幕来看待，也就是说普通程序完全可以忽略层的概念。前面的第 4 章中，我们都是在基础层上进行绘制的。

有了多图层的特性后，我们可以在基础层上绘制背景图（在激活新层之前我们的内容都默认画到基础层中），在新建层上绘制文字、图形及图像。

提 示

系统一般在进入 EntryNewScreen 函数时会自动将基础层激活，为避免特殊情况下使用层混乱，通常在激活新层并在上面绘画完毕后，我们会主动将基础层还原为激活状态。

这就好比在 C 语言中动态 Malloc 分配内存后，在不使用的时候应该将先前申请的内存释放（free）。开发人员有必要遵守上述规则。

```
gdi_handle  new_layer;
gdi_handle  base_layer;
void EntryFirstAppScreen(void)
{
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);

  gdi_layer_create(10, 10, 200, 200, & new_layer);
  gdi_layer_set_active(new_layer);

  gdi_layer_get_base_handle(&base_layer);
  gdi_layer_set_active(base_layer);

  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

7.5 合并

我们现在来运行上面的代码，进入模拟器中我们新建的屏幕，发现似乎没什么变化，新增的层没有体现出来。这是由于新建的层都是相互独立的，必须使用一个函数来将所有需要绘制的图层进行合并，然后再统一调用屏幕绘制函数。

MTK 图形系统支持 4 个图层。

函数 gui_BLT_double_buffer 用来合并层并绘制，但在使用之前先要用 gdi_layer_set_blt_layer 函数指明是哪几个层需要合并。

因此函数 gdi_layer_set_blt_layer 能接受 4 个层句柄，也就是说系统同一时刻最多只能合并 4 个层。

需要注意的是参数的顺序：第 1 个传入的层是放在最底层的，后面加入的图层依次向上重叠，如图 7.5 所示。

下面是一个合并图层的代码，为了直观地展示，我们将新图层刷成蓝色，并在上面绘制一幅图片，并同时将基础层刷成白色。

```
gdi_handle new_layer;
gdi_handle base_layer;
void EntryFirstAppScreen(void)
{
 gdi_color     blue_color;
 gdi_color     white_color;
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 blue_color = GDI_RGB_TO_BUFFER_FORMAT(0,0,255);          //获取蓝色
 white_color = GDI_RGB_TO_BUFFER_FORMAT(255,255,255);
 entry_full_screen();
 clear_screen();
 gdi_layer_create(10, 10, 200, 200, &new_layer);
 gdi_layer_set_active(new_layer);
 gdi_layer_clear(blue_color);                             //将新建层刷成蓝色
 gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);           //绘制一幅图片

 gdi_layer_get_base_handle(&base_layer);
 gdi_layer_set_active(base_layer);
 gdi_layer_clear(white_color);                            //将基础层刷成白色

 gdi_layer_set_blt_layer(base_layer, new_layer, NULL, NULL);
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

编译并运行模拟器，效果如图 7.6 所示。

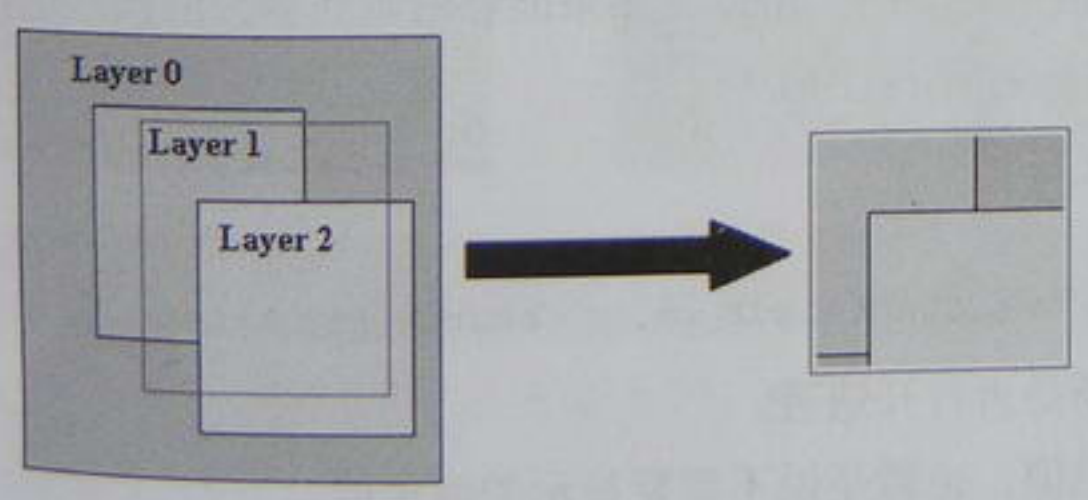


图 7.5　合并图层示意图

图 7.6　合并后的效果

可以看到，新建图层以及图层上的元素被正确地绘制出来。

也可以使用函数 gdi_layer_blt 来合并图层。gdi_layer_blt 函数接受需要合并的图层句柄，并同时设置合并显示的图层区域。

```
void EntryFirstAppScreen(void)
{
 …
 gdi_layer_create(10, 10, 200, 200, &new_layer);
 gdi_layer_set_active(new_layer);
 gdi_layer_clear(blue_color);//将新建层刷成蓝色
 gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);//绘制一幅图片

 gdi_layer_get_base_handle(&base_layer);
 gdi_layer_set_active(base_layer);
 gdi_layer_clear(white_color);//将基础层刷成白色
 //使用该合并函数，就可以省去之前的重绘屏幕的函数 gui_BLT_double_buffer
 gdi_layer_blt(base_layer, new_layer,
               NULL,NULL,0, 0,
               UI_device_width-1,UI_device_height-1);
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

7.6 通透

在讲解通透之前，我们先来完成一个工作：在基础层上先绘制一幅与层尺寸一样大小的图片 IMG_FIRST_BACKGROUND，如图 7.7 所示。

```
 …
 gdi_layer_get_base_handle(&base_layer);
 gdi_layer_set_active(base_layer);
 gdi_layer_clear(white_color);
 gdi_layer_get_base_handle(&base_layer);
 gdi_layer_set_active(base_layer);
 gdi_layer_clear(white_color);
 //绘制背景图片 IMG_FIRSTAPP_BACKGROUND
 gdi_image_draw_id(0,0,IMG_FIRSTAPP_BACKGROUND);
 …
```

编译并运行模拟器，可以看到如图 7.8 所示画面。

很明显，新建层的背景色（模拟器上为蓝色）遮盖了基础层的背景图片，显得很不美观。因为我们是将新层和基础层合并在一起，然后重绘整个屏幕的。所以，如果能够做到在合并的同时让背景色在基础层上消失即可解决遮盖的问题，这也是通透所做的工作。

在 MTK 图形系统中实现通透功能的函数如下：

```
GDI_RESULT
gdi_layer_set_source_key(BOOL  source_key_enable,gdi_color  source_key_value)
```

参数 source_key_enable：布尔变量，判断是否打开通透。

参数 source_key_value： 需要通透的颜色值，也就是说不需要显示的颜色值。

之前的界面是新建层的颜色遮盖了基础层，所以，我们利用上述函数将颜色通透掉。

```
gdi_handle new_layer;
gdi_handle base_layer;
void EntryFirstAppScreen(void)
{
```

```
    gdi_color      blue_color;
    gdi_color      white_color;
    EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
    blue_color = GDI_RGB_TO_BUFFER_FORMAT(0,0,255);
    white_color = GDI_RGB_TO_BUFFER_FORMAT(255,255,255);
    entry_full_screen();
    clear_screen();
    gdi_layer_create(10, 10, 200, 200, &new_layer);
    gdi_layer_set_active(new_layer);
    gdi_layer_clear(blue_color);
    gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);
    //将蓝色blue_color设置为通透色
    gdi_layer_set_source_key(TRUE, blue_color);

    gdi_layer_get_base_handle(&base_layer);
    gdi_layer_set_active(base_layer);
    gdi_layer_clear(white_color);
    gdi_image_draw_id(0,0,IMG_FIRSTAPP_BACKGROUND);

    gdi_layer_set_blt_layer(base_layer, new_layer, NULL, NULL);
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

按照上述代码修改后，编译、运行模拟器，进入我们的屏幕，即可看到原来带颜色的背景已经不再显示，两幅图片很自然地融合在一起，如图 7.9 所示。

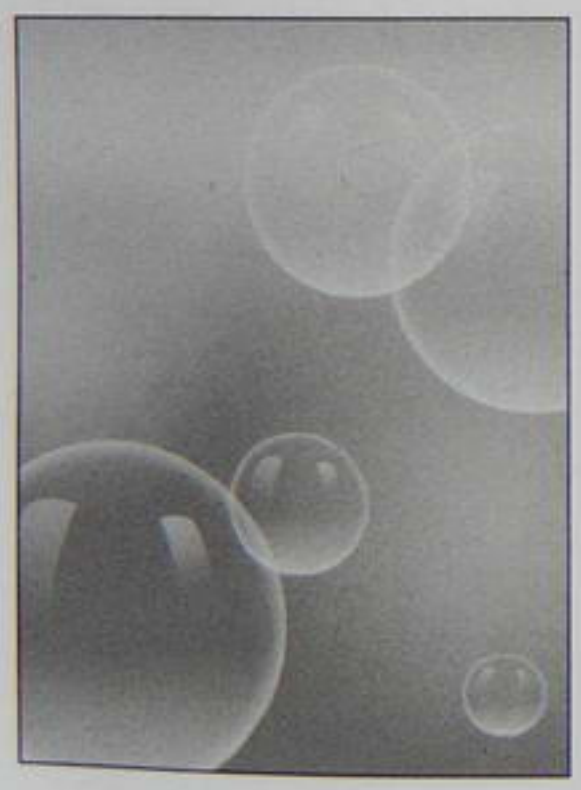


图 7.7　测试图片

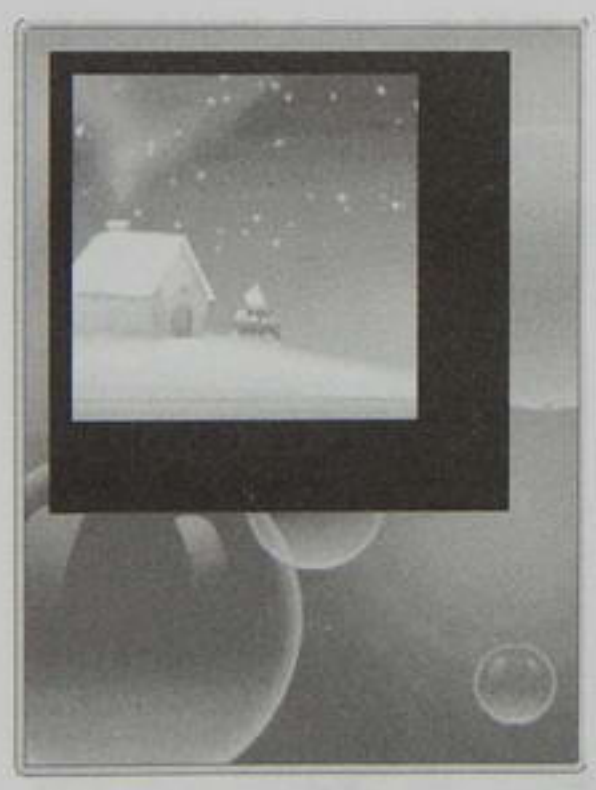


图 7.8　目前的通透效果

图 7.9　最终的通透效果图

7.7　图层剪切

在上一章中，我们对字符串及图片进行了显示区域的剪切，同样，在图层显示中，也可以使用对应的图层剪切函数来实现类似的功能。

在这里仍然要声明一点，所谓剪切，并不是真正的将图层截掉一部分，而是在层中设置一个限制区域，只有在这个区域中绘画才是有效的，否则就会被自动忽略。

剪切特效有两个特点。

（1）每个层有且只有一个剪切区域。

（2）剪切区域一经设置，永久生效，所以剪切区域用完后需要调用 gdi_layer_reset_clip 函数还原（否则以后的绘制工作都不能进行）。

函数 void gdi_layer_set_clip(S32 x1, S32 y1, S32 x2, S32 y2)中的参数即为裁剪区域的左上角和右下角坐标。

现在，如果只需要显示以新图层原点为起始点的 50 × 50 的区域，则对应代码如下：

```
void EntryFirstAppScreen(void)
{
  …
  clear_screen();
  gdi_layer_create(10, 10, 200, 200, &new_layer);
  gdi_layer_set_active(new_layer);
  gdi_layer_clear(blue_color);
  //设置图层的剪切区域，50 × 50 大小
  gdi_layer_set_clip(10, 10, 60, 60);

  gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);
  gdi_layer_set_source_key(TRUE, blue_color);

  …
  gdi_layer_reset_clip();
}
```

按照上述代码修改后，编译、运行模拟器，进入我们的屏幕，即可看到新建图层只显示了一小块 50 × 50 的区域，如图 7.10 所示。

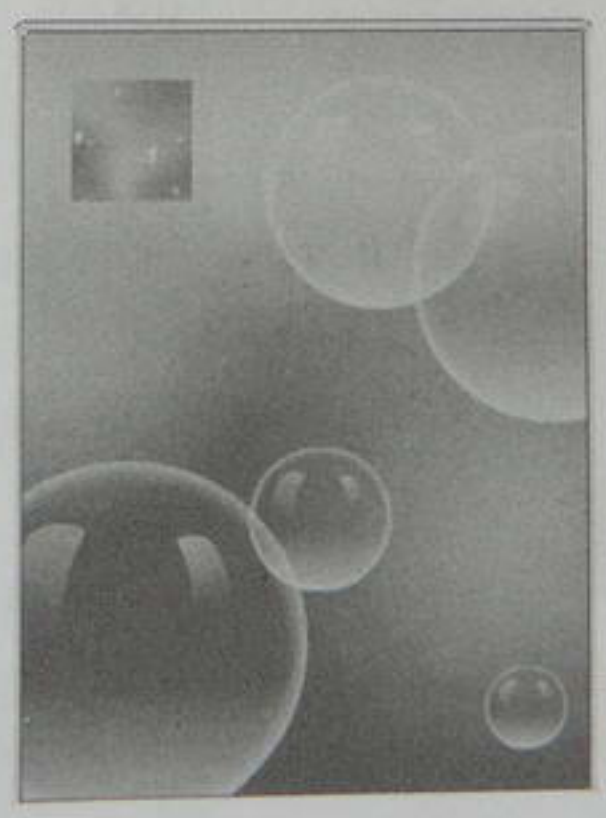


图 7.10 “裁剪”后的效果

7.8 透明效果

透明，也是上层 UI 界面开发中一个重要功能。在 MTK 图形系统中，提供响应的 API 来实现透明功能，函数如下：

```
GDI_RESULT gdi_layer_set_opacity(BOOL opacity_enable, U8 opacity_value)
```

参数 opacity_enable：布尔值，设置当前是否启用透明效果。

参数 opacity_value：透明度的具体取值，范围从 0 至 255，值越小表示透明度越高，即取值为 0 时完全透明，取值为 255 时不透明。

现在，让我们实现新建层的半透明效果。

```
void EntryFirstAppScreen(void)
{
  …
  gdi_layer_create(10, 10, 200, 200, &new_layer);
  gdi_layer_set_active(new_layer);
  gdi_layer_clear(blue_color);
  gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);

  gdi_layer_set_source_key(TRUE, blue_color);
  //设置透明度为 100，几乎半透明的效果
  gdi_layer_set_opacity(TRUE,100);

  gdi_layer_get_base_handle(&base_layer);
  gdi_layer_set_active(base_layer);
  …
}
```

按照上述代码修改后，编译、运行模拟器。进入我们的屏幕，即可看到新建图层中的图片已经处于半透明状态（基础层中气泡元素若隐若现），如图 7.11 所示。

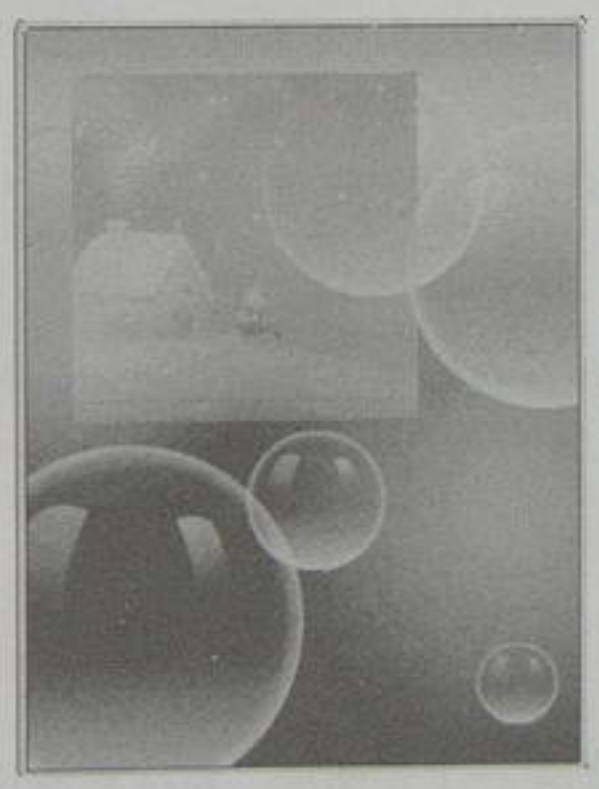


图 7.11　透明效果

7.9　释放图层

我们使用系统资源创建了新层 new_layer，当使用完毕后，应该主动将其释放，否则，会在后期出现很多不可预知的错误。读者可以做一个测试：在当前屏幕上单击右软键返回上一层界面，会报出一个严重的 ASSERT 异常，如图 7.12 所示。

我们通过其提示的文件信息，找到 gdi_layer.c 文件中的第 354 行，如图 7.13 所示。

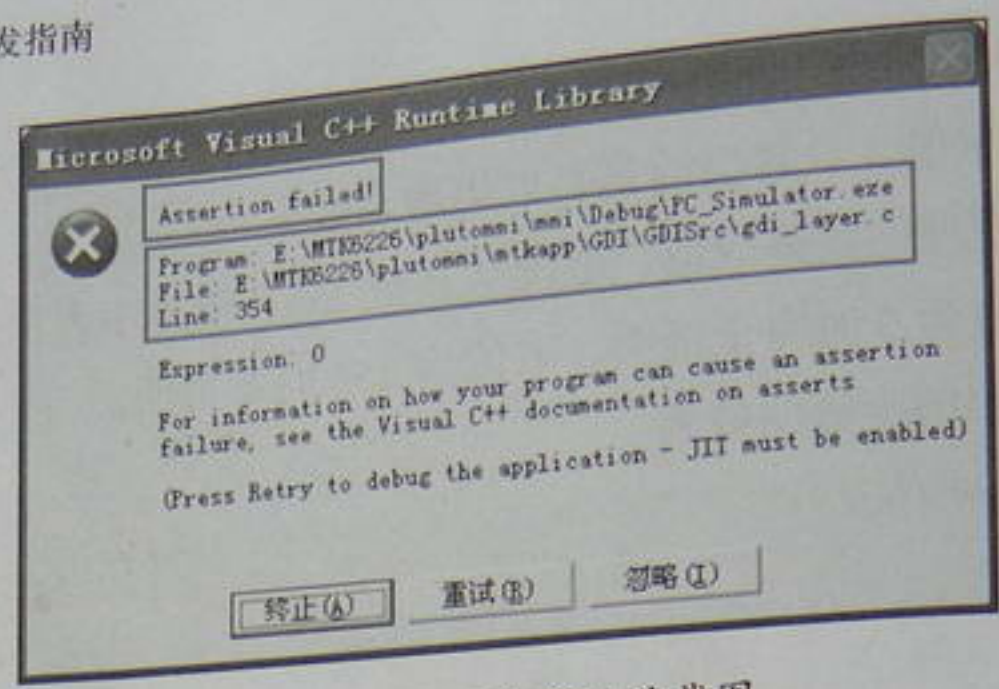


图 7.12　出现异常时的截图

```
00351:    if (next_available_ptr + size - 1 > gdi_layer_buf_pool_end_ptr)
00352:    {
00353:        /* no more buffer */
00354:        GDI_ASSERT(0);
00355:        return (NULL);
00356:    }
```

图 7.13　相关代码

由于在完成新建层操作后，没有对层进行释放，导致上层界面无法使用资源创建图层，所以报出了 ASSERT 异常，提示开发人员当前没有足够的 buffer 创建图层。因此，有必要养成良好的编程习惯，使用完毕后，及时释放图层。

我们将释放语句写在屏幕退出函数 ExitFirstAppScreen 中，如下面代码所示：

```
void ExitFirstAppScreen()
{
    ...
  gdi_layer_free(new_layer);
  gdi_layer_multi_layer_disable();
}
```

保存代码，编译并运行模拟器，现在在屏幕上单击右软键，程序再无异常，正常地退出了我们的新屏幕。

7.10　分配更多的内存

如果当前系统没有足够的资源来创建一个新图层，就应该使用系统提供的函数 gdi_layer_create_using_outside_memory 创建图层。

我们来看一下该函数的原型：

```
#define  gdi_layer_create_using_outside_memory(
X,Y,WIDTH,HEIGHT,HANDLE_PTR,OUTMEM_PTR,OUTMEM_SIZE) \
        gdi_layer_create_cf_using_outside_memory(
GDI_LCD->cf,X,Y,WIDTH,HEIGHT,HANDLE_PTR,
OUTMEM_PTR,OUTMEM_SIZE)
```

gdi_layer_create_using_outside_memory 实际上是对更下层的图层创建函数的一个宏定义。开发人员需要传入如下参数。

X，Y：　　新图层的起始坐标。

WIDTH，HEIGHT：新图层的宽度和高度。

HANDLE_PTR：新图层的句柄地址。

OUTMEM_PTR：新分配内存的地址。

OUTMEM_SIZE：内存的容量。

现在，在原有代码基础上对其稍做修改：

```
gdi_handle new_layer;
gdi_handle base_layer;
void EntryFirstAppScreen(void)
{
 gdi_color     blue_color;
 gdi_color     white_color;
  //声明一个 U8 类型指针 outside_mem，该指针将会指向动态分配的内存地址
  U8 *outside_mem = NULL;

  gdi_layer_multi_layer_enable();

  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  //动态分配内存，并将该内存地址赋值给 outside_mem
  outside_mem = (U8 *)mmi_frm_scrmem_alloc
 (((GDI_LCD_WIDTH*GDI_LCD_HEIGHT*GDI_MAINLCD_BIT_PER_PIXEL)>>3));
  blue_color = GDI_RGB_TO_BUFFER_FORMAT(0,0,255);
  white_color = GDI_RGB_TO_BUFFER_FORMAT(255,255,255);
  entry_full_screen();
  clear_screen();

   gdi_layer_create_using_outside_memory(10,  10,  150,  150,  &new_layer,  outside_mem,
((GDI_LCD_WIDTH*GDI_LCD_HEIGHT*GDI_MAINLCD_BIT_PER_PIXEL)>>3));
   gdi_layer_set_active(new_layer);
   gdi_layer_clear(blue_color);
   gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);

   gdi_layer_set_source_key(TRUE, blue_color);
   gdi_layer_set_opacity(TRUE,100);

   gdi_layer_get_base_handle(&base_layer);
   gdi_layer_set_active(base_layer);
   gdi_layer_reset_clip();
   gdi_layer_clear(white_color);
   gdi_image_draw_id(0,0,IMG_FIRSTAPP_BACKGROUND);


   gdi_layer_set_blt_layer(base_layer, new_layer, NULL, NULL);
   gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
   SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

按照上述代码修改后，编译、运行模拟器，进入我们的屏幕，一切正常。通透、半透明的特效也正常显示。但是，当我们单击右软键返回上一层界面时，前面发生过的异常又出现了。不过这个异常的内容有所不同，如图 7.14 所示。

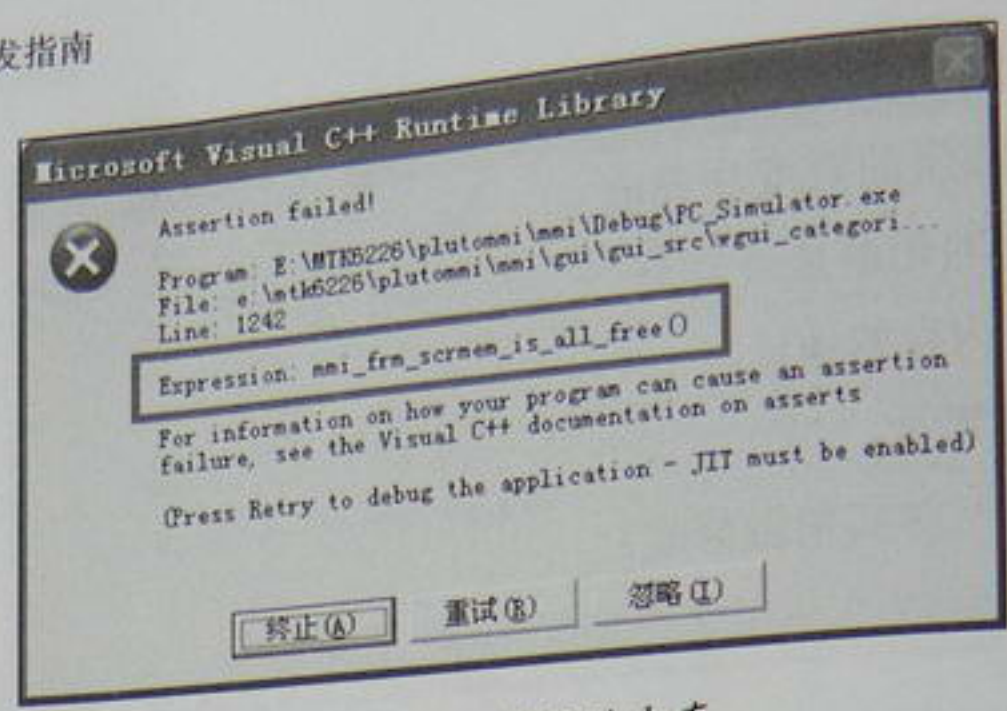


图 7.14 没有释放内存

该异常已经很清楚地告诉我们问题的所在：动态分配的内存没有释放。因此，我们在函数的末尾增加一句“mmi_frm_scrmem_free(outside_mem);”代码，用以释放先前手工分配的内存。

这和前面用 gdi_layer_free()函数释放自动分配的内存效果一样。

现在，我们的代码内容如下：

```
gdi_handle new_layer;
gdi_handle base_layer;
void EntryFirstAppScreen(void)
{
  gdi_color     blue_color;
  gdi_color     white_color;
  //声明一个 U8 类型指针 outside_mem，该指针将会指向动态分配的内存地址
  U8 *outside_mem = NULL;

  gdi_layer_multi_layer_enable();

  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  //动态分配内存，并将该内存地址赋值给 outside_mem
  outside_mem =
(U8 *)
mmi_frm_scrmem_alloc(((GDI_LCD_WIDTH*GDI_LCD_HEIGHT*GDI_MAINLCD_BIT_PER_PIXEL)>>3));

  blue_color = GDI_RGB_TO_BUFFER_FORMAT(0,0,255);
  white_color = GDI_RGB_TO_BUFFER_FORMAT(255,255,255);
  entry_full_screen();
  clear_screen();

  gdi_layer_create_using_outside_memory(10, 10, 150, 150, &new_layer, outside_mem,
((GDI_LCD_WIDTH*GDI_LCD_HEIGHT*GDI_MAINLCD_BIT_PER_PIXEL)>>3));
  gdi_layer_set_active(new_layer);
  gdi_layer_clear(blue_color);
  gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);

  gdi_layer_set_source_key(TRUE, blue_color);
  gdi_layer_set_opacity(TRUE,100);

  gdi_layer_get_base_handle(&base_layer);
  gdi_layer_set_active(base_layer);
  gdi_layer_reset_clip();
  gdi_layer_clear(white_color);
  gdi_image_draw_id(0,0,IMG_FIRSTAPP_BACKGROUND);
```

```
    gdi_layer_set_blt_layer(base_layer, new_layer, NULL, NULL);
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
    //在图层使用完毕后，释放刚才申请的内存
    mmi_frm_scrmem_free(outside_mem);
    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

7.11 锁层

一个手机的显示屏幕，由很多不同的界面元素组成，可能有状态栏、标题栏、菜单、按钮等。每个元素都可能属于不同的图层，而且状态随时都可能改变，因此元素与元素之间也会发生图层合并事件。

一般情况下，只要相关图层有改变都应该调用 gdi_layer_blt 等相关函数进行图层合并。但是，从用户使用角度来讲，有的图层合并在当前屏幕中并不是必需的，来看下面这个例子。

假如屏幕上有一个状态栏，一个菜单和一个按钮。

那么按照前面讲的流程，屏幕绘制过程可能如下。

（1）进入屏幕绘制函数，做好绘图准备。

（2）启动状态栏绘制。

- 在图层上绘制各种状态图标，绘制运营商名字，绘制时间显示条等。
- 调用函数 gdi_layer_blt 合并图层并重绘屏幕。
- 状态栏绘制结束。

（3）启动菜单绘制。

- 在图层上绘制列表项，绘制菜单框架，绘制相关特效。
- 调用函数 gdi_layer_blt 合并图层并重绘。
- 菜单绘制结束。

（4）启动按钮绘制。

- 在图层上绘制一个按钮。
- 调用函数 gdi_layer_blt 合并图层并重绘。
- 按钮绘制结束。

（5）最后调用函数 gdi_layer_blt 将所有的元素重绘。

（6）退出屏幕绘制函数。

从上述流程可以清楚地看到，整个屏幕绘制过程中总共调用了 4 次 gdi_layer_blt 函数，这极大地消耗了系统资源，但实际上起作用的是最后一次的合并。道理很简单：所有的元素都已绘制完毕，只需要在函数结尾整体绘制一次即可。有读者可能会说：那将前面的调用 gdi_layer_blt 函数的步骤去掉不就可以了？这样做是不行的。

因为每个元素都是一个独立的个体可以供其他模块调用，内部都必须有图层合并重绘函数才能将自己显示出来。只是由于在当前屏幕中，多个元素集中在一起，才使得每个元素内部的图层合并显得多余。

那么该怎样解决上述问题？

MTK 图形系统提供了一组锁层的函数：

```
gdi_layer_lock_frame_buffer()
gdi_layer_unlock_frame_buffer()
```

其原理是在绘画时加入一个计数器，gdi_layer_lock_frame_buffer 函数会将计数器加 1，gdi_layer_unlock_frame_buffer 函数将计数器减去 1，这样，只在当计数器为零时 gdi_layer_blt 才会真正起作用。

下面，让我们通过一个例子来具体讲解其原理。

（1）进入屏幕绘制函数。

· gdi_layer_lock_frame_buffer();（锁定当前图层）

（2）启动状态栏绘制。

· 各种图标，时间显示……

· 调用函数 gdi_layer_blt 合并图层并重绘（不起作用）

· 状态栏绘制结束

（3）启动菜单绘制。

· 列表项，相关特效

· 调用函数 gdi_layer_blt 合并图层并重绘（不起作用）

· 菜单绘制结束

（4）启动按钮绘制。

· 相关的工作……

· 调用函数 gdi_layer_blt 合并图层并重绘（不起作用）

· 按钮绘制结束

· gdi_layer_unlock_frame_buffer ();（释放当前图层，gdi_layer_blt 可以起作用了）

（5）最后调用函数 gdi_layer_blt 将所有的元素重绘。

（6）退出屏幕绘制函数。

最终的相关代码如下：

```
void EntryFirstAppScreen(void)
{
  gdi_color     blue_color;
  gdi_color     white_color;
  U8 *outside_mem = NULL;
  gdi_layer_multi_layer_enable();
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  gdi_layer_lock_frame_buffer();
  outside_mem = (U8 *)mmi_frm_scrmem_alloc
(((GDI_LCD_WIDTH*GDI_LCD_HEIGHT*GDI_MAINLCD_BIT_PER_PIXEL)>>3));
  blue_color = GDI_RGB_TO_BUFFER_FORMAT(0,0,255);
  white_color = GDI_RGB_TO_BUFFER_FORMAT(255,255,255);
  entry_full_screen();
  clear_screen();

  gdi_layer_create_using_outside_memory(10,  10,  150,  150,  &new_layer,  outside_mem,
```

```
((GDI_LCD_WIDTH*GDI_LCD_HEIGHT*GDI_MAINLCD_BIT_PER_PIXEL)>>3));
    //gdi_layer_set_active(new_layer);
    gdi_layer_push_and_set_active(new_layer);
    gdi_layer_clear(blue_color);
    gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);

    gdi_layer_set_source_key(TRUE, blue_color);
    gdi_layer_set_opacity(TRUE,100);
    gdi_layer_pop_and_restore_active();

    gdi_layer_get_base_handle(&base_layer);
    gdi_layer_set_active(base_layer);
    gdi_layer_reset_clip();
    gdi_layer_clear(white_color);
    gdi_image_draw_id(0,0,IMG_FIRSTAPP_BACKGROUND);

    gdi_layer_unlock_frame_buffer();
    gdi_layer_set_blt_layer(base_layer, new_layer, NULL, NULL);
    //gdi_layer_blt(base_layer,
                  new_layer,
                  NULL,NULL,
                  0, 0,UI_device_width-1,UI_device_height-1);
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
    mmi_frm_scrmem_free(outside_mem);
    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

总之，当一个屏幕中有多个元素，且每个元素都有独立的图层合并和重绘函数时，就需要调用锁层函数来减少系统资源的开销。

7.12 GDI 锁

在整个 MTK 系统运行的过程中，不仅是某一个模块的执行，在同一个时间可能有多个线程或者说 Task 在运行，而每个线程或者 Task 都可能定义有自己的 GDI 绘制函数。每一个 GDI 函数在运行的过程中都可能受到来自整个系统运行中的线程的影响。为了防止在绘图过程中发生不必要的意外，MTK 图形系统提供了一对互斥保护宏：

```
GDI_LOCK
GDI_UNLOCK
```

我在系统代码中搜索了下，系统里面函数（例如音乐播放器等）均没有使用这两个宏，所以我觉得还是不太加的好

这两个宏实际上是对应下面两个函数的，关于互斥的深入理解，请参考相关操作系统书籍。

```
#define GDI_LOCK     gdi_fast_mutex_lock()
#define GDI_UNLOCK     gdi_fast_mutex_unlock()
```

在你的应用程序中需要的位置加上上述两把锁，可以提高应用程序的执行效率，避免多线程模式下出现绘制异常和效率低下的情况。

```
void EntryFirstAppScreen(void)
{
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
GDI_LOCK;
  gdi_layer_lock_frame_buffer();
```

```
    outside_mem = (U8 *)
                  mmi_frm_scrmem_alloc(
                  ((GDI_LCD_WIDTH*GDI_LCD_HEIGHT*
                  GDI_MAINLCD_BIT_PER_PIXEL)>>3));
    …
    gdi_layer_create(10, 10, 200, 200, & new_layer);
    //gdi_layer_set_active(new_layer);
    gdi_layer_push_and_set_active(new_layer);
    gdi_image_draw_id(10,10,IMG_FIRSTAPP_PICTURE);

    gdi_layer_set_source_key(TRUE, blue_color);
    gdi_layer_set_opacity(TRUE,100);
    gdi_layer_pop_and_restore_active();

    gdi_layer_get_base_handle(&base_layer);
    gdi_layer_set_active(base_layer);
    gdi_layer_reset_clip();
    gdi_image_draw_id(0,0,IMG_FIRSTAPP_BACKGROUND);

    gdi_layer_unlock_frame_buffer();
    gdi_layer_set_blt_layer(base_layer, new_layer, NULL, NULL);
    //gdi_layer_blt(base_layer, new_layer,
    //                NULL,NULL,0, 0,
    //                UI_device_width-1,UI_device_height-1);
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
    GDI_UNLOCK;
    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
    }
```

结语：

在本章中，我们学习了如何创建一个层、激活层和合并层，并掌握了如何在层上进行各种特效的开发，对层开发中容易出错的地方进行了重点讲解。学完本章，读者应该能够进行一些简单的图层相关的开发工作。同时，我们还需要重申如下几点容易忽视的地方。

（1）必须设置图层为激活状态，才能将图层上的效果体现出来。如绘图、设置裁剪区域、透明、通透等。

（2）在离开其他图层后确保将基础层复原。

（3）如下函数必须保证成对出现：

```
gdi_layer_lock_frame_buffer
gdi_layer_unlock_frame_buffer

gdi_layer_push_clip
gdi_layer_pop_clip

gdi_layer_push_text_clip
gdi_layer_pop_text_clip

gdi_layer_push_and_set_active
gdi_layer_pop_and_restore_active
```

（4）每个 GDI 函数都应该加上互斥锁 GDI_LOCK 和 GDI_UNLOCK。

Task 8

Task 是一个具有一定封装性的软件模块（Module），当该模块被执行的时候，你可以把它理解为一个独立的进程。在整个手机系统运行的过程中，有很多的 Task 被执行。

它可以作为“服务器”处理来自客户 Task 的消息并完成预定义的服务，也可以变身为“客服端”通过发送消息向其他 Task 请求服务。Task 对于用户（客户开发人员）来说是透明的，用户不需要关心服务的具体实现；但用户必须了解 Task 提供的服务，这样才能更有效地利用该模块提供更强大的功能。

MTK 系统有很多内置 Task（如负责处理音频、视频、图像等相关请求的 Media Task），MTK 平台也允许开发人员增加处理自己业务逻辑的自定义 Task。

在第 2 章的 2.3 节，已经提前为大家讲解了 Task 的基本功能和特征，接下来的各节中，我们将介绍如何增加自定义 Task 以及如何向增加的 Task 请求服务。

8.1 创建 Task

MTK 为我们预留了自定义 Task 的接口，可以在 Custom_config.h 和 Custom_config.c 文件中找到对应的定义。下面是代码片段：

```
/*****************************************************************************
 * [Very Important Message]
 * 1. Component Task Index (Please add before system service)
 * 2. Customers are allowed to create at most 16 tasks as defined in
 *    config\include\stack_config.h (MAX_CUSTOM_TASKS = 16)
 *****************************************************************************/
typedef enum {
  INDX_CUSTOM1 = RPS_CUSTOM_TASKS_BEGIN,
  INDX_CUSTOM2,
  RPS_CUSTOM_TASKS_END
} custom_task_indx_type;

/*****************************************************************************
 * [Very Important Message]
 * 1. Component task's module id (Please add before system service)
 * 2. Customers are allowed to create at most 16 task module ID as defined
 *    in config\include\stack_config.h (MAX_CUSTOM_MODS = 16)
```

· QQ 就在此定义了一个 ID
#if defined (__MT_TENCENT_QQ_CLIENT__)
INDX_QQ,
#endif

```
/*****************************************************************************
 ******************************************************************************
typedef enum {
  MOD_CUSTOM1 = MOD_CUSTOM_BEGIN,
  MOD_CUSTOM2,
  MOD_CUSTOM_END
} custom_module_type;

/*****************************************************************************
 * Other Declarations
 *****************************************************************************
extern custom_task_indx_type custom_mod_task_g[ MAX_CUSTOM_MODS ];
```

QQ在此定义了一个 #if defined(__MT_TENCENT_QQ_CLIENT__) MOD_MQQ, #endif

而有了这些接口使得增加一个 Task 显得如此简单。

8.1.1 增加索引和 ID

首先在 Custom_config.h 文件中的 custom_task_indx_type 和 custom_module_type 两个枚举类型中分别加入我们新增的 task index 和 module id。

```
typedef enum
{
   INDX_CUSTOM1 = RPS_CUSTOM_TASKS_BEGIN,
   INDX_CUSTOM2,
   INDX_TEST,
   RPS_CUSTOM_TASKS_END
} custom_task_indx_type;

typedef enum {
  MOD_CUSTOM1 = MOD_CUSTOM_BEGIN,
  MOD_CUSTOM2,
  MOD_TEST,
  MOD_CUSTOM_END
} custom_module_type;
```

> **注 意**
>
> MTK 默认只支持 16 个自定义 Task，见 config\include\stack_config.h 文件中的 MAX_CUSTOM_TASKS 和 MAX_CUSTOM_MODS 定义。

8.1.2 增加映射

添加完 task index 和 module id 后我们需要在 Custom_config.c 文件的 custom_mod_task_g 定义中增加 Moudle 与 Task 的映射。

```
custom_task_indx_type custom_mod_task_g[ MAX_CUSTOM_MODS ] =
{
      INDX_CUSTOM1,          /* MOD_CUSTOM1 */
      INDX_CUSTOM2,          /* MOD_CUSTOM2 */
      INDX_TEST,             /* MOD_TEST */
      INDX_NIL               /* Please end with INDX_NIL element */
};
```

> **注 意**
>
> 映射顺序必须与 custom_config.h 文件的 custom_module_type 中的 module id 定义顺序保持一致。

8.1.3 配置 Task

添加完映射后我们需要在 Task 配置表 custom_comp_config_tbl 中增加 Task 的配置信息。

```
const comptask_info_struct custom_comp_config_tbl[ MAX_CUSTOM_TASKS ] =
{
        /* INDX_CUSTOM1 */
          {"CUST1", "CUST1 Q", 211, 1024, 10, 0,
#ifdef CUSTOM1_EXIST
            custom1_create, KAL_FALSE
#else
            NULL, KAL_FALSE
#endif
        },

        /* INDX_CUSTOM2 */
        {"CUST2", "CUST2 Q", 212, 1024, 10, 0,
#ifdef CUSTOM2_EXIST
  custom2_create, KAL_FALSE
#else
  NULL, KAL_FALSE
#endif
},
 {"TEST", "TEST", 210, 1024, 10, 0, test_task_create, KAL_FALSE}
};
```

#if defined (__MT_TECENT_QQ_CLIENT__)
{"MQQ", "MQQ Q", 212, 4096, 100, 10, mqq_create, KAL_FALSE},
#endif

读者可以在 Syscomp_config.h 文件中找到 comptask_info_struct 的定义。

```
typedef struct
{
   kal_char            *comp_name_ptr;
   kal_char            *comp_qname_ptr;
   kal_uint32          comp_priority;
   kal_uint16          comp_stack_size;
   kal_uint8           comp_ext_qsize;
   kal_uint8           comp_int_qsize;
   kal_create_func_ptr  comp_create_func;
   kal_bool            comp_internal_ram_stack;
} comptask_info_struct;
```

comptask_info_struct 结构中定义了 Task 的相关配置信息。

comp_name_ptr：Task 名称。

comp_qname_ptr：外部队列名称。

comp_priority：Task 优先级。

comp_stack_size：Task 堆栈大小。

comp_ext_qsize：外部队列大小。

comp_int_qsize：内部队列大小。

comp_create_func：创建 Task 的入口函数。

comp_create_func 是函数指针，要求函数返回 kal_bool，要求有一个 comptask_handler_struct **参数，定义如下：

```
typedef kal_bool (*kal_create_func_ptr)(comptask_handler_struct **);
```

comptask_handler_struct 定义了 Task 生命周期内系统将要回调的函数入口：

```
typedef struct
{
      kal_task_func_ptr     comp_entry_func;
      task_init_func_ptr    comp_init_func;
      task_cfg_func_ptr     comp_cfg_func;
      task_reset_func_ptr   comp_reset_func;
      task_end_func_ptr     comp_end_func;
} comptask_handler_struct;
```

comp_entry_func：主函数，该函数实现为一个消息循环。

comp_cfg_func：配置函数。

comp_init_func：初始化函数。

comp_reset_func：重置函数。

comp_end_func：Task 结束时调用。

8.1.4 实现

下面代码演示了 Task 创建函数的实现：

```
//test_main.cpp
kal_bool test_task_create(comptask_handler_struct **handle)
{
  static const comptask_handler_struct test_handler_info =
  {
    test_main,        task entry function */
    NULL,            task initialization function */
    NULL,            task configuration function */
    NULL,            task reset handler */
    NULL,            task termination handler */
  };

  *handle = (comptask_handler_struct *)&test_handler_info;
  return KAL_TRUE;
}
```

在 test_task_create 函数中，定义了一个 comptask_handler_struct 结构体的静态变量，并将地址赋值给了 handle 参数。

在这个实例中我们只为 Task 提供了主函数 test_main 来获取和处理 Task 的消息，因为 Task 的运行环境相对简单，没有使用到其他资源，所以将初始化、配置处理函数等均设置为 NULL。

现在我们来看看如何实现 Task 的消息处理函数，下面是消息处理函数的原型：

```
void test_main (task_entry_struct * task_entry_ptr);
```

其中只有一个 task_entry_struct 结构体指针作参数，task_entry_struct 结构定义如下：

```
typedef struct task_entry_struct
{
    task_indx_type task_indx;
} task_entry_struct;
```

task_entry_struct 结构定义很简单，这里就不做过多解释，读者可以参考源码中的示例自己完成。

8.2 Task 的消息循环过程

接下来我们看一下消息处理循环的实现步骤。

系统运行中，会首先调用 receive_msg_ext_q 从外部消息队列中获取消息，如果消息队列中有消息则立即返回待处理的消息，否则 receive_msg_ext_q 函数将阻塞 Task，直到有消息进入消息队列。

函数原型如下：

```
kal_status receive_msg_ext_q(kal_msgqid task_ext_qid,  ilm_struct *ilm_ptr);
```

参数 1：消息队列 ID，可以在 task_info_struct 结构体数组 task_info_g 中获得。

参数 2：消息的通用结构，在 8.3 节作详细介绍。

例如：

```
receive_msg_ext_q(task_info_g[task_entry_ptr->task_indx].task_ext_qid, &current_ilm);
```

开发过程中，应该根据消息 id 分别对消息进行处理。有时可能还有 Task 内部的消息需要处理，则用 receive_msg_int_q 从内部消息队列获取消息，这里不作详细介绍。

我们来看 test_main 的完整代码：

```
static void test_main(task_entry_struct * task_entry_ptr)
{
  ilm_struct current_ilm;
  while ( 1 ) {
    receive_msg_ext_q(task_info_g[task_entry_ptr->task_indx].task_ext_qid, &current_ilm);
    switch (current_ilm.msg_id)
    {
      case MSG_ID_TEST_START:
          //消息处理
           break;
      default:
           break;
    }
    free_ilm(&current_ilm);
  }
}
```

关于加消息ID
系统预留的 MSG_ID_CUSTOM2_CUSTOM1 以及QQ添加的
消息ID均定在 Custom-sap.h 和 gsmen.pl 两文件中添加的
(我自己用的消息ID, MSG_ID_MUSIC_PAUSE 也定在这两个文件中定义的，而且我也没有定义 Task 和 Module)

前面我们已经讲到，用户不需要关心 Task 的具体实现就能使用 Task 提供的服务，用户怎么知道 Task 提供了哪些服务呢？这就需要用到接下来要讲到的消息 ID。

消息 ID 向用户展现了 Task 提供的服务，这里我们只讲创建 Task 时如何定义消息 ID，在使用 Task 一节会涉及用户如何使用消息 ID 来请求服务。

可以有两种方式定义消息 ID。

（1）在 adaptation\include\stack_msgs.h 文件中 msg_type 定义的 MSG_ID_END 之前加入要定义的消息 ID。

（2）在 interface\ps 文件夹下加入一个头文件，如：test_sap.h。在该头文件中加入要定义的消息 ID，再将头文件包含到 adaptation\include\stack_msgs.h 文件中 msg_type 定义的 MSG_ID_END 之前。

这两种方式没有本质差异，在可读性和可维护性上第二种方式表现得更好。

现在来为 test 实例定义消息 ID，我们采用第二种方式。

首先在 test_sap.h 文件中定义消息 ID：

```
MSG_ID_TEST
```

将 test_sap.h 文件包含到 adaptation\include\stack_msgs.h 文件中 msg_type 定义中。如图 8.1 所示。

```
#imclude"custom_sap.h"
#ifdef TEST_MODE
#inclde"test_sap.h"
#endif
#endif /*_MTK_TARGET_II KAL_ON_NUCLEUS II MTK_KAL_MNT II KAL_ON_OSCAR II KAL_ON_ THREADX*/
MSG_ID_END
}?end msg_type? msg_type;
```

图 8.1　stack_msg.h 文件片断

以上所有内容，完成了一个简单 Task 的定义。接下来我们将学习如何使用 Task。

8.3　使用 Task

前面讲到 Task 是基于消息驱动的，在使用 Task 之前需要向 Task 的开发人员了解我们可以向 Task 请求哪些消息，这些消息都完成什么服务，以及请求这些消息时都需要提供什么信息？当了解到这些信息后我们就可以在需要的时候向 Task 请求服务。

8.3.1　消息结构

首先让我们来看一下如何向 Task 请求服务，我们先来了解一下发送消息时的消息结构，定义如下：

```
typedef struct ilm_struct
{
 module_type       src_mod_id;
 module_type       dest_mod_id;
 sap_type          sap_id;
 msg_type          msg_id;
 local_para_struct *local_para_ptr;
 peer_buff_struct  *peer_buff_ptr;
} ilm_struct;
```

其中的参数如下。

src_mod_id：设置了请求服务的模块 ID。

dest_mod_id：接收请求的模块 ID。

sap_id：服务请求标识，如果没有定义可以填 0。

msg_id：请求的消息 ID。

local_para_ptr：请求需要提供的附加信息，没有则为 NULL。

peer_buff_ptr：请求需要提供的附加信息，没有则为 NULL。

8.3.2　填充消息结构体

现在来看一下如何定义与 local_para_ptr 相对应的结构体，首先要包含 LOCAL_PARA_HDR，接下

来定义与消息相关的参数。

```
typedef struct
{
 LOCAL_PARA_HDR
 kal_int16 result;
 kal_uint16 seq_num;
}media_vid_play_finish_ind_struct;
```

LOCAL_PARA_HDR 是一个宏定义，其中定义了引用次数和消息长度。

```
#define LOCAL_PARA_HDR \
 U8  ref_count; \
 U16 msg_len;
```

peer_buff_ptr 对应的结构体定义与 local_para_ptr 相似，把 LOCAL_PARA_HDR 替换为 PEER_BUFF_HDR 即可。在应用中很少会使用到 peer_buff_ptr。

8.3.3 投递到消息队列

将消息结构填充完成后我们就需要把消息放入到消息队列中，当源模块和目的模块用相同的 Task 处理时我们需要调用如下函数将消息放入内部消息队列。

```
kal_bool msg_send_int_queue(ilm_struct *ilm_ptr);
```

在音乐层

如果源模块和目的模块的 Task 不相同，我们需要将消息放入外部消息队列。

```
kal_bool msg_send_ext_queue(ilm_struct *ilm_ptr);
```

→ 一般我们引用 and_send_msg，它是对 msg_send_ex

我们也可以用 MTK 为我们封装消息发送的宏来完成消息发送，这样我们不需要关注源模块与目的模块的 Task 是否相同的问题，代码如下所示：

queue 的一个封装.

```
#define SEND_ILM( src_mod, dest_mod, sap, ilm_ptr)\
{ \
        ilm_ptr->src_mod_id  = src_mod;  \
            ilm_ptr->dest_mod_id = dest_mod; \
            ilm_ptr->sap_id = sap; \
            if (mod_task_g[src_mod] == mod_task_g[dest_mod]) { \
            msg_send_int_queue(ilm_ptr); \
            } else { \
            msg_send_ext_queue(ilm_ptr); \
            } \
}
```

在此说明一下我在 client_msg.dat 中定义的消息 ID 会没有定义 Task，是因为我是要向已有的 Task 发消息

现在我们来看一个完整的消息发送的应用。

```
void vid_send_play_finish_ind(kal_int16 result)
{
    ilm_struct *ilm_ptr = NULL;
    ilm_ptr = allocate_ilm(MOD_MMI);
    ilm_ptr->msg_id = (msg_type) MSG_ID_TEST;
    ilm_ptr->local_para_ptr = NULL;
    ilm_ptr->peer_buff_ptr = NULL;

    SEND_ILM (MOD_MMI, MOD_TEST, 0, ilm_ptr);
}
```

接下来我们再看一个关于 local_para_ptr 的应用：该函数位于文件 Aud_media.c 文件中。

```
void vid_send_play_finish_ind(kal_int16 result)
{
    media_vid_play_finish_ind_struct *ind_p;
    ilm_struct *ilm_ptr = NULL;

    //参数结构体变量构造
    ind_p = (media_vid_play_finish_ind_struct*)
        construct_local_para(sizeof(media_vid_play_finish_ind_struct), TD_CTRL);

    //向结构体中填充相应的参数值
    ind_p->result = result;
    ind_p->seq_num = vid_context_p->seq_num;

    ilm_ptr = allocate_ilm(MOD_MED);
    ilm_ptr->src_mod_id = MOD_MED;
    ilm_ptr->dest_mod_id = vid_context_p->src_mod;
    ilm_ptr->sap_id = MED_SAP;

    ilm_ptr->msg_id = (msg_type) MSG_ID_MEDIA_VID_PLAY_FINISH_IND;
    ilm_ptr->local_para_ptr = (local_para_struct*) ind_p;
    ilm_ptr->peer_buff_ptr = NULL;

    msg_send_ext_queue(ilm_ptr);
}
```

读者可以在 MTK 源码中看到大量类似上述代码的语句，Task 的应用大大方便了各个模块之间的通信交流。

结语：

本章，对系统中的一个重要部分 Task 做了一个细致的梳理，并设计了实例来讲解如何创建一个新的 Task 及如何使用 Task。其实，大家完全可以将 Task 比作一个进程，Task 之间的交流好比是进程通信。虽然不是绝对准确，但也是一个理解 Task 思想的好方法。

通过本章的学习，读者可以独立完成一个 Task 的设计，为以后设计高级的应用程序做好准备。

9 文件操作

文件系统是对一个存储设备上的数据和元数据进行组织的机制。

从系统角度来讲，文件系统是对文件存储器空间进行组织和分配，负责文件的存储并对存入的文件进行保护和检索的系统。具体地说，它负责为用户建立文件，提供存入、读出、修改、转储文件、控制文件的存取等操作。

MTK 提供了一套完整的文件系统（以下简称 FS），其主要特性如下。

· 支持 FAT-16 和 FAT-32 文件格式

FS 支持与 MS-DOS、Windows 等相同的 FAT 格式，各种媒体文件都能在 FS 和其他操作系统中交互访问。

· 长文件名支持

文件名允许最大 255 个字符，绝对路径允许最大为 260 个字符。

· 支持 UNICODE 宽字符

FS 支持 UNICODE 编码，所有的在 FS 中访问到的文件和路径名字符串，其每个字符都将占有两个字节，也就是说在 FS 中全路径字符串将需要 520（260 × 2）个字节进行存储。

· 支持软盘、闪存、SRAM 卡

通过 FS 的自定义设备和驱动程序接口使得添加自定义的驱动器非常简单。

· 高效的缓存支持

临时储存最近访问过的数据，将读取频繁的那些数据存放到缓存中，当系统需要时，直接从设备缓存中调用，大大提高读写性能。

· 多任务支持

提供在多任务情况下，对输入/输出数据正确性的保证。

· 安装设备驱动

FS 的设备驱动程序接口是非常简单的，只需要调用 3 个函数：MountDevice、ReadSectors 和 WriteSectors。所有文件系统包装好的并且有完整资源代码的驱动都可以作为实现客户驱动的模型。

MTK 使用的 FAT 文件系统，在模拟器中放在目录\plutommi\WIN32FS 下，如 DRIVE_C、

DRIVE_D 等。

例如：FS_Open(L"C:\\test.txt", FS_READ_ONLY);

上面代码以只读模式打开 C 盘下的 test.txt 文件，文件名的字符编码是 UCS-2，即宽字符。

9.1 常用函数

MTK 提供了大量的文件操作函数，在头文件 fs_func.h 中可以找到绝大部分的声明，实现位于文件 FSSim_core.c 中。

常用的文件操作函数有：

```
extern int FS_Open(const WCHAR * FileName, UINT Flag);
extern int FS_Close(FS_HANDLE FileHandle);
extern int FS_Read(FS_HANDLE FileHandle, void * DataPtr, UINT Length, UINT * Read);
extern int FS_Write(FS_HANDLE FileHandle, void * DataPtr, UINT Length, UINT * Written);
extern int FS_Seek(FS_HANDLE FileHandle, int Offset, int Whence);
extern int FS_Commit(FS_HANDLE FileHandle);
extern int FS_Delete(const WCHAR * FileName);
extern int FS_GetCurrentDir(WCHAR * DirName, UINT MaxLength);
extern int FS_SetCurrentDir(const WCHAR * DirName);
extern int FS_CreateDir(const WCHAR * DirName);
extern int FS_RemoveDir(const WCHAR * DirName);
extern int FS_FindFirst(const WCHAR * NamePattern, BYTE Attr, BYTE AttrMask,
                FS_DOSDirEntry * FileInfo, WCHAR * FileName, UINT MaxLength);
extern int FS_FindNext(FS_HANDLE FileHandle, FS_DOSDirEntry * FileInfo, WCHAR *
                FileName, UINT MaxLength);
```

其实，MTK 的文件操作 API 和 C 语言的文件操作没有多大的差异，如果读者有良好的 C 语言基础，那么这一章就不会有什么难题。

MTK 提供的文件 API 很多，限于篇幅不可能一一列举。我们抽取几个常用的函数进行讲解。

9.1.1 FS_Open

· 功能

FS_Open 将打开一个文件（如果文件存在）或者创建一个文件，为随后的读写访问作准备。

· 原型

```
FS_HANDLE FS_Open(const WCHAR * FileName, UINT Flags)
```

· 参数

文件名（FileName）：需要打开或创建的文件名。在传入函数时，请保证其是宽字符串。文件名对大小写不敏感，最终传入函数的文件名都会被转换为大写。文件名可以有如下几种格式：

· [Drive:][\][Path\]Name[.Ext]

该文件名包含其所在的位置的路径。其中，Drive，Path 等都是可变化的，据情况而定。

比如：D:\File\newFile.txt。

· [Drive:]\

根路径。

· \\.\Drive:

逻辑上的相对路径。

· \\.\PHYSICALDRIVEx

物理硬盘。小写的 x 必须是一个数字（从“0”开始），用于指定所需的硬盘。

提 示

Drive（盘符）必须是一个不小于大写字母“A”的字符。如：

A:\ D:\

在实际开发过程中，必须使用双斜杠，避免转义符被过滤掉。如：

“C:\\sdk\\test.txt”

主要标志（Flags）

FS_READ_WRITE：文件打开后，可读可写。

FS_READ_ONLY：只读访问。

FS_OPEN_SHARED：设置该标志，可以避免“Access Denied”的异常。默认情况下，FS 允许具有只读访问属性的文件打开多次，但如果又需要进行写操作，就会出现上面的异常。设置该标志即可。

FS_CREATE：如果传入的文件名所指示的文件不存在的话，该标志指示 FS_Open 创建文件。

FS_CREATE_ALWAYS：该标志指示 FS_Open 总是创建一个文件，即使文件存在。

FS_COMMITTED：所有修改过的文件立即写入物理设备。

FS_CACHE_DATA：指示 FS 为该文件不要丢弃缓存中的数据。这个标志在随机访问那些被经常读取的数据块时很有用。

FS_LAZY_DATA：指示不要刷新脏数据（修改过的数据）缓冲。这个标志可以在相同数据写入多次的情况下，大大提高性能，因为重复的数据仅仅是不断地覆盖先前内容，而不会实际写在物理设备上。这个标志会自动地设置 FS_CACHE_DATA 标志。

· 返回值

如果该函数成功，返回当前打开的文件句柄，且返回值为 0。

如果该函数失败，返回一个错误码，且返回值将会小于 0。

9.1.2 FS_Close

· 描述

FS_Close 将关闭打开的文件并返回一个整形值，释放所有与打开文件句柄相关的资源。

· 原型

```
int FS_Close (FS_HANDLE File)
```

· 参数

File：必须保证传入的是之前已经在 FS_Open 成功创建的文件句柄。

· 返回值

函数成功，返回 FS_NO_ERROR。

函数失败，返回一个负数的错误码。

此外，还有一个与 FS_Close 类似的函数：FS_CloseAll。该函数将会关闭当前所有被打开的文件，其原型为 int FS_CloseAll(void)，使用方式和 FS_Close 没有太大差别。

9.1.3 FS_Read

· 描述

从打开的文件中读取数据。

· 原型

```
int FS_Read(FS_HANDLE File, void * DataPtr, UINT Length, UINT * Read).
```

· 参数

File：将要用于读取数据的文件句柄。

DataPtr：指明将要读取的数据地址。

Length：指明将要读取的数据字节数。

Read：无符号整形指针，用于接收实际读入的字节数。如果你的程序不需要该信息可以将其设置为 NULL。

· 返回值

函数成功，返回 FS_NO_ERROR。

函数失败，返回一个负数错误码。

> *提 示*
>
> 假设之前的 Read 没有设置为 NULL，比如函数
>
> FS_Read(op->fileHandle, op->pos, length, &rd_no)，那么在 FS_Read 结束的时候，如果 rd_no 的值小于 length，即读入的字节数小于预想的值，则说明已经读到了文件的末尾。
>
> 在 MTK 中，读取操作略过文件结束符不会视为是一个错误。

9.1.4 FS_Write

· 描述

将数据写入打开的文件。

· 原型

```
int FS_Write(FS_HANDLE File, void * DataPtr, UINT Length, UINT * Written)
```

· 参数

File：将要用于写入数据的文件句柄。

DataPtr：指明将要写入的数据地址。

Length：指明将要写入的数据字节数。

Written：无符号整形指针，用于计算实际写入的字节数。如果你的程序不需要该信息，

可以将其设置为 NULL。

· 返回值

函数成功，返回 FS_NO_ERROR。

函数失败，返回一个负数错误码。

提 示

写入数据的过程中，如果已经略过了文件结束符，那么，系统会自动地扩展文件。当文件的数据容量超过了当前文件已分配的容量时，系统就会立即在当前的 Cluster 簇之后为文件分配新的 Cluster 簇。

扩展知识

Cluster 簇：

磁盘是由一个个扇区组成的，若干个扇区合为一个簇，文件存取是以簇为单位的，即使这个文件只有 1 个字节。每个簇在文件分配表中都有对应的表项，簇号即为表项号，每个表项占 1.5 个字节（磁盘空间在 10MB 以下）或 2 个字节（磁盘空间在 10MB 以上）。

9.1.5 FS_Seek

· 描述

复位文件指针，并可能扩展文件。通过对文件指针的移动，使得可以在某个位置添加数据。

· 原型

```
int FS_Seek(FS_HANDLE File, int Offset, int Whence)
```

· 参数

File：用于当前操作的文件句柄。

Offset：指明用于数据读写的文件指针移动的距离。

Whence：指明文件指针从何处开始移动，有 3 种方式。

FS_FILE_BEGIN：　　文件指针移动到文件头。

FS_FILE_CURRENT：　　文件指针移动到当前位置。

FS_FILE_END：　　文件指针移动到文件末尾。

· 返回值

函数成功，将会返回一个新的文件指针。而且，实际的文件指针能够通过 FS_GetFileInfo 来查询。

函数失败，返回一个为负数的错误码。

比如，我们需要在文件头部写入数据：

```
FS_Seek(handleFind, 0, FS_FILE_BEGIN);
```

9.1.6 FS_Commit

· 描述

立即刷新文件缓冲区的数据并写入到到磁盘。

· 原型

```
int FS_Commit(FS_HANDLE File)
```

· 参数

File：用于提交数据的文件。

· 返回值

函数成功，返回 FS_NO_ERROR。

函数失败，返回一个负数的错误码。

> 提 示
>
> FS_Commit 保证所有文件的数据被刷新。如果该文件打开了 FS_LAZY_DATA 标志，那么文件缓存中的脏数据将会被刷新。否则，该文件的数据缓冲区，目录项和完整的 FAT 表都将全部刷新。

9.1.7 FS_GetFileInfo

· 描述

返回当前打开文件的具体信息。

· 原型

```
int FS_GetFileInfo(FS_HANDLE File, FS_FileInfo * FileInfo)
```

· 参数

File：请求信息的文件句柄。

FileInfo：输出的文件信息将存入 FS_FileInfo 结构体实例。结构体定义如下：

```
typedef struct {

  //目录信息，当文件打开时才有效
  const FS_DOSDirEntry            * DirEntry;
  //当前文件指针位置。结合 FS_Seek 分析
  UINT                            FilePos;
  //当前分配的文件大小
  UINT                            AllocatedSize;
  //指明当前文件已经分配了多少个 cluster 簇
  UINT                            ClusterChains;
  //卷标序列号。很少使用到
  UINT                            VolumeSerialNumber
  //完整的文件名，包括其路径，只在文件打开时才有效
  const char                      * FullName;
} FS_FileInfo;
```

· 返回值

函数成功，返回 FS_NO_ERROR。结构体实例 FileInfo 将会填满各种文件信息。

函数失败，返回一个负数的错误码。

9.1.8 FS_GetAttributes

· 描述

获得当前文件的属性。

· 原型

```
int FS_GetAttributes(const WCHAR * FileName)
```

· 参数

FileName：请求获得文件属性的文件名。

· 返回值

如果返回的值是正数，那么它包含文件的属性，可以和 MTK 系统定义的若干宏进行组合。

```
#define FS_ATTR_READ_ONLY        0x01    //该文件只读
#define FS_ATTR_HIDDEN           0x02    //该文件标志为隐藏
#define FS_ATTR_SYSTEM           0x04    //该文件标志为系统文件
#define FS_ATTR_VOLUME           0x08    //该文件是一个卷标
#define FS_ATTR_DIR              0x10    //该文件是一个目录
#define FS_ATTR_ARCHIVE          0x20    //该文件作为备份
```

如果函数失败，返回一个负数错误码。

与 FS_GetFileInfo 对应的是 FS_SetFileInfo，原理差不多，读者可以自己研究。

9.1.9 FS_FindFirst

· 描述

在目录下搜索匹配 NamePattern 模式的文件。

· 原型

```
int FS_FindFirst(const WCHAR * NamePattern, BYTE Attr,
                 BYTE AttrMask, FS_DOSDirEntry * FileInfo,
                 WCHAR * FileName, UINT MaxLength)
```

· 参数

NamePattern：类似一种正则表达式，传入的文件名可以包含通配符，如“*”，“?”等。为了与 MS－DOS 模式兼容，MTK 还将模式“*.*”转换为“*”。关于正则表达式，请读者参考相关书籍。

Attr：　指明将要搜索的文件具有哪些属性。

AttrMask：　规定了一系列的文件不需要匹配的属性。

需要注意的是，上面两个属性在实际开发中都设置为 0。

FileInfo：　指向 FS_DOSDirEntry 结构体实例的指针。如果当前函数返回成功，该结构体实例会填充文件的目录信息。

FileName：　已找到的文件的文件名。

MaxLength：　文件名最大长度。如果文件的文件名长度大于最大长度－1 将无法找到。同时要注意，

它的长度应该是宽字符的长度。

· 返回值

如果函数成功，就表明至少有一个文件符合要求。*FileInfo 将包含找到的第一个文件的目录项，并且返回值大于等于 0。

返回值是一个文件句柄，该句柄可能会作为 FS_FindNext 的参数。

如果当前函数调用失败，将返回一个为负数的错误码。

举个例子，假设我们要查找 C 盘 MTK 目录中的一个 TXT 文件，代码如下：

```
FS_FindFirst(L"C:\\MTK\\?*.txt", 0, 0,
             &file_info, (U16*)pFileName[0],
             MAX_COURSE_FILENAME_LEN);
```

9.1.10 FS_FindNext

· 描述

根据 FS_FindFirst 返回的句柄，继续查找满足相同条件的文件。

· 原型

```
int FS_FindNext(FS_HANDLE File, FS_DOSDirEntry * FileInfo,
                WCHAR * FileName, UINT MaxLength)
```

· 参数

File：　由 FS_FindFirst 成功返回的文件句柄。

FileInfo：　指向 FS_DOSDirEntry 结构体实例的指针。如果当前函数返回成功，该结构体实例会填充文件的目录信息。

FileName：　已找到的文件的文件名。

MaxLength：　文件名最大长度。如果文件的文件名长度大于最大长度-1 将无法找到。同时要注意，它的长度应该是宽字符的长度。

· 返回值

如果函数调用成功，将返回一个大于零的值 FS_NO_ERROR，并且*FileInfo 和*FileName 将分别填充目录项的具体内容和文件的文件名。

如果调用失败，函数将返回一个负数的错误码。

提　示

FS_FindNext 所需要的文件句柄，必须是之前 FS_FindFirst 成功返回的文件句柄。

9.1.11 FS_FindClose

该函数功能简单，将关闭之前由 FS_FindFirst 所打开的文件句柄。

函数成功即返回 FS_NO_ERROR，失败即返回一个负数的错误码。

注　意

之前 FS_FindFirst 打开的文件句柄，在不使用的时候一定要将其关闭。

9.1.12 FS_CreateDir

· 描述

创建一个新的目录。

· 原型

```
int FS_CreateDir(const WCHAR * DirName)
```

· 参数

DirName：需要创建的目录名，可以包含任何合法的文件命名语法。

· 返回值

函数成功，返回 FS_NO_ERROR。

函数失败，返回一个负数的错误码。

9.1.13 FS_GetDrive

· 描述

根据当前系统的特性获取磁盘盘符。

MTK 系统现在可以支持 1 NOR 闪存（NOR Flash）、1 NAND 闪存（NAND Flash）、最高 1 记忆卡（最多 5 个驱动器）的容量。如果用户通过数据线使得当前手机作为一个 USB，系统就可能面临分区的问题：一些私有的数据，比如 NVRAM 或者具有数字版权的文件等，是不应该暴露在用户的 PC 上的。NOR 闪存和 NAND 闪存结构如图 9.1 和图 9.2 所示。

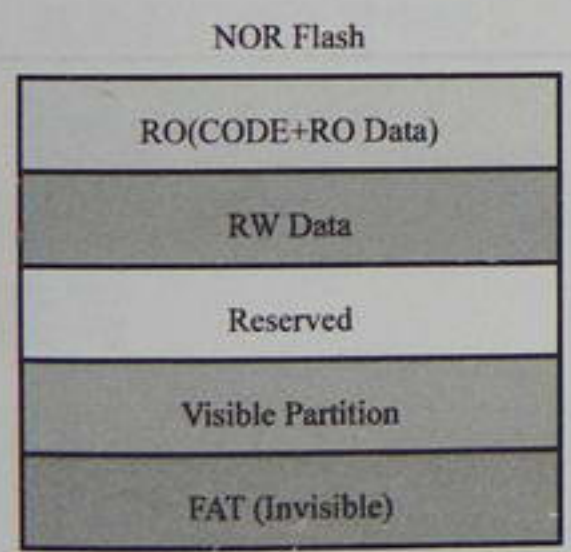


图 9.1　NOR Flash

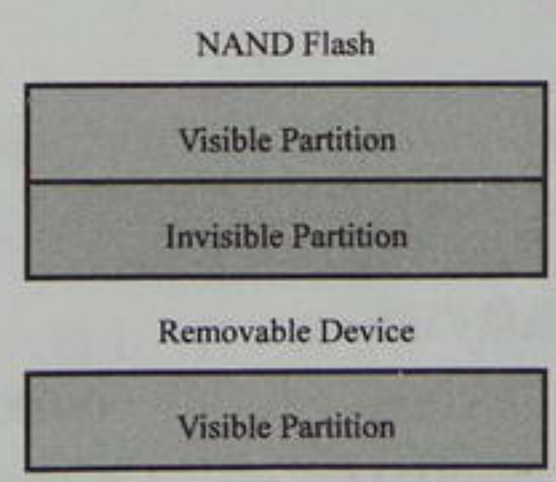


图 9.2　NAND Flash

提　示

NOR 和 NAND 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 NOR Flash 技术。1989 年，东芝公司发表了 NAND Flash 结构，强调降低每比特的成本，更高的性能，并且像磁盘一样可以通过接口轻松升级。NOR 的器件支持随机访问，而 NAND 的只支持顺序访问。因此 NAND 一般用在大规模存贮，而 NOR 的用于存程序代码，直接运行程序。而 NAND 放程序时，要先 LOAD 到 RAM 中再运行。总线接口上，NOR 的接口类型一般是总线接口，与 SRAM 器件相似；NAND 则是 IO 接口，需要另外逻辑才能作为 MEMORY。

具体的原理请参考相关的书籍。

为了解决上面提到的数据私有问题，MTK 系统必须在磁盘上创建一个分区。为了降低驱动的负担，将第 1 个分区设为可见，第 2 个分区设为不可见。这样，当手机通过数据线连接，作为一个 USB 装置连在电脑上时，驱动就会“骗过”电脑，使得电脑认为整个磁盘大小就是之前 MTK 系统设置为可见的区域。这样做，就保证了数据私有性。

总共有 12 种磁盘组合方式，如下图 9.3 所示。

IDX	DRIVES NUM	NOR	NAND	MEM CARD	NORMAL 1	NORMAL 2	SYS 1	SYS 2	CARD
1	5	2	2	1	C	D	E	F	G
2	4	2	1	1	C	D	E	…	F
3	4	2	2	0	C	D	E	F	…
4	4	1	2	1	D	…	C	E	F
5	3	2	0	1	C	…	D	…	E
6	3	2	1	0	C	D	E	…	…
7	3	1	1	1	D	…	C	…	E
8	3	1	2	0	D	…	C	E	…
9	3	2	0	0	C	…	D	…	…
10	2	1	0	1	…	…	C	…	D
11	2	1	1	0	D	…	C	…	…
12	1	1	0	0	…	…	C	…	…

图 9.3　磁盘组合方式

提　示

SYS1、SYS2 表示在手机作为 USB 存在时，不可见分区的盘符。

NORMAL1、NORMAL2 表示在手机作为 USB 存在时，可见部分的盘符。

同时，要注意到系统盘总是存在的。

· 原型

```
int  FS_GetDrive(UINT Type, UINT Serial, UINT AltMask)
```

· 参数

· Type：3 种磁盘类型。

FS_DRIVE_I_SYSTEM：　　0x00000004。

FS_DRIVE_V_NORMAL：　　0x00000008。

FS_DRIVE_V_REMOVABLE：　0x00000010。

· Serial：盘符数量。

SYSTEM：1 或 2；NORMAL：1 或 2；REMOVABLE：1。

· AltMask：提供 3 种选择不同盘符的方式。

FS_NO_ALT_DRIVE：　　该标志不能与其他方式组合。

FS_ONLY_ALT_SERIAL：　该标志不能与其他方式组合。

3 种类型的组合：(FS_DRIVE_I_SYSTEM|FS_DRIVE_V_NORMAL|FS_DRIVE_V_NORMAL)。

· 返回值

返回整型值的盘符名。

C：67

D：68

E：69

F：70

如果返回 FS_PARAM_ERROR(-2)，说明指定的类型没有定义或者盘符号越界，或者 AltMask 使用了不正确的组合方式。

返回 FS_DRIVE_NOT_FOUND (-4)，说明没有找到指定的盘符。

示例如下：

```
int  drvLetter;
TCHAR  K_filename[50];
FS_HANDLE hFile;
drvLetter = FS_GetDrive(FS_DRIVE_V_NORMAL, 2, FS_DRIVE_I_SYSTEM );
if(drvLetter > 0)
{
     kal_wsprintf(K_filename, "%c:\\ TestFile", drvLetter);
     hFile = FS_Open(K_filename,
    FS_READ_WRITE|FS_CREATE_ALWAYS|FS_OPEN_NO_DIR);
}
```

上述代码将会在系统盘中创建一个名为 TestFile 的文件。读者可以打开目录

\plutommi\WIN32FS\DRIVE_C 进入模拟器的盘符中看到该文件。

如果想创建在 D 盘，只需要修改 FS_GetDrive 中的 AltMask 参数为 FS_DRIVE_V_NORMAL，如果想创建在可移动盘（一般为 E 盘），将 AltMask 修改为 FS_DRIVE_V_REMOVABLE 即可。

以上的函数，就是我们在开发中经常用到的，系统中还有大量的 API，这里无法一一列举，请读者参考其他书籍。

9.2 文件读写

在任何平台上开发应用程序，最基础的就是文件的输入输出。任何需要持久化存储的数据，都离不开文件读写。

MTK 使用 FAT 文件系统，将模拟器文件系统放在目录\plutommi\WIN32FS 下，如 DRIVE_C、DRIVE_D 等。

我们通过一系列实例来具体讲解文件的读写过程。

1. 创建目录

首先在 E 盘根目录下创建文件夹 FileTest，编写一个函数 CreateFileDir 来实现该功能。

```
#include "fs_type.h"
#include "Fs_func.h "
#include "fs_errcode.h"
#include "SettingProfile.h"
#include "CommonScreens.h"
```

```
#define MAX_FILENAME_LEN   40
#define FILE_PATH          L"E:\\FileTest"//文件路径

S32 CreateFileDir(U16* DirName)
{
FS_HANDLE fh;
FS_DOSDirEntry  file_info;
U16 filename[MAX_CUSTOM_FILENAME_LEN];
fh = FS_FindFirst(DirName, FS_ATTR_DIR,
0, &file_info, filename,MAX_CUSTOM_FILENAME_LEN);
//如果系统已有该目录，就关闭先前的文件句柄
if(fh >= FS_NO_ERROR)
{
      FS_FindClose(fh);
      return fh;
}
else
{
      //如果没找到目录则创建目录
      fh = FS_CreateDir(DirName);
      FS_FindClose(fh);
      return fh;
  }
}
//在屏幕函数中调用CreateFileDir
void EntryFirstAppScreen(void)
{
  FS_HANDLE                handleFind;
  FS_DOSDirEntryfile_info;
  U16                      filename[MAX_FILENAME_LEN];
  S32 i = 0;
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();
   //创建目录
  handleFind = CreateFileDir(L"E:\\FileTest");
if(handleFind >= 0)
  {
    //只要创建成功，就显示一个弹出窗口，显示成功
   DisplayPopup((PU8)GetString(STR_GLOBAL_OK), IMG_GLOBAL_OK,
                1, 1000, SUCCESS_TONE);
//显示完毕，返回到Organizer界面
DeleteUptoScrID(ORGANIZER_SCREENID);
  }
}
```

保存上述代码，编译并运行模拟器。在屏幕显示弹出窗口之后，即可在模拟器的文件系统中找到新创建的目录 FileTest。\plutommi\WIN32FS\DRIVE_E\FileTest 创建成功。

2. 创建新文件

在之前创建的目录中，搜索 customFile.txt 文件。

```
void EntryFirstAppScreen(void)
{
```

```
    FS_HANDLE                handleFind;
    FS_DOSDirEntry           file_info;
    U16                      filename[MAX_CUSTOM_FILENAME_LEN];
    S32 i = 0;
   EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
   entry_full_screen();
   clear_screen();

      //创建目录
      handleFind = CreateFileDir(L"E:\\FileTest");
  if(handleFind >= 0)
   {
handleFind = FS_Open(L"E:\\FileTest\\customFile.txt",FS_CREATE|FS_READ_WRITE);
   }
   else   //创建失败
   {
        return;
   }
   FS_Close(handleFind);//在不需要的时候，一定要将句柄关闭释放
  }
```

保存修改，编译并运行模拟器。目录\plutommi\WIN32FS\DRIVE_E\FileTest 下成功地生成了文件customFile.txt。

3. 写入数据

我们尝试写入一段学生数据信息到刚刚创建的文件中。

学生数据包括：姓名、年龄、性别、学号、总成绩等。所以，使用一个结构体来存储。

```
typedef struct
{
  U8   stuName[16];
  U8   stuAge;
  U8    stuSex[8];
  S32    stuNumber;
  S16    stuScore;//上一次保存时的记忆模式
} StudentData;
```

声明一个全局 StudentData 实例：

```
StudentData stuData;
```

定义一个函数，封装写操作：

```
void WriteStuData(U16 *FileName,void *DataPtr,S32 offSet,UINT DataSize)
{
      FS_HANDLE WriteHandle;
      UINT Leng=0;
      WriteHandle = FS_Open(FileName,FS_CREATE|FS_READ_WRITE);
      if (WriteHandle >=0)
      {
           FS_Seek(WriteHandle, offSet, FS_FILE_BEGIN);
           FS_Write(WriteHandle,DataPtr,DataSize,&Leng);//
      }
      else
      {
```

```
            DisplayPopup((PU8)GetString(STR_GLOBAL_ERROR),
                         IMG_GLOBAL_ACTIVATED,
                         1, 2000, ERROR_TONE);
            DeleteNHistory(0);
        }
        FS_Close (WriteHandle);
}
```

接下来，我们想实现在当前屏幕中按左键就存储数据的功能。用这样一句代码来实现：

```
SetLeftSoftkeyFunction(WriteStuData, KEY_EVENT_UP);
```

表面上看起来似乎可行，但是实际运行中，按左键存储的时候，系统会出现异常。

这是由于函数 SetLeftSoftkeyFunction 的第一个参数是一个如下形式的函数指针：

```
void SetLeftSoftkeyFunction(void (*f) (void), MMI_key_event_type k)
```

可以看到函数指针 void (*f) (void)已经明确了传入的函数的形式，即返回值和参数都为 void 的函数。

所以，我们必须再封装一次：

```
void SaveData(void)
{
  WriteStuData(L"E:\\FileTest\\customFile.txt",&stuData,0,sizeof(StudentData));
}
```

请一定要将上面代码中的 stuData 声明为全局变量。

现在，我们就可以在按左键的时候正确调用写数据的函数了，最终代码如下：

```
typedef struct
{
  U8    stuName[16];
  U8    stuAge;
  U8    stuSex[8];
  S16    stuNumber;
  S16    stuScore;
} StudentData;

StudentData stuData; //声明一个全局实例

void WriteStuData(U16 *FileName,void *DataPtr,S32 offSet,UINT DataSize)
{
  FS_HANDLE WriteHandle;
UINT Leng=0;
WriteHandle = FS_Open(FileName,FS_CREATE|FS_READ_WRITE);
if (WriteHandle >=0)
{
      FS_Seek(WriteHandle, offSet, FS_FILE_BEGIN);
      FS_Write(WriteHandle,DataPtr,DataSize,&Leng);//
   }
   else
   {
       DisplayPopup((PU8)GetString(STR_GLOBAL_ERROR),
                        IMG_GLOBAL_ACTIVATED,
                        1, 2000, ERROR_TONE);
      DeleteNHistory(0);
   }
   FS_Close (WriteHandle);
}
void SaveData(void)
```

```
{
    WriteStuData(L"E:\\FileTest\\customFile.txt",&stuData,0,sizeof(StudentData));
}
void EntryFirstAppScreen(void)
{
    FS_HANDLE              handleFind;
    FS_DOSDirEntry    file_info;
    U16                    filename[MAX_CUSTOM_FILENAME_LEN];
    S32 i = 0;

   EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
   entry_full_screen();
   clear_screen();
   //测试用的数据
   strcpy(stuData.stuName , "chenzhipeng");
   stuData.stuAge = 25;
   strcpy(stuData.stuSex , "male");
   stuData.stuNumber = 10115;
   stuData.stuScore = 568;
   //创建目录
   handleFind = CreateFileDir(L"E:\\FileTest");
   if(handleFind >= 0)
   {
      handleFind = FS_Open(L"E:\\FileTest\\customFile.txt",FS_CREATE|FS_READ_WRITE);
   }
   else
   {
      return;
   }
   FS_Close(handleFind);
   SetLeftSoftkeyFunction(SaveData, KEY_EVENT_UP);
   gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
   SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

保存修改，编译并运行模拟器，进入屏幕，按左键后，即可在目录 \plutommi\WIN32FS\DRIVE_E\FileTest 中看到文件 customFile.txt 已被写入了内容。

提　示

在对一个字符数组赋值时，一定要使用 strcpy。上面代码中的 stuName 是一个含有 16 个 char 字符的数组，如果使用如下方式进行赋值，将会产生严重错误。

stuData.stuName = “chenzhipeng”

因为数组名作为左值时，将被视为一个常量指针，而常量指针是不允许修改的。MTK 平台是基于 C 语言的，大家一定要注意这些特殊的语法规范。

4. 读取数据

要读取数据，首先得确定文件是否存在。

```
handleFind = FS_FindFirst(L"E:\\FileTest\\*.txt", 0, 0, &file_info,
                          (U16*)fileName, MAX_CUSTOM_FILENAME_LEN);
   FS_FindClose(handleFind);
```

上面的代码，是在目录 FileTest 下查找后缀名为.txt 的文件。如果找到就将文件名赋值给 fileName，

并将该文件信息赋值给 file_info。

只要是 handleFind 不小于 0，且文件名也和我们之前已创建的相同，就说明文件已经找到。接下来要做的就是读取。

清空 EntryFirstAppScreen 函数先前的代码，然后添加读代码。

```
void EntryFirstAppScreen(void)
{
FS_HANDLE            handleFind;
  FS_DOSDirEntry     file_info;
  U16 fileName[MAX_CUSTOM_FILENAME_LEN];
  S8  receStr[32];
  S8  ageOutput[8];
  S8  numOutput[32];
  S8  scoreOutput[32];
  S8  sexOutput[8];
  S32  i = 0;
  StudentData receStuData; //声明结构体实例用于存储读取的数据
EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
entry_full_screen();
clear_screen();
gui_set_text_color((gui_color)(255,0,0));
handleFind = FS_FindFirst(L"E:\\FileTest\\*.txt", 0, 0, &file_info,
                          (U16*)fileName, MAX_CUSTOM_FILENAME_LEN);
FS_FindClose(handleFind);
if(handleFind >= 0)
{
   handleFind = FS_Open(L"E:\\FileTest\\customFile.txt",FS_READ_ONLY);
   If(handleFind>=0)
   {
      //从文件的头开始读
      FS_Seek(handleFind, 0, FS_FILE_BEGIN);
      //读取的数据放在 receStuData 中
      FS_Read(handleFind,&receStuData,sizeof(StudentData),NULL);
      //完毕后，关闭文件句柄
      FS_Close(handleFind);
   }

}
//将读取的数据显示在屏幕上
gui_move_text_cursor(50,100);
AnsiiToUnicodeString(receStr, receStuData.stuName);
gui_print_text((UI_string_type)receStr);

gui_move_text_cursor(50,120);
gui_itoa(receStuData.stuAge,(UI_string_type)ageOutput,10);
gui_print_text(ageOutput);

 gui_move_text_cursor(50,140);
 AnsiiToUnicodeString(sexOutput, receStuData.stuSex);
 gui_print_text(sexOutput);

gui_move_text_cursor(50,160);
gui_itoa(receStuData.stuNumber,(UI_string_type)numOutput,10);
gui_print_text(numOutput);
```

```
gui_move_text_cursor(50,180);
gui_itoa(receStuData.stuScore,scoreOutput,10);
gui_print_text(scoreOutput);

 SetLeftSoftkeyFunction(SaveData, KEY_EVENT_UP);
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

保存上述修改，运行模拟器，即可看到之前我们写入文件的数据被正确地读取出来，如图 9.4 所示。

图 9.4　输出结果

结语：

在本章中，我们列举了一些常用的文件操作函数。并通过一个实例，详细地叙述了如何创建文件夹，如何写入数据，如何读取数据的方法。通过本章的学习，读者可以掌握基本的文件操作方法，更深层的功能还需要自己多探索。

10 控件

在对MTK控件进行讲解之前，先了解一下什么是控件。

控件，是图形用户界面中（如屏幕上）的一种对象，用户可操作该对象来执行某一行为。控件是用户可与系统交互以输入或操作数据的对象。在Windows编程中，对话框、复选框、单选框、按钮等都属于控件的范畴。

MTK的GUI仍然延续这一传统：有基于GUI层的控件，也有预先定义好的控件模板和后期直接使用的高级WGUI控件。

10.1 GUI基本控件

GUI层的控件使用一般分为3个步骤：创建、设置和显示。

· 创建

创建一个该控件的结构体对象。MTK系统原始代码基本上都是声明一个全局对象。

由于MTK的动态内存管理做得不是很好，另外也有可能是为了减少程序复杂度，因此，用全局变量很方便。

系统会为每个GUI控件提供一个形如gui_create_<ControlName>（）的函数，用于初始化。假如我们需要使用一个按钮（Button），那么在声明一个全局结构体对象后，就将该对象作为参数传入如下形式的函数：

```
void gui_create_button(button *b, S32 x, S32 y, S32 width, S32 height)
```

先前声明的全局对象地址就将替换指针b，然后，在该函数内部进行按钮的初始化工作。函数内部代码如下：

```
void gui_create_button(button *b, S32 x, S32 y, S32 width, S32 height)
{
    b->x = x;                          /* start x position of button */
    b->y = y;                          /* start y position of button */
    b->width = width;                   /* width of button */
    b->height = height;                 /* height of button */
    b->flags = 0;
```

```
    b->clear_background_fp = NULL;
    gui_set_button_current_theme(b);    /* set the theme of button to current theme */
}
```

可以看到，内部的初始化其实很简单，一般都是对位置和大小的设置。

· 设置

控件的设置既可以发生在创建之初，如上述代码所示；也可以是在使用过程中，比如想调整控件的位置和大小，想更换控件内部元素的内容和坐标。

假设：在某种情况下，需要调整 Button 主题。对应如下形式的函数：

```
gui_set_button_theme
```

假设我们现在要调整按钮内部元素显示的位置，就可以使用如下函数：

```
gui_text_button_set_text_position
```

· 显示

初始化和设置工作完成后，就需要显示在屏幕上，负责显示的函数调用很简单。

如果显示我们之前创建的 Button，则调用方式如下：

```
void gui_show_button(button *b)
```

其他类型的控件，如列表、复选框等都是类似的形式。限于篇幅，这里就不再一一列举。接下来，我们将通过几个控件实例来详细讲解。在编写实例之前，我们先将之前的屏幕入口函数清空，以方便后续的代码编写。

```
void EntryFirstAppScreen(void)
{
    /*清空代码*/
}
```

10.1.1 按钮（Button）

首先声明一个全局变量 Button。

```
Button button;
```

在屏幕函数内部调用创建按钮函数，在坐标（50, 50）位置，创建一个宽度为 80，高度为 40 的按钮，并设置一个主题，然后显示出来。

```
void EntryFirstAppScreen(void)
{
EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();
  gui_create_button(&b, 50, 50,80, 40);
  gui_show_button(&b);
gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

保存修改，编译并运行模拟器，即可看到一个按钮位于屏幕之上，效果如图 10.1 所示。

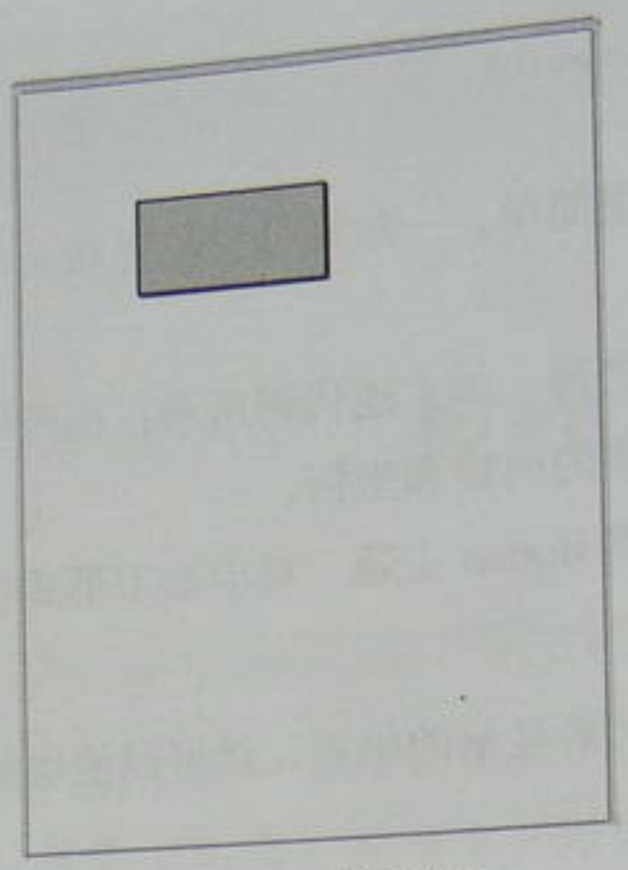


图 10.1 基本按钮

10.1.2 文本按钮（TextButton）

同样的道理，先声明一个全局的文本按钮变量：

```
text_button  text_b;
```

然后，在屏幕绘制函数中进行创建和显示。效果如图 10.2 所示。

```
void EntryFirstAppScreen(void)
{
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();
  //将字符串传入创建函数，作为按钮上的文本
  gui_create_text_button(&text_b, 50, 50,80, 40,
                         (UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK));
  gui_show_text_button(&text_b);
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

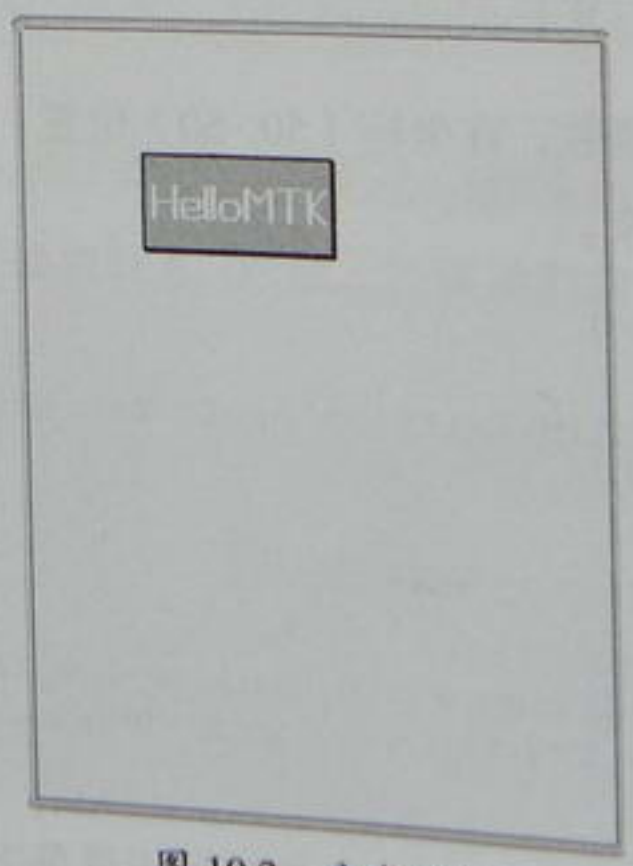


图 10.2 文本按钮

10.1.3 图标按钮（IconButton）

通过如下代码可以绘制如图 10.3 所示图标按钮。

```
icon_button   icon_b;
void EntryFirstAppScreen(void)
{
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();
 gui_create_icon_button(&icon_b, 50, 50,80, 40,get_image(IMG_FIRSTAPP));
 gui_show_icon_button(&icon_b);
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

图标按钮还可以调用函数 gui_resize_icon_button 对按钮进行放大或缩小操作，如图 10.4 所示。下面的程序通过左按键来完成 button 的尺寸修改。

首先定义一个函数 resize_button 来封装 gui_resize_icon_button。

```
icon_button icon_b;
void resize_button(void)
{
gui_resize_icon_button(&icon_b,150,100);
gui_show_icon_button(&icon_b);
gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
}
void EntryFirstAppScreen(void)
{
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();
 gui_create_icon_button(&icon_b, 50, 50,80, 40,get_image(IMG_FIRSTAPP));
 gui_show_icon_button(&icon_b);
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
 SetLeftSoftkeyFunction(resize_button,KEY_EVENT_UP);
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

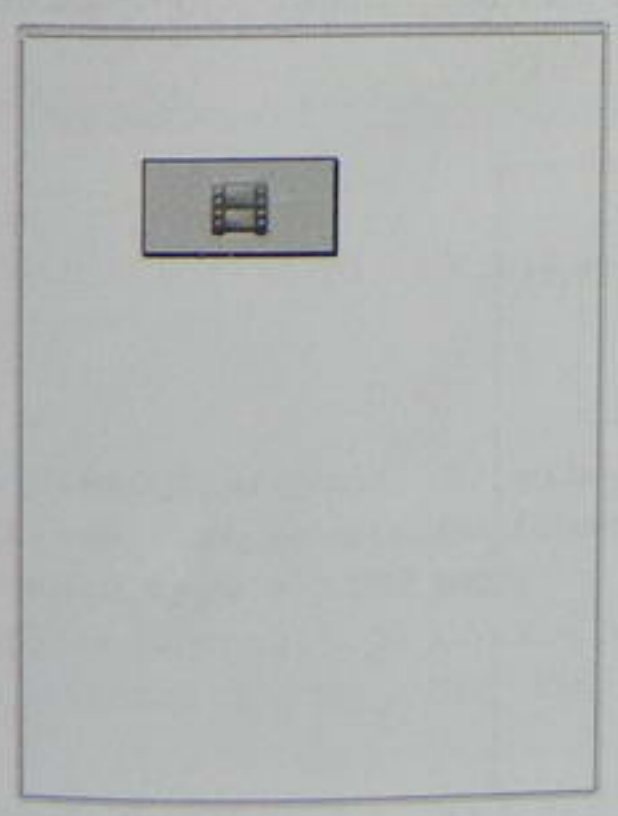


图 10.3　图标按钮

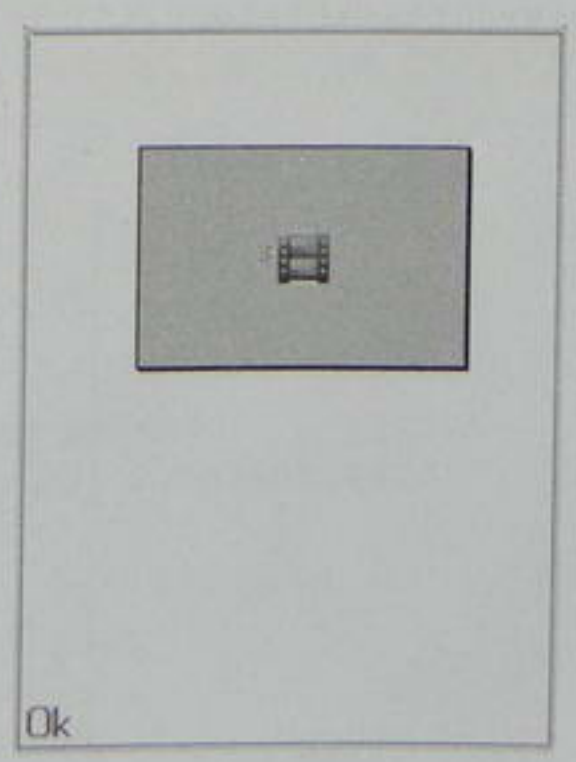


图 10.4　放大按钮

为什么 SetLeftSoftkeyFunction 不直接调用 gui_resize_icon_button 而且还要对其进行封装？

我们首先来看一下函数 SetLeftSoftkeyFunction 的原型：

```
void SetLeftSoftkeyFunction(void (*f) (void), MMI_key_event_type k)
```

该函数的第一个参数是一个函数指针 f，返回值为 void，参数为 void。这就表明，作为实参传入的函数也必须是这种类型。而函数 gui_resize_icon_button 的类型完全不同。因此，必须通过对其进行封装，才能让函数 SetLeftSoftkeyFunction 正常调用。

10.1.4 单行输入框

如图 10.5 所示的单行输入框和 Button 的创建方式差不多。

首先，创建一个全局变量 single_line_input_box。

```
#include "gui_inputs.h"
single_line_input_box input_box;
```

然后，定义一个回调函数用于输入字符。

```
void input_charactor(void)
{
    gui_single_line_input_box_insert_character(&input_box, 'a');
    gui_show_single_line_input_box(&input_box);
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);

}
```

上面的代码，按一次左键就会在输入框中输入一个字符“a”。

```
void EntryFirstAppScreen(void)
{
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();
 //创建单行输入框，1024 表示将要输入内容的大小（字节）
 gui_create_single_line_input_box(&input_box,50,50,150,80,1024);
 gui_show_single_line_input_box(&input_box);
 SetLeftSoftkeyFunction(input_charactor, KEY_EVENT_UP);
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

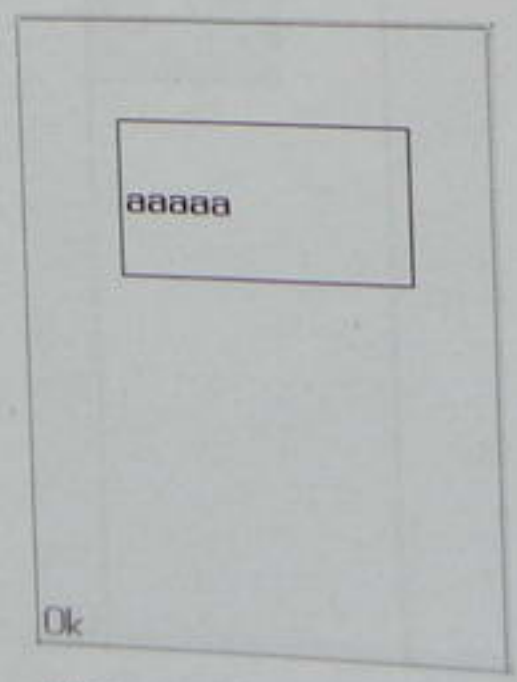


图 10.5 单行输入框示意

多行输入框的原理和单行的没有太大的区别，留给读者自己实践。

10.1.5 提示框

提示框的使用很简单，只需要传递相关的文本和图片 ID 即可。比如，在一个函数中，发生了某个错误，就需要通知用户。那么调用下面的语句即可。

```
DisplayPopup((PU8)GetString(STR_GLOBAL_ERROR), IMG_GLOBAL_ACTIVATED,
            1, 2000, ERROR_TONE);
```

其中，2000 表示弹出框持续的时间，单位为毫秒，这里就表示弹出框持续 2 秒钟。

通常情况下，如果一个弹出框用于提示错误信息，那么在其消失的时候，当前屏幕就应该回到上一个界面之中。看起来很复杂，其实只需要在 DisplayPopup 函数后面增加一句：

```
DeleteNHistory(1);
```

加上此句代码就可以实现上述功能。DeleteNHistory 用于回到某个界面，因为之前我们已经讲过，系统在进入新屏幕之前会将前一个屏幕压入栈，而弹出框并不是全屏占有的，只是附属于当前出错的屏幕，因此将数字 1 作为参数传入，即表示回到上一层。

10.1.6 文本图标列表控件

前面讲到的一些控件，只是一些很基础的，在实际项目中并不会经常用到。而现在我们要讲的列表控件，才是控件之中最常用和相对复杂的。

列表其实是由若干个菜单项和列表框架组成。因此，创建一个图标列表控件就要做两个工作：

一是创建一个能容纳多个菜单项的列表框架（Fixed List Menu）；

二是创建若干个菜单项（Fixed List MenuItem）。

1. 列表框架

首先声明一个列表框架全局变量。

```
fixed_list_menu  new_fixed_list_menu;
```

然后，在函数内部初始化该列表框架。

```
void EntryFirstAppScreen(void)
{
  S8 i;
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();

  memset(&new_fixed_list_menu, 0, sizeof(fixed_list_menu));
  gui_create_fixed_list_menu(&new_fixed_list_menu, 20, 20, 200 100);
  MMI_current_menu_type = LIST_MENU;
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

首先调用内存初始化函数 memset 对全局变量 fixed_listed_menu 进行初始化。然后，调用列表创建

函数创建一个宽度为 200，高度为 100 的列表框架。接着将当前列表类型设置为 LIST_MENU。不过该步骤似乎多余，但不加的话会有异常。

上面的工作只是一个初步阶段。还无法显示在屏幕上。因为，还没有对菜单项的属性进行设置。MTK 提供了一个结构体 fixed_icontext_menuitem，用于菜单项的公共属性设置。

fixed_icontext_menuitem 结构体的具体声明如下：

```
typedef struct _fixed_icontext_menuitem
    {
        S32 width, height;
        UI_filled_area *normal_filler;
        UI_filled_area *disabled_filler;
        UI_filled_area *selected_filler;
        UI_filled_area *focussed_filler;
        U32 flags;
        U32 ext_flags;
        S32 text_x, text_y;
        color normal_text_color;
        color disabled_text_color;
        color selected_text_color;
        color focussed_text_color;
        UI_font_type text_font;
        S32 icon_x, icon_y;
        UI_transparent_color_type icon_transparent_color;
        S32 scroll_x, scroll_width;
        S32 text_width;
        fixed_list_menu *parent_list;
        fixed_matrix_menu *parent_matrix;
    } fixed_icontext_menuitem;
```

可以看到，结构体内部包含了大量的菜单项属性（包括宽、高、背景、需要显示的文本等）。

2. 设置菜单项的公共属性

定义一个全局变量：

```
fixed_icontext_menuitem  new_fixed_icontext_menuitem;
```

然后，对结构体对象进行具体的赋值。

```
//设置每一个菜单项的宽度和高度
gui_create_fixed_icontext_menuitem(&new_fixed_icontext_menuitem, 200, 18);
//设置菜单项里面文本的起始位置，坐标是相对于菜单项左上角的
gui_fixed_icontext_menuitem_set_text_position( &new_fixed_icontext_menuitem, 50, 0);
//关闭菜单项背景
new_fixed_icontext_menuitem.flags |= UI_MENUITEM_DISABLE_BACKGROUND;
```

3. 设置菜单项具体内容

我们创建 5 个菜单项，并将它们放入先前的列表框架之中。

首先宏定义列表中菜单项个数。

```
#define new_fixed_list_n_items  5
//声明一个文本图标列表菜单项数组
```

```
fixed_icontext_menuitem_type new_fixed_list_menuitems[new_fixed_list_n_items];
//声明一个void*指针数组，用于指向文本图标列表菜单项
void *new_fixed_menuitem_pointers[new_fixed_list_n_items];
```

然后对数组循环赋值，为了方便，在这里我们将所有菜单项的文本都置为“HelloMTK!”，图标也使用系统已有的。

```
memset(&new_fixed_list_menuitems, 0,
       sizeof(fixed_icontext_menuitem_type)
       * new_fixed_list_n_items);

for (i = 0; i < new_fixed_list_n_items; i++)
{
//菜单项文本内容
new_fixed_list_menuitems[i].item_text =
                                    (UI_string_type)GetString(STR_FIRSTAPP_HELLOMTK);
//菜单项图标
new_fixed_list_menuitems[i].item_icon = (PU8) GetImage(IMG_GLOBAL_L1+i );
//标志位使得文本居中对齐
new_fixed_list_menuitems[i].flags = (UI_MENUITEM_CENTER_TEXT_Y
                                    | UI_MENUITEM_CENTER_ICON_Y);
new_fixed_list_menuitems[i].item_icon_handle = GDI_ERROR_HANDLE;
// new_fixed_menuitem_pointers 为指向菜单项数组
new_fixed_menuitem_pointers[i] = (void*)&new_fixed_list_menuitems[i];
}
```

说　明

由于列表的内部属性值 items 的类型为 void**，因此，在上面的代码中，我们就将一个指针数组指向原始的列表菜单项数组，以使后面的代码 new_fixed_list_menu.items 能够正确被赋值。

4. 绑定菜单项的内容和列表框架

```
//菜单项索引列表，这里就解决上述代码中有关指针数组的疑惑
new_fixed_list_menu.items = new_fixed_menuitem_pointers;
//将菜单项的属性传递给列表框架
new_fixed_list_menu.common_item_data = (void*)&new_fixed_icontext_menuitem;
//设置菜单项个数
new_fixed_list_menu.n_items = new_fixed_list_n_items;
//设置每个菜单项的行为
gui_set_fixed_list_menu_item_functions(&new_fixed_list_menu,
                                       //显示菜单项函数
                                       gui_show_fixed_icontext_menuitem,
                                       //测量菜单项函数
                                       gui_measure_fixed_icontext_menuitem,
                                       //高亮函数
                                       gui_highlight_fixed_icontext_menuitem,
                                       //失去高亮函数
                                       gui_remove_highlight_fixed_icontext_menuitem,
                                       //菜单项隐藏函数
                                       gui_hide_fixed_icontext_menuitem,
```

```
                                    //菜单项重设大小函数
                                    NULL
);
```

5. 显示列表

```
//菜单高亮项索引，必须初始化为-1
new_fixed_list_menu.highlighted_item = -1;
//默认高亮的菜单项是 0 号
gui_fixed_list_menu_goto_item(&new_fixed_list_menu, 0);
//显示整个列表
gui_show_fixed_list_menu(&new_fixed_list_menu);
```

为了使演示的效果更好，我们使用系统中已有的字符串 ID：STR_GLOBAL_DIAL。最终的代码如下：

```
fixed_list_menu          new_fixed_list_menu;
fixed_icontext_menuitem  new_fixed_icontext_menuitem;
#define new_fixed_list_n_items (5)
fixed_icontext_menuitem_type new_fixed_list_menuitems[new_fixed_list_n_items];
void *new_fixed_menuitem_pointers[new_fixed_list_n_items];
void EntryFirstAppScreen(void)
{
  S8 i;
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();
  memset(&new_fixed_list_menu, 0, sizeof(fixed_list_menu));
  gui_create_fixed_list_menu(&new_fixed_list_menu, 20, 20, 200, 100);
  MMI_current_menu_type = LIST_MENU;

  gui_create_fixed_icontext_menuitem(&new_fixed_icontext_menuitem, 200, 18);
  gui_fixed_icontext_menuitem_set_text_position( &new_fixed_icontext_menuitem, 50,8);
  new_fixed_icontext_menuitem.flags |= UI_MENUITEM_DISABLE_BACKGROUND;
  memset(&new_fixed_list_menuitems, 0,
        sizeof(fixed_icontext_menuitem_type) * new_fixed_list_n_items);
  for (i = 0; i < new_fixed_list_n_items; i++)
  {
  new_fixed_list_menuitems[i].item_text =
                                    (UI_string_type)GetString(STR_GLOBAL_DIAL+i);
  new_fixed_list_menuitems[i].item_icon = (PU8) GetImage(IMG_GLOBAL_L1+i );
  new_fixed_list_menuitems[i].flags = (UI_MENUITEM_CENTER_TEXT_Y
  |                                   UI_MENUITEM_CENTER_ICON_Y);
  new_fixed_list_menuitems[i].item_icon_handle = GDI_ERROR_HANDLE;
  new_fixed_menuitem_pointers[i] = (void*)&new_fixed_list_menuitems[i];
  }

  new_fixed_list_menu.items = new_fixed_menuitem_pointers;
  new_fixed_list_menu.common_item_data = (void*)&new_fixed_icontext_menuitem;
  new_fixed_list_menu.n_items = new_fixed_list_n_items;
  gui_set_fixed_list_menu_item_functions(&new_fixed_list_menu,
                                    gui_show_fixed_icontext_menuitem,
                                    gui_measure_fixed_icontext_menuitem,
                                    gui_highlight_fixed_icontext_menuitem,
                                    gui_remove_highlight_fixed_icontext_menuitem,
                                    gui_hide_fixed_icontext_menuitem,
                                    NULL
                                    );
```

```
new_fixed_list_menu.highlighted_item = -1;
gui_fixed_list_menu_goto_item(&new_fixed_list_menu, 0);
gui_show_fixed_list_menu(&new_fixed_list_menu);
gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

上面代码中，我们使用系统自带的字符串，并通过循环方式显示连续的 5 个文本。

```
new_fixed_list_menuitems[i].item_text = (UI_string_type)GetString(STR_GLOBAL_DIAL+i);
```

保存修改，编译并运行模拟器，效果如图 10.6 所示。

还有一种模板实现方式，将在 WGUI 高级控件中详细讲解。

目前，显示是正常了，可是我们还无法通过按键和触摸笔进行菜单项的选择和切换（按方向键或者数字键无反应）。这是由于菜单项没有和键盘进行绑定，我们将在下一节中继续完善。

6. 键盘响应

键盘是用户与手机系统进行交互的重要组成部分。用户的任何请求，从根源上讲都是通过键盘来完成的。图 10.7 所示是 MTK 手机的标准键盘。

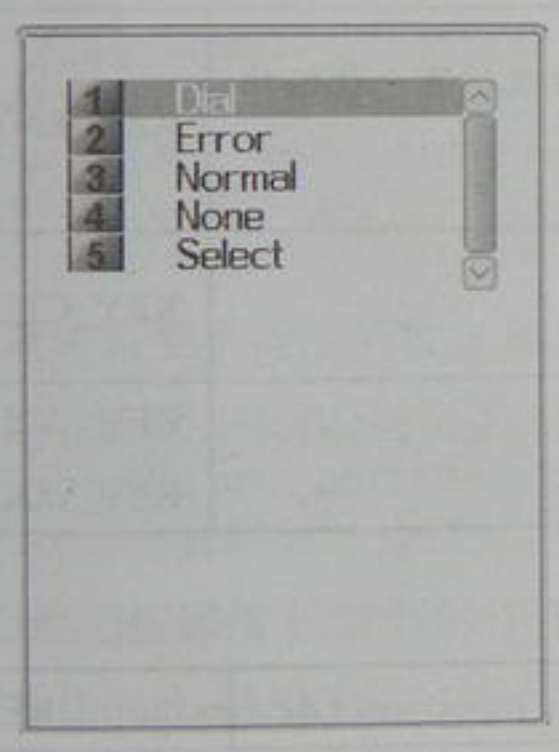


图 10.6　文本列表

图 10.7　标准键盘

MTK 系统内部会给每种功能的按键指定一个代码，不管键盘外观及布局如何发生变化，只要按键功能不变，其按键代码也就不必改变。比如 OK 键，一般都用于确定功能。因此，其按键代码一直都是 KEY_ENTER。

表 10.1 所示就是各个按键对应的按键代码及功能描述。表 10.2 所示是按键操作方式及效果描述。

表 10.1　按键代码及描述

按 键 名	描　　述	代码（KEY CODE）
数字键 0～9	最原始的数字键盘	KEY_0 — KEY_9
拨号键	主要有两个用途：一是打电话时用来拨号， 二是在待机界面时按此键进入通话记录列表	KEY_SEND

续表

按 键 名	描 述	代码（KEY CODE）
终止键	终止键主要有 3 个作用 （1）打电话时用来挂断 （2）绝大部分程序用此键来返回待机界面 （3）长按此键开关机	KEY_END
“*”键	功能键，具体作用依实际情况定	KEY_STAR
“#”键	功能键，有时用于输入内容的结束符	KEY_POUND
左右中软键	一般左软键用来进入，右软键用来退出，中软键一般只在触摸屏中会用到	KEY_LSK KEY_RSK KEY_CSK
上下左右方向键	方向按键	KEY_UP_ARROW KEY_DOWN_ARROW KEY_LEFT_ARROW KEY_RIGHT_ARROW
确认（OK）键	通常情况下，确认键和左软键绑定，这样就与左软键有一样的功能，用于确认某一个操作	KEY_ENTER
清除键	编辑界面时此键用来删除字符 （相当于 PC 键盘的退格键）	KEY_CLEAR
上下音量键	用来调节音量，一般放在手机左侧面	KEY_VOL_UP KEY_VOL_DOWN

表 10.2 按键操作方式及效果

操 作 方 式	描 述	KEY TYPE
按下	将按键按下	KEY_EVENT_DOWN KEY_FULL_PRESS_DOWN
放开	放开按键	KEY_EVENT_UP
长按	按下按键后不动，一定时间后将会触发长按事件，一般情况下是 1.5 秒后，可参考 KPD_LONGPRESS_PERIOD	KEY_LONG_PRESS
重复	长按事件触发以后，系统将会以一定频率不断触发“重复”事件，一般是间隔 0.5 秒，可参考 KPD_REPEAT_PERIOD。长按事件只会触发一次，重复事件将会一直持续下去，直到放开按键	KEY_REPEAT
半按	将按键按下一半，只有特殊的按键支持此事件（比如经过特殊处理的拍照键），目前很少用到	KEY_HALF_PRESS_DOWN
半按放开	半按后放开按键	KEY_HALF_PRESS_UP

在应用开发中，最常见的就是如下两个函数：

```
void SetKeyHandler( FuncPtr funcPtr, U16 keyCode, U16 keyType );
void SetLeftSoftkeyFunction( void (*f)(void), MMI_key_event_type k );
```

（1）SetKeyHandler

主要作用就是将需要响应的函数的指针（funcPtr）传入，然后根据实际的按键执行这个函数。全局二维数组 currKeyFuncPtrs[keyCode][keyType]将保证需要执行的函数被正确执行。

（2）SetLeftSoftkeyFunction

该函数内部主要流程如下：

```
register_left_softkey_handler();
set_left_softkey_function(f, k);
SetInputboxLSKFunction(f);
```

① 调用 register_left_softkey_handler 函数，这个函数会调用 SetKeyHandler，将左键的按下和正常两种状态的实现函数 left_softkey_down/left_softkey_up 和按键方式（KEY_ EVENT_DOWN/KEY_EVENT_UP）进行绑定。

```
    …
    case MMI_LEFT_SOFTKEY:
          SetKeyHandler(left_softkey_down, KEY_LSK, KEY_EVENT_DOWN);
          SetKeyHandler(left_softkey_up, KEY_LSK, KEY_EVENT_UP);
          left_softkey_keyboard_handler_active = 1;
          break;
…
```

② 接下来，就将需要执行的函数 f 赋值给全局的二维数组 softkey_functions。

```
void (*softkey_functions[WGUI_MAX_SOFTKEYS]
              [WGUI_MAX_SOFTKEY_FUNCTIONS]) (void);
softkey_functions[key][k] = f;
```

这样，就成功地把按键按下/放开的作用函数与具体的动作关联起来了。

③ 最后调用 SetInputboxLSKFunction（f）函数将上述函数与特定的输入（如触摸笔）关联起来。

（3）以上两个函数的主要区别：SetLeftSoftkeyFunction 可以识别长按状态并且可以关联触摸笔操作等。

此外，比较常用的还有 ChangeLeftSoftkey/ChangeRightSoftkey 函数。

主要执行流程如下。

- 调用 change_left_softkey：设置左软键图标，文字。
- 调用 redraw_softkey：刷新左软键/右软键显示区域。

现在，就将上面讲到的函数增加到代码中。

（1）方向键

首先，需要编写两个上下方向键绑定的函数。

```
/*前一个菜单项*/
void new_fixed_list_previous_item(void)
{
    gui_lock_double_buffer();
```

```
    gui_fixed_list_menu_goto_previous_item(&new_fixed_list_menu);
    gui_show_fixed_list_menu(&new_fixed_list_menu);
    gui_unlock_double_buffer();
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
}
/*下一个菜单项*/
void new_fixed_list_next_item(void)
{
    gui_lock_double_buffer();
    gui_fixed_list_menu_goto_next_item(&new_fixed_list_menu);
    gui_show_fixed_list_menu(&new_fixed_list_menu);
    gui_unlock_double_buffer();
    //由于列表状态发生变化，所以需要重绘
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
}
```

现在，我们就需要将按键事件和菜单项绑定。

```
SetKeyHandler(new_fixed_list_previous_item, KEY_UP_ARROW, KEY_EVENT_DOWN);
SetKeyHandler(new_fixed_list_next_item, KEY_DOWN_ARROW, KEY_EVENT_DOWN);
```

最终调用代码如下：

```
void EntryFirstAppScreen(void)
{
  S8 i;
  EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
  entry_full_screen();
  clear_screen();
  …
  new_fixed_list_menu.highlighted_item = -1;
  gui_fixed_list_menu_goto_item(&new_fixed_list_menu, 0);
  gui_show_fixed_list_menu(&new_fixed_list_menu);
  SetKeyHandler(new_fixed_list_previous_item, KEY_UP_ARROW, KEY_EVENT_DOWN);
  SetKeyHandler(new_fixed_list_next_item, KEY_DOWN_ARROW, KEY_EVENT_DOWN);
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

编译并运行模拟器，进入列表所在的屏幕。现在按上下键，即可正常地切换。假如我们连续按下方向键 3 次，即高亮显示 Normal 菜单项，如下图 10.8 所示。

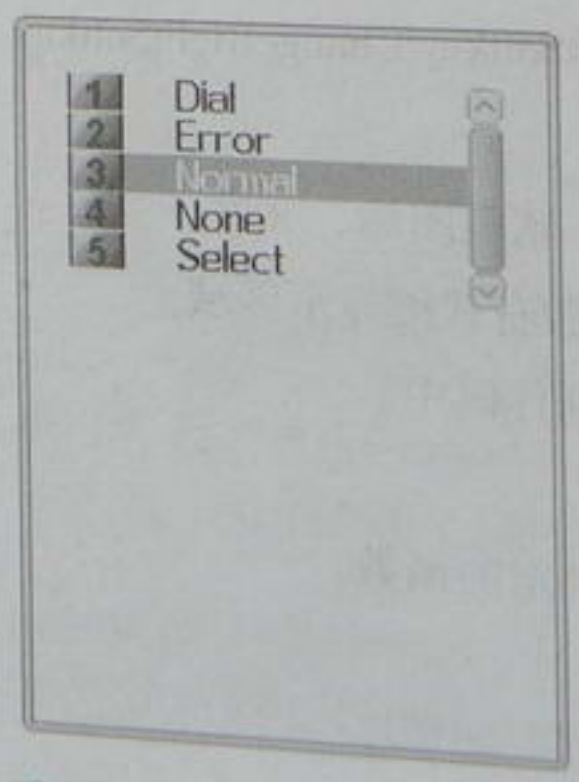


图 10.8 添加按键响应的列表

（2）组按键

同样，我们也可以通过按数字键进行切换。数字键总共 10 个，我们不可能像上面绑定方向键一样，一个个地绑定。这时，我们就需要使用群组按键。

```
SetGroupKeyHandler(FuncPtr funcPtr, PU16 keyCodes, U8 len, U16 keyType)
```

参数 funcPtr：按键对应的函数指针。

参数 keyCodes：按键代码列表。

参数 len：总共注册的按键数量。

参数 keyType：按键操作方式。

同样，我们定义一个函数来封装按键响应函数。

```
void new_fixed_list_goto_item(void)
{
U16 keycode, keytype;
S32 index = 0;
gui_lock_double_buffer();
GetkeyInfo(&keycode, &keytype); //获取当前用户操作的按键代码
index = keycode - KEY_1; //计算出当前按的是哪一个数字键
gui_fixed_list_menu_goto_item(&new_fixed_list_menu, index);
gui_show_fixed_list_menu(&new_fixed_list_menu);
gui_unlock_double_buffer();
gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
}
```

最终调用代码如下：

```
void EntryFirstAppScreen(void)
{
 S8 i;
 //因为只有 5 个列表项
 U16 shortcut_keys[new_fixed_list_n_items] =
 {
 KEY_1, KEY_2, KEY_3, KEY_4,KEY_5
 };
 EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
 entry_full_screen();
 clear_screen();
 …
 new_fixed_list_menu.highlighted_item = -1;
 gui_fixed_list_menu_goto_item(&new_fixed_list_menu, 0);
 gui_show_fixed_list_menu(&new_fixed_list_menu);
 SetGroupKeyHandler(new_fixed_list_goto_item, (PU16)shortcut_keys,
                    new_fixed_list_n_items, KEY_EVENT_UP);
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

运行模拟器后，进入列表所在的屏幕。这时侯数字键有反应了。如果我们按 5 键，列表就会高亮显示第 5 个菜单项，如下图 10.9 所示。

此外，我们还可以定义一个回调函数，用于在高亮状态的时候显示当前菜单项的文本内容。

```
//显示当前的文本内容
void firstapp_draw_text(S32 item_index)
{
    gui_move_text_cursor(20,20);
    gui_fill_rectangle(20, 20, UI_device_width - 1, 20+20, UI_COLOR_WHITE);
    gui_print_bordered_text((UI_string_type)GetString(STR_GLOBAL_DIAL + item_index));
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
}
//菜单项高亮显示时被系统调用的回调函数
void firstapp_highlight_handler(S32 item_index)
{
     firstapp_draw_text(item_index);
}
```

现在在 EntryFirstAppScreen 函数中注册高亮回调函数：

```
new_fixed_list_menu.item_highlighted = firstapp_highlight_handler;
```

item_highlighted 是一个带有 S32 参数的函数指针：void (*item_highlighted) (S32 i)。因此，我们定义的回调函数必须要和函数指针的参数相同。

现在，再次运行模拟器，在切换到列表菜单项的时候，就会在列表上方显示图 10.10 所示内容。

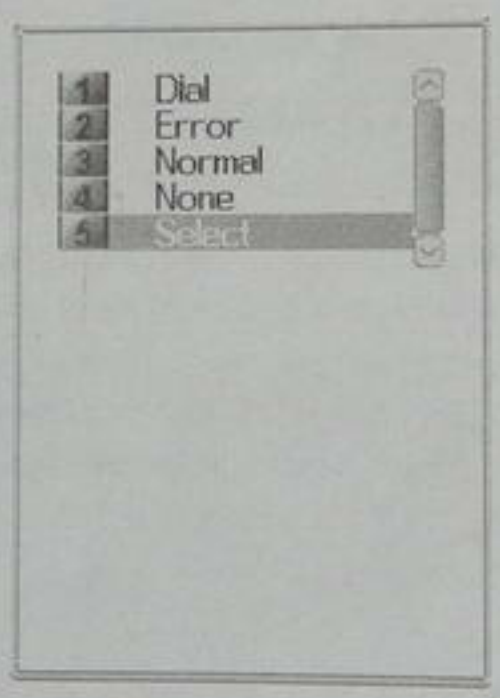


图 10.9　添加组按键的列表

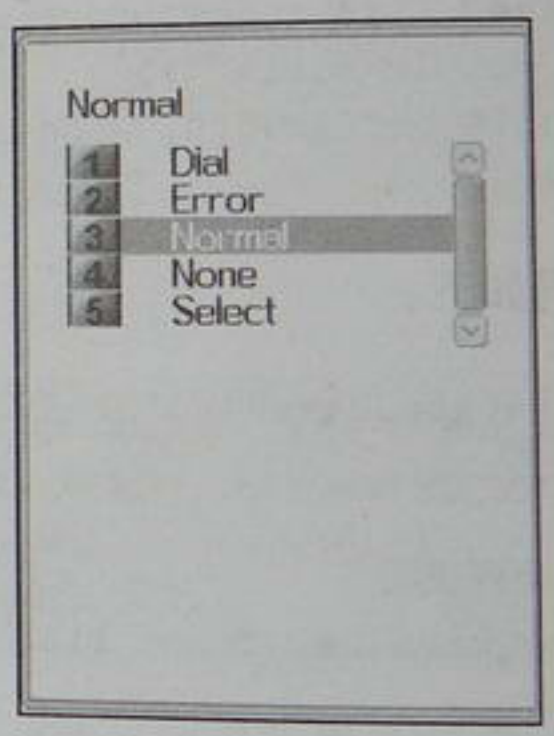


图 10.10　显示菜单项内容

7. 触摸屏响应

从程序开发的角度来讲，MTK 平台的触摸屏操作分为 3 个层次：硬件级、驱动级和 MMI 级。

硬件级：有触摸屏硬件产生，大部分情况下硬件都会产生一种操作方式，即触摸笔点击和放开。

驱动级：驱动收到硬件通知后，会根据前一次操作状态来决定当前状态是点下还是放开。同时，在触摸笔点下的瞬间，就启动定时器，然后根据一定的时间间隔来产生特殊的操作，如长按、重复等。

MMI 级：基本上与驱动操作一一对应。只不过在 MMI 的应用中提供了封装过的接口。

这里我们只关注 MMI 级的触摸屏操作，硬件和驱动级的触摸屏原理请读者参考相关的文档。

接下来我们为前面的列表增加触摸屏处理功能。

首先，定义一个触摸笔事件的响应函数。

```
void mmi_pen_list_menu_hdlr(mmi_pen_point_struct point,
                            mmi_pen_event_type_enum pen_event)
{
   gui_list_pen_enum  menu_event;
   gui_lock_double_buffer();
   if(!gui_fixed_list_menu_translate_pen_event(&new_fixed_list_menu, pen_event,
                                               point.x, point.y, &menu_event))
   {
      return; //如果触摸笔不在菜单显示区域内则不处理
   }
   //如果列表菜单项事件是以下 3 种情况
   if(menu_event == GUI_LIST_PEN_HIGHLIGHT_CHANGED ||
     menu_event==GUI_LIST_PEN_NEED_REDRAW||
     menu_event == GUI_LIST_PEN_ITEM_SELECTED)
   {
         //菜单处理后使菜单显示内容有了变化，我们需手动重画菜单
         gui_show_fixed_list_menu(&new_fixed_list_menu);
   }
   gui_unlock_double_buffer();
   gui_BLT_double_buffer( new_fixed_list_menu.x, new_fixed_list_menu.y,
                          new_fixed_list_menu.x + new_fixed_list_menu.width,
                          new_fixed_list_menu.y + new_fixed_list_menu.height);
}
```

触摸笔在单击某一个菜单项的时候触发事件 GUI_LIST_PEN_HIGHLIGHT_CHANGED。

说　明

在重绘控件的时候，可以只重绘控件所占据的区域，而不需要重绘整个屏幕。如上面函数中的 gui_BLT_double_buffer 只重绘了列表区域。推荐使用这种方法，可以提高程序执行效率。

接着，手动注册上述函数到每一个触摸笔事件。

```
void mmi_pen_up_hdlr(mmi_pen_point_struct point)
{
   mmi_pen_list_menu_hdlr(point, MMI_PEN_EVENT_UP);
}
void mmi_pen_down_hdlr(mmi_pen_point_struct point)
{
   mmi_pen_list_menu_hdlr(point, MMI_PEN_EVENT_DOWN);
}
void mmi_pen_move_hdlr(mmi_pen_point_struct point)
{
   mmi_pen_list_menu_hdlr(point, MMI_PEN_EVENT_MOVE);
}
void mmi_pen_repeat_hdlr(mmi_pen_point_struct point)
{
   mmi_pen_list_menu_hdlr(point, MMI_PEN_EVENT_REPEAT);
}
void mmi_pen_long_tap_hdlr(mmi_pen_point_struct point)
{
   mmi_pen_list_menu_hdlr(point, MMI_PEN_EVENT_LONG_TAP);
}
```

这样，触摸笔的所有动作都可以响应函数 mmi_pen_list_menu_hdlr，该函数根据当前触摸笔的位置和操作方式进行相应的处理。

然后，在调用的屏幕中，注册每个触摸笔事件。

```
void EntryFirstAppScreen(void)
{
    EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
    entry_full_screen();
    clear_screen();
    memset(&new_fixed_list_menu, 0, sizeof(fixed_list_menu));
    gui_create_fixed_list_menu(&new_fixed_list_menu, 20, 20, 200, 100);
    …
    gui_show_fixed_list_menu(&new_fixed_list_menu);
    mmi_pen_register_down_handler(mmi_pen_down_hdlr); //注册“点下”操作响应函数
    mmi_pen_register_up_handler(mmi_pen_up_hdlr); //注册“放开”操作响应函数
    mmi_pen_register_move_handler(mmi_pen_move_hdlr); //注册“移动”操作响应函数
    mmi_pen_register_repeat_handler(mmi_pen_repeat_hdlr); //注册“重复”操作响应函数
    mmi_pen_register_long_tap_handler(mmi_pen_long_tap_hdlr);// 注册“长按”操作响应
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);//重绘屏幕
    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

保存修改，编译运行模拟器，进入新增的屏幕。在出现的列表界面中，单击某个菜单项，即可将其高亮显示。

添加触摸响应的列表效果图如图 10.11 所示。

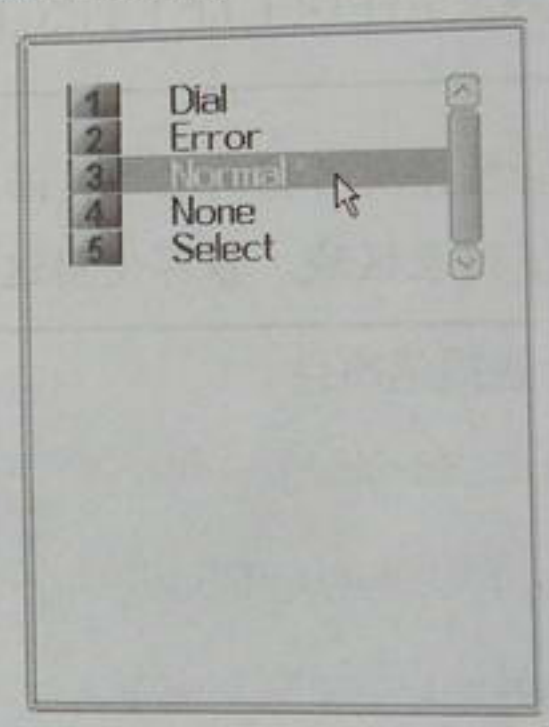


图 10.11　添加触摸响应的列表

为了能够更好地演示，读者可以按照前面章节的讲述，给被单击菜单项增加一个显示区域，显示的内容就是当前被单击高亮显示的菜单项文本。

触摸笔事件的实现其实很简单，读者通过对这一节中例子的学习，可以快速地掌握触摸笔更深层的原理知识和方法。我们这里只是做一个简单的指引。

10.2　WGUI 高级控件

上面介绍的控件都是基于 GUI 层的，相对来说比较简单，但是实现却相当繁琐。为了提高代码重用率和提高开发效率。MTK 还提供了一种封装过的高级控件：WGUI Control。

开发人员只需要在使用的时候将该实体重新初始化一遍即可。

10.2.1 文本图标列表

我们首先以文本图标列表菜单为例，采用 WGUI 提供的方式。

第 1 步：确定列表的位置和大小。

```
move_fixed_list(20, 50); //设置位置
resize_fixed_list(200, 200); //设置宽高
MMI_current_menu_type = LIST_MENU; //设置显示风格
```

说 明

大家可能会觉得疑惑，还没有创建初始化列表为什么就先设置位置和大小了？

前面讲过，WGUI 控件都是预先创建了一个实例，在使用的时候初始化即可。因此，其属性大多都是全局变量。

函数 move_fixed_list、resize_fixed_list 都是对列表的全局属性变量作修改。所以，这样的顺序没有任何问题。

读者可以查看上面两个函数的源码。

第 2 步：创建并初始化列表菜单项的通用属性。

```
create_fixed_icontext_menuitems();
```

第 3 步：关联列表和菜单项。

```
associate_fixed_icontext_list();
```

第 4 步：初始化每一个菜单项。

```
for (i = 0; i < new_fixed_list_n_items; i++)
  {
        add_fixed_icontext_item(get_string(STR_GLOBAL_DIAL+i),
                                (PU8)GetImage(IMG_GLOBAL_L1 + i));
  }
```

然后注册按键事件。

```
register_fixed_list_highlight_handler(firstapp_highlight_handler); //注册高亮接口
register_fixed_list_keys(); //注册方向键
register_fixed_list_shortcut_handler(); //注册数字键
fixed_list_goto_item_no_redraw(0); //先高亮第一项
show_fixed_list(); //显示菜单
```

最终代码如下：

```
void EntryFirstAppScreen(void)
{
    EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
    entry_full_screen();
    clear_screen();
    move_fixed_list(20, 50); //设置位置
    resize_fixed_list(200, 200); //设置宽高
    MMI_current_menu_type = LIST_MENU; //设置显示风格
    create_fixed_icontext_menuitems();
```

```
    associate_fixed_icontext_list();
    for (i = 0; i < new_fixed_list_n_items; i++)
    {
    add_fixed_icontext_item(get_string(STR_GLOBAL_DIAL+i),
                            (PU8)GetImage(IMG_GLOBAL_L1 + i));

}

  register_fixed_list_highlight_handler(firstapp_highlight_handler); //注册高亮接口
  register_fixed_list_keys(); //注册方向键
  register_fixed_list_shortcut_handler(); //注册数字键
  fixed_list_goto_item_no_redraw(0); //先高亮第一项
  show_fixed_list(); //显示菜单
 gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
                                    }
```

保存修改，编译并运行模拟器。效果和之前一样，如图 10.12 所示。

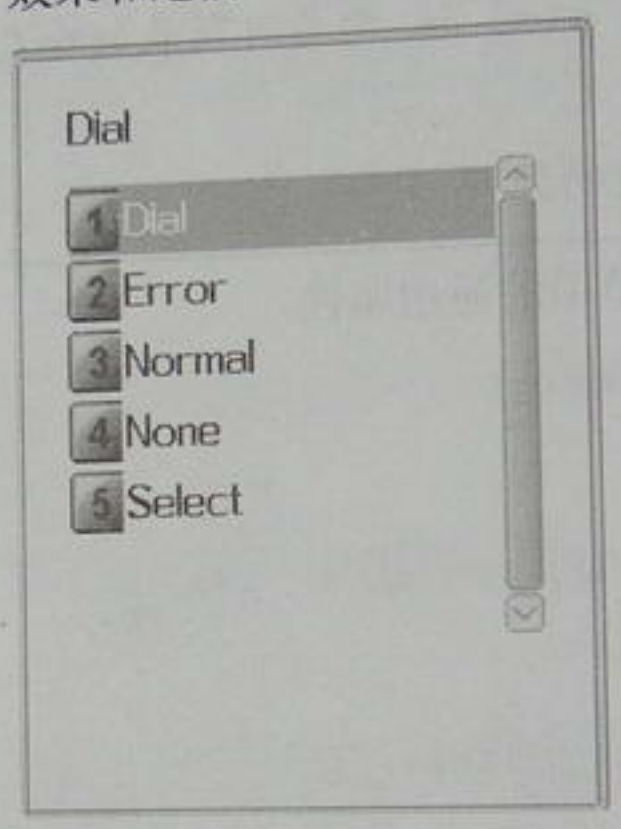


图 10.12　列表最终效果

10.2.2　动态列表控件

前一节讲到的文本图标列表的菜单项数目是固定的，但是在实际开发过程中，菜单项可能会随机动态地增加或减少，于是 MTK 提供了动态列表控件。

动态列表实际上包含了固定列表的特性，因此，其创建过程和固定列表无异。

```
create_fixed_icontext_menuitems();
MMI_current_menu_type=LIST_MENU;
associate_fixed_icontext_list();
```

关键是初始化动态列表。

函数 init_dynamic_item_buffer 用于动态列表的初始化工作，它接受 4 个参数。

参数 1 表示当前列表菜单项的个数。

参数 2 表示列表菜单项的具体内容（包括图标、文本）的回调函数。

参数 3 表示高亮显示某个菜单项时提示信息的回调函数。

参数 4 表示刷新缓存数据的回调函数。

例如：

```
init_dynamic_item_buffer(items, GetItemCallback, NULL, UI_dummy_function);
```

在实际调用过程中，系统就会根据变量 items 的值决定当前列表有多少个菜单项，并根据 items 调用回调函数设置各个菜单项的内容。由于高亮提示功能和刷新缓存很少用到，因此分别设为 NULL 和 UI_dummy_function.。

接下来，就注册各种按键事件。

```
register_menu_shortcut_keys();
```

基本流程就是以上 3 个部分。在这里，我们只给大家一个启蒙性的认识。由于动态列表最好是通过模板实现，因此具体实现细节我们放在后面的第 15 章（高仿 NOKIA N73）中进行介绍。

10.2.3 左右按键

左软键和右按键同样也预先定义了实例，使用的时候很方便。

```
void EntryFirstAppScreen(void)
{
…
//设置左软键文本及图标
change_left_softkey(STR_GLOBAL_OK, IMG_GLOBAL_OK);
//设置右软键文本及图标
change_right_softkey(STR_GLOBAL_BACK, IMG_GLOBAL_BACK);
//显示左软键
show_softkey(MMI_LEFT_SOFTKEY);
//显示右软键
show_softkey(MMI_RIGHT_SOFTKEY);
…
}
```

结语：

本章，我们为大家讲解了多种形式的控件实现方式，包括 GUI 的基本控件、文本图标列表控件和 WGUI 高级控件，并描述了触摸屏和按键等响应方式。读者学完本章，可以对整个 MTK 系统的控件绘制和事件处理有一个完整的认识。

11 屏幕模板

浏览 MTK 工程中的各个文件，读者可能经常见到或者已经使用过下面这种形式的函数：ShowCategoryXScreen。这种形式的函数在 MTK 系统庞大的源代码中扮演着重要的角色，它们都是基于屏幕模板的。为了提高开发效率，MTK 将许多类型相似的窗口升级为模板，在需要的时候传入特定的参数来进行重绘。

系统中的屏幕模板接口都类似于 ShowCategoryXScreen。

其中，X 表示数字，可以是 ShowCategory6Screen、ShowCategory33Screen 等。数字是随便定的，没有任何规则和意义，这可能是 MTK 设计之初没有做好的地方。

你可以在 wgui_categories.h 文件中看到大量类似屏幕模板函数的声明。这些都是 MTK 系统已经做好，提供给开发人员使用的。

本章，我们将自定义一个屏幕模板，并讲解模板的绘制原理。

11.1 系统提供的屏幕模板

MTK 系统提供了大量的屏幕模板，大多是可以通用的，也包含少数为实现特定功能而定义的屏幕模板。

典型的屏幕有以下几种。

绘制待机界面（如图 11.1 所示）的 ShowCategory33Screen 函数。

绘制矩形主菜单界面（如图 11.2 所示）的 ShowCategory14Screen 函数。

绘制文本列表菜单屏幕（如图 11.3 所示）的 ShowCategory6Screen 函数。

全屏弹出提示对话框（如图 11.4 所示）的 ShowCategory165Screen 函数。

绘制全屏多行文本输入框（如图 11.5 所示）的 ShowCategory5Screen 函数。

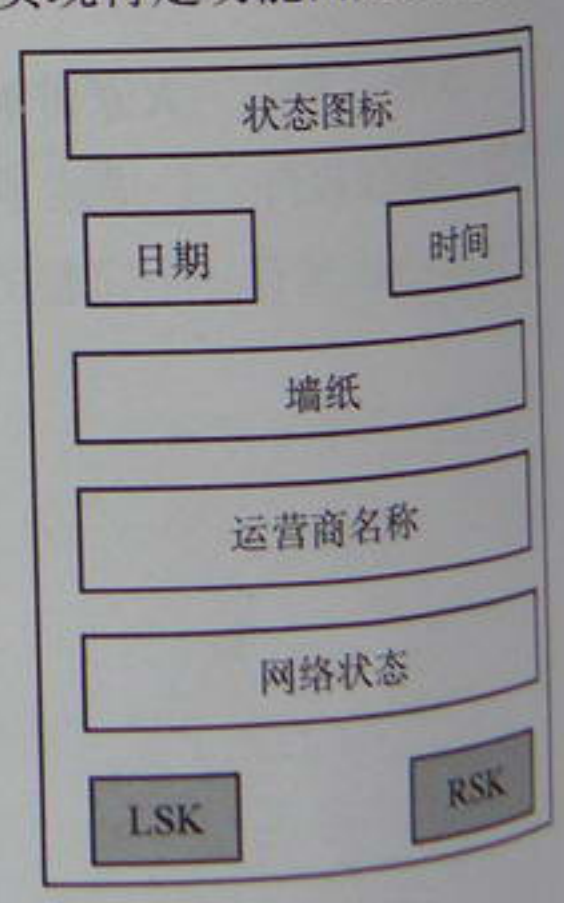


图 11.1 待机界面

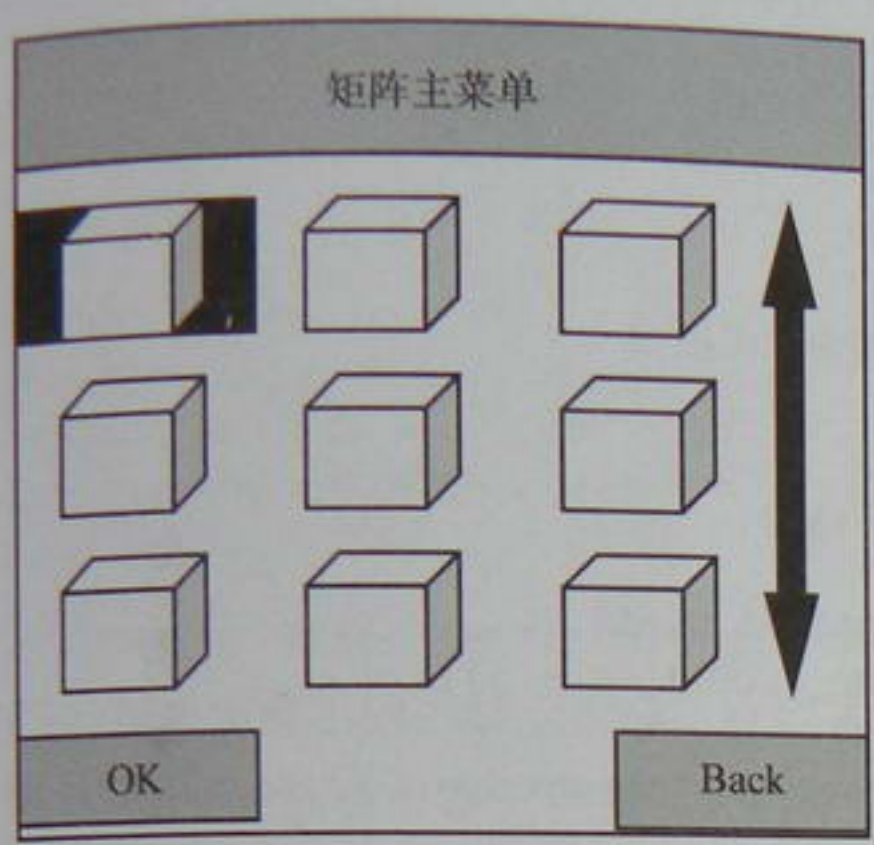


图 11.2 主菜单

图 11.3 文本列表

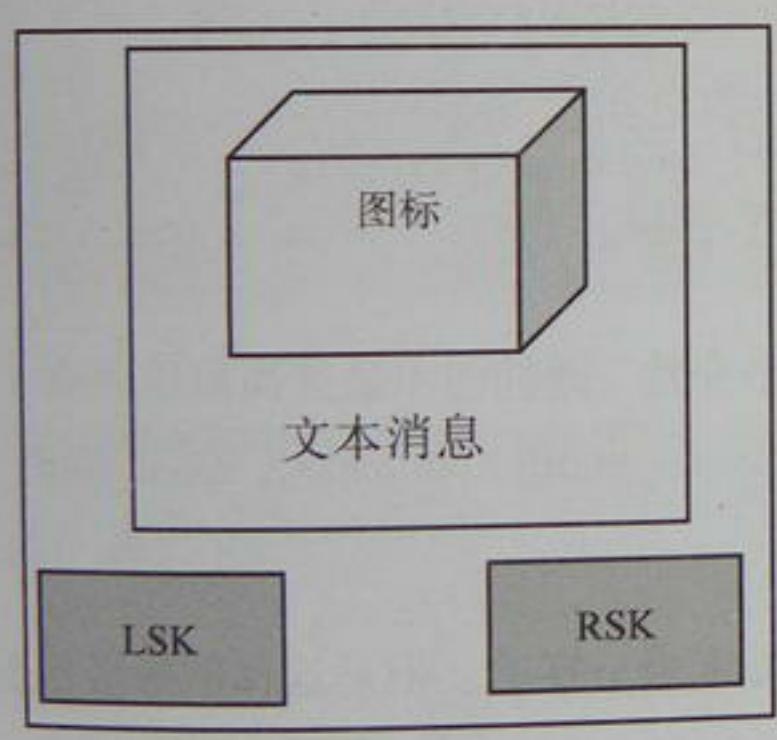


图 11.4 提示对话框

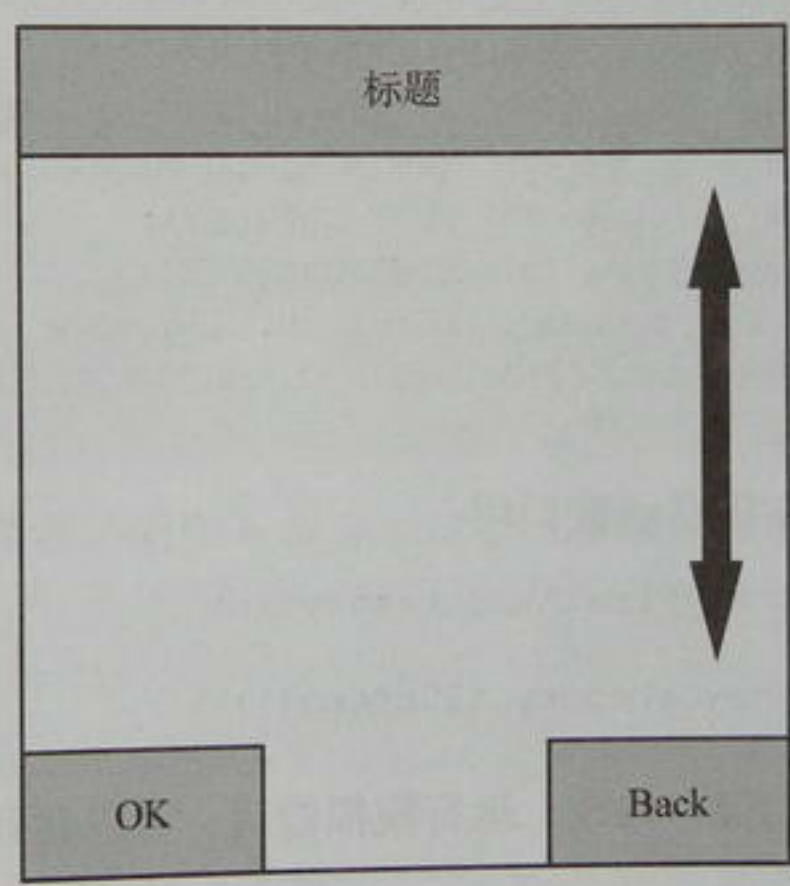


图 11.5 多行输入框

上面列举的只是几个典型的屏幕，限于篇幅，我们不可能将上百个屏幕模板一一列举出来。我们在接下来的章节里为读者讲解模板的构造原理，并让读者能够自行定义自己需要的模板。

11.2 自定义屏幕模板

本章之前，我们所讲的列表控件等都是按照当前需要临时绘制的，没有通用性。因此，可以将其中具有通用性的元素提取出来，形成模板。

11.2.1 定义新的屏幕模板函数

遵循 MTK 的命名规则，我们将新的屏幕命名为 ShowCategory668Screen。然后，将前面的文本图标列表绘制加入到新的屏幕函数之中。

```
void ShowCategory668Screen(void)
{
    S8 i;
    entry_full_screen();
    clear_screen();
    move_fixed_list(20, 50); //设置位置
    resize_fixed_list(200, 200); //设置宽高
    MMI_current_menu_type = LIST_MENU; //设置显示风格
    create_fixed_icontext_menuitems();
    associate_fixed_icontext_list();
    for (i = 0; i < new_fixed_list_n_items; i++)
    {
    add_fixed_icontext_item(get_string(STR_GLOBAL_DIAL+i),
                            (PU8)GetImage(IMG_GLOBAL_L1 + i));
    }
    register_fixed_list_highlight_handler(firstapp_highlight_handler); //注册高亮接口
    register_fixed_list_keys(); //注册方向键
    register_fixed_list_shortcut_handler(); //注册数字键
    fixed_list_goto_item_no_redraw(0); //先高亮显示第一项
    show_fixed_list(); //显示列表
    change_left_softkey(STR_GLOBAL_OK, IMG_GLOBAL_OK);
    change_right_softkey(STR_GLOBAL_BACK, IMG_GLOBAL_BACK);
    show_softkey(MMI_LEFT_SOFTKEY);
    show_softkey(MMI_RIGHT_SOFTKEY);
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
    SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

然后在屏幕函数调用：

```
void EntryFirstAppScreen(void)
{
    ShowCategory668Screen();
}
```

当我们保存修改，运行模拟器后，可以看到效果和原来是一样的。但是这样的屏幕模板没有任何意义。

11.2.2 提取变化的元素

经常变化的元素有：左右按键文本和图标，菜单项的内容，按键响应函数等。找到了这些元素，就可以将其提取出来作为模板屏幕函数的参数。

```
void ShowCategory668Screen(
U16 left_softkey, U16 left_softkey_icon, //左软键文本及图标
U16 right_softkey, U16 right_softkey_icon, //右软键文本及图标
S32 number_of_items, //菜单项项数
U16 *list_of_items, //菜单项文本列表
U16 *list_of_icons, //菜单项图标列表
S32 highlighted_item)//高亮项
{
for (i = 0; i < number_of_items; i++)
{
add_fixed_icontext_item(get_string(list_of_items[i]), (PU8)GetImage(list_of_icons[i]));
```

```
}
register_fixed_list_shortcut_handler();
register_fixed_list_keys();
register_fixed_list_highlight_handler(category668_highlight_handler);
fixed_list_goto_item_no_redraw(highlighted_item);
show_fixed_list();
//显示左右软键
change_left_softkey(left_softkey, left_softkey_icon);
change_right_softkey(right_softkey, right_softkey_icon);
show_softkey(MMI_LEFT_SOFTKEY);
show_softkey(MMI_RIGHT_SOFTKEY);
}
```

list_of_items 和 list_of_icons 用于存储文本 ID 和图标 ID。当需要使用该模板的时候，就可以灵活地传入实际的 ID 来实现不同的显示内容。这就是模板的一个重要特性。

接着，还需要重新修改高亮回调函数的实现方式。

```
void firstapp_draw_text(S32 item_index)
{
    gui_move_text_cursor(20,20);
    gui_fill_rectangle(20, 20, UI_device_width - 1, 20+20, UI_COLOR_WHITE);
    gui_print_bordered_text(MMI_fixed_icontext_menuitems[item_index].item_text);
    gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
}
void category668_highlight_handler(S32 item_index) //菜单项高亮时被系统调用的回调函数
{
    firstapp_draw_text(item_index);
}
```

这样，在菜单项高亮显示的时候，就会根据此菜单项的索引 item_index，读出该菜单项的文本内容：MMI_fixed_icontext_menuitems[item_index].item_text。

目前的代码如下：

```
void ShowCategory668Screen(
U16 left_softkey, U16 left_softkey_icon, //左软键文本及图标
U16 right_softkey, U16 right_softkey_icon, //右软键文本及图标
S32 number_of_items, //菜单项项数
U16 *list_of_items, //菜单项文本列表
U16 *list_of_icons, //菜单项图标列表
S32 highlighted_item)//高亮项
{
    S8 i;
    entry_full_screen();
    clear_screen();
    move_fixed_list(20, 50); //设置位置
    resize_fixed_list(200, 200); //设置宽高

    MMI_current_menu_type = LIST_MENU; //设置显示风格

    create_fixed_icontext_menuitems();
    associate_fixed_icontext_list();
    for (i = 0; i < number_of_items; i++)
```

```
  }
  register_fixed_list_shortcut_handler();
  register_fixed_list_keys();
  register_fixed_list_highlight_handler(category668_highlight_handler);

 fixed_list_goto_item_no_redraw(highlighted_item);
 show_fixed_list();
 //显示左右软键
change_left_softkey(left_softkey, left_softkey_icon);
change_right_softkey(right_softkey, right_softkey_icon);
show_softkey(MMI_LEFT_SOFTKEY);
show_softkey(MMI_RIGHT_SOFTKEY);
  gui_BLT_double_buffer(0,0,UI_device_width-1,UI_device_height-1);
  SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
//现在，就可以在屏幕函数中调用模板
void EntryFirstAppScreen(void)
{
S32 i;
U16 list_of_items[new_fixed_list_n_items];
U16 list_of_icons[new_fixed_list_n_items];
//设置每个菜单项的文本和图标
for (i = 0; i <new_fixed_list_n_items; i++)
{
  list_of_items[i] = STR_GLOBAL_DIAL + i;
  list_of_icons[i] = IMG_GLOBAL_L1 + i;
}
EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
//传入需要的图标、文本和菜单项内容到模板
ShowCategory668Screen(STR_GLOBAL_OK, IMG_GLOBAL_OK,
STR_GLOBAL_BACK, IMG_GLOBAL_BACK,
new_fixed_list_n_items,
list_of_items,list_of_icons,
0);
}
```

保存修改，编译并运行模拟器，效果如图 11.6 所示。

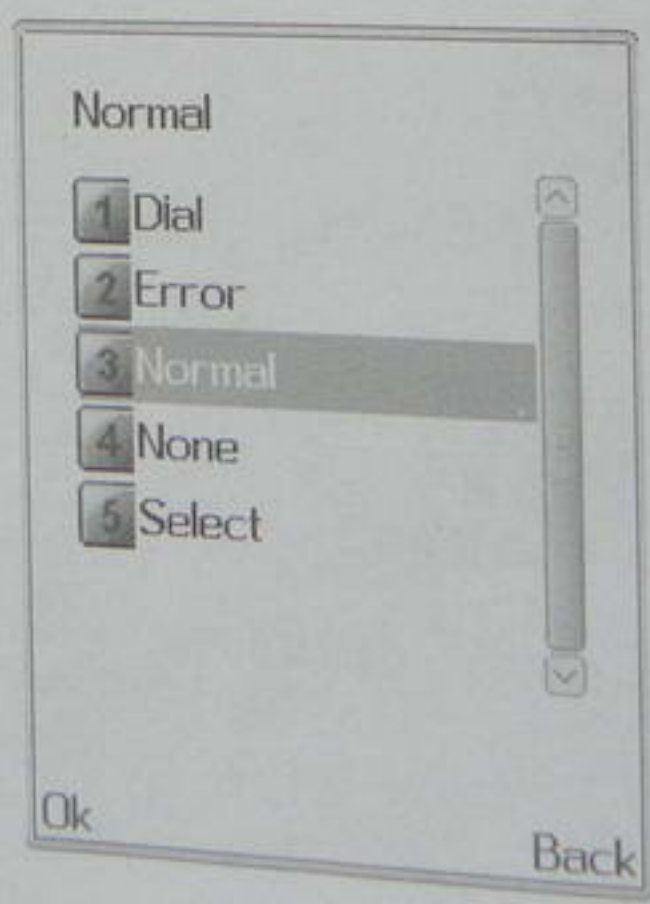


图 11.6 列表效果

11.2.3 提取重绘函数

我们可以看到 MTK 系统提供的模板屏幕中，都有如下形式的函数：

```
ExitCategoryFunction = ExitCategoryXScreen;
GetCategoryHistory = dummy_get_history;
GetCategoryHistorySize = dummy_get_history_size;
RedrawCategoryFunction = dm_redraw_category_screen;
```

这就表明，在 MTK 系统中，屏幕重绘和屏幕逻辑是分开的。因此，要把之前写在 ShowCategory668Screen 函数中的屏幕重绘代码提取出来。

```
void RedrawCategory668Screen(void)
{
  gdi_layer_lock_frame_buffer();
  //显示菜单
  show_fixed_list();
  //显示左右软键
  show_softkey(MMI_LEFT_SOFTKEY);
  show_softkey(MMI_RIGHT_SOFTKEY);
  gdi_layer_unlock_frame_buffer();
  gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

这样，模板函数的内部需要稍微作一些修改。

```
void ShowCategory668Screen(
U16 left_softkey, U16 left_softkey_icon, //左软键文本及图标
U16 right_softkey, U16 right_softkey_icon, //右软键文本及图标
S32 number_of_items, //菜单项项数
U16 *list_of_items, //菜单项文本列表
U16 *list_of_icons, //菜单项图标列表
S32 highlighted_item)//高亮项
{
  S8 i;
  entry_full_screen();
  clear_screen();
  move_fixed_list(20, 50); //设置位置
  resize_fixed_list(200, 200); //设置宽高

  MMI_current_menu_type = LIST_MENU; //设置显示风格

  create_fixed_icontext_menuitems();
  associate_fixed_icontext_list();
  for (i = 0; i < number_of_items; i++)
  {
    add_fixed_icontext_item(get_string(list_of_items[i]), (PU8)GetImage(list_of_icons[i]));
  }
register_fixed_list_shortcut_handler();
register_fixed_list_keys();
register_fixed_list_highlight_handler(category668_highlight_handler);
//显示菜单
fixed_list_goto_item_no_redraw(highlighted_item);
//show_fixed_list();
```

```
change_left_softkey(left_softkey, left_softkey_icon);
change_right_softkey(right_softkey, right_softkey_icon);
//show_softkey(MMI_LEFT_SOFTKEY);
//show_softkey(MMI_RIGHT_SOFTKEY);
 //将我们定义的重绘函数地址传递给全局的函数指针
RedrawCategoryFunction = RedrawCategory668Screen;
RedrawCategoryFunction();
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

RedrawCategoryFunction 及相关的模板函数重绘原理和机制将在下一章 12.1 小节中讲解。

11.2.4 模板历史管理

和正常的屏幕函数一样，屏幕模板也需要历史机制进行管理。

比如在当前屏幕中，突然弹出一个提示框，在提示框结束后，先前的屏幕是否能够正常地显示和保留先前状态。

我们来举个例子，用于测试弹出框对屏幕模板函数的影响。

```
void mmi_test_popup()
{
 EntryNewScreen(0, NULL, mmi_test_popup, NULL);
//弹出窗口，窗口内容就是当前列表的高亮项内容
 ShowCategory165Screen(
STR_GLOBAL_OK, IMG_GLOBAL_OK,
STR_GLOBAL_CANCEL, IMG_GLOBAL_BACK,
get_string(STR_GLOBAL_DIAL  + MMI_fixed_list_menu.highlighted_item),
IMG_GLOBAL_ACTIVATED, NULL
);
SetLeftSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
}
```

然后，在 EntryFirstAppScreen 函数中将左键和弹出框函数绑定，这样就可以按左键将提示框弹出。

```
void EntryFirstAppScreen(void)
{
    …
    SetLeftSoftkeyFunction(mmi_test_popup, KEY_EVENT_UP);
}
```

现在，让我们重新运行模拟器，进入列表屏幕。按下方向键直到菜单项 5 高亮显示。这时，再按左软键，一个弹出框就出现了，如图 11.7 所示。

图 11.7 弹出框效果

然后，单击 Ok 或者 Cancel 按键，返回到列表界面，却发现一个问题，刚才明明高亮显示的是第 5 个菜单项，可是现在高亮项变成了第 1 项。

很明显，这是由于没有正确地将历史加入屏幕模板而导致的。添加模板历史的方法如下。

首先，定义两个函数用于获取与历史相关的信息。

```
U8 *cat888_get_category_history(U8 *history_buffer)
{
    get_list_menu_category_history(0, history_buffer); //返回屏幕模板的历史记录
}
S32 cat888_get_category_history_size()
{
    return (sizeof(list_menu_category_history)); //返回屏幕模板的历史记录大小
}
```

然后，将 ShowCategory668Screen 增加一个参数*history_buffer，用于保存在屏幕流动过程中的历史。

```
void ShowCategory668Screen(
U16 left_softkey, U16 left_softkey_icon, //左软键文本及图标
U16 right_softkey, U16 right_softkey_icon, //右软键文本及图标
S32 number_of_items, //菜单项项数
U16 *list_of_items, //菜单项文本列表
U16 *list_of_icons, //菜单项图标列表
S32 highlighted_item, //高亮项
U8 *history_buffer)
```

接着，我们就需要在屏幕模板函数中进行判断，以决定当前屏幕中某些元素的状态。在本例中，我们要检查的元素就是列表高亮项。将先前代码中的函数

fixed_list_goto_item_no_redraw(highlighted_item)替换为如下代码：

```
if (set_list_menu_category_history(0, history_buffer))
{
  //如果屏幕是 GoBack 进来的，则历史不为空
  fixed_list_goto_item_no_redraw(MMI_fixed_list_menu.highlighted_item);
}
else
{   //如果历史为空则用 highlighted_item 初始化
    fixed_list_goto_item_no_redraw(highlighted_item);
}
```

添加上述代码后，现在如果从弹出框返回到列表屏幕，由于是 goback 过来的，history_buffer 不为空，就将前面全局变量保存的高亮项传递给 fixed_list_goto_item_no_redraw 用于重绘。这就避免了返回后，高亮项无法复原的异常。随后，我们再做一些后续工作，将先前定义的函数地址传递给对应的函数指针。

```
RedrawCategoryFunction = RedrawCategory888Screen;
GetCategoryHistory = cat888_get_category_history;
GetCategoryHistorySize = cat888_get_category_history_size;
RedrawCategoryFunction();
```

在 EntryFirstAppScreen 中，声明一个变量 history_buffer 用于获取屏幕历史，为了体现模板的作用，我们用另外的字符串替换先前的 ID。

```
void EntryFirstAppScreen(void)
{
U16 list_of_items[new_fixed_list_n_items];
U16 list_of_icons[new_fixed_list_n_items];
U8* history_buffer;
for (i = 0; i <new_fixed_list_n_items; i++)
{
```

```
list_of_items[i] = STR_GLOBAL_FILENAME + i;
list_of_icons[i] = IMG_GLOBAL_L1 + i;
}
EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
history_buffer = GetCurrGuiBuffer(SCR_FIRSTAPP_MAIN);//获取当前屏幕模板的历史
ShowCategory668Screen(STR_GLOBAL_OK, IMG_GLOBAL_OK,
                      STR_GLOBAL_BACK, IMG_GLOBAL_BACK,
                      new_fixed_list_n_items,
                      list_of_items,list_of_icons,
                      0,history_buffer);
SetLeftSoftkeyFunction(mmi_test_popup, KEY_EVENT_UP);
}
```

在弹出框出现的时候，系统就会调用当前列表屏幕的退出函数，我们就将模板屏幕历史加入。

```
void ExitFirstAppScreen()
{
    history currHistory;
    currHistory.scrnID = SCR_FIRSTAPP_MAIN;
    currHistory.entryFuncPtr = EntryFirstAppScreen;
    GetCategoryHistory(currHistory.guiBuffer);
    pfnUnicodeStrcpy((S8*) currHistory.inputBuffer, (S8*) L"");
   GetCategoryHistory(currHistory.guiBuffer); //添加屏幕历史时顺便将模板历史加入其中
    AddHistory(currHistory);
}
```

最后，保存代码，运行模拟器，效果如图 11.8 所示。

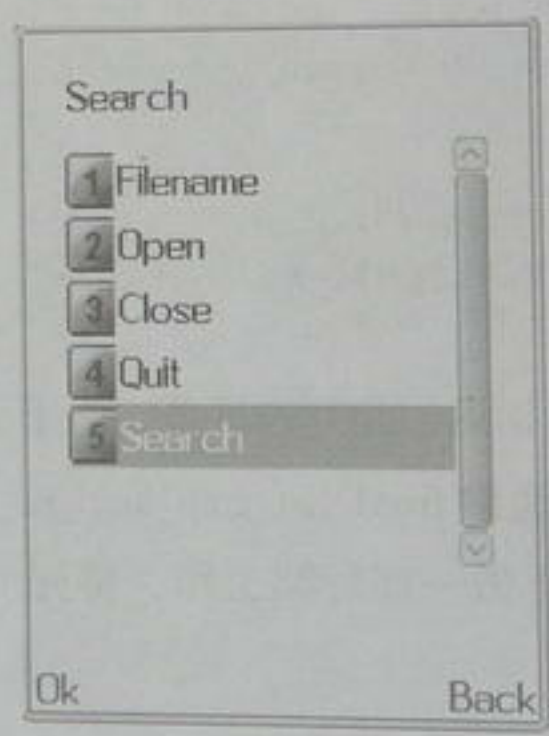


图 11.8　列表效果

11.2.5　退出模板

当屏幕模板完成使命退出的时候，需要做一些还原工作，将这些工作封装在一个函数中。

```
void ExitCategory668Screen(void)
{
ClearHighlightHandler(); //清掉所有高亮响应函数
reset_softkeys(); //重置左右软键
reset_menu_shortcut_handler(); //重置快捷序号框
reset_fixed_list(); //重置列表菜单
}
```

然后将该函数地址赋值给全局的函数指针。

```
RedrawCategoryFunction = RedrawCategory668Screen;
ExitCategoryFunction = ExitCategory668Screen;
GetCategoryHistory = cat668_get_category_history;
```

结语：

本章，我们首先罗列了系统提供的屏幕模板，给读者一个初步的屏幕模板印象。然后，通过一个实例讲解了如何自定义模板，读者通过本章的学习，可以自己设计需要的屏幕模板，提高开发效率。

12 高级屏幕模板

在 WGUI 中除了屏幕模板和控件模板外，还有两个重要的模块：触摸屏（Touch Screen）和重绘管理器（Draw Manager）。

· Touch Screen：以下简称（TS），主要是为了协调屏幕中的触摸屏操作管理，要协调触摸屏操作，就要知道当前屏幕中含有哪些控件。

为此，MTK 建立了一个模板数据库，在数据库中记录了每个模板的控件列表，如下所示：

```
const U8 category14_matrix[] =
{
    5,
    DM_BASE_LAYER_START,
    DM_SCR_BG,
    DM_BASE_CONTROL_SET1,
    DM_MATRIX_MENU1

};
```

数组 category14_matrix 在系统中表示主菜单界面包含组件（控件）。数组中每个元素就代表一个控件或者说组件。当有触摸屏事件发生的时候，TS 就会从数据库中取出当前模板的控件集合，实际是一个数组，并且依次“询问”每个控件要不要处理当前的触摸屏操作，如果控件注册过该屏幕事件，那么控件就会获得当前的控制权。轮询控件如图 12.1 所示。

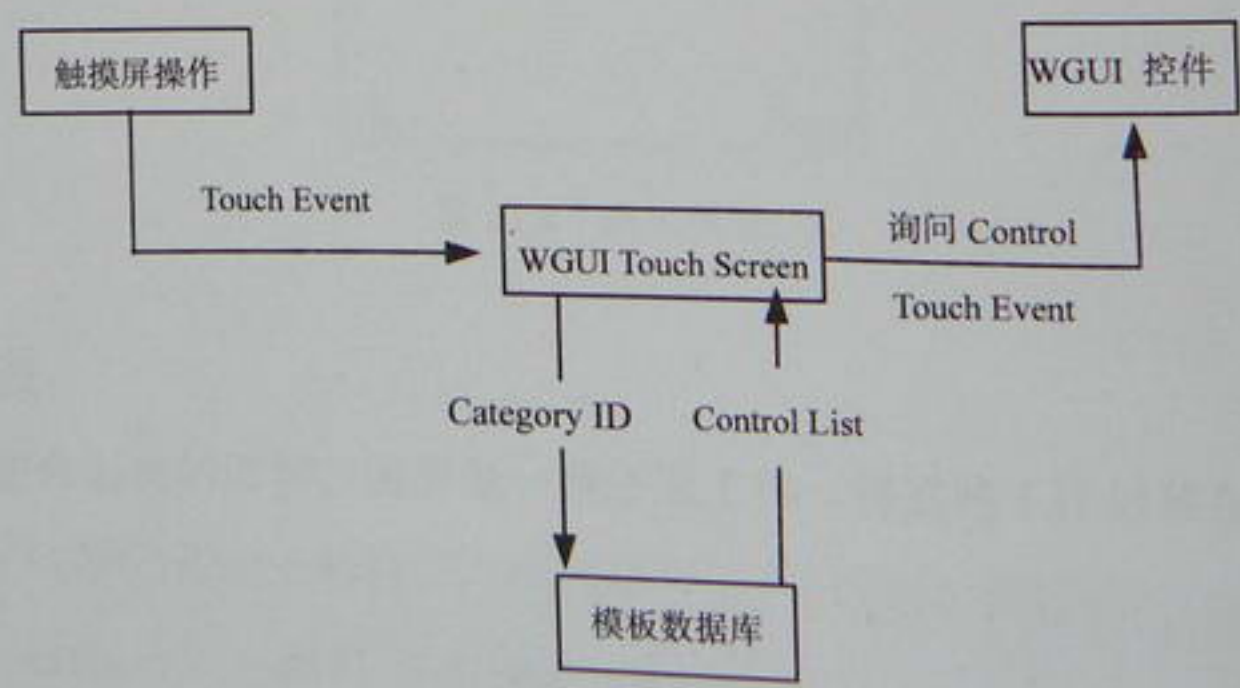


图 12.1　轮询控件

· Draw Manager：以下简称（DM），主要是为了提高开发效率。TS 在模板数据库中保存了每个模板

的控件列表，DM 一并将每个控件的属性集（如排版数据，控制标志）也加入其中，当要绘制模板时 DM 将控件集与属性集一起取出来，然后依次通知每个控件，控件收到 DM 的通知与相关的属性集合后立即重绘，如图 12.2 所示。

这样做非常方便，在绘制过程中可以迅速地取得控件 ID 和这个控件的相关属性，使得绘制的流程简洁而高效。

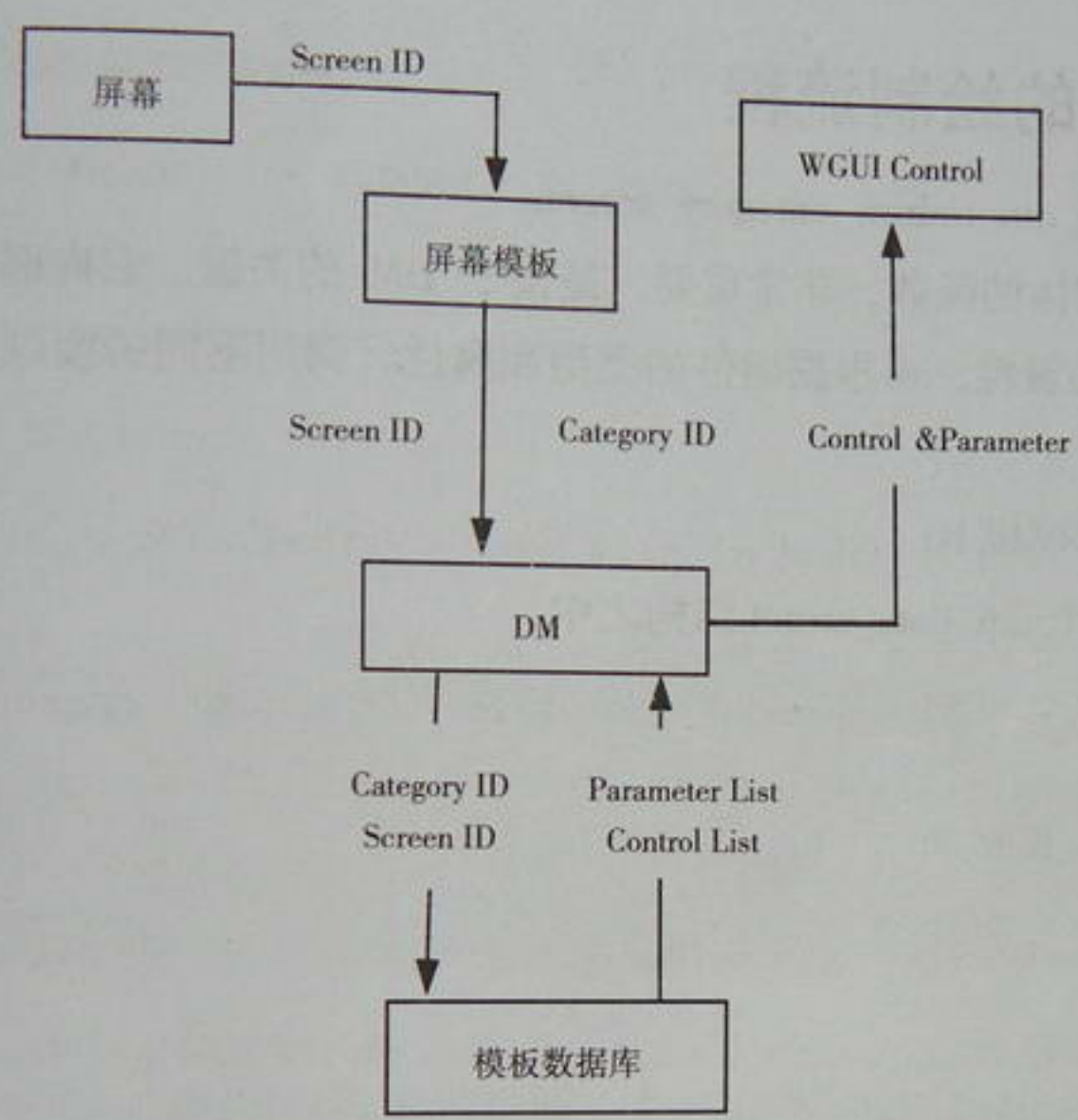


图 12.2 获取控件属性

在系统中，很多形如 ShowCategoryXScreen 的屏幕模板都会调用如下 3 个函数：

```
dm_setup_data(&dm_data);
dm_register_category_controlled_callback(dm_category_33_controlled_area);
dm_redraw_category_screen();
```

我们首先来看一下待机界面使用的屏幕模板 ShowCategory33Screen。

```
void ShowCategory33Screen(
        U16 left_softkey,
        U16 left_softkey_icon,
        U16 right_softkey,
        U16 right_softkey_icon,
        U8 *history_buffer)
{
    GDI_LOCK;
    ExitCategoryFunction = ExitCategory33Screen;
    GetCategoryHistory = dummy_get_history;
    GetCategoryHistorySize = dummy_get_history_size;
    RedrawCategoryFunction = dm_redraw_category_screen;
    ...
    dm_setup_data(&dm_data);
```

```
    dm_redraw_category_screen();
    GDI_UNLOCK;
}
```

粗体部分的语句是所有屏幕模板函数都含有的，这几个函数也是模板绘制的核心，接下来我们为读者进一步讲解其具体细节和作用。

12.1 高级模板的绘制流程

我们首先来讲解函数 dm_redraw_category_screen。

这个函数便是显示窗体的函数，非常重要，是整个 DM 的关键。它内部的实现是先获得该窗体所包含的组件元素及它们的属性，再根据组件的类型和属性，调用不同的接口，然后逐一绘制各组件，主要流程如下。

· 获取待绘制的窗体模板 ID。

模板 ID 被封装在一个 dm_data_struct 结构之中。

```
typedef struct
{
    S32 s32ScrId;
    S32 s32CatId; //模板 ID
    S32 s32flags;
} dm_data_struct;
```

使用示例：

```
dm_data_struct  g_dm_data
g_dm_data.s32CatId 将获得当前模板 ID 的值
```

· 根据模板 ID 获取控件集。

```
control_set_ptr = dm_search_control_set((U16) g_dm_data.s32CatId, &DeafultCoordinateSet_p);
u8NoOfUICtrls = control_set_ptr[0];
```

u8NoOfUICtrls 即屏幕控件的数量，用于后面 DM 重绘屏幕时的循环检测，获取的组件集将存放在 DeafultCoordinateSet_p 变量之中。

· 根据模版 ID 得到控件属性标识的集合。

```
UICtrlAccessPtr_p = dm_search_coordinate_set(g_dm_data.s32ScrId);
```

属性标识包括控件绘制时的坐标、控件大小、是否有按键、是否是全屏等。属性都对应着一个宏，集中定义在文件 wgui_draw_manager.h 之中，如图 12.3 所示。

```
#define DM_FULL_SCREEN_COORDINATE_FLAG      (-10002)
#define DM_FULL_SCREEN_COORDINATE           (-10003)
#define DM_CONTENT_COORDINATE_FLAG          (-10004)
#define DM_CONTENT_COORDINATE               (-10005)
#define DM_POPUP_SCREEN_COORDINATE_FLAG     (-10006)
#define DM_POPUP_SCREEN_COORDINATE          (-10007)
#define DM_DEFAULT_STATUS_BAR_FLAG          (-10008)
#define DM_DEFAULT_STATUS_BAR               (-10009)
```

图 12.3 属性宏定义的片段

· 锁定当前图层，各个组件绘制前的任务准备就绪。

```
gdi_layer_lock_frame_buffer()
```

· 在属性的集合中，取第一个标志作为窗体的基础属性，准备绘制窗体。

```
UICtrlAccessPtr_p = dm_get_cat_scr_coordinates(UICtrlAccessPtr_p, &dm_cat_scr_info);
```

· 循环获取组件，分发到各个绘制接口，并传入相应的属性。请看下面的 MTK 代码片段：

```
for (u8CtrlCt = 1; u8CtrlCt <= u8NoOfUICtrls; u8CtrlCt++)
  {
     switch (control_set_ptr[u8CtrlCt])
     {
        case DM_BASE_LAYER_START:
        {
           if (redraw_type != REDRAW_2_REDRAW)
           {
              //设置基础层
              dm_setup_base_layer();
              …
        case : DM_SCR_BG:
           {  …
              if (g_dm_cntx.redraw_screen_main != REDRAW_2_REDRAW)
              {
    //根据属性 UICtrlAccessPtr_p 调用接口 dm_setup_and_draw_scr_bg 设置并绘制背景
               dm_setup_and_draw_scr_bg(&UICtrlAccessPtr_p);
              }
               break;
           }
            …
        case DM_LIST1:
        {
//根据属性 UICtrlAccessPtr_p 调用接口 dm_setup_and_draw_fixed_lis 绘制列表
           dm_setup_and_draw_fixed_list(&UICtrlAccessPtr_p);
           break;
                  }
           …
```

· 根据窗体标志变量，绘制软键盘等特殊组件。

```
/* if virtual keypad show flag is set, show the virtual keypad */
   if (g_dm_data.s32flags & DM_SHOW_VKPAD)
   {
      if (dm_show_vkpad_callback != NULL)
      {
         dm_show_vkpad_callback();
      }
      /* wgui_show_virtual_keyboard(); */
   }
```

· 全部绘制完毕，整合图层，释放图层。

```
gdi_layer_unlock_frame_buffer();
if (!(g_dm_data.s32flags & DM_NO_BLT))
   {
      gdi_layer_blt(
         dm_layers[0],
         dm_layers[1],
         dm_layers[2],
         dm_layers[3],
         dm_cat_scr_info.x1,
```

```
            dm_cat_scr_info.y1,
            dm_cat_scr_info.x2 - dm_cat_scr_info.x1 - 1,
            dm_cat_scr_info.y2 - dm_cat_scr_info.y1 - 1);
    }
```

12.2 模板 ID

屏幕模板都对应一个模板数据库，数据库存放在文件 CustCoordinates.c 中。

映射表 g_categories_controls_map：

```
typedef struct
{
    U16 category_id; //模板 ID，categoryScreen，DM，TS 之间通过该模板 ID 交互
    U8 *control_set_p;//控件列表
    S16 *default_coordinate_set_p; //属性列表
    S16 *rotated_coordinate_set_p; //暂时不起作用
} dm_category_id_control_set_map_struct;
```

可以看到，无论是窗体所包含的组件，还是组件的属性，都是根据模板 ID 获取的。所以，我们必须要清楚两个问题：

（1）模板 ID 是如何传递到这个屏幕模板函数之中的；

（2）模板 ID 和窗体组件、组件的属性是如何关联的。

12.2.1 模板 ID 的传递

模板 ID 是屏幕模板函数 ShowCategoryXXScreen 所调用的屏幕 ID，这里的屏幕 ID 不要与函数 EntryNewScreen 的参数中的 scrID 相混淆，它实际表示绘制时调用的模板编号。这里的屏幕 ID 都是以 MMI_为前缀，被定义在枚举结构 MMI_CATEGORY_ID_LIST 中的。打开文件 wgui_categories_defs.h，即可看到每个模板 ID 的定义。

```
…
#include "MMI_features.h"
#include "gui_windows.h"

enum MMI_CATEGORY_ID_LIST
{
    MMI_LIST_CATEGORY_ID = 1,
    MMI_CATEGORY5_ID,
    MMI_CATEGORY6_ID,
    MMI_CATEGORY7_ID,
    MMI_CATEGORY9_ID,
    /* START VIJAY PMT 20050930 */
#ifdef UI_SMALL_LIST_SCREEN
    MMI_CATEGORY10_ID,
#endif
…
```

结构体 dm_data_struct 的成员 s32CatId，就表示这个模板 ID。我们来看一下结构体 dm_data_struct 的定义：

```
typedef struct
{
    S32 s32ScrId;
    S32 s32CatId; //模板 ID
    S32 s32flags;
} dm_data_struct;
```

参数 s32ScrId：当前窗口 ID，也就是我们使用 EntryNewScreen 函数时传入的那个参数。

参数 s32CatId：屏幕模板 ID，即我们前面提到的以 MMI_为前缀的屏幕 ID。

参数 flag：是模板执行显示软键盘、清屏等操作时，所置的标志变量，它在 dm_redraw_category_screen 函数中被判断。

我们重点讲解结构体成员 s32CateId 的使用。以 ShowCategory33Screen 函数为例，这个显示模板的函数中，可以看到这样的语句：

```
dm_data.s32ScrId = (S32)GetActiveScreenId();
dm_data.s32CatId = MMI_CATEGORY33_ID;
dm_data.s32flags = 0;
dm_setup_data(&dm_data);
```

上述代码也是所有屏幕模板函数都具备的，这 4 句代码其实就是对全局的模板属性进行赋值，然后，调用函数 dm_setup_data。

```
void dm_setup_data(dm_data_struct *dm_data)
{
    g_dm_data.s32CatId = dm_data->s32CatId;
    g_dm_data.s32ScrId = dm_data->s32ScrId;
    g_dm_data.s32flags = dm_data->s32flags;
    ...
}
```

通过上述代码， MMI_CATEGORY33_ID 就被赋值到 g_dm_data.s32CatId 中，然后随着这个全局变量，顺利地被带到了 dm_redraw_category_screen 函数中。

12.2.2 模板 ID 和窗体组件、组件的属性的关联和交互

分析 dm_search_control_set 函数，可以看到窗体模板的组件和属性相关信息都隐藏在一个很庞大的 dm_category_id_control_set_map_struct 类型的结构体数组中。

结构体 dm_category_id_control_set_map_struct 定义如下：

```
typedef struct
{
    U16 category_id;
    U8 *control_set_p;
    S16 *default_coordinate_set_p;
    S16 *rotated_coordinate_set_p;
}dm_category_id_control_set_map_struct;
```

第 1 个结构体成员，是窗体模板的 ID。

第 2 个结构体成员，是组件数组的首地址。

第 3 个结构体成员，是默认的组件属性数组的首地址。

第 4 个结构体成员，是特殊的组件属性数组的首地址。

函数 dm_search_control_set 和函数 dm_search_coordinate_set 就是通过匹配模板 ID 在结构体数组中分别获取组件集合和组件属性集合的。

dm_search_control_set 函数的原型：

```
U8 *dm_search_control_set(S32 category_id, S16 **default_coordinate_set_p)
```

通过函数 dm_search_control_set 获得控件集合和控件个数。

```
dm_search_coordinate_set 函数的原型：
S16 *dm_search_coordinate_set(S32 ScrId)
```

调用该函数获得每个控件的具体绘制属性和标志。

```
UICtrlAccessPtr_p = dm_search_coordinate_set(g_dm_data.s32ScrId);
    if (UICtrlAccessPtr_p == NULL)
    {
        UICtrlAccessPtr_p = DeafultCoordinateSet_p;
    }
```

在系统调用函数 dm_redraw_category_screen 重绘的过程中，还会检查当前模板的控件集合中是否有属性标志。如果没有，就会将默认的属性标识集赋值给 UICtrlAccessPtr_p，用于接下来的窗体绘制。

12.3 模板组件

12.3.1 构成模板的组件定义

模板组件的定义与窗体模板 ID 是通过结构体 dm_category_id_control_set_map_struct 关联的，现在来看一下组件数组的结构。打开文件 Custcoodinates.c，可以看到大量的组件数组定义。

下面将以 6 号模板为例：

```
const U8 category6[] =
{
    4,
    DM_BASE_LAYER_START,
    DM_SCR_BG,
    DM_BASE_CONTROL_SET1,
    DM_LIST1
};
```

在这个数组中：

- 4 表示当前屏幕模板的组件数量；
- DM_BASE_LAYER_START 表示开始使用基础图层；
- DM_SCR_BG 表示背景图；
- DM_BASE_CONTROL_SET1 表示窗体将使用 1 号控件集；
- DM_LIST1 表示列表组件。

这样，就定义了这个屏幕模板所需要的控件元素，读者从字面上就可以猜到这个屏幕将会是什么样子。在 MTK 环境的原始版本中，组件特别丰富。各组件的外观和功用，从组件名称的字面意思上就可

略知一二，读者可以在具体使用过程中进行了解。

说 明

MTK 系统将基本的窗体元素集中起来组成一个控件集，方便开发使用。DM_BASE_CONTROL_SET1 表示当前屏幕的基本组件包括状态栏、标题栏、左右按键等。这样，开发人员定义这一个标识在组件数组中就可以了，大大地节省了时间。

MTK 还有一种控件集 DM_BASE_CONTROL_SET2，和前面的控件集作用一样。

12.3.2 各组件的属性定义

前面声明了模板的组件，接下来就需要定义各个组件的属性。

```
const S16 coordinate_set6[] =
{
    DM_FULL_SCREEN_COORDINATE_FLAG,
    DM_CONTENT_COORDINATE_FLAG
};
```

这两个常量都是代表组件属性的标志，定义在 Wgui_draw_manager.h 中。

```
…
#define DM_FULL_SCREEN_COORDINATE_FLAG  -10002
#define DM_CONTENT_COORDINATE_FLAG  -10004
…
```

每个控件需要多少个属性由控件本身决定，读取属性也需要由控件自己来做。控件只需要取得属性后通知 DM 其获得了几个即可。

大部分的控件属性集定义都是有规律的，一般是 5 个属性：X、Y、Width、Height、Flags。

MTK 提供了大量的定义好的属性，如表 12.1 所示。

表 12.1 预定义属性

第一个属性值	X	Y	Width	Height	Flags
DM_DUMMY_COORDINATE	0	0	0	0	无
DM_NULL_COORDINATE	−1	−1	−1	−1	无
DM_FULL_SCREEN_COORDINATE_FLAG	0	0	LCD Width	LCD Height	无
DM_CONTENT_COORDINATE_FLAG	Content X	Content Y	Content Width	Content Height	无
DM_POPUP_SCREEN_COORDINATE_FLAG	Popup X	Popup Y	Popup Width	Popup Height	无
DM_DEFAULT_TITLE_BAR_FLAG	Title X	Title Y	Title Width	Titel Height	无
DM_DEFAULT_BUTTON _BAR_FLAG	0	Button Bar Y	LCD Width	Button Bar Height	无

还需要注意的是，属性集数组中的第一个属性不是属于某一个控件的，而是整个模板的基础属性。

12.4 模板组件类型

模板数据库中的组件总体来说可以分为 3 种类型。

12.4.1 控制类型控件

这种控件比较特殊，其并不是用来显示的，而是在后台控制其他控件的显示，如表 12.2 所示。

表 12.2 控制型控件

名　称	描　述	属 性 集
DM_BASE_LAYER_START	激活基础层，从 DM_BASE_LAYER_START 到 DM_BASE_LAYER_END 之间的控件都画在基础层上	布局：无 Flags：无
DM_BASE_LAYER_END	结束基础层活动状态，此控件基本上不用	布局：无 Flags：无
DM_NEW_LAYER_START	创建一个新层，并将其激活。从 DM_NEW_LAYER_START 到 DM_NEW_LAYER_END 之间的控件都画在新层上	布局：标准（DM） Flags：DM_NO_FLAGSS
DM_NEW_LAYER_END	结束新层活动状态，此控件基本不用，新层在屏幕退出时由 DM 自动释放	布局：无 Flags：无
DM_ALIGNED_AREA_START	控件对齐开始。从此控件开始，至 DM_ALIGNED_AREA_END 结束，中间的所有控件将上下等距对齐。（注：如若中间有单行或多行编辑框，则自动压缩使其大小不超过对齐区域的大小）	布局：标准（DM） Flags：DM_ALLIGNED_AREA_EQUAL_SPACE_TOP_AND_BOTTOM，间距为零，上下居中
DM_ALIGNED_AREA_LAYER_END	控件对齐结束	布局：无

12.4.2 标准显示类控件

标准显示类控件都有一个 WGUI 控件原型，如表 12.3 所示。

表 12.3 标准显示型控件

名　称	描　述	属 性 集
DM_CIRCULAR_MENU1	循环 3D 菜单。只有主菜单才会用到	布局：标准（DM） Flags：DM_NO_FLAGS
DM_LIST1	列表菜单	布局：标准（CategoryScreen）Flags：DM_CATEGORY_CONTROL_COORDINATES，菜单布局强制使用 DM 数据
DM_INLINE_FIXED_LIST1	内嵌编辑列表菜单	布局：标准（DM） Flags：DM_NO_FLAGS
DM_MATRIX_MENU1	矩阵菜单	布局：标准（DM） Flags：DM_NO_FLAGS
DM_DYNAMIC_LIST1	动态列表菜单。 菜单项在显示的时候动态加载（WGUI 菜单项数组容量有限，当实际项数超出容量时，就只能显示到哪一项再初始化哪一项）。修改待机界面会用到该控件	布局：标准（DM） Flags：DM_NO_FLAGS
DM_ASYNCDYNAMIC_LIST2	异步动态列表菜单。 菜单加载时采用异步机制（加载很耗时，为了不阻塞用户操作，只能异步加载完后再显示）	布局：标准（DM） Flags：DM_NO_FLAGS
DM_SINGLELINE_1	单行编辑框	布局：无

12.4.3 扩展显示类控件

扩展显示类控件如表 12.4 所示。

表 12.4 扩展显示型控件

名　称	描　述	属 性 集
DM_POPUP_BACK GROUND	弹出提示框背景	布局：标准（DM） Flags： DM_DRAW_POPUP_BACKGROUND_3D，提示框用 3D 效果 DM_POPUP_BACKGROUND_HATCH_FILL：用十字编纹填充整个屏幕 DM_POPUP_BACKGROUND_GREYSCALE：将整个屏幕半灰色显示

续表

名　称	描　述	属 性 集
DM_WALL_PAPER	桌面背景	布局：标准（DM） Flags：DM_NO_FLAGS
DM_SCR_BG	屏幕背景	布局：无 Flags：无
DM_CATEGORY_CONTROLLED_AREA	自画区域 1	布局：标准（用户自定） Flags：DM_NO_FLAGS
DM_CATEGORY_CONTROLLED_AREA2	版画区域 2	布局：标准（用户自定） Flags：DM_NO_FLAGS
DM_NWAY_STOPWATCH	多向秒表	布局：标准（DM）+31 个自定义参数 Flags：DM_NO_FLAGS
DM_TYPICAL_STOP	普通秒表	布局：标准（DM）+31 个自定义参数

12.5 自定义模板

12.5.1 模板数据库

高级模板的重点就是“模板数据库”，数据库存在 CustCoordinates.c 文件之中。下面结合一个例子来详细讲述数据库的构成。

首先来看映射表 g_categories_controls_map，映射表中每一项代表一个模板，其结构定义如下：

```
typedef struct
{
U16 category_id; //模板 ID，CategoryScreen、DM、TouchScreen 之间通过它相互交流
U8 *control_set_p; //控件列表
S16 *default_coordinate_set_p; //属性列表
S16 *rotated_coordinate_set_p; //旋转屏幕的属性列表，目前基本不用
} dm_category_id_control_set_map_struct;
```

模板 ID 定义在 wgui_categories_defs.h 文件中。现在，我们将新模板的 ID 命名为 MMI_CATEGORY668_ID。

```
enum MMI_CATEGORY_ID_LIST
{
MMI_LIST_CATEGORY_ID = 1,
MMI_CATEGORY5_ID,
…
MMI_CATEGORY_NSM275,
MMI_CATEGORY668_ID //我们屏幕的模板 ID
};
```

然后在映射表中加上一项：

```
const dm_category_id_control_set_map_struct g_categories_controls_map[] =
{
```

```
{MMI_CATEGORY5_ID, (U8 *) category5, (S16 *) coordinate_set5, NULL},
{MMI_CATEGORY6_ID, (U8 *) list_menu_category, (S16 *) common_coordinate_set, NULL},
{MMI_CATEGORY_NSM275, (U8 *) categoryNSM275, (S16 *) coordinate_setNSM275, NULL},
...
{MMI_CATEGORY888_ID, (U8 *) category668, (S16 *) coordinate_set668, NULL}
};
```

其中，category668 代表模板中所包含的控件，模板需要一个列表，还有一个按键栏，定义如下：

```
const U8 category668[] =
{
3,
DM_BASE_LAYER_START,
DM_LIST1,
DM_BUTTON_BAR1,
};
```

第 1 个参数表示模板包含的空间数，很明显，目前的控件数为 3。

DM_BASE_LAYER_START 是控制类型的控件，后面再详细讲解。

控件在列表中放置的顺序也有一定的特点，排在越后的控件，显示越靠近上层，也是最容易接受到触摸屏事件的。

定义了所包含的控件，接下来就需要定义模板的属性集。

```
const S16 coordinate_set668[] =
{
DM_FULL_SCREEN_COORDINATE_FLAG,
20, MMI_CONTENT_Y + 5, 136, MMI_CONTENT_HEIGHT - 40, DM_NO_FLAGS,
DM_DEFAULT_BUTTON_BAR_FLAG, MMI_SOFTKEY_WIDTH,
};
```

12.5.2 普通模板加入 DM

将上一章的 ShowCategory668Screen 函数稍作修改：

```
void ShowCategory668Screen(
U16 left_softkey, U16 left_softkey_icon, //左软键文本及图标
U16 right_softkey, U16 right_softkey_icon, //右软键文本及图标
S32 number_of_items, //菜单项项数
U16 *list_of_items, //菜单项文本列表
U16 *list_of_icons, //菜单项图标列表
S32 highlighted_item,
U8* history_buffer)//高亮项
{
S8 i;

  dm_data_struct dm_data; //模板参数
  entry_full_screen();
  clear_screen();


  create_fixed_icontext_menuitems();
  associate_fixed_icontext_list();
  for (i = 0; i < number_of_items; i++)
  {
```

```
    add_fixed_icontext_item(get_string(list_of_items[i]), (PU8)GetImage(list_of_icons[i]));
  }
register_fixed_list_shortcut_handler();
register_fixed_list_keys();
register_fixed_list_highlight_handler(category668_highlight_handler);

change_left_softkey(left_softkey, left_softkey_icon);
change_right_softkey(right_softkey, right_softkey_icon);

if (set_list_menu_category_history(MMI_CATEGORY668_ID, history_buffer))
  {
    fixed_list_goto_item_no_redraw(MMI_fixed_list_menu.highlighted_item);
  }
  else
  {
    fixed_list_goto_item_no_redraw(highlighted_item);
  }


RedrawCategoryFunction = RedrawCategory668Screen;
RedrawCategoryFunction = dm_redraw_category_screen; //DM 绘制模板函数
GetCategoryHistory = dm_get_category_history; //DM 获取模板历史函数
GetCategoryHistorySize = dm_get_category_history_size; // DM 获取模板历史记录大小
//当前屏幕 ID，运行 EntryNewScreen(SCR_XXX，… )函数时自动保存
dm_data.s32ScrId = (S32) GetActiveScreenId();
dm_data.s32CatId = MMI_CATEGORY668_ID; //当前屏幕模板的 ID
dm_data.s32flags = DM_NO_FLAGS; //当前模板的 Flags，详细标志见下表
dm_setup_data(&dm_data); //将模板参数传入 DM
dm_redraw_category_screen(); //由 DM 绘制模板
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

修改后，运行效果和之前一样。现在我们不仅可以进行触摸屏操作，而且函数的代码量也减少了。

表 12.5　　dm_data.s32flags 的值

Flag	描　述
DM_NO_FLAGS	默认标志
DM_NO_SOFTKEY	不显示软键
DM_NO_TITLE	不显示标题条
DM_NO_STATUS_BAR	不显示状态条
DM_NO_POPUP_BACKGROUND	不显示弹出提示框背景
DM_LEFT_ALIGN_TITLE	标题条文本左对齐（默认居中显示）
DM_SCROLL_TITLE	标题条文本过长时自动滚动
DM_CLEAR_SCREEN_BACKGROUND	清理整个屏幕背景（为新屏幕绘制作准备）
DM_SHOW_VKPAD	显示虚拟键盘
DM_SUB_LCD_CONTEXT	此屏幕模板用于副屏显示
DM_NO_BLT	DM 不自动合并层（默认会自动合并）

12.5.3 自绘制控件

如果 MTK 提供的控件没有我们实际开发所需要的，就可以使用 DM_CATEGORY_CONTROLLED_AREA 控件进行自定义。

该控件实际是一个白板，其式样和内容以及触摸屏操作都由该控件自行处理。

下面，我们为“668”模板增加一个自定义控件，也就是一个自画区域，并将此区域用作显示触摸屏操作时的自定义内容。

修改模板数据库：

```
const U8 category668[] =
{
4,
DM_BASE_LAYER_START,
//将自画区域放在系统提供的控件最前面，这样就不妨碍后面控件显示及操作
DM_CATEGORY_CONTROLLED_AREA,
DM_LIST1,
DM_BUTTON_BAR1,
};
const S16 coordinate_set668[] =
{
DM_FULL_SCREEN_COORDINATE_FLAG,
DM_FULL_SCREEN_COORDINATE_FLAG, //设置控件为全屏大小，也可为其他尺寸
20, MMI_CONTENT_Y + 5, 200, 200, DM_NO_FLAGS,
DM_DEFAULT_BUTTON_BAR_FLAG, MMI_SOFTKEY_WIDTH
};
```

然后，对具体的屏幕模板函数中对自绘制控件进行绘制。

```
void DrawCate668CategoryControlArea (dm_coordinates *coordinate)
{
  RedrawControl();//具体的绘制函数。可以绘制任意图形、文字等。读者可以自行试验。
}
```

最后，对自绘制控件的函数进行注册。

```
void ShowCategory668Screen(…)
{
…
dm_setup_data(&dm_data);
dm_register_category_controlled_callback(DrawCate668CategoryControlArea);
dm_redraw_category_screen();
 SetKeyHandler(GoBackHistory,KEY_RSK,KEY_EVENT_UP);
}
```

这样，自绘制的控件就正确地加入了屏幕模板，开发人员可以利用该控件特性创造出更加别致的效果。

12.5.4 更加漂亮的界面

在前面的控件章节中，我们讲了一些按钮控件的基本情况。在高级屏幕模板中，还可以实现自绘制按钮的效果。

按钮，一般都有按下和松开的两种状态。在高级模板中，可以通过两张图片的切换来实现这一

效果。

假如现在需要开发一个软件，其界面包含多个按钮操作，而且按钮的布局有一定规则。并且这个界面可能被软件其他的界面所重用，那么最好就是通过高级屏幕模板来实现。

首先，准备如下图 12.4 所示形式的图片。

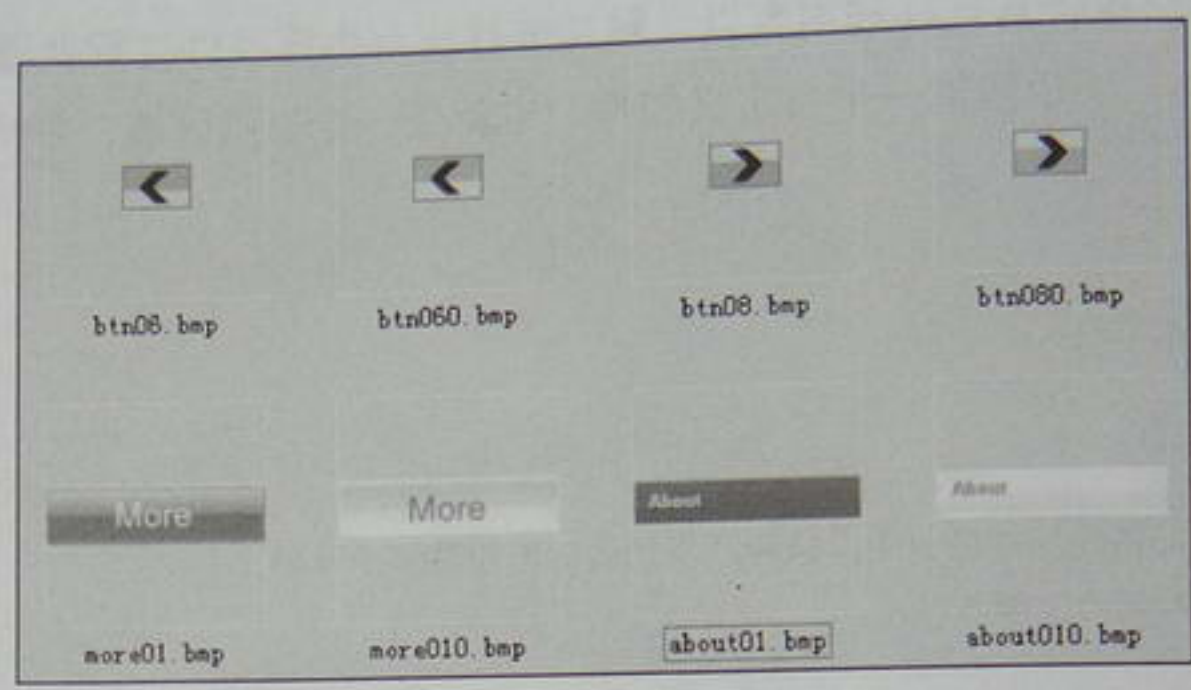


图 12.4 各个状态的按钮图片

为描述方便，我们总共定义 4 个按钮，每个按钮两种状态，因此共需要 8 张图片。

然后，在文件 FirstAppDefs.h 中声明每个图片的 ID。

```
typedef enum
{
    IMG_FIRSTAPP = FIRSTAPP_BASE +1,
    IMG_FIRSTAPP_PICTURE,
    IMG_FIRSTAPP_ANIMATE,
    IMG_FIRSTAPP_BACKGROUND,
    IMG_FIRSTAPP_BACKGROUND1,
    IMG_FIRSTAPP_BACKGROUND2,
    IMG_FIRSTAPP_ABOUT01,
    IMG_FIRSTAPP_ABOUT010,
    IMG_FIRSTAPP_MORE01,
    IMG_FIRSTAPP_MORE010,
    IMG_FIRSTAPP_BTN06,
    IMG_FIRSTAPP_BTN060,
    IMG_FIRSTAPP_BTN08,
    IMG_FIRSTAPP_BTN080
}IMAGEID_LIST_FIRSTAPP;
```

在 Res_FirstApp.c 文件中添加图片的资源路径。

```
    ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP_ABOUT01,CUST_IMG_PATH"\\\\MAINLCD\\\\FirstApp\\\\
about01.bmp","BTN Image");
    ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP_ABOUT010,CUST_IMG_PATH"\\\\MAINLCD\\\\FirstApp\\\\
about010.bmp","BTN Image");
    ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP_MORE01,CUST_IMG_PATH"\\\\MAINLCD\\\\FirstApp\\\\
more01.bmp","BTN Image");
    ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP_MORE010,CUST_IMG_PATH"\\\\MAINLCD\\\\FirstApp\\\\
more010.bmp","BTN Image");
    ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP_BTN06,CUST_IMG_PATH"\\\\MAINLCD\\\\FirstApp\\\\
btn06.BMP","BTN Image");
```

```
ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP_BTN060,CUST_IMG_PATH"\\\\MAINLCD\\\\FirstApp\\\\btn
060.BMP","BTN Image");
ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP_BTN08,CUST_IMG_PATH"\\\\MAINLCD\\\\FirstApp\\\\btn
08.BMP","BTN Image");
ADD_APPLICATION_IMAGE2(IMG_FIRSTAPP_BTN080,CUST_IMG_PATH"\\\\MAINLCD\\\\FirstApp\\\\btn
080.BMP","BTN Image");
```

接下来，打开 wgui_categories_defs.h 文件，定义一个新的模板，声明在前面提到的“668”模板的后面。

```
enum MMI_CATEGORY_ID_LIST
{
    …
    _CATEGORY668_ID,
    MMI_CATEGORY669_ID
    };
```

打开 CustCoordinates.c 文件。

```
const dm_category_id_control_set_map_struct g_categories_controls_map[] =
    /* The sequence of category id should be listed in ascending order */
{
{MMI_CATEGORY5_ID, (U8 *) category5, (S16 *) coordinate_set5, NULL},
…
    {MMI_CATEGORY668_ID, (U8 *) category668, (S16 *) coordinate_set668, NULL},
    {MMI_CATEGORY669_ID,(U8 *) category669, (S16 *) coordinate_set669, NULL}
};
```

接下来，分别定义 category669 和 coordinate_set669 模板。由于总共有 4 个按钮，故 DM_BUTTON 重复声明 4 次。为了增强效果，我们还在模板中增加背景的绘制，因此增加一个变量 DM_SCR_BG。

```
const U8 category669[] =
{
      7,
      DM_BASE_LAYER_START,
      DM_SCR_BG,
      DM_BUTTON,
      DM_BUTTON,
      DM_BUTTON,
      DM_BUTTON_BAR1
};
```

紧接着，设置每个控件的具体属性。

```
const S16 coordinate_set669[] =
{
   DM_FULL_SCREEN_COORDINATE_FLAG,
   30,40,30,20,DM_NO_FLAGS,     //对应实际绘制的第 4 个按钮
   180 ,40,30,20,DM_NO_FLAGS,  //对应实际绘制的第 3 个按钮
   50,160,145,30,DM_NO_FLAGS,  //对应实际绘制的第 2 个按钮
   50,210,145,40,DM_NO_FLAGS,  //对应实际绘制的第 1 个按钮
                               //ButtonBar 的属性
   DM_DEFAULT_BUTTON_BAR,
   DM_BUTTON_DISABLE_BACKGROUND|
```

```
    DM_BUTTON_BAR_LEFT_ARROW|DM_
    BUTTON_BAR_RIGHT_ARROW,
    MMI_SOFTKEY_WIDTH
  };
```

DM_SCR_BG 如果没有指明具体的属性，系统进行默认绘制。其余的事情，就是在代码中的具体实现了，我们将模板取名为 ShowCategory669Screen。

在 FirstAppSrc.c 文件中定义 ShowCategory669Screen 模板函数。

```
void ShowCategory669Screen(U16 left_softkey,U16 left_softkey_icon,
                           U16 right_softkey,U16 right_softkey_icon,U8* history_buffer)
  {
    /*----------------------------------------------------------------*/
    /*                                                                */
    /* Local Variables                                                */
    /*----------------------------------------------------------------*/
    dm_data_struct   dm_data;
    //4 个按键变量
    U16 btn_id6,btn_id8,btn_about, btn_more;
    /*----------------------------------------------------------------*/
    /*                                                                */
    /* Code Body                                                      */
    /*----------------------------------------------------------------*/
    dm_reset_context();
    UI_UNUSED_PARAMETER(history_buffer);
     entry_full_screen();
     clear_screen();

    gdi_layer_lock_frame_buffer();

    change_left_softkey(left_softkey,left_softkey_icon);
    change_right_softkey(right_softkey,right_softkey_icon);
    SetupCategoryKeyHandlers();

    dm_data.s32flags=0;
    if((left_softkey==0)&&(left_softkey_icon==0)
   &&(right_softkey==0)&&(right_softkey_icon==0))
      dm_data.s32flags |= DM_NO_SOFTKEY;
    else
      dm_data.s32flags &= ~DM_NO_SOFTKEY;
    gdi_layer_unlock_frame_buffer();
    #ifdef _MMI_TOUCH_SCREEN_
     //dm_setup_and_draw_fixed_list(&UICtrlAccessPtr_p);
        //设置每一个按钮的 up 状态和 down 状态
          btn_id6= dm_add_button(NULL,
           get_image(IMG_FIRSTAPP_BTN06),
           get_image(IMG_FIRSTAPP_BTN060),NULL);
  btn_id8 = dm_add_button(NULL,
           get_image(IMG_FIRSTAPP_BTN08),
           get_image(IMG_FIRSTAPP_BTN080),NULL);
  btn_about = dm_add_button(NULL, get_image(IMG_FIRSTAPP_ABOUT01),
           get_image(IMG_FIRSTAPP_ABOUT010), NULL);
  btn_more = dm_add_button(NULL,
           get_image(IMG_FIRSTAPP_MORE01),
           get_image(IMG_FIRSTAPP_MORE010),NULL);
  ExitCategoryFunction=UI_dummy_function;
#endif
```

```
dm_setup_category_functions(dm_redraw_category_screen,dm_get_category_history,dm_get_categ
ory_history_size);

    dm_data.s32ScrId = (S32)GetActiveScreenId();
    dm_data.s32CatId = MMI_CATEGORY669_ID;
    dm_data.s32flags |= DM_CLEAR_SCREEN_BACKGROUND;
    dm_setup_data(&dm_data);
    dm_redraw_category_screen();

  }
```

至此，模板函数就定义完毕，然后就可以放在 EntryFirstAppScreen 主函数中进行调用了。

```
void EntryFirstAppScreen(void)
  {
   EntryNewScreen(SCR_FIRSTAPP_MAIN,ExitFirstAppScreen,NULL,NULL);
   ShowCategory669Screen(STR_GLOBAL_OK, IMG_GLOBAL_OK,
                         STR_GLOBAL_BACK, IMG_GLOBAL_BACK,
                         NULL);
  SetRightSoftkeyFunction(GoBackHistory,KEY_EVENT_UP);
}
```

保存修改，运行模拟器，效果如图 12.5 所示。

读者可以对比之前我们绘制的按钮，是不是效果完全不同，显得更有质感了？

当触摸笔单击每个按钮时，都有对应的按下效果，如图 12.6 所示。

图 12.5　按钮效果

图 12.6　按钮按下的效果

一般地，用户单击一个按钮，就有相应的事件发生。比如单击播放按钮，播放音乐；单击照相按钮，打开摄像头等。在这里，我们也要讲解如何将事件与按钮进行关联。

为了方便讲述，我们决定在单击按钮时，在屏幕上绘制一个字符串，以此证明按钮的单击动作。

首先，定义一个字符串绘制函数。

```
void DisplayButtonContent(void)
{
   gui_move_text_cursor(90,120);
   gui_set_text_color(gui_color(255,0,0));
   gui_print_text(get_string(STR_FIRSTAPP_HELLOMTK));
   gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

然后，在模板函数 ShowCategory669Screen 中增加事件响应代码。

```
void ShowCategory669Screen(U16 left_softkey,U16 left_softkey_icon,
                           U16 right_softkey,U16 right_softkey_icon,U8* history_buffer)
{
  …

  dm_setup_category_functions(dm_redraw_category_screen,dm_get_category_history,
                              dm_get_category_history_size);


#if defined(_MMI_TOUCH_SCREEN_)
  dm_register_button_functions(btn_id6, KEY_EVENT_UP,
            NULL);
  dm_register_button_functions(btn_id8, KEY_EVENT_UP,
            NULL);
  //单击按钮 About，执行函数 DisplayButtonContent
  dm_register_button_functions(btn_about, KEY_EVENT_UP,
            DisplayButtonContent);
  dm_register_button_functions(btn_more, KEY_EVENT_UP,
            NULL);
#endif

  dm_data.s32ScrId = (S32)GetActiveScreenId();
  dm_data.s32CatId = MMI_CATEGORY669_ID;
  dm_data.s32flags |= DM_CLEAR_SCREEN_BACKGROUND;
  dm_setup_data(&dm_data);
  dm_redraw_category_screen(); //把控件等绘制出来

}
```

dm_register_button_functions 函数将按钮需要响应的函数绑定。其原型如下：

```
void dm_register_button_functions(U16 button_index, U16 key_type, FuncPtr f)
```

参数 button_index：按钮的编号。也就是先前我们声明的按钮变量。

参数 KEY_TYPE：按键的方式。判断是按下 KEY_EVENT_UP 还是松开 KEY_EVENT_DOWN 的时候执行。

参数 f：按钮将要执行的函数地址。这是一个函数指针。

现在，保存修改，运行模拟器。当我们单击按钮 About 的时候，屏幕上就会绘制一个（模拟器上显示为红色）字符串“HelloMTK”，如图 12.7 所示。

图 12.7　响应事件

注 意

上面代码中，其余 3 个按钮我们没有注册执行函数，留给读者自己试验。比如我们要单击 More 按钮，在屏幕上显示一张图片，该怎么做？

结语：

本章，我们对高级屏幕模板的内部实现机制做了一个梳理，讲解了模板 ID，模板组件以及这些组件的绘制方式，详细描述了如何自定义一个我们需要的模板。在最后，我们还通过一个例子讲解了如何实现绚丽的按钮效果。读者可以在我们讲解的基础上，完成本章留给大家的小问题，融会贯通，更好地掌握屏幕模板的原理，并在以后的开发中熟练运用。

13 多国语言

使用手机的用户，不可能只局限于某一个国家或地区，这就要求手机设计公司提供多国语言包，使得欧洲的用户可以正常输入英语，拉丁美洲的用户可以输入西班牙或葡萄牙语。

MTK 解决方案当然不会遗漏这一重要的部分。MTK 默认提供的英语输入法和字库可以作为开发人员移植多国语言的实例。

13.1 添加新的语言

13.1.1 前期准备

在手机上添加一个新的语言，需要有字库和输入法，两者缺一不可。

但是前期准备工作还必须完成如下任务。

（1）确定该语言输入法的输入方式，比如是拼音还是其他方式。

（2）找到对应的输入法的所有字符，并且导入到手机中。

（3）给这些字符建立一个索引表。

（4）把这个字符索引表和我们的输入法对应起来。

以上这些都是我们要做的前期工作。

13.1.2 MTK 支持该语言

我们通过中文输入法来进行讲解，修改步骤如下。

1. 定义宏开关

打开文件 MMI_featuresPLUTO.h（plutommi\customer\custresource\pluto_mmi）：

```
#define _MMI_LANG_SM_CHINESE_
```

2. 修改字库资源文件

打开文件 FontRes.c（plutommi\customer\lcdresource\mainlcd240x320）：

```
#ifdef _MMI_16X16_CHINESE_FONT_
#include "L_Pluto_16x16_ch.h"      //引入字库头文件
```

```
#else
……
const RangeData XXX_RangeData[a]={…} //XXX 为具体的语言编码，如 GB2312
…
const U16 mtk_gMaxDeployedLangs = 1;
…
#if defined(_MMI_LANG_SM_CHINESE_)
sFontFamily pluto_ch_standardFamily =
{6,{&Pluto_ch, &Pluto_ch,
&Pluto_ch, &Pluto_ch, &Pluto_ch,&gVirtualKeyboardFont}};
#endif
…
#if defined(_MMI_LANG_SM_CHINESE_)
{
    {(char)0x80,(char)0x7B,(char)0x53,(char)0x4F,(char)0x2D,
     (char)0x4E,(char)0x87,(char)0x65,(char)0x00,(char)0x00},//Language Name
     "*#0086#",//SSC string name
     0,
     { &pluto_ch_standardFamily },
     "zh"
},
#endif
```

13.1.3 MTK 不支持该语言

如果 MTK 不支持新加的语言，那么可以分为 3 种情况来考虑。

1. 有字库和输入法

（1）添加字库（参照上面），设定对应的开关。

一般包含两个文件：FontRes.c 和字库对应的头文件。

（2）增加输入法（可参考已有的输入法），设定对应的开关。

· 输入法存放位置

```
\ MTK6226\custom_inputMethod\
```

比如添加了 Zi 输入法，那么路径为：\ MTK6226\Zi\。

· 打开/关闭输入法

请打开文件 Wgui_inputs.c，这个文件中定义了大量的语言宏开关。如：

```
_MMI_LANG_SPANISH_              //西班牙语
_MMI_LANG_FINNISH_              //芬兰语
_MMI_LANG_GERMAN_               //德语
```

MTK 就是通过这些开关来确定当前版本中使用的语言的。

· 输入法的 make 文件

```
\ MTK6226\make\ custom_inputMethod \
```

同理，如果添加了 Zi 输入法，那么路径为：\ MTK6226\make\ Zi \。

（3）修改 MTK 默认的输入法，我们要将默认的输入法修改为拼音。

在\ MTK6226\custom\app\MCUTEL26_06B _BB\nvram_cust_pack.c 文件中进行修改。

```
kal_uint8 const NVRAM_CACHE_SHORT_DEFAULT[NVRAM_CACHE_SIZE] =
{
```

```
…
0xFF, 0xFF, /* NVRAM_SETTING_CTR_TIME, */
#ifndef _N73_STYLE_      //修改为拼音输入法
  0x0D, 0x00, /* NVRAM_SETTING_PREFER_INPUT_METHOD, */
#else
  0x0D,0x00,   //INPUT_TYPE_SM_PINYIN
#endif
0x01, 0x00, /* RESTORE_DEFAULT_CURRENT_SCREENSVER_ID, */
0x01, 0x00, /* RESTORE_DEFAULT_FUNANDGAMES_SETWALLPAPER, */
#ifndef _N73_STYLE_      //修改为拼音输入法
0x0D, 0x00, /* RESTORE_PREFER_INPUT_METHOD, */
#else
0x0D,0x00,  //INPUT_TYPE_SM_PINYIN
#endif
0xFF, 0xFF, /* NVRAM_GX_NINJA_SCORE, */
…
```

（4）添加该语言对应的用于手机显示的字符串。

添加到 Ref_list.txt 文件中，第 4 章已经讲解过如何添加字符串。开关定义在 MMI_featurePLUT0.h 文件中。

2. 有字库没有输入法

方法和前一种情况一样，惟一不同的是需要自己设计和实现输入法。

3. 没有字库没有输入法

（1）自己制作字库

常用的就是用字库编辑器，用打点的方式描点，再导出字库。一般格式为 pdf 或者 bdf。bdf 可以通过使用 MCT 工具转换出字库文件（即最终我们需要的字库文件）。

（2）添加字库（参照上面），设定对应的开关。

一般包含两个文件：FontRes.c 和字库对应的头文件。

（3）自己实现该语言对应的输入法并且添加该输入法到 MTK 工程（可参考已有的输入法），设定对应的开关。

如下步骤和第一种情况一致，不再赘述。

13.1.4 自己实现输入法的总结

如果自定义的输入法的编码系统没有定义，则：

（1）需要码库，完成编码方案和编码存储，可以是矢量库或点阵字库；

（2）调用编码的算法及程序；

（3）应用软件显示字体及打印的程序。

如果存在系统字库，则只需要完成编码算法及程序即可。

13.2 移植多国语言

移植多国语言主要牵涉到对 mmi_features.h（整个工程的宏控定义文件）和 fontres.c（字体资源文件）文件的修改，并且要添加相应的字库文件。

13.2.1 语言宏控的修改

在 mmi_features.h 文件中，用于控制各种语言的开关就是前一节我们已经提到过的类似于_MMI_LANG_ENGLISH_的宏。

打开该宏，那么工程就支持该种语言。有时侯，需要同时打开与该语言相关的一些宏，比如阿拉伯语的编码模式、汉语的编码模式（当然这个编码模式一般都是设置好了的，但也有些许例外，根据实际情况酌情处理）。

13.2.2 字库文件的添加

在 MTK 中，字库主要是两个文件，一个是*.h 文件，还有一个是对应的 fontres.c 文件。

1. 字库的获取

有两种方式可以获取字库。

（1）直接去网上找对应语言的字库文件。

（2）自己利用 MTK 公司所提供的 MCT 6.0 工具来制作字库。不过 MCT 6.0 制作字库需要.bdf 的点阵字库源文件，而网上能找的基本上都是.ttf 格式的矢量字库文件，这就存在一个需要将 TTF 文件转换成 BDF 文件的第三方软件，并且转换的好坏将直接影响到最终用 MCT 6.0 生成的字库文件。

2. 字库的使用

首先我们查看一下 MTK 版本中自带的英文字库 L_1_Pluto_Large.h 是存放在什么位置的，然后将我们的*.h 文件也放入相同的文件夹下；接下来需要将我们的 fontres.c 文件与 MTK 中原有的 fontres.c 文件合并为一个 fontres.c 文件。合并的方法就参考原有的 fontres.c 文件中英文字库的代码分布方式，将我们的 fontres.c 文件里面的代码利用语言宏控拷入 MTK 原有的 fontres.c 文件中。

13.2.3 修改默认显示语言

请参考 13.1.3 小节。

13.2.4 移植 Zi 输入法

1. 打开 Zi 开关

（1）文件：..\make\ MCUTEL26_06B_W07_GPRS.mak。

修改输入法为 MMI_ZI：

```
...
IRDA_SUPPORT          =  FALSE          # IRDA support feature
INPUT_METHOD          =  MMI_ZI         # Zi 输入法
DICT_SUPPORT          =    NONE         # Dictionary library: NONE, MMI_GV
...
```

（2）在 MMI_features.h 文件中打开相应的 Zi 语言输入法宏开关，如_MMI_ZI_PRC_ENGLISH_。

```
#if !defined(_MMI_TOUCH_SCREEN_) || defined(_N73_STYLE_)
    #if defined(_MMI_LANG_ENGLISH_)
```

```
        #define _MMI_ZI_PRC_ENGLISH_
  /* Detail Info: [turn on Zi English inputs]*/
     #endif /* _MMI_LANG_ENGLISH_ */
#endif
```

2. 添加 Zi 代码

将 Zi 目录添加到工程根目录，如下所示：

```
\MTK6226\Zi
```

Zi 目录中包含所有的源代码，其含有两个文件夹：Include 和 Src。

3. 添加 Zi 的编译选项

工程文件要最终通过 Makefile 编译成目标文件，每个模块的编译路径都写入该模块的编译目录中。

```
\MTK6226\make\Zi\
```

包含文件有 zi.def、zi.inc、zi.lis 和 zi.pth。

编译过程中，还有其他的模块用到了 Zi 的文件，因此必须把 Zi 的头文件目录 Zi\include 包含到下面几个文件中，否则编译时报错，提示“不能打开×××.h 文件”。在作者的编译过程中，如下模块会提示找不到头文件，因此，请将目录包含到如下文件中，读者可以根据具体版本进行修改。

- make\gdi_arm\gdi_arm.inc
- make\mmiresource\mmiresource.inc
- make\vendorapp\vendorapp.inc
- make\plutommi\conn_app.inc
- make\plutommi\inet_app.inc
- make\plutommi\media_app.inc
- make\plutommi\mmi_framework.inc

4. 添加 Zi 输入法库文件

在 make\option.mak 文件里面，查找 MMI_ZI，找到类似如下代码：

```
 ifeq ($(strip $(INPUT_METHOD)),MMI_ZI)
     CUSTOM_COMMINC  +=
vendor\InputMethod\ZI\project\$(strip $(INPUT_METHOD_BASE))\v6_official\inc
     COMPOBJS   +=
vendor\InputMethod\ZI\project\$(strip
$(INPUT_METHOD_BASE))\v6_official\lib\zi8clib.a
CUS_REL_OBJ_LIST     +=
vendor\InputMethod\ZI\project\$(strip
$(INPUT_METHOD_BASE))\v6_official\lib\zi8clib.a
     CUS_INPUTMETHOD_BASE     =
 vendor\InputMethod\ZI\project\$(strip $(INPUT_METHOD_BASE))\v6_official
 endif
```

从上面的脚本中可以看到，Zi 输入法的库文件的存放位置：

```
\MTK6226\vendor\InputMethod\ZI\project\plutommi\v6_official\lib
```

头文件存放位置：

```
\MTK6226\vendor\InputMethod\ZI\project\plutommi\v6_official\inc
```

因此我们开始添加 Zi 输入法的库文件和头文件到上面的目录，如果目录不存在，则自己创建。其中，头文件放在 inc 文件夹下，.lib 和.a 文件放在 lib 文件夹下。

当然，版本不同，库文件和头文件存放的路径也许不同，只要参照上面的步骤即可。上面的展示是以 06B 版本为基础的。

提 示

在第 3 章，我们讲过如何配置第三方库。如果在移植过程中，读者还有不熟悉的地方，请回顾第 3 章所讲的内容。

13.2.5 移植手写汉王（hanwang）输入法

1. 打开 hanwang 开关

文件：\MCUTEL26_06B_W07_GPRS.mak。

修改手写输入法为 MMI_HANWANG：

```
…
IRDA_SUPPORT     =  FALSE        # IRDA support feature
INPUT_METHOD     =    MMI_ZI     # Input methods: NONE, MMI_T9, or MMI_ZI
HAND_WRITING     =  MMI_HANWANG # MMI_HANWANG,  MMI_PENPOWER
DICT_SUPPORT     =    NONE       # Dictionary library: NONE, MMI_GV,
…
```

2. 添加 hanwang 代码

```
\MTK6226\hanwang
```

将 hanwang 目录添加到工程根目录。

3. 添加 hanwang 的编译选项

工程文件要最终通过 Makefile 编译成目标文件，每个模块的编译路径都写入该模块的编译目录中。

```
\MTK6226\make\hanwang\
```

包含文件有 hanwang.def，hanwang.inc，hanwang.lis 和 hanwang.pth。

4. 添加 hanwang 输入法库文件

在 make\option.mak 文件里面，查找 MMI_HANWANG，找到类似如下代码：

```
ifeq ($(strip $(HAND_WRITING)),MMI_HANWANG)
  COM_DEFS     += _MMI_HANWANG_
  COMPLIST     += hanwang
  CUS_REL_SRC_COMP += hanwang
  COMPOBJS     += vendor\handwriting\hanwang\v1_official\lib\hw.a
  CUS_REL_OBJ_LIST    += vendor\handwriting\hanwang\v1_official\lib\hw.a
  CUS_REL_OBJ_LIST    += vendor\handwriting\hanwang\v1_official\lib\hw.lib
  CUS_REL_OBJ_LIST    += vendor\handwriting\hanwang\v1_official\lib\hw.dll
  CUS_REL_OBJ_LIST    += vendor\handwriting\hanwang\v1_official\lib\hws.lib
  CUS_REL_OBJ_LIST    += vendor\handwriting\hanwang\v1_official\inc
  CUSTOM_COMMINC  +=vendor\handwriting\hanwang\v1_official\inc
endif
```

从上面的脚本中可以看到，hanwang 输入法的库文件的存放位置：

```
\MTK6226vendor\ handwriting\hanwang\v1_official\lib
```

头文件存放位置：

```
\MTK6226vendor \vendor\ handwriting\hanwang\v1_official\inc
```

因此我们开始添加 hanwang 输入法的库文件和头文件到上面的目录中，如果目录不存在，则自己创建。其中，头文件放在 inc 文件夹下，.lib 和.a 文件放在 lib 文件夹下。

> *注　意*
>
> 汉王手写输入模块，是必须取得原开发商授权的软件。

13.2.6　打开 MTK 自带输入法支持

在 mmi_features.h 文件里面打开形如__MMI_MULTITAP_THAI__的宏开关，每种语言都对应着一个宏开关，根据需要打开。

13.3　虚拟键盘

1．添加虚拟键盘布局

修改 Wgui_virtual_keyboard.c 文件，其中有写好的标准键盘布局模式，可以参考其代码自己写其他语言的虚拟键盘布局模式。另外我们也可以比较 MTK 原始的代码，将对应的虚拟键盘布局代码复制过来即可。注意，别遗漏一些部分，这个文件里面有好几处都需要添加代码，自己搜语言宏控来加代码就可以。

2．虚拟键盘的显示与隐藏

虚拟键盘的显示与隐藏主要体现在输入法的切换方面，因此需要修改代码的地方在文件 EditorPen.c 里面。主要修改的函数是 mmi_pen_editor_vk_show 和 mmi_pen_editor_set_virtual_keyboard_by_input_mode。

大家按照函数里面已经写好的代码模式很容易就可以加上其他语言虚拟键盘的支持代码。

13.4　输入法提示符的显示

每当我们切换一个输入法的时候，左上角就会显示当前输入法的名称。该功能在 wgui_categories_inputs.c 文件中的 wgui_get_inputbox_information_icon 和 wgui_redraw_input_information_bar 函数中实现。逻辑很简单，读者可以通过调试查看其具体步骤。

结语：

本章，我们对如何移植多国语言，做了一个比较详细的讲解。读者学完本章，可以加深对库移植操作的理解，并能够在实际项目中做好相应的语言移植工作。

14 SOCKET

在本章中，我们将介绍 MTK 平台中通过 Socket 进行网络通信的方法。

熟悉 PC 网络开发的朋友，一定对套接字接口非常了解。伯克利版本的 UNIX 系统最早引入了套接字这一概念。一台机器上的进行可以使用套接字和另外一台机器上的进程进行通信。

MTK 为应用程序提供了一套基于标准套接字的 API，在代码里简称为 SOC。应用程序调用 SOC 提供的接口发送链接请求，SOC 在处理后会以异步的方式通知应用程序，同时，在底层的调用中，SOC 还提供接口供 TCP/IP、PPP 等调用。读者可以通过示意图 14.1，仔细体会。

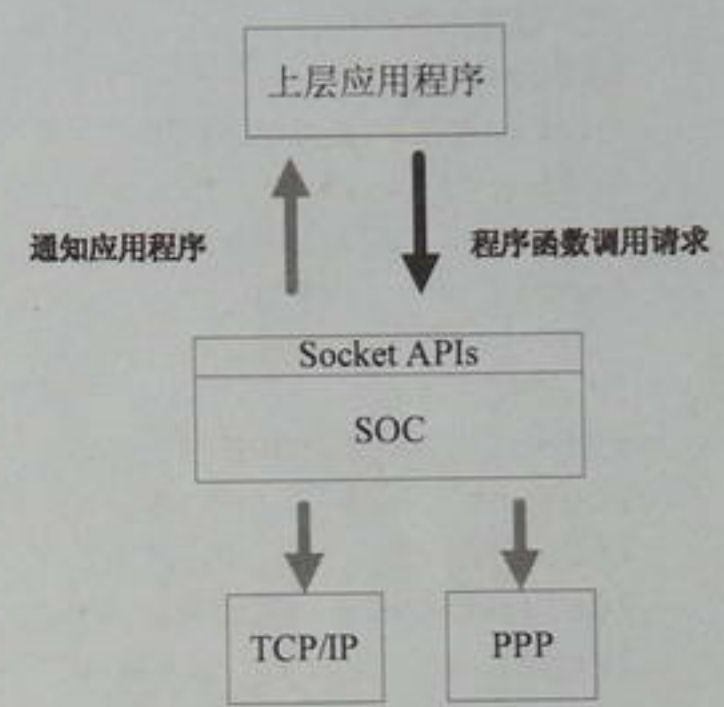


图 14.1 SOC 基本结构

> **说 明**
>
> （PPP：Point to Point Protocol）点对点协议（PPP）为在点对点连接上传输多协议数据包提供了一个标准方法。

本章，我们不会详细地介绍网络链接过程中所有的内容，这也不是本书所涉及的范围，我们将结合 MTK 源代码中工程模式中的 SOC 例子来讲解 MTK 平台基本的网络开发过程。读者通过这个例子，应该可以进行基本的网络通信程序开发。

在继续下面的内容之前，我们先介绍一下什么是 CMNET 和 CMWAP。

CMNET 和 CMWAP 都是手机上网使用的接入点的名称。通过 CMNET 可以获得完全的 Internet 访问权，通过 CMWAP 只能访问 WAP 网站，不过 CMWAP 使用 HTTP 代理协议和 WAP 网关协议可以访问到 Internet，而 CMNET 则适用于所有协议，它也是标准的 TCP/IP 协议。CMWAP 的端口为 80、8080（HTTP 代理）、9201（WAP 网关协议）。

总之，CMWAP 相当于局域网通过路由器上网，CMNET 相当于直接给你一个外网 IP 地址上网。

14.1 SOC 定义浅析

在 MTK 的 SOC 模块中，所有关于套接字的声明都位于文件 soc_api.h 中。打开上述文件，我们可

以看到其内部定义了大量的枚举、常量、Socket 相关的函数原型等。让我们来看一下该文件的主要部分。

14.1.1 SOCKET 宏常量

在常量区集中了多种类型的 Socket 宏定义，用于区别套接字的使用方式。

```
/*==============================
 * Constants
 *==============================*/

/* couting total socket numbers by compile options */
/* Define number of socket used by applications */
/* WARNING: THESE CONSTANTS SHOULD BE CONFORM WITH FLOW CONTROL
POOL BLOCKS */
/* TFTP */
#ifdef _TFTP_
#define TFTP_SOCKETS    0   /* 1, use spared */
#else /* !_TFTP_ */
#define TFTP_SOCKETS    0
#endif /* !_TFTP_ */

/* DHCP */
#ifdef _WIFI_SUPPORT_
#define DHCP_SOCKETS   0    /* 1, use spared */
#else /* !_WIFI_SUPPORT_ */
#define DHCP_SOCKETS   0
#endif /* !_WIFI_SUPPORT_ */

/* IPERF, iperf will block the screen, use the spared sockets */

/* DNS */
#ifdef _TCPIP_
#define DNS_SOCKETS    2
#else /* !_TCPIP_ */
#define DNS_SOCKETS    0
#endif /* !_TCPIP_ */
…
```

14.1.2 套接字类型

```
/*
 * Socket Type
 */
typedef enum
{
    SOCK_STREAM = 0,  /* stream socket, TCP */
    SOCK_DGRAM,       /* datagram socket, UDP */
    SOCK_SMS,         /* SMS bearer */
    SOCK_RAW          /* raw socket */
} socket_type_enum;
```

· 流式套接字（SOCK_STREAM）

提供了一个面向连接、可靠的数据传输服务，数据无差错、无重复地发送，且按发送顺序接收。内设流量控制，避免数据流超限；数据被看做是字节流，无长度限制。文件传送协议（FTP）即使用流

式套接字。

· 数据报式套接字（SOCK_DGRAM）

提供了一个无连接服务。数据包以独立包形式被发送，不提供无错保证，数据可能丢失或重复，并且接收顺序混乱。网络文件系统（NFS）使用数据报式套接字。

· 原始式套接字（SOCK_RAW）

该接口允许对较低层协议如 IP、ICMP 直接访问。常用于检验新的协议实现或访问现有服务中配置的新设备。

如果你要保证将你的数据正确地传给客户，就需要可靠的连接，就应用 SOCK_STREAM，即 TCP。像腾讯 OICQ 这样的不可靠的连接，就是用 SOCK_DGRAM，即 UDP。

14.1.3 套接字事件

```
/*
 * event
 */
typedef enum
{
   SOC_READ    = 0x01,  /* Notify for read */
   SOC_WRITE   = 0x02,  /* Notify for write */
   SOC_ACCEPT  = 0x04,  /* Notify for accept */
   SOC_CONNECT = 0x08,  /* Notify for connect */
   SOC_CLOSE   = 0x10   /* Notify for close */
} soc_event_enum;
```

上面的这些事件 ID，在应用程序监视消息 MSG_ID_APP_SOC_NOTIFY_IND 的过程中会频繁地使用到。特别是在连接网络后，会以异步方式触发上述消息，并通知注册过该消息的响应函数进行事件处理。

14.1.4 封装的套接字函数

```
/*==================================================
 *
 * Socket APIs prototypes
 *
 *==================================================*/
extern kal_int8 soc_create(kal_uint8          domain,
                           socket_type_enum   type,
                           kal_uint8          protocol,
                           module_type        mod_id,
                           kal_uint32         nwk_account_id);

extern kal_int8 soc_close(kal_int8 s);

extern kal_int8 soc_bind(kal_int8 s, sockaddr_struct *addr);

extern kal_int8 soc_listen(kal_int8 s, kal_uint8 backlog);

extern kal_int8 soc_accept(kal_int8 s, sockaddr_struct *addr);
```

上面的套接字函数在模拟器和真机中都是一样的名字，只是底层实现方式不同而已。同时可以看到在 MTK 中的套接字函数名称和标准的 SOCKET 差不多，这也是为了方便开发人员理解和使用。具体的使用我们随后进行讲解。

14.2 SOC 实例分析

MTK 在其工程模式中有一个 Socket 的示例（DEMO），这个示例包含了各种类型的连接方式和套接字的基本使用方法。接下来的章节，我们就结合这个示例讲解如何通过 Socket 连接网络。

14.2.1 进入工程模式

在待机界面通过键盘输入“*#3646633#”即可进入工程模式（EngineerMode），如图 14.2 所示。

注：每款机子可能不一样，这个定在 (nstRes Ref.h 中

#define SSC_ENGINEERING_MODE

图 14.2 进入工程模式

14.2.2 输入 URL 地址

选择菜单项 Socket Test，进入套接字测试列表。单击“HTTP Get”菜单项，会进入一个输入界面，用于输入 URL 地址，如图 14.3 所示。

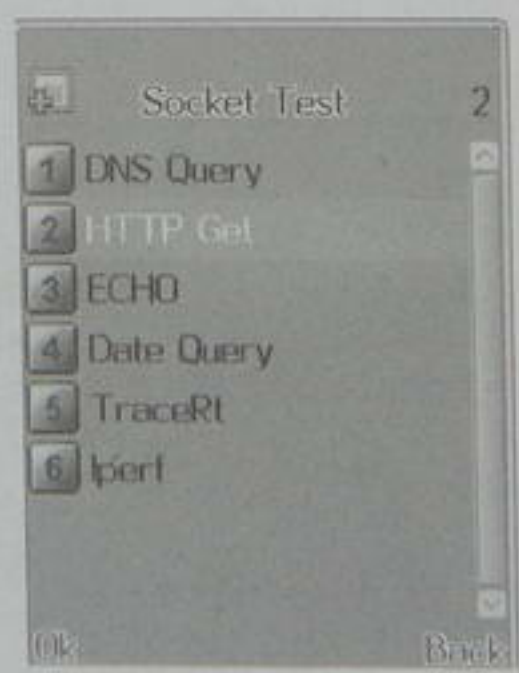


图 14.3 选择 HTTP Get

上述行为都是通过一个函数 HighlightEmHTTPGet 进行绑定的。

```
void HighlightEmHTTPGet(void)
{
    currentSocketAppType = HTTP;

    /* Register function for left soft key */
    SetLeftSoftkeyFunction(EntryEmHTTPGet, KEY_EVENT_UP);

    /* Register function for right soft key */
    SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
}
```

在这个函数中，设置了当前套接字的应用类型为 HTTP。

按下左键时，进入函数 EntryEmHTTPGet，程序将提供一个输入框，用于获取用户输入的网络地址，获取的地址放在 EmSocketInputBuff 变量中。需要注意的是，这时候地址的编码为 UNICODE。

```
void EntryEmHTTPGet(void)
{
    //首先将表示套接字状态的变量 EmSocketResult 设置为 EN_SOC_SUCCESS
    EmSocketInitResult();
    …
    if (history_buffer == NULL)
    {
        //EmSocketInputBuff 在后期将用于表示网络地址，因此，调用函数 kal_wsprintf
        //给用户输入的原始数据增加一个前缀“http://”。如果用户输入 www.sohu.com
        //该函数就会将其合成为“http://www.sohu.com”
        kal_wsprintf((U16*) EmSocketInputBuff, "%s", "http://");
    }
    ShowCategory5Screen(
        EM_SOCKET_HTTP_URL_TEXT,
        (U16) NULL,
        STR_GLOBAL_OPTIONS,
        (U16) NULL,
        STR_GLOBAL_BACK,
        (U16) NULL,
        INPUT_TYPE_ALPHANUMERIC_SENTENCECASE,
        (PU8) EmSocketInputBuff,
        EM_SOCKET_MAX_INPUT_LEN,
        history_buffer);
    //左键绑定设置界面
    SetLeftSoftkeyFunction(EntryEmHTTPOption, KEY_EVENT_UP);
}
```

在如图 14.4 所示的界面中，输入完毕后，在用户按下左键时，即会进入选项（Option）界面。

图 14.4　输入 URL

```
void EntryEmHTTPOption(void)
{
EntryNewScreen(EM_SOCKET_HTTP_OPTION_SCR, NULL, EntryEmHTTPOption, NULL);

    guiBuffer = GetCurrGuiBuffer(EM_SOCKET_HTTP_OPTION_SCR);
    numItems = GetNumOfChild(EM_SOCKET_HTTP_OPTION_MENUID);
    GetSequenceStringIds(EM_SOCKET_HTTP_OPTION_MENUID, nStrItemList);
    SetParentHandler(EM_SOCKET_HTTP_OPTION_MENUID);
    RegisterHighlightHandler(ExecuteCurrHiliteHandler);
    ShowCategory52Screen(
        STR_GLOBAL_OPTIONS,
        0,
        STR_GLOBAL_OK,
        IMG_GLOBAL_OK,
        STR_GLOBAL_BACK,
        IMG_GLOBAL_BACK,
        numItems,
        nStrItemList,
        (U16*) gIndexIconsImageList,
        NULL,
        0,
        0,
        guiBuffer);
}
```

该界面实际定义了一个列表，该列表总共有两个菜单项，如图 14.5 所示。

图 14.5　Option 界面

第 1 个菜单项绑定了一个输入界面，用户通过执行 HighlightEmHTTPGetInputMethod，可以切换输入法。

```
SetHiliteHandler(EM_SOCKET_HTTP_INPUT_METHOD_MENUID,
                 HighlightEmHTTPGetInputMethod);
```

第 2 个菜单项绑定了输入完成后将要执行的函数。就是说前面网址输入完毕之后，单击该菜单项就会调用函数 EntryEmSocketInProgress，代码如下：

```
SetHiliteHandler(EM_SOCKET_HTTP_DONE_MENUID, HighlightEmHTTPGetDone);
```

初看函数 EntryEmHTTPOption，似乎只是在显示一个列表屏幕，并没有看见上面提到的菜单项，以及菜单项所要执行的函数。其实这些函数都通过上面的 SetHiliteHandler 在初始化的时候进行了绑定。

因为在 MTK 中，列表菜单项的行为可以在函数中动态绑定，也可以在初始化时通过菜单 ID 绑定。

当前屏幕也是一个特殊的菜单，ID 是 EM_SOCKET_HTTP_OPTION_MENUID。这个 ID 下还有两个子菜单，其中一个就是 EM_SOCKET_HTTP_DONE_MENUID。

```
ADD_APPLICATION_MENUITEM((EM_SOCKET_HTTP_OPTION_MENUID, 0, 2,
                          EM_SOCKET_HTTP_DONE_MENUID,
                          EM_SOCKET_HTTP_INPUT_METHOD_MENUID,
                          SHOW, NONMOVEABLE, DISP_LIST,
                STR_GLOBAL_OPTIONS, EM_SOCKET_HTTP_OPTION_ICON));
```

上述定义都实现于 Res_EngineerMode.c 文件，读者可以仔细查看。

这样，当用户输入完毕，切换到 HTTPGetDone 菜单项，再按下左键时，就会进入绑定的函数：EntryEmSocketInProgress。

```
void HighlightEmHTTPGetDone(void)
{

    /* Register function for left soft key */
    SetLeftSoftkeyFunction(EntryEmSocketInProgress, KEY_EVENT_UP);

    /* Register function for right soft key */
    SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
}
```

跟踪 EntryEmSocketInProgress 函数进入，即可看到该函数实际上是一个显示进度的屏幕（如图 14.6），同时后台开始处理链接事件。

处理完后显示的是请求的页面的文本显示

图 14.6 显示进度

当然 UI 不是这里关注的重点，我们关心的是其后台连接的实现。

```
void EntryEmSocketInProgress(void)
{
    EntryNewScreen(EM_SOCKET_PROGRESS_SCR,NULL,EntryEmSocketInProgress,NULL);
    guiBuffer = GetCurrGuiBuffer(EM_SOCKET_PROGRESS_SCR);
     if (guiBuffer == NULL)
     {
         mmi_soc_demo_app_request_abort();
         if (currentSocketAppType != TRACERT)
```

```
        {
                EmSocketSendReq();//该函数负责发送 Socket 请求
        }
        else
        {
                ...
        }
}
```

14.2.3 Socket 初始化

在上面的函数 EmSocketSendReq 中：

```
void EmSocketSendReq(void)
{
#ifdef MMI_ON_HARDWARE_P
    U8 emSocketAccountId = custom_get_csd_profile_num();
#else
    U8 emSocketAccountId = 5; //CMNET 设置为 5
#endif
if (currentSocketAppType == HTTP || currentSocketAppType == DNS
   || currentSocketAppType == ECHO || currentSocketAppType == TRACERT)
 {
        //将存储 URL 地址的变量 EmSocketInputBuff 转换为 ANSII 编码。EmSocketInputdata
        //就获得了正确编码格式的网络地址，后面可以传入 mmi_soc_demo_app_request
         UnicodeToAnsii(emSocketInputdata, EmSocketInputBuff);
 }
#ifndef MMI_ON_HARDWARE_P
    soc_init_win32();
#endif
...
else if (currentSocketAppType == HTTP)
    {
         emSocketTmpResult = mmi_soc_demo_app_request(
                                HTTP,
                                emSocketAccountId,
                                NULL,
                                emSocketInputdata,
                                strlen(emSocketInputdata),
                                NULL,
                                0,
                                EmSocketRsp);
 }
 ...
}
```

如果当前是在模拟器而非真机上操作，则需要调用函数 soc_init_win32 来初始化基于 Windows 平台的套接字，MTK 会调用 Windows 的套接字来模拟整个连接过程。

说　明

真机或者模拟器，代码只是在 SOC 层以下有所区分。上层开发的 Socket 接口和消息都是统一的。实现的方式一个是 Winsock，一个是真机上的具体库。

在模拟器测试的时候，一定要在创建套接字之前调用函数 soc_init_win32 进行初始化。然后，判断当前的套接字应用类型是哪一种，由于我们只讲 HTTP，故省略了 DNS、ECHO 等方式。

不一定，我就没调用，然后就连网

14.2.4 请求准备工作

函数 mmi_soc_demo_app_request 的原型：

```
int mmi_soc_demo_app_request(
        mmi_soc_demo_app_enum  app_id,
        int account_id,
        char server_ip_address[4],
        char *url,
        int url_len,
        char *echo_text,
        int echo_text_len,
        mmi_soc_demo_app_rsp_t  callback)
```

参数如下。

app_id：标识当前的套接字应用类型，可以为 DNS、HTTP、ECHO 等。

Account_id：联网账号 ID。

server_ip_address：服务器 IP 地址。

url：统一资源定位符，即联网地址，如：http://www.sohu.com。

callback：回调函数，用于应用程序得到事件响应后的处理，这里的实参是函数 EmSocketRsp。

```
void EmSocketRsp(mmi_soc_demo_app_enum app_id_enum, int result,
                 char *response, int response_len)
{
if (currScrnID == EM_SOCKET_PROGRESS_SCR || currScrnID
               == EM_SOCKET_TRACERT_PROCESS_SCR)
    {
        EntryEmSocketResult();//用于显示连接后收到的信息
    }
}
```

让我们重点分析一下该请求函数。

```
int mmi_soc_demo_app_request(
        mmi_soc_demo_app_enum app_id,
        int account_id,
        char server_ip_address[4],
        char *url,
        int url_len,
        char *echo_text,
        int echo_text_len,
        mmi_soc_demo_app_rsp_t callback)
{
  /* 首先判断soc_demo_transaction是否非空。如果非空就说明当前的链接正在处理中，
   那么函数将返回*/
  if (soc_demo_transaction)
  {
     return;
  }
  else
  {
```

```
    /* Set  initial values to soc_demo_transaction */
    memset((kal_int8*) soc_demo_transaction->rcvd_buffer, 0, MAX_RCV_BUFFER_SIZE);
    memset((kal_int8*) soc_demo_transaction->snd_buffer, 0, MAX_SND_BUFFER_SIZE);
    soc_demo_transaction->soc_demo_app_id = app_id;
    soc_demo_transaction->nwt_acount_id = account_id;
    if (server_ip_address)
    {
            memcpy(soc_demo_transaction->server_ip_addr.addr, server_ip_address, 4);
            soc_demo_transaction->server_ip_addr.addr_len = 4;
    }

        soc_demo_transaction->url = (kal_int8*) url;//URL 地址
        soc_demo_transaction->url_len = url_len;//地址的长度
        soc_demo_transaction->snd_counter = 0;//发送计数
        soc_demo_transaction->rcvd_counter = 0;//接受计数
        soc_demo_transaction->callback = callback; //链接成功时返回信息的处理函数
    }
```

14.2.5 发送请求

接下来将当前状态设置为 HTTP_DNS_QUERY。调用函数 en_soc_demo_http_send_request 发送 HTTP 的请求。

```
    switch (app_id)
    {
            case HTTP:
                {
                    soc_demo_transaction->state = HTTP_DNS_QUERY;
                    return en_soc_demo_http_send_request();
                }
                ...
    }
```

· 解析域名

函数 en_soc_demo_http_send_request 会根据传入的 URL 地址调用函数 en_soc_demo_get_host_by_name 解析主机地址和端口：

```
    int en_soc_demo_http_send_request(void)
    {
        if (soc_demo_transaction)
        {
          if (soc_demo_transaction->state == HTTP_DNS_QUERY &&
          (soc_demo_app_http_url_struct*) soc_demo_transaction->url_parsed_info == NULL)
           {
             soc_demo_transaction->url_parsed_info =
                 (soc_demo_app_http_url_struct*)
               en_soc_demo_http_get_parsed_url(soc_demo_transaction->url,
                                               soc_demo_transaction->url_len);
            if (soc_demo_transaction->url_parsed_info)
            {
                  (void)en_soc_demo_get_host_by_name(
                        HTTP,
```

```
                    (kal_uint8*)
                     soc_demo_transaction->url_parsed_info->host_name);
                     return EN_SOC_SUCCESS;
        }
        else
        {
            en_soc_output_result(EN_SOC_INVAL_PARA, NULL, 0);
            return EN_SOC_INVAL_PARA;
        }
    }
```

soc_demo_transaction->url_parsed_info 变量将获得主机地址和端口信息。接着，通过调用函数 en_soc_demo_get_host_by_name 获得具体的 IP 地址。

```
void en_soc_demo_get_host_by_name(mmi_soc_demo_app_enum app_id,
                                  const kal_uint8 *hostname)
{
soc_demo_transaction->state = REQ_SENDING;
ret = soc_gethostbyname(…)
…
else if (app_id == HTTP)
        {
            memcpy(soc_demo_transaction->server_ip_addr.addr,
                   soc_demo_transaction->rcvd_buffer, 4);
            soc_demo_transaction->rcvd_counter = 0;
            soc_demo_transaction->server_ip_addr.addr_len = 4;
            soc_demo_transaction->state = HTTP_TCP_CON_CREATING;
            en_soc_demo_http_send_request();

        }
}
```

如果 soc_gethostbyname 函数的返回值 ret>=0，说明程序成功地获得了 IP 地址。如果为负数，表明不成功。但有一种情况例外，就是返回值为 SOC_WOULDBLOCK(-2)。这个返回值很特殊，虽然为负数，但并不表示当前函数有错误。

它表示当前函数处于阻塞状态，会等待监听消息 MSG_ID_APP_SOC_NOTIFY_IND 的通知函数处理完毕才会返回。

接着，会把当前 Socket 状态设置为 HTTP_TCP_CON_CREATING，表示链接正在进行中。

· 设置 HTTP 报文

然后，再次进入函数 en_soc_demo_http_send_request，此时函数会进入另外一个判断：

```
int en_soc_demo_http_send_request(void)
{
    …
    else if (soc_demo_transaction->state == HTTP_TCP_CON_CREATING)
    {
            char CRLF[3] = {0x0d, 0x0a, 0x00};//作为后面 HTTP 报文的换行标志
            soc_demo_transaction->state = HTTP_TCP_CON_CREATED;

    }
```

```
    ...
}
```

可以看到，整个网络链接的过程都伴随着状态的切换。在上面代码片段中，套接字的状态又被设置成 HTTP_TCP_CON_CREATED，说明 TCP 连接已经创建完成。接着就可以开始设置要发送的数据，发送的数据都做成一种标准的 HTTP 报文用于从 Web 服务器中获取对象。

```
...
strcat(soc_demo_transaction->snd_buffer, "GET");
soc_demo_transaction->snd_data_len += 4;
...
strcat((char*)soc_demo_transaction->snd_buffer, "HTTP/1.1");
soc_demo_transaction->snd_data_len += 8;
strcat((char*)soc_demo_transaction->snd_buffer, "Host:");
soc_demo_transaction->snd_data_len += 6;
...
```

通过 Strcat 进行字符串组合，最后 snd_buffer 的内容如下：

```
"GET /HTTP1.1 HOST:WWW.SOHU.COM Connection: Closed"
```

小 知 识

HTTP 报文格式：

请求消息 = 请求行（实体头信息）CRLF[实体内容]

请求行 = 方法　URL　HTTP 版本号　CRLF

方法 = GET|HEAD|POST|扩展方法

URL = 协议名称 + 宿主名 + 目录与文件名

其中“CRLF”表示回车换行。

“请求行”中的“方法”描述了对指定资源执行的动作，常用的方法有“GET”、“HEAD”和“POST”3 种，如下表 14.1 所示。

表 14.1　HTTP 请求方法

取　值	描　述
GET	从 Web 服务器中获取对象，不同类型的对象将获取不同的信息，比如： · 文件类型对象，获取该文件的内容 · 程序类型对象，获取该程序执行的结果 · 数据库查询类型对象，获取该查询的结果
HEAD	要求服务器查找对象的元信息
POST	从客户端向 Web 服务器发送数据

HTTP 报文的具体细节请参考相关书籍。

14.2.6　连接

一切都准备就绪，现在就可以连接目标地址了。

```
ret = soc_connect(soc_demo_transaction->socket_id, &soc_demo_transaction->server_ip_addr);
```

将对应的目标 IP 地址传入 soc_connect。

如果连接成功，程序将会再次进入函数 en_soc_demo_http_send_request，设置套接字状态 soc_demo_transaction->state = REQ_SENDING，这样，就可以进行发送了。

```
ret = soc_send(soc_demo_transaction->socket_id,
               (kal_uint8*) (soc_demo_transaction->snd_buffer +
               soc_demo_transaction->snd_counter),
                (soc_demo_transaction->snd_data_len - soc_demo_transaction->snd_counter),
                0);
```

如果发送成功，那么 ret 将大于等于 0。并将套接字状态设置为 REQ_SENT，表示已经发送。启动定时器，等待服务器端的响应。

```
…
if (ret > 0)
{
      if (ret == (kal_int32) soc_demo_transaction->snd_data_len)
      {
              soc_demo_transaction->state = REQ_SENT;//已发送
              soc_demo_transaction->post_retry_counter++;
              em_soc_demo_app_start_timer(); //等待服务器端响应
              en_soc_demo_tcp_app_recv_response();//接受服务器端返回的响应
              return EN_SOC_SUCCESS;
      }
      else
      {
                soc_demo_transaction->snd_counter += ret;
                return EN_SOC_SUCCESS;
      }
}
…
```

14.2.7 等待服务器响应

紧接着，系统将调用函数准备接收服务器端的响应。

```
void en_soc_demo_tcp_app_recv_response(void)
{
//将套接字状态设置为 RSP_RCVING，表明处于接受状态
soc_demo_transaction->state = RSP_RCVING;
ret_val = soc_recv(soc_demo_transaction->socket_id,
                   (kal_uint8*) (soc_demo_transaction->rcvd_buffer +
                    soc_demo_transaction->rcvd_counter),
                   (MAX_RCV_BUFFER_SIZE - soc_demo_transaction->rcvd_counter),
                   0);
}
```

一般情况下，如果 soc_recv 的返回值 ret_val 等于 SOC_WOULDBLOCK，表示接收数据并不是马上就可以进行，而必须等待监听消息 MSG_ID_APP_SOC_NOTIFY_IND 的函数完成。监听该消息的工作由函数 em_soc_demo_app_socket_notify 负责。

```
else
 {
      if (ret_val == SOC_WOULDBLOCK)
      {
          SetProtocolEventHandler(em_soc_demo_app_socket_notify,
                                  MSG_ID_APP_SOC_NOTIFY_IND);
```

```
                return;
        }
```

14.2.8 处理消息事件

在监听函数 em_soc_demo_app_socket_notify 中，对各种事件做了对应的处理。

```
switch (soc_notify->event_type)
    {
        case SOC_WRITE:
        {
        …
        case SOC_READ:
        …
        SOC_CONNECT:
        …
```

只有当相应的事件处理完毕，程序才会返回到之前处于 BLOCK 位置的地方，继续执行。如果返回值大于等于 0，就可以将返回的结果输出。

14.2.9 回调函数执行

在函数中会调用之前注册的回调函数 EmSocketRsp。

```
void en_soc_output_result(int ret, char *out_str, int len)
{
    if (soc_demo_transaction)
    {
    //这里的 callback 就是之前注册的 EmSocketRsp
    soc_demo_transaction->callback(soc_demo_transaction->soc_demo_app_id, ret, out_str, len);
      if (ret != EN_SOC_SUCCESS)
        {
            soc_demo_transaction->state = SOC_CLOSING;
        }
        mmi_soc_demo_app_request_abort();
    }
}
```

14.2.10 输出结果

最后，通过 EmSocketRsp 函数进入一个屏幕，输出服务器返回给客户端的信息。返回结果如图 14.7 所示。

```
void EmSocketRsp(mmi_soc_demo_app_enum app_id_enum, int result,
                 char *response, int response_len)
{
if (currScrnID == EM_SOCKET_PROGRESS_SCR || currScrnID ==
EM_SOCKET_TRACERT_PROCESS_SCR)
    {
        EntryEmSocketResult();//输出最终的结果
    }
}
```

一般情况下，如果返回结果首行内容中包含“200 OK”，就表示一切正常，对 GET 和 POST 请求的应答文档跟在后面。

到此为止，我们就把这个例子完整地梳理了一遍。

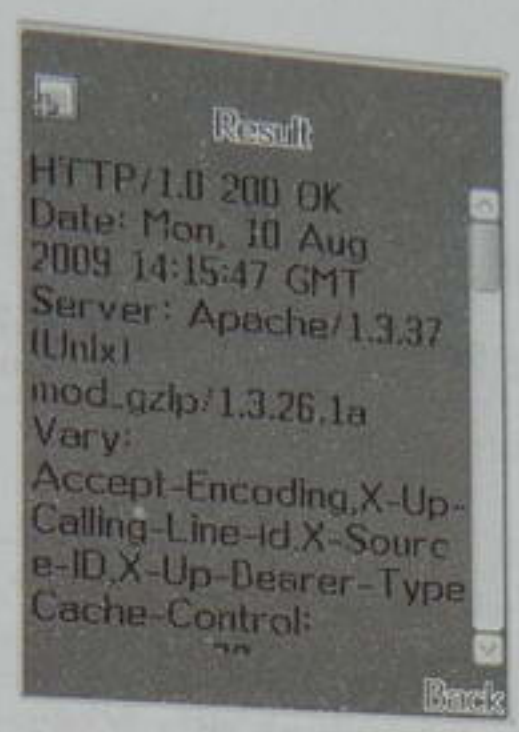


图 14.7 返回结果

上面的例子，囊括了在 MTK 平台中进行 Socket 连接的所有过程。该 DEMO 已经很完整了，读者根据我们的讲解，应该熟悉了其内部的运作方式。接下来我们就根据这个 DEMO 做一个总结。

14.3 HTTP 连接方式编程总结

在 MTK 上实现 Socket 连接远程服务器并下载数据需要用到以下各小节中提到的函数。

14.3.1 创建 Socket 连接

```
kal_int8 soc_create(kal_uint8  domain,
                    socket_type_enum  type,
                    kal_uint8  protocol,
                    module_type mod_id,
                    kal_uint32 nwk_account_id)
```

创建 Socket 连接，成功后将返回一个 socket_id 供以下函数使用。成功运行在手机上的效果是打开黄色边框的 GPRS 图标，对应手机 nwk_account_id 的值有 10 和 14，分别代表网络接入点 CMWAP 和 CMNET。参数说明请参照相关文档。

14.3.2 设置 Socket 非阻塞/异步模式

```
kal_int8 soc_setsockopt(kal_int8 s, kal_uint16 option, kal_uint8 *val, kal_uint8 val_size)
kal_uint8 val = 1;
soc_setsockopt(socket_id,SOC_NBIO,&val, sizeof(val))
val = SOC_READ | SOC_WRITE | SOC_CLOSE | SOC_CONNECT;
soc_setsockopt(socket_id,SOC_ASYNC,&val,sizeof(val))
```

14.3.3 链接准备

如果是 CMNET 联网并且请求中用到了英文域名，还需要调用 soc_gethostbyname 函数解析域名，解析出来的 IP 作为我们建立连接的目标 IP。

如果是 CMWAP 联网，直接跳到第下一步，直接连接移动网关 10.0.0.172:80 即可。

14.3.4 连接

```
kal_int8 soc_connect(kal_int8 s, sockaddr_struct *addr)
```

手机通过 WAP 上网与 CMNET 上网不同，手机端需要首先连接移动网关才能进行网络通信，这里需要填充结构体 sockaddr_struct 的 addr 及 port 成员。

14.3.5 发送 HTTP 请求

```
kal_int32 soc_send(kal_int8 s, kal_uint8 *buf, kal_int32 len, kal_uint8 flags)
```

其中 buf 字符串需要遵守 HTTP 连接协议，也就是必须写成 HTTP 报文格式。

例如访问 www.sohu.com/的格式如下：

```
GET http://www.sohu.com HTTP/1.1\r\nHost:www.baidu.com:80\r\nUser-Agent: */*\r\nAccept:
*/*\r\nX-Online-Host:www.baidu.com\r\nConnection:Keep-Alive\r\n\r\n
```

14.3.6 接收数据

```
kal_int32 soc_recv(kal_int8 s, kal_uint8 *buf, kal_int32 len, kal_uint8 flags)
```

buf 用来保存接收到的字符串，这里需要注意 buf 的长度不能超过 2000，否则上真机的时候一到申请内存空间时就会死机。

14.3.7 关闭 Socket 连接

```
kal_int8 soc_close(kal_int8 s)
```

参数传入 soc_create 的返回值。

14.3.8 启动/关闭定时器

启动定时器：

```
void StartTimer(U16 timerid, U32 delay, FuncPtr funcPtr)
```

在等待服务器响应时需要用到。

关闭定时器：

```
void StopTimer(U16 timerid)
```

14.3.9 注册事件回调函数

```
SetProtocolEventHandler(PsFuncPtr funcPtr, U16 eventID)
```

在 funcPtr 里实现响应 Socket 事件。

此函数贯穿 Socket 连接、发送、收取过程。合理设计并应用此函数是成功完成一次 Socket 通信的关键。

图 14.8 是链接中国移动网关的基本状态图。

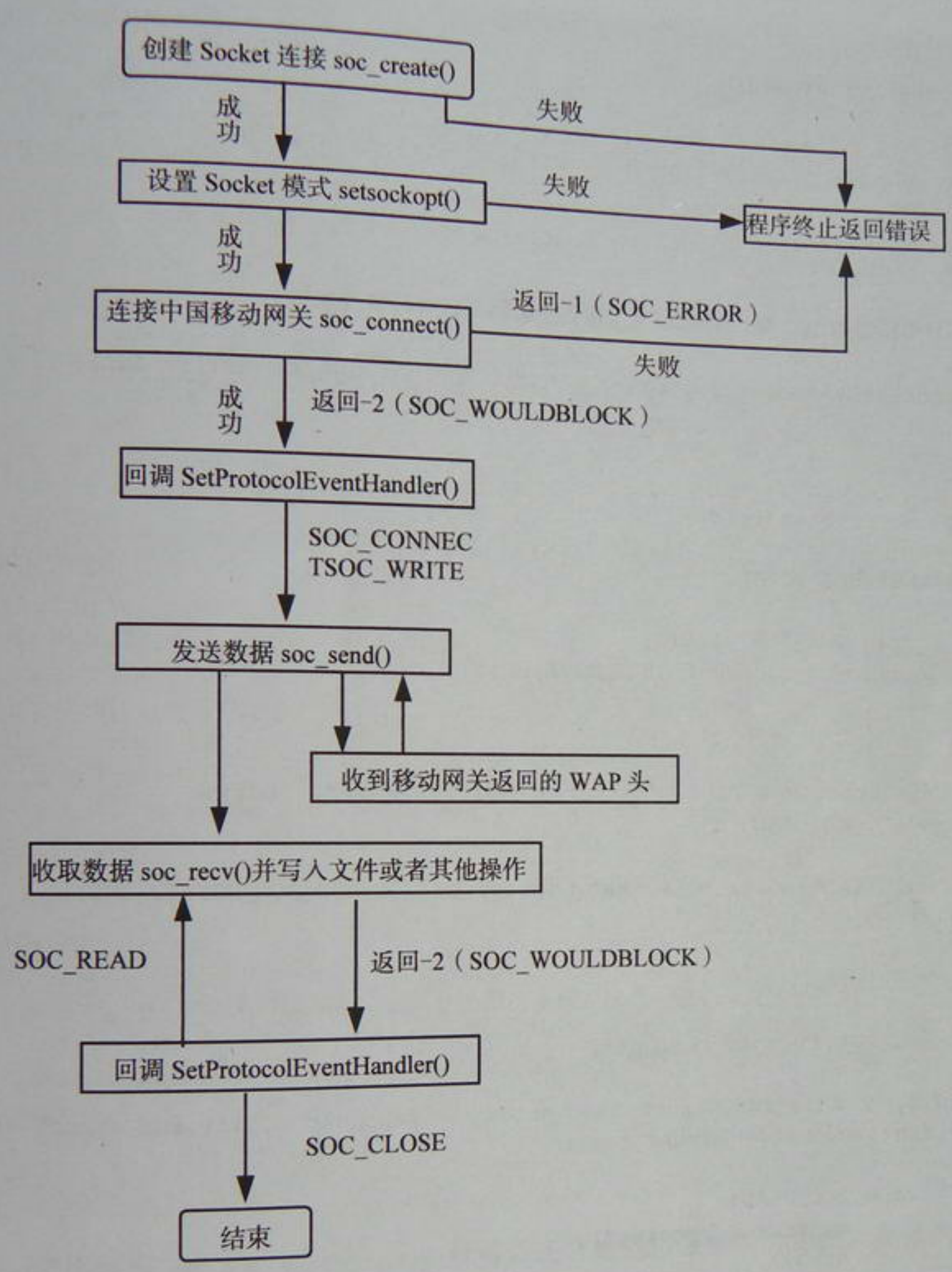


图 14.8 连接中国移动网关

14.4 精简的代码

现在，让我们根据上述步骤来整理先前的代码，将整个的链接访问过程浓缩为如下代码。

```
//#include "soc_consts.h"
#include "soc_api.h"
static kal_int8 soc_id;
void my_socket_receive(void);
void my_socket_notify(void *msg_ptr);
sockaddr_struct my_ip_addr;
#define BUF_SIZE 200
```

```
//发送
void my_socket_send(void)
{
 kal_int32 ret;
 ret = soc_send(soc_id, (unsigned char*)URL, strlen(URL), 0);
 if( ret > 0)
 {
  my_socket_receive();
 }
 else //如果返回值为 SOC_WOULDBLOCK，就要注册回调，等待事件通知
 {
  SetProtocolEventHandler(my_socket_notify, MSG_ID_APP_SOC_NOTIFY_IND);
 }
}
//接收
void my_socket_receive(void)
{
 kal_uint8 rbuf[BUF_SIZE];
 int ret;
 unsigned short wBuf[BUF_SIZE+1];
 ret = soc_recv( soc_id, rbuf, BUF_SIZE, 0);
 if( ret > 0)
 {
  int i;
  //kal_prompt_trace(MOD_TCPIP," <<<<<<<- %d >>>>>>>>", rbuf);
  if( ret == SOC_WOULDBLOCK)
  {
   SetProtocolEventHandler(my_socket_notify, MSG_ID_APP_SOC_NOTIFY_IND);
  }
 }
}
//事件通知
void my_socket_notify(void *msg_ptr)
{
        app_soc_notify_ind_struct *soc_notify = (app_soc_notify_ind_struct *)msg_ptr;
        switch(soc_notify->event_type)
        {
          case SOC_READ:
               my_socket_receive();
               break;
          case SOC_WRITE:
               my_socket_send();
               break;
          case SOC_CONNECT:
              my_socket_send();
              break;
          case SOC_CLOSE:
              break;
          default:
          break;
        }

}
//总调函数
int my_test_socket_entry()
{
 kal_uint8 val=KAL_TRUE;
```

```
kal_int8 soc_ret;
#ifndef MMI_ON_HARDWARE_P
soc_init_win32();
#endif
soc_ret = soc_create(PF_INET, SOCK_STREAM, 0, MOD_MMI, 14);
if(soc_ret < 0)
 return 0;
if(soc_setsockopt(soc_id, SOC_NBIO, &val, sizeof(val)) <0)
{
 return 0;
}
val=SOC_READ | SOC_WRITE | SOC_CLOSE | SOC_CONNECT;
if(soc_setsockopt(soc_id, SOC_ASYNC, &val, sizeof(val)) <0)
{
 return 0;
}

memset( &my_ip_addr, 0, sizeof(sockaddr_struct));
//连接中国移动网关
my_ip_addr.addr[0] = 10;
my_ip_addr.addr[1] = 0;
my_ip_addr.addr[2] = 0;
my_ip_addr.addr[3] = 172;
my_ip_addr.addr_len = 4;
my_ip_addr.port = 80;
soc_ret = soc_connect(soc_id, &my_ip_addr);
if( soc_ret >= 0)
{
 my_socket_send();
 return soc_ret;
}
else if( soc_ret == SOC_WOULDBLOCK) /* 一般此条件会成立，也就是异步事件*/
{
 SetProtocolEventHandler(my_socket_notify, MSG_ID_APP_SOC_NOTIFY_IND);
 return soc_ret;
}
return 1;
}
```

这样，我们调用函数 my_test_socket_entry 就可以正常地连接服务器。当然，具体的细节以及扩展操作还需要进一步完善，这里，我们仅仅为讲解方便。

14.5 WAP 浏览器

如果你的工程已经移植了 WAP 模块，那么，在 WAP 中可以通过浏览器直接获得网页的全部内容，其调用方式很简单。

```
void wap_internet_goto_URL(U8 *url)
{

    ilm_struct *ilm_ptr = NULL;
    wap_browser_startup_req_struct *para = (wap_browser_startup_req_struct*)
     construct_local_para((kal_uint16)sizeof(wap_browser_startup_req_struct), 0);
```

```
    if (!wap_is_ready())
    {
        wap_entry_no_ready_screen(WAP_SERVICE_BROWSER);
        return;
    }
    mmi_brw_wap_entry_busy_progressing_screen_in_idle();
    if (wap_browser_is_open())
    {
        /* Abnormal case. BRA should already terminate its UI. */
      WAP_DBG_ASSERT(0);
      ilm_ptr = allocate_ilm( MOD_MMI );
      ilm_ptr->msg_id = MSG_ID_WAP_BROWSER_SHUTDOWN_REQ;
      ilm_ptr->peer_buff_ptr = NULL;
      ilm_ptr->local_para_ptr = (local_para_struct *) NULL;
      SEND_ILM( MOD_MMI, MOD_WAP, WAP_MMI_SAP, ilm_ptr);
    }

    para->type = WAP_BROWSER_GOTO_URL;

    memset(para->url, 0, sizeof(para->url));
    UnicodeNToAnsii((S8 *)para->url, (S8 *) url, sizeof(para->url));

    ilm_ptr = allocate_ilm(MOD_MMI);
    ilm_ptr->local_para_ptr = (local_para_struct*) para;
    ilm_ptr->msg_id = MSG_ID_WAP_BROWSER_STARTUP_REQ;
    ilm_ptr->peer_buff_ptr = NULL;


    ilm_ptr->src_mod_id  = MOD_MMI;
    ilm_ptr->dest_mod_id = MOD_WAP;
    ilm_ptr->sap_id = WAP_MMI_SAP;
    //msg_send_ext_queue(ilm_ptr);
    SEND_ILM( MOD_MMI, MOD_WAP, WAP_MMI_SAP, ilm_ptr);
    return;
}
```

执行这个函数，即可调出浏览器界面。然后，浏览器会访问传入的 URL 地址页面，应用很方便。其中涉及的 MOD_MMI、MOD_WAP 等 Task 相关的知识请参考先前的 Task 一章的内容。

结语：

本章对 MTK 的网络做了一个大概的介绍，并对其自带的 Socket 例子做了详细的梳理，读者学完本章，可以对 Socket 在 MTK 平台的应用有更深的认识，为以后开发功能更加复杂的网络应用程序打下良好的基础。本章只是一个启蒙式的讲解，更深层的方面还需要读者自己试验和体会。

15 高仿 NOKIA N73

作为诺基亚 N 系列中最经典的一款产品，N73 以其简洁大方的界面风格赢得相当多的消费者的青睐。

N73 的 UI 设计以实用为主，界面简洁大方。待机界面的元素排列、主菜单的十二宫格等，都对整个手机业界的 UI 设计产生了深远影响。在本书所用的 MT6226 06B 的软件版本中，其待机界面的风格也留有明显的 NOKIA 痕迹。

在本章中，我们将通过一个完整的实例，来讲解如何修改 MTK 的待机界面。对其中每一个步骤和牵涉到的细节都将做出详细的解释。通过本章的学习，可以为读者以后进行界面设计打下良好的基础。仿制哪个系统，哪款手机不重要，重要的是通过这些修改过程，对 MTK 图形系统有一个更深刻的认识，这才是本章的核心所在。

本章重点放在待机界面的修改上。

15.1 认识 N73 界面

既然是高仿 N73 界面，那么我们首先要对它有一个全面的认识。先来看一下待机界面中的元素，如图 15.1 所示。

因为手机系统开机后，最先呈现在用户眼前的就是待机界面，因此，不管这个手机基于什么平台，待机界面永远都是一个手机界面的门面。更夸张地说，用户甚至可以直接从待机界面来确定是否选购该产品。

N73 的待机界面元素多而不杂，可显示的信息量大。在整个界面中集中了时间显示、运营商显示、快捷菜单、日历项等众多模块。各个显示条排列紧凑，功能实用，中上部的快捷键排列更加为用户考虑。用户可以直接在待机界面进行常见模块的操作，而不需要一层层地进入界面深处，极大地提高了操作效率。

当用户从待机界面进入主界面后，看到的是如图 15.2 所示的经典的十二宫格界面（也可能是九宫格），这样的布局能够显示更多的功能图标，尽可能地使用户操作时少翻页，少按键，如图 15.2 所示。

当我们单击“信息”图标，再深入一层界面时，可以看到图 15.3 所示的界面样式。在屏幕的正上方是一个状态条，含有当前进入的菜单 ID 和图标以及电量和信号条；接下来就是一个图标列表，列出了当前菜单下所有支持的功能菜单；最下方是一个按钮条。这 3 部分元素组成了 N73 约 80%的图形界面。

图 15.1　N73 待机界面

图 15.2　N73 主菜单界面

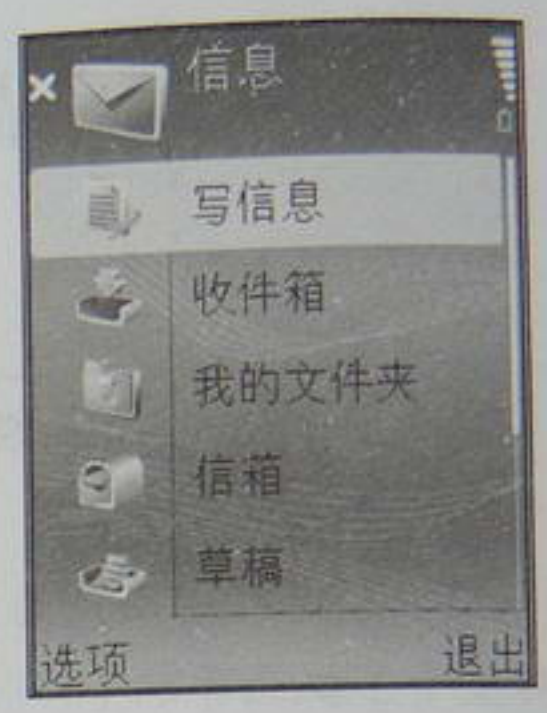


图 15.3　列表界面

从宏观上来讲，上面 3 个图就可以代表 N73 的界面基调。当然在仿制过程中，不可能一下子就达到上面的效果，任何一个细节都不能忽略。因此在后面的修改过程中，我们将在上述界面的基础上进行深入分析和讲解，使得大家在修改过程中学到更多的东西。

15.2　待机界面分析

在模仿 N73 待机界面之前，我们首先应该弄清楚待机界面的组成，如图 15.4 所示。

图 15.4　界面拆解

通过上图，我们可以将待机界面大致分为 4 部分。

· 状态栏区，包括 4 个元素

➢ 运营商显示。

➢ 日期和星期显示。

➢ 时钟显示。

➢ 电量和信号状态显示。

· 快捷菜单区

标准情况下是 6 个图标。

· 日历及备忘录区

当用户增加了一个日历项或者备忘录时，当前区域就会显示细节。

· 按键栏区

左右按键，特殊情况下可能有中间键。

15.3 基本布局

现在，让我们运行一下模拟器，看看 MTK 原始的待机界面的样子，如图 15.5 所示。

图 15.5 原始待机界面

从图中可以看到，MTK 的原始界面和 N73 在布局上还是有很大差别的，最关键的左右按键栏在原始界面中没有显示，并且运营商和快捷菜单的位置都明显不同。因此，在进行细化修改之前，我们首先需要将界面的布局纠正过来，就如同在修房子之前必须把梁搭好一样。

15.3.1 定义宏开关

和第 4 章讲的一样，在对 MTK 源代码进行自定义修改时，为了保证源代码干净和后期版本复原，需要定义一个开关宏。

在 option.mak 文件末尾定义我们的宏：→这是我们通常情况下的选择

```
COM_DEFS += __N73_STYLE__
```

或者在文件 MMI_featuresPLUTO.h 末尾定义：

```
#define __N73_STYLE__
```

打开文件 wgui_categories_idlescreen.c。

15.3.2 打开左右按键

找到函数 ShowCategory33Screen，该函数也就是待机界面的屏幕模板。

将 DM_NO_SOFTKEY 宏（不显示左右按键）屏蔽。

```
#ifndef __N73_STYLE__
        #ifdef __MMI_TOUCH_IDLESCREEN_SHORTCUTS__
        if (mmi_bootup_get_active_flight_mode() == 0)
        {
            dm_data.s32flags |= DM_NO_SOFTKEY;
        }
        #endif
#endif
```

现在，左右按键即可正常地显示出来。但是，由于原始界面的 4 个快捷图标显示在屏幕底部，因此，刚才显示的按键会与快捷图标产生冲突。因此，必须将快捷图标移至屏幕中上部，这其实也是 N73 的风格。

15.3.3 上移快捷菜单

首先定义快捷菜单的位置宏，也就是距离屏幕原点的纵坐标，设置为 96 像素。

```
#define  SHORTCUT_POS              96
```

然后，进入函数 setup_touch_idle_screen_shortcuts，修改 shortcut 的纵坐标。

```
#ifdef __N73_STYLE__
      touch_idle_shortcuts[i].y = SHORTCUT_POS;
#else
      touch_idle_shortcuts[i].y = UI_device_height - touch_idle_shortcuts[i].height;
#endif
```

编译并运行模拟器，效果如图 15.6 所示。

图 15.6 左右按键显示

另外，当前待机界面中的背景图片名称为 WALL01.gif，读者可以另外绘制一幅 240 × 320 大小的图片并命名为 WALL01.gif，将原有的背景图片替换。

现在，待机界面的效果开始接近 N73。接下来，我们开始屏幕细节的修改。

15.4 状态栏修改

15.4.1 电量和信号状态图标

首先应该修改的是电量和信号状态图标，N73 的风格与 MTK 原始图标有很大差别。所以，图标的替换也是重要的一环，大家应该提前准备好图片。

（1）绘制 5 个尺寸为 12 × 49、BMP 格式的电量图标，如图 15.7 所示。

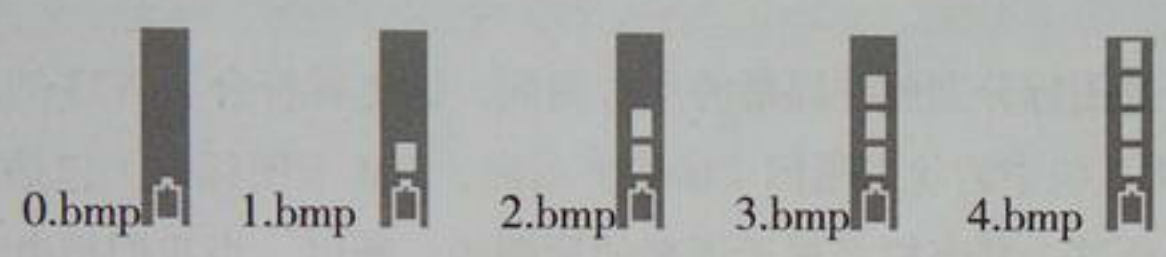


图 15.7　5 帧电量状态图片

很明显，上面的每个图标都表示电池电量状态的一帧。

接下来，用上面的图标替换电量图标文件夹

\plutommi\Customer\Images\PLUTO240X320\ MainLCD\ IdleScreen\Statusicons\battery\SI_BAT 中的全部图片内容。由于原始代码中的电池显示是 4 格，而现在增加了一格。所以要在 interval.txt 文件中增加一行“200”字符。即现在 interva.txt 的内容为 5 行“200”字符。

（2）绘制 5 个 12 × 49 尺寸的信号状态图标，如图 15.8 所示。

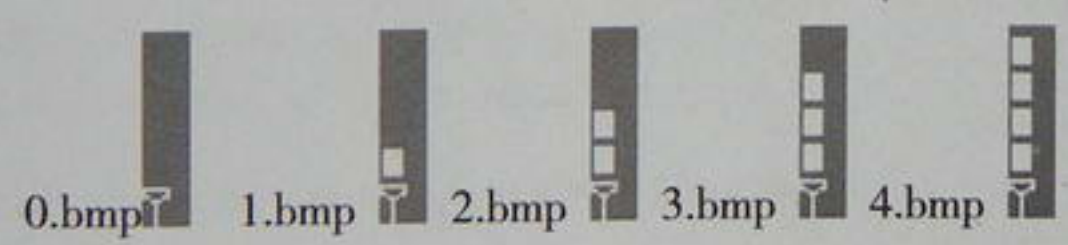


图 15.8　5 帧信号状态图片

和电池图标一样，上面的每个图标都表示信号状态的一帧。

接下来，用上面的图标替换 3 个信号图标文件夹

\plutommi\Customer\Images\PLUTO240X320\MainLCD\IdleScreen\Statusicons\signal\SI_SIG,\plutommi\Customer\Images\PLUTO240X320\MainLCD\IdleScreen\Statusicons\signal\SI_SIG_L1 和\plutommi\Customer\Images\PLUTO240X320\ MainLCD\ IdleScreen\ Statusicons\signal\SI_SIG_L2 中的全部图片内容。

由于原始代码中的电量显示是 4 格，而现在增加了 1 格。所以，要在 interval.txt 文件中增加一行“200”字符，即现在 interva.txt 的内容为 5 行“200”字符。

图片替换完毕后，需要在命令行中对整个工程进行 Update 操作。完成后，开始修改逻辑。

打开文件 wgui_status_icons.c，查找函数 arrange_status_icons，在内部如下位置进行修改：

```
    …
  {
      MMI_status_icons[0].y = cy - (MMI_status_icons[0].height >> 1);
      MMI_status_icons[1].y = cy - (MMI_status_icons[1].height >> 1);
    }
    /* PMT VIKAS END 20051010 */
#endif /* (STATUS_ICONS_DRAW_FROM_BOTTOM_BASELINE) */
#endif
    /* PMT HIMANSHU END */
#ifdef _N73_STYLE_
    //电量图标坐标
    MMI_status_icons[STATUS_ICON_BATTERY_STRENGTH].y = 0;
    MMI_status_icons[STATUS_ICON_BATTERY_STRENGTH].x=226;
    //信号图标坐标
    MMI_status_icons[STATUS_ICON_SIGNAL_STRENGTH].y = 0;
    MMI_status_icons[STATUS_ICON_SIGNAL_STRENGTH].x=3;
#endif
    i = 2;
    …
```

我们将电量和信号的图标分别放在屏幕的左右两侧，这也就符合了 N73 的风格。

由于 MTK 默认是将系统中的状态图标（如锁屏图标、SIM 卡图标等）包括电量、信号图标放在一个状态栏中进行管理，而这个状态栏是居于屏幕的顶端的，所以，当我们把电量和信号图标的位置进行了修改之后，也要将其余的后期可能显示的图标放在另外一个地方，也就是状态栏的下方。

15.4.2 修改状态栏重绘区域

由于我们现在已经使用了新的电量和信号图标，其尺寸已经增加，之前的状态栏剪切区域已经不能覆盖现在的状态栏，因此，需要增加状态栏的裁剪区域。

打开 wgui_categories_idlescreen.c 文件，找到函数 ShowCategory33Screen_ext，修改其内部部分代码。

```
    …
  /* PMT VIKAS START 20051010 */
#ifdef _MMI_UI_STATUS_BAR_AT_BOTTOM_
    set_status_icon_bar_clip(
        0,
        0,
        UI_device_height - 1 - MMI_button_bar_height - MMI_status_bar_height,
        UI_device_width - 1,
        UI_device_height - 1 - MMI_button_bar_height - 1);
#else /* _MMI_UI_STATUS_BAR_AT_BOTTOM_ */
      set_status_icon_bar_clip(0, 0, 0, UI_device_width - 1, MMI_status_bar_height+42 - 1);
#endif /* _MMI_UI_STATUS_BAR_AT_BOTTOM_ */
    /* PMT VIKAS END 20051010 */

    /* 110105 status bar1 Start */
#ifdef _MMI_UI_TECHNO_IDLESCREEN_BAR_
if (PhnsetGetShowDateTimeStatus() == MMI_TRUE)
    …
```

上面代码的黑体部分，将 0 号状态栏下边界增加 42 个像素，之前的 MMI_status_bar_height 为 18，UI_device_width 为 240，所以，现在 0 号状态栏的 clip 区域为（0, 0, 239, 59）。

请注意，set_status_icon_bar_clip 的参数为 5 个，其中第一个是表示状态的编号，之后才是真正的裁剪区域坐标。

MTK 系统共设置了 3 个状态栏，分别是我们目前使用的：

0 号，位于屏幕顶端；

1 号，位于屏幕右端并且垂直显示；

2 号，子屏幕的状态栏。

15.4.3 移动小图标绘制坐标

现在，让我们把之前屏幕上的小图标显示区域位置移到状态栏下方。并且将图标的增加方式改为从右向左。

```
…
#ifdef _N73_STYLE_
    x = 200;//从横坐标 200 处开始添加图标
#else
    x = x1 + STATUS_ICON_X_GAP;
#endif

    while (!done)
    {
…
```

上面的代码，确定了小图标的起始位置。

```
…
#endif /* _MMI_UI_STATUS_BAR_AT_BOTTOM_ */  /* PMT VIKAS END 20051010 */
y = cy - (MMI_status_icons[k].height >> 1);
#endif /* (STATUS_ICONS_DRAW_FROM_BOTTOM_BASELINE) */
#ifdef _N73_STYLE_
        y = 53;  //将小图标的纵坐标设置为 53，这样，小图标就将位于 0 号状态栏之下
#endif
        MMI_status_icons[k].x = x;
        MMI_status_icons[k].y = y;

#ifdef _N73_STYLE_
      //原来是+= ，改为-= ，这样就可以将图标从右向左排列
        x-=MMI_status_icons[k].width+STATUS_ICON_X_GAP;
#else
      x += MMI_status_icons[k].width + STATUS_ICON_X_GAP;
#endif
      MMI_status_icons_list1[MMI_status_icon_bars[0].n_icons] = (S16) k;
      MMI_status_icon_bars[0].n_icons++;
      if (MMI_status_icon_bars[0].n_icons >= MMI_N_STATUS_ICONS_LIST1)
      {
          done = 1;
      }
…
```

到这里，电量和信号的图标以及默认显示的图标都已经调整好了位置，接下来就需要对时间显示进行调整。

> **提　示**
>
> 小图标用于标识情景模式、闹钟、背景播放等状态。因为现在小图标的起始坐标位于屏幕的右端，如果仍然按照先前由左至右的显示顺序，将会影响后期图标的显示。因此，将其顺序颠倒。

15.4.4 时间显示修改

打开文件 wgui_datetime.c。首先，将时间显示改为数字时钟，之前默认的是图形时钟。

在文件的 381 行附近，将时钟替换为 DIGITAL：

```
#ifdef _N73_STYLE_
U8 g_clock_type = DIGITAL;   /* default clock for Techno Idlescreen Bar */
#else
U8 g_clock_type = ANALOG;   /* default clock for Techno Idlescreen Bar */
#endif
```

找到函数 show_main_LCD_date_time。这个函数很关键，几乎所有跟时间显示有关的功能都实现在该函数中。在函数的变量声明处，增加两个字体变量。

定义两种用于显示时间的字体：

```
#ifdef _N73_STYLE_
     stFontAttribute    font1=  {   0,0,0,DIALER_FONT,0,1 }; //用于显示数字时间
     stFontAttribute    font2=  {   0,0,0,MEDIUM_FONT,0,1 };//用于显示日期
#endif
```

后面的代码中，注意如下代码段：

```
…
switch (main_LCD_dt_object.flags & DT_FLAGS_TYPE)
{
    case DT_IDLE_SCREEN:
        {
        …
            case DIGITAL:
                …
…
```

这个 switch 代码段实现的是根据当前时间模式和所处屏幕进行时间显示。由于我们需要修改的屏幕位于待机界面，所以，程序将进入“case:DT_IDLE_SCREEN”语句，接下来，由于当前需要显示的是数字时钟，程序会进入“case: DIGITAL:”代码段，因此在其内部修改代码，如下所示（从第 3515 行开始）：

```
…
  switch (main_LCD_dt_object.flags & DT_FLAGS_TYPE)
   {
       case DT_IDLE_SCREEN:
       {
       #ifdef _MMI_UI_TECHNO_IDLESCREEN_BAR_
            switch (g_clock_type)
            {
                 case ANALOG:
                     if (enable_main_LCD_date_time_display)
                     {
                         show_analog_clock();
                     }
                     break;
                  case DIGITAL:

            #endif /* _MMI_UI_TECHNO_IDLESCREEN_BAR_ */
```

```
#ifdef _N73_STYLE_
        gui_set_font(&font1);
        /* get current time in string format */
        time_string(&main_LCD_dt_object.time.t, (UI_string_type) s,
                    main_LCD_dt_object.flags);
        /* calculate string width of time string */
        gui_measure_string(s, &w, &h);
        x = 18; //数字时间显示的 x 坐标
        y = 10; //数字时间显示的 y 坐标
       x1=main_LCD_dt_object.time.x=x;
       y1=main_LCD_dt_object.time.y=y;
        main_LCD_dt_object.time.width = w + (DT_DISPLAY_GAP << 1);
      main_LCD_dt_object.time.height = h + (DT_DISPLAY_GAP << 1);
      if (enable_main_LCD_date_time_display)
      {
         gui_set_text_color(main_LCD_dt_object.time.text_color);
         gui_set_text_border_color(main_LCD_dt_object.time.text_border_color);
         if (r2lMMIFlag)
         {
             gui_move_text_cursor(x + main_LCD_dt_object.time.width - 1, y + 1);
         }
        else
        {
            gui_move_text_cursor(x, y + 1);
        }
        x2 = x1 + main_LCD_dt_object.time.width - 1;
        y2 = y1 + main_LCD_dt_object.time.height - 1;
        #ifdef _MMI_UI_DALMATIAN_IDLESCREEN_
         gui_set_text_clip(x1 - 1, y1, x2, y2);
        #else
         if(!PhnsetGetTimeFormat())//如果当前时间格式是 12 小时制
         {
            //修改上面的字符串显示区域
           gui_set_text_clip(x1,y1,x2 - 34,y2);
         }
         else
         {
           gui_set_text_clip(x1,y1,x2 ,y2);
         }
        #endif
        gui_print_bordered_text(s);
        if(!PhnsetGetTimeFormat())//add by chenzhp 12/10, 修改 12 小时格式的时间显示
        {
             gui_set_text_clip(35,y1 - 1,115,78);
             x = -5;
             y = 40;
             gui_set_font(main_LCD_dt_object.time.font);
             gui_move_text_cursor(x,y+1);
             gui_print_bordered_text(s);
        }
           }
        #ifdef _MMI_UI_TECHNO_IDLESCREEN_BAR_
                  break;
          }
      #endif /* _MMI_UI_TECHNO_IDLESCREEN_BAR_ */
             gui_set_font(&font2);
             /* get current date in string format */
```

```
date_string(&main_LCD_dt_object.date.t,
            (UI_string_type) s, main_LCD_dt_object.flags);
/* calculate date string width */
gui_measure_string(s, &w, &h);
/* calculate start x,y and end x,y position of date string display */

x = 100;//日期显示的x坐标
y = 30; //日期显示的y坐标
x1 = main_LCD_dt_object.date.x = x;
y1 = main_LCD_dt_object.date.y = y;
main_LCD_dt_object.date.width = w + (DT_DISPLAY_GAP << 1);
main_LCD_dt_object.date.height = h + (DT_DISPLAY_GAP << 1);
if (enable_main_LCD_date_time_display)
{
    gui_set_text_color(main_LCD_dt_object.date.text_color);
    gui_set_text_border_color(main_LCD_dt_object.date.text_border_color);

#ifdef _MMI_UI_DALMATIAN_IDLESCREEN_
    /* To avoid passing y+1 to the curser move function */
    if (r21MMIFlag)
    {
        gui_move_text_cursor(x + main_LCD_dt_object.date.width - 1, y);
    }
    else
    {
        gui_move_text_cursor(x, y);
    }
#else /* _MMI_UI_DALMATIAN_IDLESCREEN_ */
    if (r21MMIFlag)
    {
        gui_move_text_cursor(x + main_LCD_dt_object.date.width - 1, y + 1);
    }
    else
    {
        gui_move_text_cursor(x, y + 1);
    }
#endif /* _MMI_UI_DALMATIAN_IDLESCREEN_ */
    x2 = x1 + main_LCD_dt_object.date.width - 1;
    y2 = y1 + main_LCD_dt_object.date.height - 1;
#ifdef _MMI_UI_DALMATIAN_IDLESCREEN_
    gui_set_clip(x1 - 1, y1 - 1, x2, y2);
#else
    gui_set_clip(x1 - 1, y1 - 1, x2 + 1, y2 + 1);
#endif

#ifdef _MMI_UI_TECHNO_IDLESCREEN_BAR_
    gui_print_bordered_text(s);
#endif
//调用显示星期的函数
show_main_LCD_day_display();
#ifdef _MMI_UI_TECHNO_IDLESCREEN_BAR_
gdi_layer_blt_previous(x1<old_x1?x1:old_x1,
                       y1<old_y1?y1:old_y1,
                       224,
                       y2+28>old_y2+28?y2+28:old_y2+ 28);
#endif /* _MMI_UI_TECHNO_IDLESCREEN_BAR_ */
```

```
        }
        if (enable_main_LCD_date_time_display)
        {
            dm_coordinates coordinates;

            coordinates.s16X = main_LCD_dt_object.time.x;
            coordinates.s16Y = main_LCD_dt_object.time.y;
            coordinates.s16Width = main_LCD_dt_object.time.width;
            coordinates.s16Height = main_LCD_dt_object.time.height;
            setup_previous_control(&coordinates);
        }
#else
  /*原始代码*/
  gui_set_font(main_LCD_dt_object.time.font);
  …
#endif
      }
    break;
```

找到函数 hide_main_LCD_date_time，修改如下部分代码：

```
#ifdef _N73_STYLE_
            //日期显示的裁剪区域起始横坐标距离字符串 5 个像素
            x1=main_LCD_dt_object.date.x - 5;
            y1=main_LCD_dt_object.date.y;
            x2=main_LCD_dt_object.date.width+x1-1;
            y2=main_LCD_dt_object.date.height+y1-1;
            main_LCD_dt_object.date.hide_function(x1,y1,x2,y2);
            if ( x1 > main_LCD_dt_object.time.x )
                 x1=main_LCD_dt_object.time.x;
            y1=main_LCD_dt_object.time.y-1;
            x2 = main_LCD_dt_object.time.width + main_LCD_dt_object.time.x-1;
            y2=main_LCD_dt_object.time.y+main_LCD_dt_object.time.height-1 + 21;
           if(!PhnsetGetTimeFormat())//add by chenzhp 2007/12/11
            {
                main_LCD_dt_object.time.hide_function(x1,y1,x2 - 34,y2);
            }
            else
            {
                //call hide function of idle,netwroksearch,call screen
               main_LCD_dt_object.time.hide_function(x1,y1,x2,y2);
           }

           if((main_LCD_dt_object.flags & DT_FLAGS_TYPE) == DT_IDLE_SCREEN)
           {
                x1 = 195; //设置星期显示时的显示区域
                y1 = 30;
                x2 = 217;
                y2 = 52;
                MMI_main_LCD_dt_day_hide_function(x1, y1, x2, y2);
           }
#else
      /*原始代码*/
      …
       #endif /* _MMI_UI_TECHNO_IDLESCREEN_BAR_ */
#endif
      }
    break;
```

```
    case DT_TABBED_CALL_SCREEN:
    case DT_ACTIVE_CALL_SCREEN:
…
```

接下来，需要调整星期的显示，修改函数 show_main_LCD_day_display 内部代码：

```
#ifdef _N73_STYLE_
      /* 星期显示的裁剪区域坐标*/
      x1 = 195;
      y1 = 30;
      x2 = 217;
      y2 = 52;
      gdi_layer_push_clip();
      gui_set_text_clip(x1 , y1, x2, y2);
      //PMT HIMANSHU START 20051012
      /* Day string is not coming at right location in case of r2lMMIFlag is ON.
      So move the cursor at correct position */
      if(r2lMMIFlag)
          gui_move_text_cursor(x1 + w, y1);
      else
          gui_move_text_cursor(x1, y1);
#else
  /*原始代码*/
  …
#endif
    /* PMT HIMANSHU END 20051012 */
    gui_print_bordered_text(day_string);
    gdi_layer_pop_clip();
…
```

由于目前我们不需要模拟时钟，因此屏蔽 wgui_set_clock_type 函数中对模拟时钟的设置。

```
void wgui_set_clock_type(U8 clock_type)
{
    /*----------------------------------------------------------------*/
    switch (clock_type)
    {
 #ifdef _N73_STYLE_
      case DIGITAL:
          g_clock_type = DIGITAL;
          break;
      default:
      g_clock_type = DIGITAL;
      break;
#else
  /*原始代码*/
  …
#endif
    }
}
```

由于原始代码使用 dt_update_analog_clock 来读取时间，而目前数字时钟显然不能够使用这种方式，因此，使用函数 update_main_LCD_date_time 来读取系统内部时间，并且将时间转换为一个字符串的形式。修改如下：

```
    …
    MMI_main_LCD_dt_display_function = show_main_LCD_date_time;
#ifdef _N73_STYLE_
```

```
    MMI_main_LCD_dt_update_function = update_main_LCD_date_time;
#else
    #ifdef __MMI_UI_TECHNO_IDLESCREEN_BAR__
        MMI_main_LCD_dt_update_function = dt_update_analog_clock;
        #else
       MMI_main_LCD_dt_update_function = update_main_LCD_date_time;
     #endif
#endif
    MMI_main_LCD_dt_update_function();  /* upadet date tiem */
    …
```

现在，保存所有修改，编译并运行模拟器，初步效果如图 15.9 和图 15.10 所示。

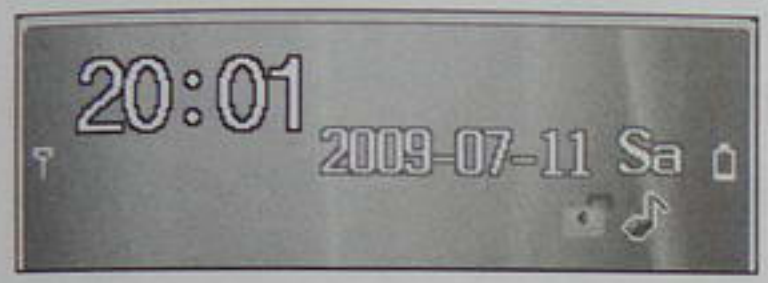


图 15.9　24 小时格式

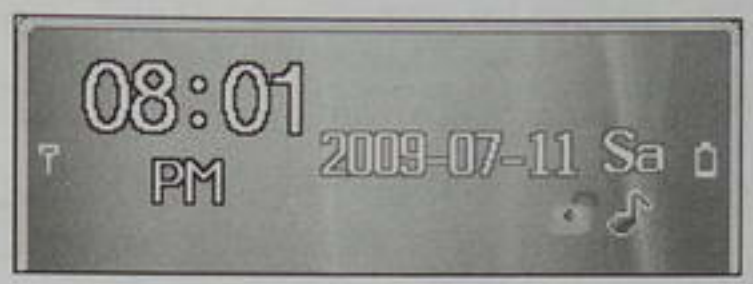


图 15.10　12 小时格式

如此，目前的修改能够保证两种时间格式都正常显示和刷新。读者还可以将时间调整到晚上的 23:59 分，来验证过一分钟后，状态栏的时间是否正常刷新。

提　示

上述代码中，我们为了讲解方便，直接使用了具体的数字用于坐标调整。更好的方法是设置一些宏来进行控制。

另外，如果在开发过程中，出现某些区域不能正常显示的情况，请检查是否被其他区域如星期显示的区域遮盖了。

另外，读者可能会遇到这样的问题，当每次新版本烧到真机板上的时候，系统时间都会默认为 MTK 软件版本定制的初始日期，如：2006/01/01。这显然不利于产品的发布，因此需要修改每个新版本的默认日期。

打开文件 Custom_hw_default.c，修改如下语句：

```
const kal_uint8 DEFAULT_HARDWARE_YEAR = 9; //9 表示 2009 年，同理，8 表示 2008 年
const kal_uint8 DEFAULT_HARDWARE_MON = 1;//具体的月份
const kal_uint8 DEFAULT_HARDWARE_DAY = 1;//具体的号
```

15.4.5　状态栏刷新

现在，我们看到日期、电池和信号都已显示正常，是不是表示状态栏的修改就已经完成了？答案是否定的。

让我们通过一个试验来验证一下。

单击 Network Simulator 窗口中的电量状态模拟图标“Battery/Charger Status”，如图 15.11 所示。

这时，待机界面中的电量状态图标会显示满格。但是，随之问题也出现了：状态栏中的时间、日期和星期的显示全部被覆盖，如图 15.12 所示。

图 15.11 模拟电量状态

图 15.12 文字被遮盖

让我们仔细分析这个异常产生的过程。很明显，当电池电量状态改变时，将会调用相关的函数进行图标重绘。MTK 原始代码会调用函数 ChangeStatusIconLevel 来绘制电量或信号等会进行帧改变的图标。

在函数 ChangeStatusIconLevel 中，我们看到如下一段代码：

```
if ((icon_ID == STATUS_ICON_SIGNAL_STRENGTH) ||
    (icon_ID == STATUS_ICON_BATTERY_STRENGTH))
{
     refresh_status_icon_display();
}
```

这也就是说，当信号图标或者电量图标的状态改变时，将调用函数 refresh_status_icon_display 进行重绘。而上述函数会将整个状态栏全部重绘，这样就带来了一个问题：先前显示的日期时间等内容都会被覆盖。这也就是前面发生的日期和时间消失的原因。

也许有读者会问，为什么在我们没有修改之前，用哪个函数都是可以的？

这是因为按照 N73 的风格修改了状态栏之后，状态栏的高度增加了，达到了 60 个像素。日期和时间及星期也显示在新的状态栏中。

而没有修改之前，信号和电量图标都是显示在高度为 18 的状态栏里，日期和时间以及星期的显示都位于状态栏下方，那么当需要重绘电量或者信号图标的时候，调用 refresh_status_icon_display 刷新整个状态栏而不会影响屏幕其他内容，也就不会有被遮盖的异常了。

现在，为了解决被遮盖的问题，我们就不能直接重绘整个状态栏了，而是根据需要绘制具体的某一个图标。MTK 提供了这样一个函数：

```
void refresh_fixed_status_icon(S16 icon_id, S16 bar_id)
```

参数 icon_id：需要重绘的图标 ID。

参数 bar_id：状态栏 ID，icon_id 附属于它。

很明显，只要将需要重绘的图标 ID 传入上述函数，即可实现单独重绘而不会影响到整个状态栏。而我们的电量和信号属于 0 号状态栏，代码修改如下：

```
…
#else
 if ((icon_ID == STATUS_ICON_SIGNAL_STRENGTH)
    || (icon_ID == STATUS_ICON_BATTERY_STRENGTH))
 {
#ifdef __N73_STYLE__
     refresh_fixed_status_icon(icon_ID, 0);
#else
     refresh_status_icon_display();
```

```
#endif
}
#endif /* __MMI_UI_DALMATIAN_STATUSBAR__ */
...
```

保存代码，编译并运行模拟器。再次单击电量模拟图标，可以看到现在的电量图标已经变为满格，而状态栏上其他的内容仍然正常显示。同时我们也可以模拟信号显示，单击 Network Simulator 窗口中的异步事件菜单，如图 15.13 所示。

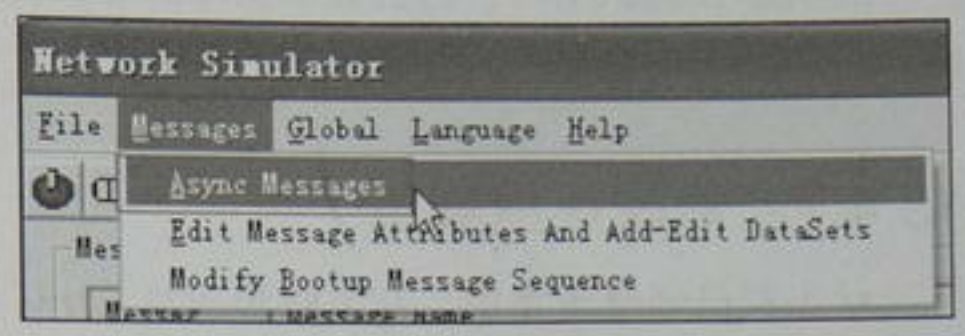


图 15.13　异步消息模拟

进入菜单后，双击展开“Events”节点，可以看到有大量的事件供开发者模拟。找到信号级数标识“Signal Level Indication”节点，展开后，就可以进行 5 种信号状态的模拟。

单击 Signal 3 就表示信号格数为 3 格，如图 15.14 所示。

现在，即可看到状态栏上电量和信号的图标与日期等内容共处，不会再有遮盖异常了，如图 15.15 所示。

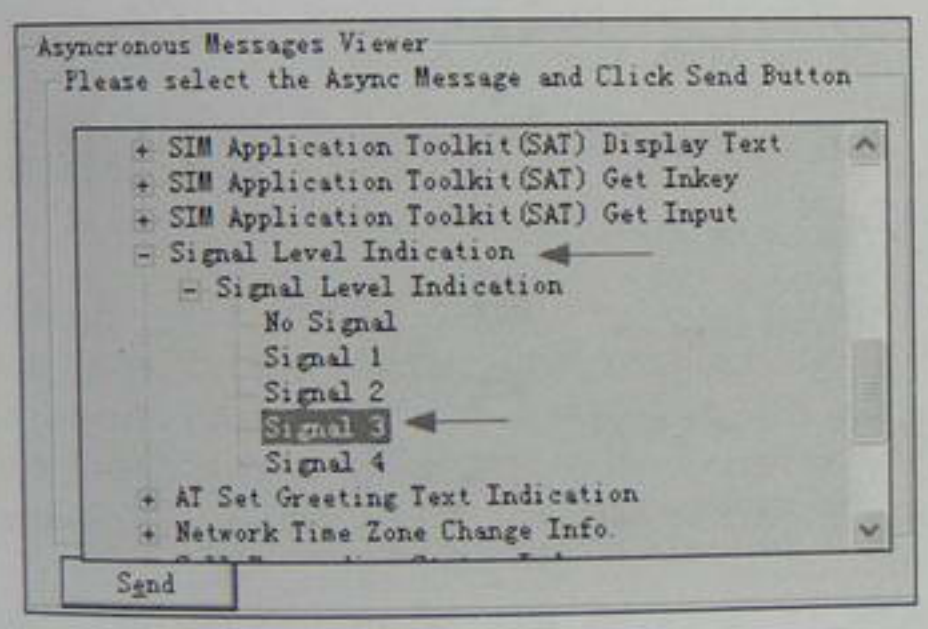


图 15.14　模拟 3 格信号量

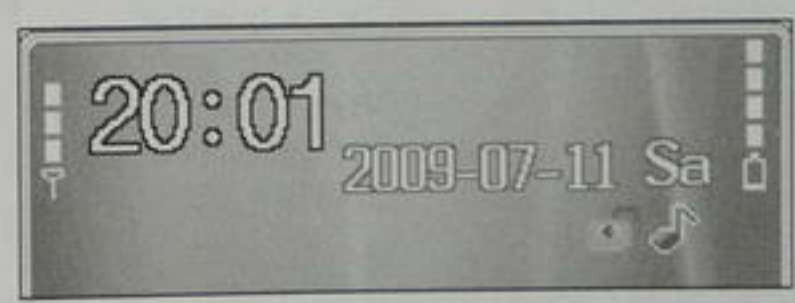


图 15.15　正常显示

15.4.6　运营商名称

默认情况下，运营商的名称是放在待机界面中下部分的，请看本章前面的截图。

按照 N73 的风格，应该跳转到日期显示的上方。修改运营商名称位置很简单，打开文件 wgui_categories_idlescreen.c，查找函数 idle_screen_show_network_details，在内部如下位置调整坐标：

```
...
    #ifdef __MMI_UI_STATUS_BAR_AT_BOTTOM__
        y = MMI_status_bar_height + 4;
    #else
        y = MMI_status_bar_height + 2;
    #endif
    #endif

    #ifdef __N73_STYLE__
     x += 25;
     y = 5;
```

```
    #endif

    gui_move_text_cursor(x, y);
    gui_print_bordered_text(idle_screen_network_name);
...
```

保存代码，编译并运行模拟器，效果如图 15.16 所示。

当烧制在真机上时，就会显示实际的运营商名称，如“中国联通”或“中国移动”。

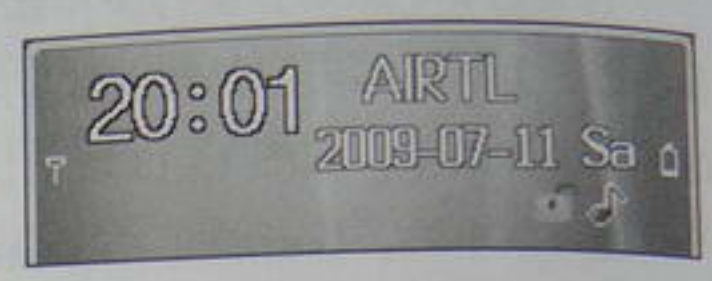


图 15.16 运营商显示

15.4.7 修改快捷菜单

状态栏已经修改完毕，其风格和 N73 如出一辙。现在，让我们开始对待机界面中最重要的快捷菜单进行修改。

默认情况下，MTK 提供了 4 个快捷菜单，并且位于屏幕的下方。在之前我们已经将其整体挪动到了屏幕上方，但是其尺寸过大，而且只有 4 个，没有达到 N73 中快捷菜单的个数。

我们首先要制作如图 15.17 所示的 6 对 GIF 格式、38 × 38 尺寸的图片。

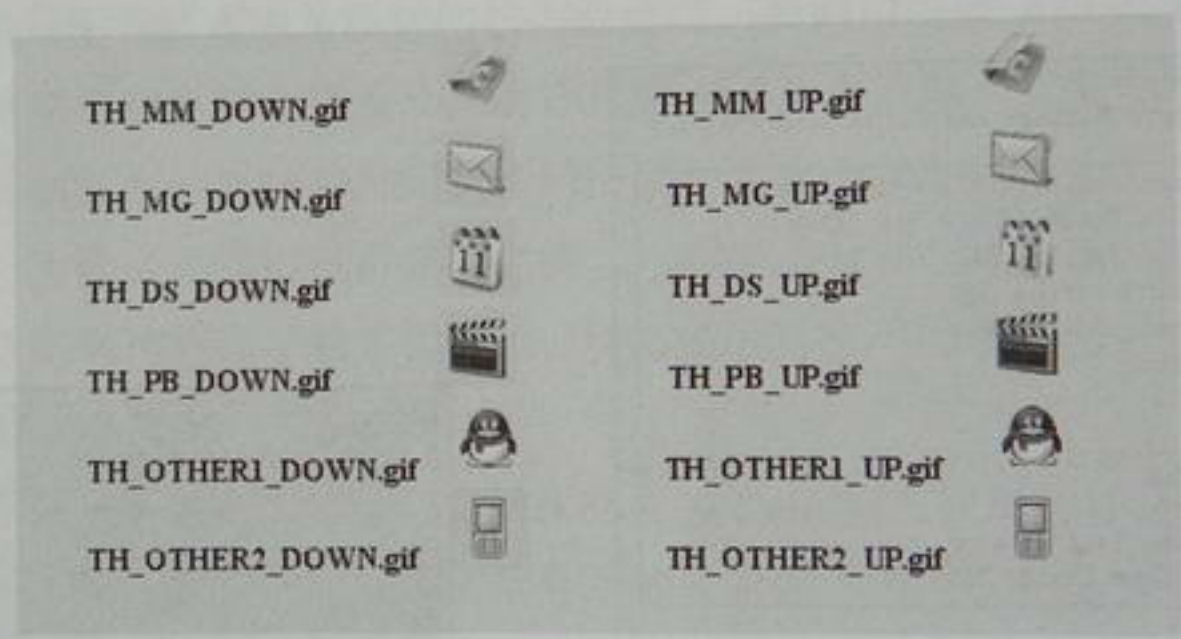


图 15.17 快捷图标的图片资源

进入文件目录\plutommi\Customer\Images\PLUTO240X320\MainLCD\IdleScreen\Touch，该目录中即是待机界面中快捷菜单的集合。原始代码中只有 4 对图片，由于我们的修改需要 6 个快捷图标，该目录将会保存 6 对 GIF 图片。将上述做好的图片复制到该文件目录，将原有文件全部覆盖掉。

说 明

上面的图片列表中，前 8 个图标保持和原始代码中一样的文件名和 ID，这样可以减少工作量。而 TH_OTHER1_DOWN、TH_OTHER1_UP、TH_OTHER2_DOWN、TH_OTHER2_UP 等图片属于新增的。因此，需要在文件 wgui_categories_defs.h 中定义新增的 4 个图片 ID。

定义图片 ID：

```
...
#if 1 /* DON'T CHANGE THE ORDER */
    IMG_IDLE_MAINMENU_SHORCUT,
    IMG_IDLE_MESSAGE_SHORTCUT,
```

```
    IMG_IDLE_PHONEBOOK_SHORTCUT,
    IMG_IDLE_DIALING_SCREEN_SHORTCUT,
#ifdef __N73_STYLE__  //图片正常状态
    IMG_IDLE_OTHER1_SHORTCUT,
    IMG_IDLE_OTHER2_SHORTCUT,
#endif
    IMG_IDLE_MAINMENU_SHORCUT_PRESSED,
    IMG_IDLE_MESSAGE_SHORTCUT_PRESSED,
    IMG_IDLE_PHONEBOOK_SHORTCUT_PRESSED,
    IMG_IDLE_DIALING_SCREEN_SHORTCUT_PRESSED,
#ifdef __N73_STYLE__  //图片按下状态
    IMG_IDLE_OTHER1_SHORTCUT_PRESSED,
    IMG_IDLE_OTHER2_SHORTCUT_PRESSED,
#endif
#endif
…
```

然后，进入目录.\plutommi\customer\custresource\pluto_mmi\res_mmi 中，找到文件 Res_IdleScreen.c，将图片从资源中载入。

```
  #if defined(_MMI_TOUCH_IDLESCREEN_SHORTCUTS_)
  ADD_APPLICATION_IMAGE2(IMG_IDLE_MAINMENU_SHORCUT,CUST_IMG_PATH"\\\\MainLCD\\\\IdleScre
en\\\\Touch\\\\TH_MM_UP.gif","Idle Screen Main Menu shorcut image.");
  …
  ADD_APPLICATION_IMAGE2(IMG_IDLE_DIALING_SCREEN_SHORTCUT_PRESSED,CUST_IMG_PATH"\\\\MainL
CD\\\\IdleScreen\\\\Touch\\\\TH_DS_DOWN.gif","Idle Screen Dialing Screen shorcut pressed
image.");

  #ifdef __N73_STYLE__
  ADD_APPLICATION_IMAGE2(IMG_IDLE_OTHER1_SHORTCUT,CUST_IMG_PATH"\\\\MainLCD\\\\IdleScree
n\\\\Touch\\\\TH_OTHER1_UP.gif","Idle Screen Dialing Screen shorcut image.");
  ADD_APPLICATION_IMAGE2(IMG_IDLE_OTHER1_SHORTCUT_PRESSED,CUST_IMG_PATH"\\\\MainLCD\\\\I
dleScreen\\\\Touch\\\\TH_OTHER1_DOWN.gif","Idle Screen Dialing Screen shorcut image.");

  ADD_APPLICATION_IMAGE2(IMG_IDLE_OTHER2_SHORTCUT,CUST_IMG_PATH"\\\\MainLCD\\\\IdleScreen
\\\\Touch\\\\TH_OTHER2_UP.gif","Idle Screen Dialing Screen shorcut image.");

  ADD_APPLICATION_IMAGE2(IMG_IDLE_OTHER2_SHORTCUT_PRESSED,CUST_IMG_PATH"\\\\MainLCD\\\\
IdleScreen\\\\Touch\\\\TH_OTHER2_DOWN.gif","Idle Screen Dialing Screen shorcut image.");

  #endif
  #endif /* __MMI_TOUCH_IDLESCREEN_SHORTCUTS__ */
```

保存代码，由于修改了资源文件，因此需要在命令行中对整个工程进行 update 操作。完毕后，编译并运行模拟器，初步效果如图 15.18 所示。

目前，待机界面上只有 4 个快捷菜单，因为我们还没有修改代码逻辑，所以，接下来的主要工作就是要让 6 个快捷菜单同时显示出来。

打开 wgui_categories_idlescreen.c 文件，首先在文件中修改快捷菜单个数的宏。

```
…
#ifdef __N73_STYLE__
#define MAX_IDLE_SCREEN_SHORTCUTS   6
#else
#define MAX_IDLE_SCREEN_SHORTCUTS   4
```

```
#endif
...
```

再次编译并运行代码，在模拟器上就可以看到 6 个快捷图标出现了，如图 15.19 所示。

图 15.18　初步效果图

图 15.19　快捷菜单调整

现在，6 个图标已经很整齐地排列在屏幕上了。但是我们单击某些图标，却没有反应，无法跳转到另外一个界面去，这是由于目前我们只是增加了图标，而没有绑定图标的响应函数。

15.4.8　注册快捷菜单事件

在函数 setup_touch_idle_screen_shortcuts 中，可以看到如下代码：

```
  ...
touch_idle_shortcuts[0].entryfn = EntryMainMenuFromIdleScreen;
/* PMT HIMANSHU END 20050721 */
touch_idle_shortcuts[1].entryfn = EntryScrMessagesMenuList;
touch_idle_shortcuts[2].entryfn = mmi_phb_entry_main_menu;
touch_idle_shortcuts[3].entryfn = IdleScreenDigitHandler;
...
```

上述代码片段就是原始代码中对待机界面的 4 个快捷菜单进行响应的函数的设置。现在，我们增加了 2 个快捷菜单，所以，touch_idle_shortcuts 数组将会有 6 个元素。其中，每个图标进入的函数如下。

- 图标对应 touch_idle_shortcuts[0]，响应函数 mmi_phb_entry_main_menu，进入电话本界面。
- 图标对应 touch_idle_shortcuts[1]，响应函数 EntryScrMessagesMenuList，进入短信界面。
- 图标对应 touch_idle_shortcuts[2]，响应函数 mmi_audply_entry_main，进入音频播放界面。
- 图标对应 touch_idle_shortcuts[3]，响应函数 ClndrPreEntryApp，进入日历界面。
- 图标对应 touch_idle_shortcuts[4]，响应函数 mmi_fng_entry_screen，进入娱乐界面。
- 图标对应 touch_idle_shortcuts[5]，响应函数 IdleScreenDigitHandler，进入拨号盘界面。

由于新增了日历界面、音频界面和娱乐界面，所以，需要在当前函数之前引用 3 个函数：

```
extern void mmi_audply_entry_main(void);//add by chenzhp
extern void ClndrPreEntryApp(void);
extern void mmi_fng_entry_screen(void);
```

接着修改函数内部代码：

```
#ifdef __N73_STYLE__
    touch_idle_shortcuts[0].entryfn = mmi_phb_entry_main_menu;
    touch_idle_shortcuts[1].entryfn =EntryScrMessagesMenuList;
    touch_idle_shortcuts[2].entryfn = mmi_audply_entry_main;
    touch_idle_shortcuts[3].entryfn = ClndrPreEntryApp;
    touch_idle_shortcuts[4].entryfn = mmi_fng_entry_screen;
    touch_idle_shortcuts[5].entryfn = IdleScreenDigitHandler;
#else
    //PMT HIMANSHU START 20050721
    //touch_idle_shortcuts[0].entryfn = NULL; /* Needs to be decided */
    touch_idle_shortcuts[0].entryfn = EntryMainMenuFromIdleScreen;
    /* PMT HIMANSHU END 20050721 */
    touch_idle_shortcuts[1].entryfn = EntryScrMessagesMenuList;
    touch_idle_shortcuts[2].entryfn = mmi_phb_entry_main_menu;
    touch_idle_shortcuts[3].entryfn = IdleScreenDigitHandler;
#endif
```

保存修改，编译并运行模拟器。当我们再次分别单击 6 个图标时，即可看到图标正常地跳转了。

接着，我们继续在当前函数中修改快捷图标的位置。最初我们已经将图标的位置移到了屏幕上方，并且定义了一个宏来标识高度。

```
#ifdef __N73_STYLE__
        touch_idle_shortcuts[i].y = SHORTCUT_POS;
#else
        touch_idle_shortcuts[i].y = UI_device_height - touch_idle_shortcuts[i].height;
#endif
```

如果读者需要校准所有快捷图标的高度，可以修改 SHORTCUT_POS 的声明。这里，我们稍微将高度往上移动 10 个像素。

```
#define SHORTCUT_POS                    80
```

现在，快捷图标已经正确地摆放在待机界面上，并能够正确地响应函数了。

15.4.9 添加快捷菜单安全区

安全区实际上就是位于快捷菜单上下方的两条虚线。

在函数 display_touch_idle_screen_shortcuts 中，修改代码：

```
void display_touch_idle_screen_shortcuts(void)
{
  ...
max_height = max_touch_idle_screen_shortcut_height();
#ifdef __N73_STYLE__
    UI_draw_dotted_horizontal_line(0,239,77,gui_color(255,255,255));
    UI_draw_dotted_horizontal_line(0,239,120,gui_color(255,255,255));
#endif
  /* PMT HIMANSHU END 20050809 */
  gui_unlock_double_buffer();
  ...
}
```

虚线其实是由一个个前后有间隔的白色像素点组成的。在上面的代码中，我们调用了系统自带的

绘制水平虚线的函数进行绘制，效果如图 15.20 所示。

现在，就界面效果来说，已经和 N73 的没有什么区别了。但是，前面已经说过，UI 的设计其实不仅仅是一个界面那么简单，更为重要和关键的还是后台的工作。

如果我们在 N73 的手机后台上播放一首歌曲，那么在待机界面就会显示播放的内容。如果我们新增了日历项，那么也会在快捷菜单下方紧挨着显示一个列表项，我们通过左右上下按键能够在快捷菜单和列表项之间来回切换。

但是，这些功能在我们的待机界面上都没有实现。所以，接下来的工作就是实现待机界面上的各种控制行为。

图 15.20 绘制间隔线

15.5 动态列表

在 N73 的待机界面中，如果当前新增了日历项，会在快捷菜单下方增加一行日历项的描述信息；而此时在后台播放音乐的时候，又会在日历项下方再增加一行信息显示当前播放的内容。

很显然，需要使用动态列表控件来完成这一功能。

15.5.1 添加动态列表控件

由于待机界面也使用了屏幕模板，因此，要在该屏幕中增加一个控件，最快速的方法就是在其控件数组中增加一个具体定义。

首先打开文件 CustCoordinates.c 进行设置。

待机界面使用的屏幕 ID 是 MMI_CATEGORY33_ID。通过这个 ID 即可找到定义屏幕内部控件元素和绘制的两个数组：category33 和 coordinate_set33。

{MMI_CATEGORY33_ID, (U8 *) category33, (S16 *) coordinate_set33, NULL}，关于这两个数组的具体细节和使用，在之前的章节已经讲解过，这里不再赘述。

现在，让我们把动态列表加入到数组中去：

```
#ifdef __N73_STYLE__
    const U8 category33[]=
    {
     5,   //增加了动态列表控件，所以元素变为 5。请一定注意这里的修改
     DM_WALL_PAPER,
     DM_BASE_LAYER_START,
     DM_CATEGORY_CONTROLLED_AREA,
     DM_DYNAMIC_LIST1,   //将动态列表放进来
     DM_BUTTON_BAR1
    };

const s16 coordinate_set33[]=
{
        DM_FULL_SCREEN_COORDINATE_FLAG,
```

```
            DM_FULL_SCREEN_COORDINATE_FLAG,
            0,0,MAIN_LCD_DEVICE_WIDTH, 119,    DM_NO_FLAGS,
            /*动态列表的坐标及本身的宽度和高度*/
            0,121,MMI_CONTENT_WIDTH, 100, DM_NO_FLAGS,
  #if defined (__MMI_IDLE_FULL_SCREEN__)
            0, MAIN_LCD_DEVICE_HEIGHT-MMI_BUTTON_BAR_HEIGHT,
            MAIN_LCD_DEVICE_WIDTH,MMI_BUTTON_BAR_HEIGHT, DM_BUTTON_DISABLE_BACKGROUND, MMI_
SOFTKEY_WIDTH
  #else
            DM_DEFAULT_BUTTON_BAR_FLAG,MMI_SOFTKEY_WIDTH
  #endif
  };
  #else
  /*原始 Category33 和 coordinate_set33 的定义*/
  ...
  #endif
```

保存修改，控件就已经加入到了待机界面中。但是，目前动态列表还无法确定有几个列表项以及列表项要显示的内容，所以，动态列表此时显示的是一片空白，如图 15.21 所示。

图 15.21　空白的动态列表

因此，接下来要做其控件本身的修改。

15.5.2　今天无日历项

根据 N73 的风格，日历项显示在第一项，随后才是音乐播放。由于日历项将会调用到备忘录和日期计算相关的功能，所以需要对备忘录和系统时间相关的函数进行引用。同样的道理，还需要引用与音乐播放器相关的函数。因此，需要在 wgui_categories_idlescreen.c 文件的开头增加如下代码：

```
#ifdef __N73_STYLE__
#include "ToDoListDef.h"          //备忘录相关的头文件
#include "custom_mmi_default_value.h"
#include "DateTimeType.h"         //系统时间相关的头文件
#include "app_datetime.h"
#ifdef __MMI_AUDIO_PLAYER__
extern audply_struct g_audply;
scrolling_text scroll_audply_idleText;
```

```
#endif /* __MMI_AUDIO_PLAYER__ */
extern tdl_context_struct g_tdl_cntx;
extern void mmi_audply_entry_main(void); //音乐播放的入口函数
#endif
```

> **说　明**
>
> 上述的变量一定要放在文件开头的#ifdef __MMI_AUDIO_PLAYER__之后，因为我们新增的一些头文件和全局变量要依赖于#ifdef __MMI_AUDIO_PLAYER__之中的文件。

同时，我们需要增加一个字符串“今天无日历项”，ID 命名为 STR_TODOLIST_NONE，定义在待机界面的头文件 IdleAppDef.h 中。

```
enum STR_IDLE_SCREEN_LIST
{
…
KEYPAD_LOCKED_RSK_TEXT,
#ifdef __N73_STYLE__
   STR_TODOLIST_NONE,
#endif
#ifdef __MMI_TOUCH_SCREEN__
   SCREEN_LOCKED_TEXT,
#endif
…
```

接着，在文件 Res_idlescreen.c 中定义：

```
#ifdef __N73_STYLE__
ADD_APPLICATION_STRING2(STR_TODOLIST_NONE,
                        " No calendar entries today "," No calendar entries today ");
#endif
```

然后在目录\plutommi\Customer\CustResource\PLUTO_MMI 中打开 ref_list.txt 文件附加字符串资源。

```
STR_TODOLIST_NONE    Undefined 40    No calendar entries today    No    calendar    entries
today    今天无日历项今天无日历项
```

绘制两个图标。

日历项图标：IMG_IDLE_TODOLIST.pbm，大小 22 × 22。

音乐播放图标：IMG_IDLE_MUSIC.pbm，大小 22 × 22。

将图标文件放在\plutommi\Customer\Images\PLUTO240X320\MainLCD\IdleScreen\Touch 目录下并打包。

15.5.3 列表初始化

如何初始化和使用列表控件，大家其实可以从源代码中找到一些答案。任何控件的使用，基本上都遵循以下的流程：创建，初始化，注册响应函数，重绘。

按照上面的流程，修改 wgui_categories_idlescreen.c 文件中的代码。

```
void ShowCategory33Screen_ext(
        U16 left_softkey,
        U16 left_softkey_icon,
        U16 right_softkey,
        U16 right_softkey_icon,
        U8 *history_buffer)
```

```
{
…
#ifdef __N73_STYLE__
 //如果是飞行模式，就不显示列表
 if(mmi_bootup_get_active_flight_mode() == 0)
 {
     //创建含有图标和文本的菜单项
     create_fixed_icontext_menuitems();
     //菜单项以列表方式显示
     MMI_current_menu_type=LIST_MENU;
     //将菜单项与列表框架联系起来
     associate_fixed_icontext_list();
     /*根据 ItemsOfCat33 参数来决定列表中菜单项的个数，并为每个菜单项设置一个回调函数 Cat33
GetItemCallback*/
     init_dynamic_item_buffer(ItemsOfCat33,
                              Cat33GetItemCallback,
                              NULL,
                              UI_dummy_function);
     //菜单项的个数
     MMI_fixed_list_menu.n_items = ItemsOfCat33;
     register_menu_shortcut_keys();
  /*关闭菜单项的背景，左上角对齐，去掉滚动条*/
  MMI_fixed_list_menu.flags |= UI_LIST_MENU_DISABLE_BACKGROUND
                               |UI_LIST_MENU_ALIGN_TO_TOP
                               |UI_LIST_MENU_DISABLE_SCROLLBAR;
  MMI_fixed_icontext_menuitem.flags |= UI_MENUITEM_DISABLE_HIGHLIGHT_DISPLAY;
  resize_dynamic_icontext_menuitems_to_list_width();
  wgui_register_list_item_selected_callback_all(Cat33hlight);
   }
#endif
…
}
```

说　明

一般情况下，在待机界面中会显示日历项内容和背景播放时候的音乐标题，只有两个菜单项，因此，ItemsOfCat33 一般被赋值为 2。

在上面的代码中，我们要着重讲一下 init_dynamic_item_buffer 函数，其函数原型如下：

```
void init_dynamic_item_buffer(
        S32 n_items,
        GetItemFuncPtr get_item_func,
        GetHintFuncPtr get_hint_func,
        FlushDataFuncPtr flush_data_func)
```

参数 n_items：

函数将根据 n_items 的值来设置列表中菜单项的个数。

参数 get_item_fun：

回调函数，用于填充菜单项内容的回调函数，我们将设计一个函数 Cat33GetItemCallback 完成此任务，并将函数地址传递给 get_item_fun。

参数 get_hint_func：

回调函数，高亮菜单项时，显示提示信息，在目前的修改中，没有用到。

参数 flush_data_func：

回调函数，用于同步刷新缓存中的数据，暂时不用。

下面的代码就是用于填充菜单项内容的函数实现。

```
#ifdef __N73_STYLE__
pBOOL Cat33GetItemCallback(S32 item_idx, UI_string_type str_buff, UI_image_type *img_buff_p,
                           U8 str_img_mask)
{
    MYTIME currentTime;
    U8 i = 0;
    DTGetRTCTime(&currentTime);
    if(item_idx == 0)//当前是第一个菜单项，也就是“今天无日历项”
    {
     if(g_tdl_cntx.TotalListedTask== 0&&g_tdl_cntx.NoteBuff[0]==0)
     {
      //如果用户没有设置日历项，那么就将菜单项内容设为默认
      UCS2Strcpy((PS8)str_buff, (PS8) get_string(STR_TODOLIST_NONE));
     }
     else  //如果有日历项
     {
      memset(&g_tdl_cntx.NoteBuff,0,MAX_TODO_LIST_NOTE);
      for(i = 0;i < NUM_OF_TDL;i++)
      {   //如果当前日期在日历项的时间区间之内，就读取数据
          if(g_tdl_cntx.TaskList[i].Present == 1
           &&((g_tdl_cntx.TaskList[i].EndTime.nYear == currentTime.nYear)
           &&(g_tdl_cntx.TaskList[i].EndTime.nMonth == currentTime.nMonth)
           &&(g_tdl_cntx.TaskList[i].EndTime.nDay == currentTime.nDay)))
           {
             pfnUnicodeStrcpy(g_tdl_cntx.NoteBuff,(S8*) g_tdl_cntx.TaskList[i].Note));
            //将读出的日历项数据复制到 str_buff，等待显示
             UCS2Strcpy((PS8)str_buff, (PS8)g_tdl_cntx.NoteBuff);
              break;
           }
      }
      if(g_tdl_cntx.NoteBuff[0]==0)
      {
           UCS2Strcpy((PS8)str_buff, (PS8) get_string( STR_TODOLIST_NONE));
      }
     }
     //获得显示日历项的菜单项所用图标
     *img_buff_p = (UI_image_type)GetImage(IMG_IDLE_TODOLIST);
    }
    //如果列表菜单项是音乐播放
    if(item_idx == 1)
    {
     //将正在播放的音乐标题内容复制给 str_buff，用于随后显示
     UCS2Strcpy((PS8)str_buff, (PS8)g_audply.title);
     //获得该菜单项的图标
     *img_buff_p = (UI_image_type)GetImage(IMG_IDLE_MUSIC);
    }

    return TRUE;
}
#endif
```

通过以上代码，就确定了列表中每个菜单项显示的内容。保存修改，编译并运行模拟器，进入待机界面，效果如图 15.22 所示。

现在，进入 Organizer 界面，选择 TodoList 设置一个备忘录 Reminder，内容为“HELLO!”。保存修改，然后回到待机界面，可以看到，列表中已经正确地显示了，如图 15.23 所示。

图 15.22　动态列表默认值

图 15.23　添加日历项后的列表

但是，有这样的一个问题，如果先前用户已经设置了一些备忘录，随后用户关闭了手机，那么，第二次开机的时候，如何获取日历项的内容，如何将其再次显示出来？

这就需要我们在刚进入待机界面的时候，就读取 NVRAM 中的数据。增加判断代码，将代码添加在如下位置：

```
void ShowCategory33Screen_ext(
        U16 left_softkey,
        U16 left_softkey_icon,
        U16 right_softkey,
        U16 right_softkey_icon,
        U8 *history_buffer)
{
#ifdef __N73_STYLE__
 //如果刚开机，就需要从 NVRAM 中读取 todolist 数据
if(isFirstIn)
{
     TDLReadFromNvram();
     DTGetRTCTime(&currentTime);//add by chenzhp 20080331
     for(i = 0;i < NUM_OF_TDL;i++)
     {   //备忘录设置的时间和当前日期时间相比较，如果没有过期，则读取数据
        if(g_tdl_cntx.TaskList[i].Present == 1
             &&((g_tdl_cntx.TaskList[i].EndTime.nYear == currentTime.nYear)
             &&(g_tdl_cntx.TaskList[i].EndTime.nMonth == currentTime.nMonth)
             &&(g_tdl_cntx.TaskList[i].EndTime.nDay == currentTime.nDay)))
        {
             pfnUnicodeStrcpy(g_tdl_cntx.NoteBuff,(S8*)(g_tdl_cntx.TaskList[i].Note));
             break;
        }
    }
```

```
        isFirstIn = 0;
    }
        //飞行模式不显示 list
        if(mmi_bootup_get_active_flight_mode() == 0)
        {
    …
    }
```

通过以上修改，保证了日历项的判断和显示。

15.5.4 音乐标题显示

MTK 系统播放音乐时，有一个选项是让用户选择是否背景播放。当用户选择背景播放后，在退出播放器界面的同时，不会关闭音乐播放；而选择非背景播放，则退出播放器的同时终止音乐播放。

因此，在退出播放器界面的同时就应该对待机界面中列表的菜单项进行设置。如果当前选择了背景播放，则 ItemsOfCat33 的值由 1 增为 2，如果取消背景播放，则其由 2 减为 1。

打开文件 AudioPlayerSrc.c，在文件头部引用标识待机界面中列表菜单项数的变量 ItemsOfCat33。

```
#ifdef __N73_STYLE__
    extern U8 ItemsOfCat33;
#endif
```

接着，修改函数：

```
void mmi_audply_exit_main()
{
    …
    if(g_audply.audio_cache_proc_buf_p!=NULL)
    {
        mmi_frm_scrmem_free(g_audply.audio_cache_proc_buf_p);
        g_audply.audio_cache_proc_buf_p = NULL ;
    }
#ifdef __N73_STYLE__
    if(ItemsOfCat33 == 1)
    {
        ItemsOfCat33 = 2;
    }
    if(g_audply.state == STATE_IDLE||g_audply.background_play == 0)
    {
        ItemsOfCat33 = 1;
    }
#endif
currHistory.scrnID = SCR_ID_AUDPLY_MAIN;
…
}
```

还要注意的是，用户可能在音乐播放列表中直接退出到待机界面，而不是经过音乐播放器主界面。为了防止这种情况下待机界面的列表项个数产生异常，还需要修改退出列表的函数。

```
void mmi_audply_exit_list()
{
    …
history currHistory;
    /*----------------------------------------------------------------*/
    /* Code Body                                                      */
    /*----------------------------------------------------------------*/
```

```
#ifdef __N73_STYLE__
    if(ItemsOfCat33 == 1)
    {
        ItemsOfCat33 = 2;
    }

    if(g_audply.state == STATE_IDLE||g_audply.background_play == 0)
    {
        ItemsOfCat33 = 1;
    }
#endif
g_audply.in_list_screen = FALSE;
…
}
```

保存修改，通过命令行 remake 操作将版本烧制在板子上，播放音乐即可检验效果。当一首歌播放完毕后，会进行歌曲的切换，而标题也会随之改变。打开文件 AudioPlayerMsgHandler.c，在文件头部增加如下代码：

```
#ifdef __N73_STYLE__
extern U8 ItemsOfCat33;
extern pBOOL Cat33GetItemCallback(S32  item_idx,  UI_string_type   str_buff,
                                  UI_image_type  *img_buff_p, U8  str_img_mask);
#endif
```

当音乐播放完成时：

```
void mmi_audply_play_completed()
{
…
else
{
  if (mmi_audply_apply_picked_file())
  {
      …
      g_audply.need_restore = FALSE;
      g_audply_main_selected_button = (signed char)AUDPLY_MAIN_STOP;
      g_audply_main_selected_button_down = FALSE;
      picked_OK = TRUE;
#ifdef __N73_STYLE__
      if(g_idle_context.IsOnIdleScreen)//必须是在主界面才刷新列表
      {
          init_dynamic_item_buffer(ItemsOfCat33, Cat33GetItemCallback,
                                   NULL, UI_dummy_function);

          redraw_dynamic_list();
      }
#endif
          }
      }
…
}
```

到此，音乐播放和日历项都能够正确地显示了。但是，目前还无法通过单击列表中的菜单项跳转到对应的函数。这个功能在下一节中同按键一起进行完善。

15.5.5 按键切换

到目前为止，我们只能够通过触摸屏单击某个快捷菜单进入对应的界面，而 N73 可以通过左右按

键在这排快捷图标中进行切换，每切换一次，就有一个图标处于高亮状态（获得焦点），此时按 OK 键，就可以进入对应的函数。而且，还可以通过上下键实现在快捷菜单和列表项之间的转换。

而目前的待机界面，没有实现按键切换的功能，下面我们来做个试验。

打开模拟器，按左方向键进入了情景模式界面，按下方向键进入了闹钟设置界面。很显然，这个功能需要我们自定义实现。打开 shortcuts.c 文件，该文件控制所有快捷菜单的行为，其中，我们可以看到 4 个函数：ShctExecuteUpKey、ShctExecuteDownKey、ShctExecuteLeftKey 和 ShctExecuteRightKey。

这些函数就是控制待机界面中方向键行为的函数。为了将当前方向键的行为和 N73 进行重新绑定，就必须修改这 4 个函数。在修改之前，我们需要将该文件中的快捷菜单个数修改为 6 个。在头部定义如下：

```
#ifdef __N73_STYLE__
#define MAX_IDLE_SCREEN_SHORTCUTS  6
S8 g_shortcut_index = -1;
S8 g_index_Vetical = -1;
S8 g_shortcut_index_hide = -1;
extern gui_idle_touch_screen_display_pressed_shorcut_item(S8 index);
extern gui_idle_touch_screen_display_shorcut_item(S8 index);
#endif
```

每个快捷菜单图标都用一个索引 g_shortcut_index 来标识。

g_index_Vetical 用于表示当前焦点是处于列表还是快捷菜单中。因为用户可能通过上下键在列表和快捷菜单之间切换。

g_shortcut_index_hide 表示当前高亮项之前的那一项的索引。一般情况下，在用户按键时，g_shortcut_index 会自增，但在自增之前，需要将自身的值赋值给 g_shortcut_index_hide。

后面紧跟着的函数用来显示正常状态下的快捷菜单图标和按下状态时的快捷菜单图标。

接下来，就将源代码中的 4 个方向键响应函数进行替换。

```
#if defined(__MMI_DEDICATED_KEY_SHORTCUTS__)
...
#ifdef __N73_STYLE__
/*当用户按上键*/
void ShctExecuteUpKey(void)
{
    //默认状态下焦点不在快捷菜单也不在列表上，就将焦点定位到快捷功能
    if (-1 == g_index_Vetical)
    {
        g_index_Vetical = 0;
        g_shortcut_index = 0;
        gui_idle_touch_screen_display_pressed_shorcut_item(0);
    }
    else  if (0 == g_index_Vetical) //焦点处于快捷功能中
    {
        //快捷功能失去焦点，不再高亮显示快捷功能的当前项
        if (g_shortcut_index >= 0 && g_shortcut_index < MAX_IDLE_SCREEN_SHORTCUTS)
        {
            gui_idle_touch_screen_display_shorcut_item(g_shortcut_index);
        }
        //由于焦点已经到达快捷菜单，所以默认高亮的是第一个菜单图标
```

```
        g_shortcut_index = 0;
        //将焦点切换到列表并高亮显示列表的当前项
        g_index_Vetical = 1;
        MMI_fixed_icontext_menuitem.flags &=
        ~ UI_MENUITEM_DISABLE_HIGHLIGHT_DISPLAY;
        if ((MMI_fixed_list_menu.n_items-1) != MMI_fixed_list_menu.highlighted_item)
            dynamic_list_goto_item((MMI_fixed_list_menu.n_items-1));
        else
        {
            fixed_icontext_menuitem_type *t=
(fixed_icontext_menuitem_type*)MMI_fixed_list_menu.items[MMI_fixed_list_menu.n_items-1];
            t->flags |= UI_MENUITEM_STATE_FOCUSSED;
            t->flags |= UI_MENUITEM_STATE_ANIMATE;

            gui_highlight_fixed_icontext_menuitem(MMI_fixed_list_menu.items[MMI_fixed_ list_
menu.n_items-1],MMI_fixed_list_menu.common_item_data);
            redraw_dynamic_list();
        }
    }
    else //焦点处于列表中
    {
        if (MMI_fixed_list_menu.highlighted_item == 0)
        {
            //停止菜单项的文本滚动
            gui_fixed_icontext_menuitem_stop_scroll();
            gui_remove_highlight_fixed_icontext_menuitem(MMI_fixed_list_menu. items [MMI_
fixed_list_menu.n_items-1],MMI_fixed_list_menu.common_item_data);
            //列表失去焦点，不再高亮显示列表的当前项
            MMI_fixed_icontext_menuitem.flags |=
    UI_MENUITEM_DISABLE_HIGHLIGHT_DISPLAY;
            redraw_dynamic_list();
            //将焦点切换到列表中，并让列表第一项高亮显示
            g_index_Vetical = 0;
            g_shortcut_index = 0;
            gui_idle_touch_screen_display_pressed_shorcut_item(0);
        }
        else //让列表的上一项获得焦点
        {
            fixed_list_goto_previous_item();
            redraw_dynamic_list();
        }
    }
}

void ShctExecuteDownKey(void)
{
    //当焦点不在快捷功能也不在列表中时将焦点定位到快捷功能
    if (-1 == g_shortcut_index)
    {
        g_index_Vetical = 0;
        g_shortcut_index = 0;
        gui_idle_touch_screen_display_pressed_shorcut_item(0);
    }
    else if (0 == g_index_Vetical) //焦点处于快捷功能中
    {
```

```
        //快捷功能失去焦点，不再高亮显示快捷功能的当前项
        if (g_shortcut_index >= 0 && g_shortcut_index < MAX_IDLE_SCREEN_SHORTCUTS)
        {
            gui_idle_touch_screen_display_shorcut_item(g_shortcut_index);
        }
        g_shortcut_index = 0;

        //将焦点切换到列表并高亮显示 list 的第一项
        g_index_Vetical = 1;
        MMI_fixed_icontext_menuitem.flags &=
        ~ UI_MENUITEM_DISABLE_HIGHLIGHT_DISPLAY;
        if (0 != MMI_fixed_list_menu.highlighted_item)
            dynamic_list_goto_item(0);
        else
        {
            fixed_icontext_menuitem_type *t=
    (fixed_icontext_menuitem_type*)MMI_fixed_list_menu.items[0];
            t->flags |= UI_MENUITEM_STATE_FOCUSSED;
            t->flags |= UI_MENUITEM_STATE_ANIMATE;
            gui_highlight_fixed_icontext_menuitem(MMI_fixed_list_menu.  items[0],MMI_  fixed_
list_menu.common_item_data);
            redraw_dynamic_list();
        }
    }
    else  //焦点处于列表中
    {
         //将焦点从列表切换到快捷功能
        if (MMI_fixed_list_menu.highlighted_item>=(MMI_fixed_list_menu.n_items-1))
        {
            //停止文本滚动
            gui_fixed_icontext_menuitem_stop_scroll();
        gui_remove_highlight_fixed_icontext_menuitem(
        MMI_fixed_list_menu.items[MMI_fixed_list_menu.n_items-1],
        MMI_fixed_list_menu.common_item_data);

        //列表失去焦点，不再高亮显示 list 的当前项
        MMI_fixed_icontext_menuitem.flags |=
  UI_MENUITEM_DISABLE_HIGHLIGHT_DISPLAY;
        redraw_dynamic_list();

        //将焦点切换到列表中，并让列表第一项高亮显示
        g_index_Vetical = 0;
        g_shortcut_index = 0;
        gui_idle_touch_screen_display_pressed_shorcut_item(0);
    }
    else //将焦点放在列表的下一项
    {
        fixed_list_goto_next_item();
         redraw_dynamic_list();//add by chenzhp 1220 虚线相关
    }
  }

}

void ShctExecuteLeftKey(void)
{
```

```
int iIndex;
//焦点处于列表中，那么左右按键无效
if (1 == g_index_Vetical)
   return;

g_index_Vetical = 0;
//初始状态下往左按键，默认高亮显示第一个按键即 g_shortcut_index = 0
if (-1 == g_shortcut_index)
{
     g_shortcut_index = 0;
}
else
{
     //否则，将当前高亮的菜单项赋值给 g_shortcut_index_hide，用于随后的隐藏高亮动作
     g_shortcut_index_hide = g_shortcut_index;
     //高亮项会左移
     g_shortcut_index--;
     //如果当前是从第一个开始后退，那么需要高亮最后一个图标
     if (g_shortcut_index < 0)
     {
          g_shortcut_index = MAX_IDLE_SCREEN_SHORTCUTS - 1;//高亮最后一个图标
          g_shortcut_index_hide = 0;
     }
   }

if (g_shortcut_index_hide >= 0
&& g_shortcut_index_hide < MAX_IDLE_SCREEN_SHORTCUTS)
   {
      int iHide = g_shortcut_index_hide;
      gui_idle_touch_screen_display_shorcut_item(g_shortcut_index_hide);
   }
   iIndex = g_shortcut_index;
   gui_idle_touch_screen_display_pressed_shorcut_item(g_shortcut_index);
}

void ShctExecuteRightKey(void)
{
   int iIndex;
   if (1 == g_index_Vetical)
      return;
   g_index_Vetical = 0;
   //原理同上述 left 按键函数
   g_shortcut_index_hide = g_shortcut_index;
   g_shortcut_index++;

   if (g_shortcut_index > MAX_IDLE_SCREEN_SHORTCUTS - 1)
   {
        g_shortcut_index = 0;
   }

   iIndex = g_shortcut_index;
   if (g_shortcut_index_hide >= 0
   && g_shortcut_index_hide < MAX_IDLE_SCREEN_SHORTCUTS)
   {
      int iHide = g_shortcut_index_hide;
      gui_idle_touch_screen_display_shorcut_item(g_shortcut_index_hide);
   }
```

```
        gui_idle_touch_screen_display_pressed_shorcut_item(iIndex);
    }
    #else
    /*原始的 4 个函数代码*/
    …
    #endif
    …
    #endif
```

保存修改，编译并运行模拟器。现在可以通过 4 个方向键在快捷菜单和列表中进行任意切换了，并能够高亮显示当前获得焦点的图标，效果如图 15.24 所示。

图 15.24　高亮显示快捷键

此时，按 OK 键无法进入对应的界面。首先添加确定键（OK）的响应，在前面 4 个函数的末尾再添加一个函数。

```
void ShctExecuteOKKey(void)
{
    switch(g_shortcut_index)
    {
        case 0:
            mmi_phb_entry_main_menu();
            break;
        case 1:
            EntryScrMessagesMenuList();
            break;
        case 2:
            mmi_audply_entry_main();
            break;
        case 3:
            ClndrPreEntryApp();
            break;
        case 4:
            mmi_fng_entry_screen();
            break;
        case 5:
            IdleScreenDigitHandler();
            break;
        default:
            break;
    }
}
```

然后，将该函数进行注册。

```
void RegisterDedicatedKeyHandlers(void)
{
…
#ifdef __N73_STYLE__
   SetKeyHandler(ShctExecuteOKKey, KEY_ENTER, KEY_EVENT_DOWN);
#endif
…
}
```

保存修改，编译并运行模拟器，现在即可在图标高亮显示的时候，按 OK 键进入对应的界面了。

15.5.6 动态列表触摸事件

当我们通过上下键切换到列表时，单击菜单项，无法跳转到对应的界面。例如，我们现在按下方向键，日历项高亮。但是单击后，无法进入备忘录界面，按 OK 键也没有响应，所以，我们还需要为列表中的菜单项添加响应。

打开文件 wgui_categories_idlescreen.c，我们需要引用到 shortcuts.c 文件中的变量。

```
extern  S8 g_shortcut_index;
extern S8 g_index_Vetical;
```

然后增加函数：

```
void Cat33hlight(void)
{
   //焦点在列表
   g_index_Vetical = 1;
   if (g_shortcut_index >=0 && g_shortcut_index < MAX_IDLE_SCREEN_SHORTCUTS)
        {
              gui_idle_touch_screen_display_shorcut_item(g_shortcut_index);
   }
   switch(MMI_fixed_list_menu.highlighted_item)
   {
        case 0://添加单击时候的高亮显示
              MMI_fixed_icontext_menuitem.flags &=
              ~ UI_MENUITEM_DISABLE_HIGHLIGHT_DISPLAY;
              gui_highlight_fixed_icontext_menuitem(MMI_fixed_list_menu.items[0],
              MMI_fixed_list_menu.common_item_data);
              //重绘列表
              redraw_dynamic_list();
              //接着进入备忘录列表界面
             TDLShowAllList();
        break;
        case 1:
           MMI_fixed_icontext_menuitem.flags &=
           ~UI_MENUITEM_DISABLE_HIGHLIGHT_DISPLAY;
           gui_highlight_fixed_icontext_menuitem(MMI_fixed_list_menu.items[1],
           MMI_fixed_list_menu.common_item_data);
           //重绘列表
           redraw_dynamic_list();
           //进入音乐播放器主界面
           mmi_audply_entry_main();
           break;
        }
}
```

保存修改，编译并运行模拟器。当列表菜单项高亮的时候，单击即可进入对应的模块，这里，我们还需要在触摸笔单击时，进行索引的判断。

```
MMI_BOOL category33_pen_down_handler(mmi_pen_point_struct point)
{
  …
  if (index >= 0 && index < MAX_IDLE_SCREEN_SHORTCUTS)
  {
  …
  gui_idle_touch_screen_display_pressed_shorcut_item(index);
#ifdef __N73_STYLE__
        g_shortcut_index = index;
        g_index_Vetical = 0; //焦点处于快捷菜单区域
#endif
          }
      }
    #endif /* __MMI_TOUCH_IDLESCREEN_SHORTCUTS__ */
    …
}
```

这样，当触摸笔单击到快捷菜单区域时，将标识动态列表菜单项的索引清零，防止出现异常。

但是，按 OK 键没有反应，很显然，我们还需要在 ShctExecuteOKKey 函数中增加列表的 OK 键响应。

```
void ShctExecuteOKKey(void)
{
   //焦点处于列表中，则交给函数 Cat33hlight 处理
   if (1 == g_index_Vetical)
   {
      Cat33hlight();
      return;
   }
   switch(g_shortcut_index)
   {
   case 0:
       mmi_phb_entry_main_menu();
       break;
   …
}
```

如果当前屏幕焦点位于快捷菜单行，这时按下方向键，就会高亮显示列表菜单项，这没问题。但是，如果再次按上键回到快捷菜单行，那么先前高亮显示的列表菜单项没有更新，也就是会出现快捷图标和日历项同时高亮显示的状况，如图 15.25 所示。

图 15.25　同时高亮显示的异常

这是由于列表项没有关闭背景导致的。打开文件，修改代码：

```
void gui_show_dynamic_list_menu(fixed_list_menu *m)
{
 …
 #if !defined(MT6205B) && !defined(MT6208)
            gui_draw_filled_area(x1, y1, x2, y2, f);
 #else /* !defined(MT6205B) && !defined(MT6208) */
  /* For low-end phone, disable list menu background can greately improve menu performance.
  We use the color of filler to draw the background (typically white). */
  color c = f->c;
  gdi_draw_solid_rect(x1, y1, x2, y2, gdi_act_color_from_rgb(255, c.r, c.g, c.b));
  #endif /* !defined(MT6205B) && !defined(MT6208) */
      }
    }
#ifdef __N73_STYLE__
  else
    {
     //将列表中的菜单项背景绘制为透明
     gdi_layer_push_clip();
     gdi_layer_set_clip(x1, y1, x2, y2);
     gdi_draw_solid_rect(x1, y1, x2, y2, GDI_COLOR_TRANSPARENT);
     gdi_layer_pop_clip();

    }
#endif
    if (m->n_items <= 0)
    {
       return;
    }
 …
}
```

保存修改，运行模拟器，多次按上下键，当快捷菜单高亮显示的时候，列表项正常。刷新问题解决，一切正常，如图 15.26 所示。

图 15.26　修改同时高亮

到目前为止，我们已经顺利地完成了待机界面的修改，整个待机界面和 NOKIA N 系列的风格已经没有什么差别了，读者还可以进一步对其进行细化。

15.6 主菜单初步

现在，在待机界面中单击 Menu，就会进入主菜单界面。MTK 本身是提供了矩阵菜单排列方式的，只是在风格上和 N73 有一定的差别。打开文件 MMI_features_GUIPLUTO.h，取消宏__MMI_BI_DEGREE_MAIN_MENU_STYLE__的定义，即可实现 Matrix 菜单风格。

```
#ifndef __N73_STYLE__
#define __MMI_BI_DEGREE_MAIN_MENU_STYLE__
#endif
```

调整后，将原来的 TAB 型菜单界面换成十二宫格。注意，屏蔽这个宏之后，一定要执行 remake 操作，现在的效果如图 15.27 所示。

很显然，目前的界面非常粗糙，下面我们对其进行细化操作。

首先，准备如下图片，如图 15.28 所示。

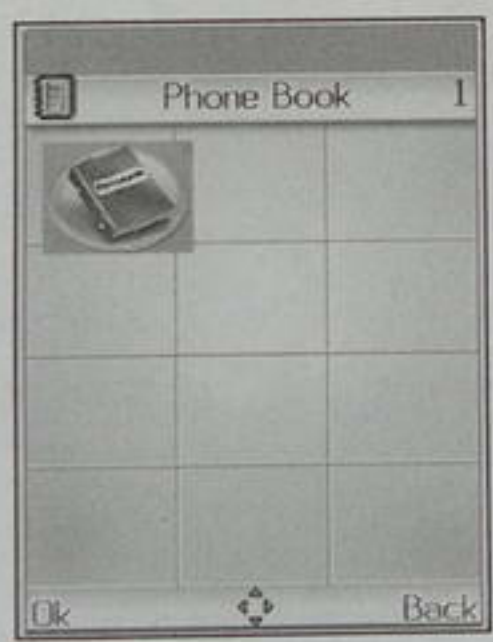


图 15.27 原始的矩阵菜单

图 15.28 替换自定义图片

读者在增加图片的同时，保证和将替换的系统图片文件名一致。然后，进入目录：

```
\plutommi\Customer\Images\PLUTO240X320\MainLCD\MainMenu\MATRIX
```

用新的图片覆盖原有的图片，覆盖完毕后，记得将文件夹中的内容打包。然后，更新工程，重新编译模拟器，即可看到下面的效果，如图 15.29 所示。

注 意

由于系统代码会根据每个图标的尺寸大小进行排版，所以如果新绘制的图标超过一定大小，就会出现某些图标不可见的情况。因此，有两种方法可以补救：或者修改排版的具体代码，或者修改图标。在这里，为了简单起见，我们对图标进行修改，读者在制作图标的时候，保证尺寸不超过 70 × 60。同时，在制作这些 GIF 格式图标的时候，要保证其背景层为透明，否则，主菜单上每个图标后面都有一块白色的底色。

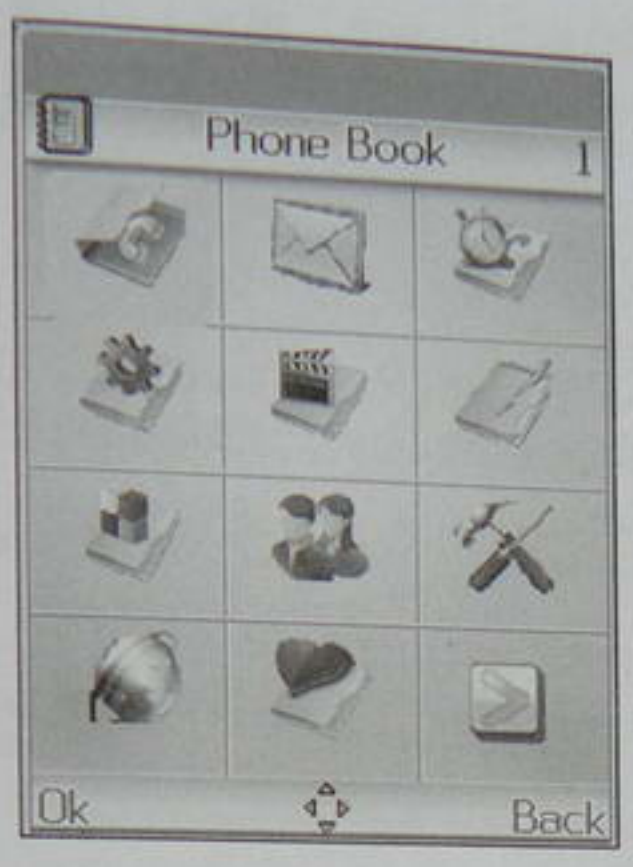


图 15.29　修改后的效果

为了达到背景主题一致的效果，我们首先去除界面底部的 ButtonBar 背景：

修改 Custcoordinates.c 文件中的屏幕模板数据库，将 DM_BUTTON_BAR1 组件移除。

```
const U8 category14_matrix[] =
{
   4,
   DM_BASE_LAYER_START,
   DM_SCR_BG,
   DM_BASE_CONTROL_SET1,
   DM_MATRIX_MENU1
};
const S16 coordinate_set14_Matrix[] =
{
   DM_FULL_SCREEN_COORDINATE_FLAG,
DM_CONTENT_COORDINATE_FLAG,

};
```

接下来，我们用当前的主题背景来绘制整个界面。例如，我们可以用当前系统的待机界面背景来绘制所有界面的屏幕，这样就可以达到风格统一的目的。

在文件 wgui_draw_manager.c 中，查找函数 dm_draw_scr_bg_image，内部代码修改部分如下：

```
void dm_draw_scr_bg_image(void)
{
    …
//#else /* __MMI_LITE_DISPLAY__ */
   {
            gdi_draw_solid_rect(0, 0, UI_device_width - 1, UI_device_height - 1,
                                gdi_layer_get_background());
            if (g_dm_scr_bg_cntx.image_id != 0) /* from resource */
            {
                if (g_dm_scr_bg_cntx.offset_x == -1 && g_dm_scr_bg_cntx.offset_y == -1)
                {

#ifndef __N73_STYLE__
        gdi_image_get_dimension_id(g_dm_scr_bg_cntx.image_id,&img_width,&img_height);
#else
```

```
            //获取主题的背景图片的尺寸
            gdi_image_get_dimension(current_MMI_theme->list_background_filler->b,
                                    &img_width,    &img_height);
#endif
                …
            else
                {
                    S32 offset_x = (UI_device_width - img_width) >> 1;
                    S32 offset_y = (UI_device_height - img_height) >> 1;
#ifndef __N73_STYLE__
            gdi_image_draw_id(offset_x, offset_y, g_dm_scr_bg_cntx.image_id);
#else
            //使用当前主题的背景图片作为窗体的背景图片
            gdi_image_draw(offset_x, offset_y, current_MMI_theme->list_background_filler->b);
#endif
                …
            else
                {
                    S32 offset_x = (UI_device_width - img_width) >> 1;
                    S32 offset_y = (UI_device_height - img_height) >> 1;
#ifndef __N73_STYLE__
            gdi_image_draw_file(offset_x, offset_y,g_dm_scr_bg_cntx.image_name);
#else
            //使用当前主题的背景图片作为窗体的背景图片
            gdi_image_draw(offset_x, offset_y, current_MMI_theme->list_background_filler->b);
#endif
                }
            }
            else
            {
                gdi_image_draw_file(g_dm_scr_bg_cntx.offset_x,  g_dm_scr_bg_cntx.offset_y,
g_dm_scr_bg_cntx.image_name);
            }
    }
```

上述函数就是绘制所有屏幕背景的公共函数，其中的 current_MMI_theme 就是刷新过程中具体包含的属性，其设置过程请参考相关文档。

然后，重新编译代码，主菜单界面效果如下图 15.30 所示。

图 15.30　替换背景后的主菜单

由于具体设置和改动部分比较多，限于篇幅，无法一一列举。我们只能给大家一个启蒙性的讲解，读者可以顺着本章的思路继续深入学习下去。

结语：

本章，我们通过一个完整的例子讲解了如何仿制 NOKIA 的待机界面。读者可以通过仿制的过程，理解 MTK 的具体绘制流程和逻辑。读者通过本章的学习，能更加深入地了解上层界面开发中的重要知识点，为以后的项目开发和学习打下基础。

16 MTK 驱动开发

当手机生产厂商根据硬件工程师的设计图生产出手机样机后，接下来的工作就由驱动工程师来完成了。驱动工程师的主要职责就是通过对驱动程序的编写，使硬件能正常工作，验证硬件设计的正确性，为改进硬件设计提供基础，并且还对应用程序的开发提供基本的支持。所以驱动开发有着十分重要的承上启下的作用，是影响手机开发进度的关键步骤之一，熟练地掌握驱动开发的知识和技巧对手机研发项目整体也有着重要的作用。

作为驱动开发工程师，需要对硬件和软件有一定的基本了解。一般来说，在进行样机生产前，主要的硬件元器件已经确定下来，这时驱动工程师就可以先了解硬件的器件和功能列表，以便决定需要对哪些硬件部分进行验证，并对如何进行驱动开发有一个基本的构思。

对于基于 MTK 方案的手机，由于方案源代码中集成了大部分默认的驱动代码，因此，我们要做的就是了解代码中驱动的框架和流程，对有变化的和新增的驱动部分进行修改。

本章首先简要介绍了 MTK 驱动开发的架构，让读者全面地认识一下 MTK 驱动涉及的各个方面；接着介绍手机启动过程，使读者对手机的启动流程有所了解；然后介绍了各个驱动模块的原理和如何在方案代码中对驱动代码进行修改，最终使我们的驱动在样机上调试通过并正常运行。主要介绍的模块有 EMI 配置、驱动定制工具的使用、设备默认值的设置、可插入设备的设置、其他设备的设置。

由于硬件方面涉及的专业术语简写比较多，初学者往往会被弄得一头雾水，因此后面涉及有简称的地方也对其进行介绍。驱动代码介绍方面一般先介绍文件的路径，再列举和讲解相应的代码。

16.1 基本架构

下面我们看一下驱动开发涉及的硬件和软件的基本架构，如图 16.1 所示。介绍一台 MTK 手机硬件主要由哪些部分组成的和其对应的相关文件代码。

一般来说 PCB 板贴上其他器件就构成了一台无外壳的手机原型，驱动开发人员会先拿到这类手机原型的板子进行驱动调试。

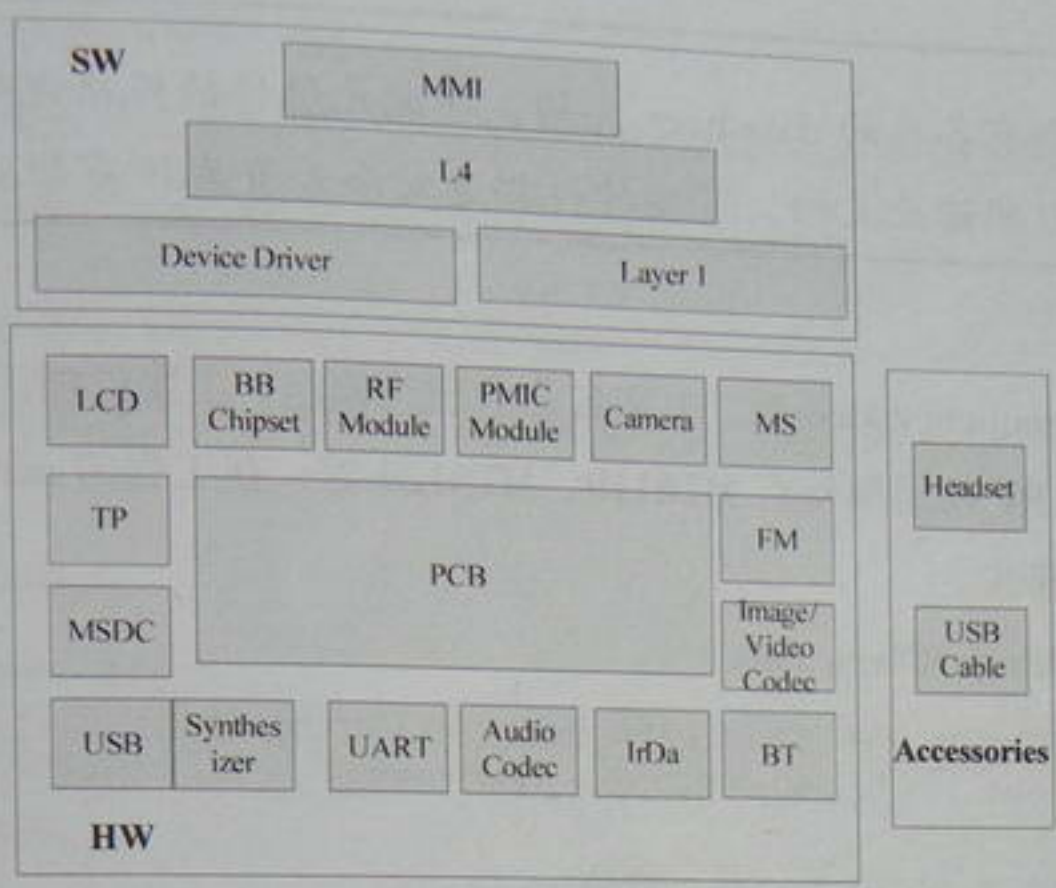


图 16.1　基本架构图

其中主要涉及的模块有以下几种，并根据不同类型给出在工程 makefile 中需要修改的部分。

1. PCB

PCB 即 PrintedCircuitBoard 的缩写，中文名称为印制电路板，是电子元器件的支撑体，是电子元器件电气连接的提供者。图 16.1 所示的下面绿色的板子不含上面贴的其他元器件。

2. BB Chipset

BB Chipset 即 Baseband Chipset 的简称，中文名称为基带芯片，是整个手机的主控制器，又叫做 MCU（Micro Controller Unit，微控制单元），还可以称为单片微型计算机（Single Chip Microcomputer）或者单片机。是指随着大规模集成电路的出现及其发展，将计算机的 CPU、RAM、ROM、定时数器和多种 I/O 接口集成在一片芯片上，形成芯片级的计算机，为不同的应用场合做不同组合控制。对于不同 MTK BB 芯片，应该知道它们的一些主要区别，例如 MT6225 的处理速度为 104MHz，快于 MT6226 的处理速度 52MHz，DSP 速度也较快，但 MT6225 的 RAM 比 MT6226 的小，是 MT6226 的 cost down 版本。

> *提　示*
>
> 在 MTK 相关文档中经常以 MCU 代表工程的根目录名。在应用层开发的时候，我们的根目录为 E:\MTK6226，但为了描述更准确和更具通用性，我们还是用 MCU 来表示根目录。

主要的 BB 芯片型号有 MT6226、MT6227 等，在工程的 makefile 中可对 BB 芯片型号、BB 主版本号及 MCU 时钟进行设置。

文件路径：mcu\make\ProjectName.mak。

```
PLATFORM=MT6226  # Hardware Platform, MT6208, FPGA, MT6218, MT6218B, MT6219, MT6226, MT6227,
MT6228, MT6229 ...etc.
BOARD_VER=PROJECTNAME_BB  # Baseband main board description.
MCU_CLOCK=MCU_52M #MCU clock setting, MCU_13M, MCU_26M, MCU_39M, MCU_52M, MCU_104M
```

BB 相关功能模块代码的目录会对 BOARD_VER 的值进行设置。可设置的 MCU 时钟的值与 BB 的类型有关，详情见各 BB 的 datasheet 文档。

说　明

各个元器件一般都有各自的 datasheet 文档，即此元器件的规格说明文档，如果要对该器件进行详细了解，可以阅读此文档，有些特性描述对驱动开发很有帮助。

3. RF Module

RF Module 即 Radio Frequency Module，中文名称为射频模组。主要处理手机的射频通信功能，有了它手机才能通话。主要的 RF 模组型号有 MT6119、MT6129 等，在工程的 makefile 中可对 RF 模组型号及其 RF 晶振类型等进行修改。

文件路径：mcu\make\ProjectName.mak。

```
RF_MODULE=PROJECTNAME_MT6129D #TOP_6227_MT6129D, FOUNTAIN2, SPRING,
                              # MT6129D, AERO1PLUS.

AFC_VCXO_TYPE= VCTCXO # VCTCXO    : VCTCXO
                      # VCXO      : VCXO
                      # VCXO_SC   : VCXO with 7 area slope control
                      # VCXO_TC   : VCXO with temperature compensation
                      # VCXO_SC_TC : VCXO with 7 area slope control & temperature compensation

COM_DEFS_FOR_PROJECTNAME_MT6129D=MT6129D_RF PROJECTNAME_MT6129D
```

除了在 makefile 中进行修改还要根据选择的类型使用相应的 RF 代码。

例如，文件路径为：

mcu\custom\l1_rf\PROJECTNAME_MT6129D\l1d_custom_rf.h 和

mcu\custom\l1_rf\PROJECTNAME_MT6129D\ m12193.c。

注意 PROJECTNAME_MT6129D 要与在 makefile 中设置的一样，然后可向 MTK 获取相应 RF 类型代码。

4. PMIC Module

PMIC Module 即 Power Management Integrated Circuit Module，中文名称为电源管理集成电路模组。主要处理手机的电源管理功能，主要的 PMIC 模组型号有 MT6305 等，在工程的 makefile 中可对 PMIC 模组型号进行选择。

文件路径：mcu\make\ProjectName.mak。

```
PMIC=MT6305 # Charge Function, NONE, MT6305 or MT6318
```

提　示

BB Chipset、RF Module 和 PMIC Module 称为 MTK 的 3 大模块，一般来说 MTK 的解决方案就是指这 3 大模块的组合。

5. LCD

LCD 即 Liquid Crystal Display，中文名称为液晶显示屏。根据颜色数量的不同可分为 6.5 万色，26 万色等，也可根据分辨率分为 176 × 220、240 × 320 等，手机上的 LCD 一般为 TFT LCD（Thin Film Transistor LCD，薄膜晶体管液晶显示器）。在工程的 makefile 中可对 LCD 型号及其尺寸等进行设置。

文件路径：mcu\make\ProjectName.mak。

```
LCD_MODULE=PROJECTNAME_LCM # Based on the LCM solutions (even multiple LCM modules for
this project)
MAIN_LCD_SIZE=240X320# To distinguish the main lcd size.
SUB_LCD_SIZE = NONE # NONE, 48X64BW, 64X96, 96X64BW, 96X64, 128X128
COM_DEFS_FOR_PROJECTNAME_LCM= PROJECTNAME_LCM TFT_MAINLCD
```

除了在 makefile 中进行修改之外还要根据选择的类型修改相应的 LCD 代码。

例如，文件路径：

- mcu\custom\drv\LCD\lcd.c；
- mcu\custom\drv\LCD\lcd_hw.h；
- mcu\custom\drv\LCD\lcd_sw.h；
- mcu\custom\drv\LCD\lcd_sw_inc.h；
- mcu\custom\drv\LCD\lcd_rnd.h。

6. Camera

Camera 即摄像头，主要用在照相、摄像模块，根据可采集的像素主要分为 30 万像素、130 万像素、500 万像素等，根据采集数据的类型分为 RGB、YUV 等，其是否支持相应的类型主要由所选择的 BB Chipset 决定。在工程的 makefile 中可以对 Camera 型号及属性进行设置。

文件路径：mcu\make\ProjectName.mak。

```
CMOS_SENSOR=OV7670#OV9640, PAS105, PAS302, NONE, MT9D011, MT9M111, OV9650
SENSOR_ROTATE=SENSOR_ROTATE_0#SENSOR_ROTATE_0, SENSOR_ROTATE_90, SENSOR_ROTATE_180, SENSOR_
ROTATE_270
HORIZONTAL_CAMERA=FALSE # FALSE, TRUE
                        # To enable horizontal camera feature.
                        # Add HORIZONTAL_CAMERA to global compile option when the value
is TRUE.
WEBCAM_SUPPORT= TRUE# WEB CAMERA support
YUV_SENSOR_SUPPORT= TRUE # TRUE, FALSE
                        # to enable YUV sensor interface
```

> **提 示**
>
> 型号的前缀一般表示 Camera 芯片厂商英文全称缩写或简写，如 OV 代表 OmniVision 公司。如果摄像头 IC 使用 YUV 格式，以后对摄像头的 Fine Tune 一般直接联系厂商；如果摄像头 IC 使用 RGB 格式，以后对摄像头的 Fine Tune 一般直接联系 MTK。

上面的相关设置有控制摄像头是否显示旋转、是否支持网络摄像头、是否支持 YUV 格式的数据采集类型。除了在 makefile 中进行修改之外还要根据选择的类型添加相应的 MS 代码。

文件路径：

- mcu\custom\drv\camera\ProjectName\camera_hw.c；
- mcu\custom\drv\camera\ProjectName\camera_info.c；
- mcu\custom\drv\camera\ProjectName\camera_para.c（或 mcu\custom\drv\camera\ProjectName\camera_yuv_para.c）；
- mcu\custom\drv\camera\ProjectName\usbvideo_attr.c。

7. FM

FM 即 Frequency Modulation 的缩写，中文名称为调频收音机，可接收电台发出的广播信号。主要的 FM 型号有 MT6189、MT6188 等。在工程的 makefile 中可对 FM 型号和是否对 FM 录音进行设置。

文件路径：mcu\make\ProjectName.mak。

```
FM_RADIO_CHIP=MT6189A1# NONE, MT6189A1, NS953
FM_RADIO_RECORD= TRUE#TRUE/FALSE
```

8. TP

TP 即 Touch Panel 的缩写，中文名称为触摸屏，支持对屏幕触摸时间的响应，主要的 TP 型号有 MT6301、MT6301B 等，在工程的 makefile 中可对 TP 型号进行设置。

文件路径：mcu\make\ProjectName.mak。

```
TOUCH_PANEL_SUPPORT=MT6301#NONE,MT6301 or MT6301B, TOUCH PANEL Feature
```

9. MSDC

MSDC 即 Mass Storage Device Controller 的缩写，中文名称为大容量设备控制器，主要使手机具有优盘的功能，能够通过插入外部存储卡存储大容量数据，如 SD/MMC（手机上的一种存储卡）卡等。在工程的 makefile 中可对是否支持 MSDC 功能进行设置并选择类型。

文件路径：mcu\make\ProjectName.mak。

```
MSDC_CARD_SUPPORT_TYPE=MSDC_SD_MMC
                        #MSDC_SD_MMC for SD/MMC card support
                        #MSDC_MS for MS card support
                        #MSDC_MSPRO for MS-PRO card support
                        #NONE
                        #MT6218B, MT6217, MT6219 support MSDC
                        #MT6218, MT6205B, MT6208 don't support MSDC
```

10. USB

USB 即 Universal Serial Bus 的缩写，中文名称为通用串行总线，主要控制 USB 线与电脑的连接和数据传输，还可当充电器用。在工程的 makefile 中可对 USB 相关属性进行设置。

文件路径：mcu\make\ProjectName.mak。

```
USB_SUPPORT=TRUE # MT6218B and MT6219 chip support USB
USB_MULTI_CHARGE_CURRENT_SUPPORT = FALSE  # TRUE, FALSE
USB_IN_META_SUPPORT = TRUE# TRUE, FALSE
                        # To enable USB suuport in META mode feature
USB_IN_NORMAL_MODE_SUPPORT = TRUE # TRUE,FALSE
                        # To enable USB mass storage support in normal mode
```

11. Synthesizer

Synthesizer 即声音合成器，主要用于音乐的合成输出。

12. Audio Codec

Audio Codec 即音频编解码器，主要对音频数据进行编解码给上层应用软件使用，可将原始音频数据（如 PCM）转换成压缩的数据格式（如 AMR 等），或将压缩好的数据格式（如 AMR）转换成原始音频数据（如 PCM）。编码的过程可以是硬件或软件，在工程的 makefile 中可对 Audio Codec 相关属性进行设置。

文件路径：mcu\make\ProjectName.mak。

```
AMRWB_ENCODE= TRUE# TRUE/FALSE
                #MT6219 DSP cannot support AMRWB_ENCODE when GPRS connection
AMRWB_DECODE= TRUE# TRUE/FALSE
DAF_DECODE = TRUE# TRUE/FALSE
AAC_DECODE= TRUE # TRUE/FALSE
MMF_DECODE=FALSE # TRUE, FALSE  MMF_DECODE support in MT6218B, MT6217, MT6219 and later chip
WMA_DECODE = FALSE # MT6228 support WMA DECODE
```

13. Image/Video Codec

Image/Video Codec 即图像/视频编解码器，主要对图像/视频数据进行编解码给上层应用软件使用，可将原始图像/视频数据（如 Raw RGB）转换成压缩的数据格式（如 JPEG），或将压缩好的数据格式（如 JPEG）转换成原始图像/视频数据（如 Raw RGB）。编码的过程可以是硬件或软件，在工程的 makefile 中可对 Image/Video Codec 相关属性进行设置。

文件路径：mcu\make\ProjectName.mak。

```
JPG_ENCODE = JPG_HW # NONE, JPG_SW, JPG_HW
JPG_DECODE = JPG_HW# NONE, JPG_SW, JPG_HW
GIF_DECODE = TRUE # TRUE/FALSE
PNG_DECODE= NONE# NONE, PNG_HW, PNG_SW
MJPG_SUPPORT= FALSE# TRUE/FALSE
MP4_CODEC= TRUE # TRUE/FALSE
```

14. BT

BT 即 Blue Tooth 的缩写，中文名称为蓝牙，使用 BT 技术可以进行数据的无线传输，传输距离比 IrDa 远。在工程的 makefile 中可对 BT 型号进行设置。

文件路径：mcu\make\ProjectName.mak。

```
BLUETOOTH_SUPPORT=BTMTK_MT6601
#BTMTK, NONE, BTVCSR_HP or BTVCSR_HCI_BCHS or BTMTK or BTMTK_MT6601
```

15. IrDa

IrDa 即 Infrared Data Association 的简写，中文名称为红外数据组织。目前广泛采用的 IrDa 红外连接技术就是由该组织提出的，使用 IrDa 技术可以进行数据的无线传输，传输距离比较近，目前使用比较少了。在工程的 makefile 中可对是否支持 IrDa 进行设置。

文件路径：mcu\make\ProjectName.mak。

```
IRDA_SUPPORT=FALSE # IRDA support feature
```

16. MS

MS 即 Motion Sensor 的缩写，中文名称为加速传感器。在工程的 makefile 中可对 MS 型号进行设置。

文件路径：mcu\make\ProjectName.mak。

```
MOTION_SENSOR_SUPPORT = MXC6202_I2C
# NONE or KXP84_SPI or KXP84_I2C or KXP74_SPI or MXC6202_I2C
# For motion sensor function support or not.
```

除了在 makefile 中进行修改之外还要根据选择的类型添加相应的 MS 代码。

> **提　示**
>
> 当设置了 MS 型号后，要建立一路径为 mcu\custom\drv\motion_sensor\MXC6202_I2C 的目录，在下面添加 MS 相应代码。

文件路径：

```
mcu\custom\drv\motion_sensor\MXC6202_I2C\motion_sensor_custom.h
mcu\custom\drv\motion_sensor\MXC6202_I2C\motion_sensor_custom.c
```

17. UART

UART 即 Universal Asynchronous Receiver/Transmitter 的缩写，中文名称为通用异步接收/发送装置，集成在 BB 中。发送数据时，BB 先把准备写入串行设备的数据放到 UART 的寄存器（临时内存块）中，再通过 FIFO（First Input First Output，先入先出队列）传送到串行设备。接收数据时将外部传来的串行数据转换为字节，供内部并行数据的器件使用。MTK 手机中，UART 主要用于连接 PC 与手机，进行系统烧录和程序调试输出。

18. 其他辅助设备

与手机相连的外设主要还有 USB 线、耳机等。

16.2 MTK 手机启动过程

上节我们介绍了 MTK 手机是由哪些部件组成的，下面我们来介绍 MTK 手机是如何启动，让这些部件运行的。

MTK 一般启动过程的步骤如图 16.2 所示（不同的结束过程以 End_X 标注，X 用数字表示）。

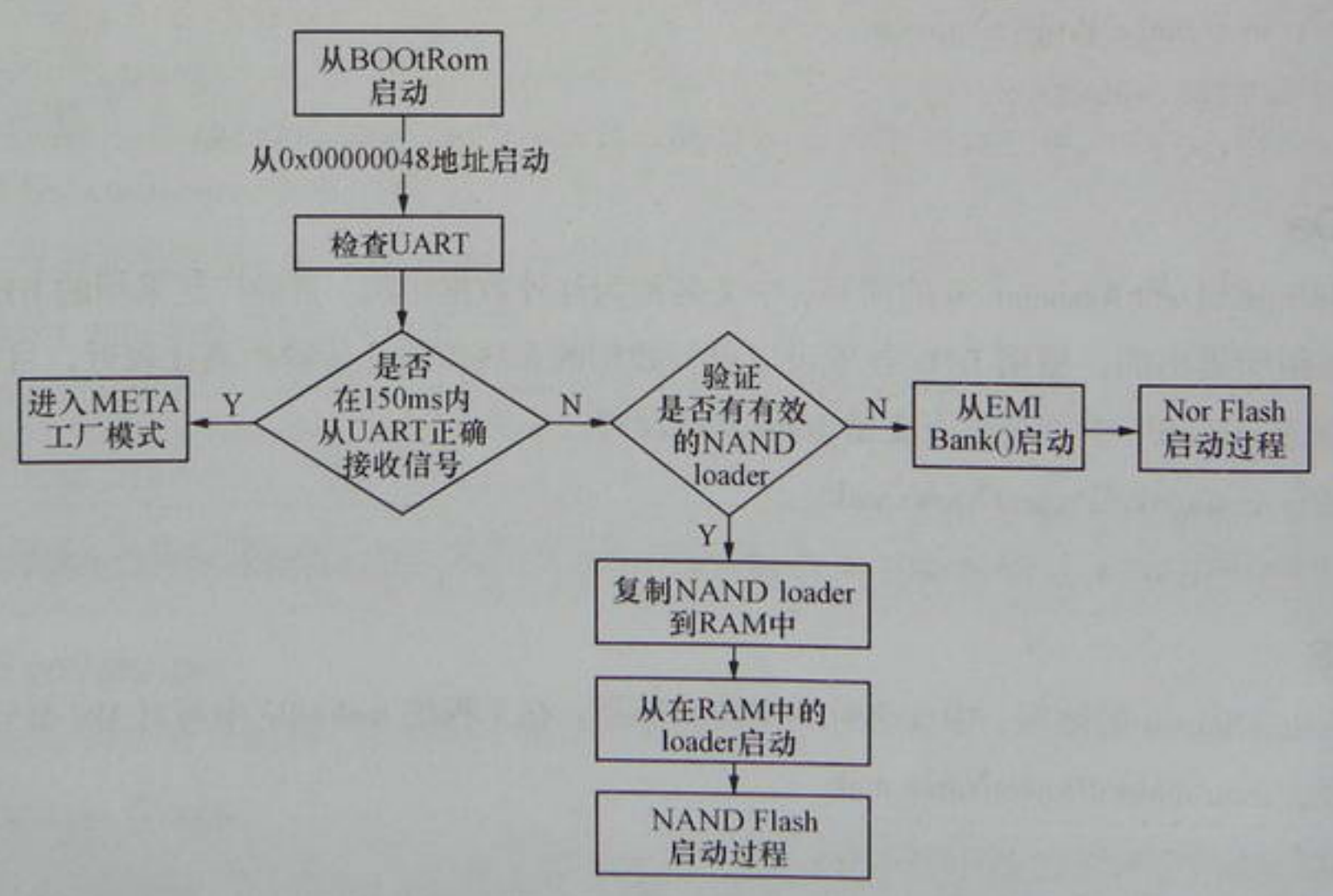


图 16.2　Flash 启动过程

（1）按下开机键，PMIC 给 BB 供电。

（2）一定的时序后，BB 时钟设置好，BB 收到复位信号。

（3）从 BB 的 BOOTROM 中启动，BOOTROM 在 BB 芯片内部，里面存储与启动相关的只读代码，不可更改，每次系统烧录的时候也不会改变其中代码。代码从 BOOTROM 的 0x00000000 地址开始运行，然后就跳转到 0x48000000 的代码运行。

（4）在 150ms 内检测是否 UART1 正确地接收到特殊的字节数据（例如，连接到 PC，启动了 META 工具）。

说　明

Meta Tool（Mobile Engineering Testing Architecture Tool，手机工程测试架构工具），是在 MTK 平台中用于测试、校准、调试手机的一个开发工具。

（5）检测到 150ms 内 UART1 正确地接收到特殊的字节数据（例如，连接到 PC，启动了 META 工具），则进入 META 模式。在 META 模式只有部分 Task 被启动起来，开发人员可在 META 工具中对相关数值进行调整、校准、测试。

（6）检测到 150ms 内 UART1 没有正确地接收到特殊的字节数据（例如，没有连接到计算机，没有启动 META 工具），则判断是否有有效的 NAND loader（即在 NAND Flash 上的程序装载器）。

① NAND Flash 启动过程：如果有有效的 NAND loader 则先将 NAND loader 从 NAND Flash 复制到 RAM 上，从 RAM 启动，装载 images，再运行 image 上的代码。

② NOR Flash 启动过程：如果没有有效的 NAND loader 则从 EMI 的 Bank0 启动。将 PC（程序计数器）跳转到外部设备的 Bank0（一般来说就是 Flash 的代码区域）。

说　明

PC（Program Counter，程序计数器），为了保证程序能够连续地执行下去，使用 PC 寄存器来确定下一条指令的地址，所以通常又称为指令计数器。在程序开始执行前，必须将它的起始地址，即程序的一条指令所在的内存单元地址送入 PC。每执行一条指令，PC 增加一个量，指向下一条指令。也可使用 PC 跳转到某个地址，指向下一条指令。

Bank，即内存库。“Bank 数”表示物理或逻辑的内存数。在 MTK 启动过程中 Bank0 一般指 Flash（主要存储烧录好的系统程序），Bank1 一般指 SRAM/PSRAM。

（7）分配 Supervisor 模式的栈顶指针。

说　明

Supervisor Mode，即监控模式，当复位或软中断指令执行时将会进入此种模式，是供操作系统使用的一种保护模式。

ARM 的基本工作模式：MTK 的 BB 芯片使用 ARM 的处理器，有 7 种工作模式，分别是 User、FIQ、IRQ、Supervisor、Abort、Undef、System，不同处理模式的权限或物理寄存器不同。

（8）根据一定的条件决定是否要重新映射。

条件如下。

① 硬件条件：如果硬件中的 iBoot 信号为高则从 BOOTROM 启动，如果 iBoot 信号为低则从硬件的\CS0

片选信号启动。

② 软件条件：通过设置 BB 中寄存器 EMI_REMAP 的 RM[1:0]来标记重新映射的类型。如表 16.1 所示。

表 16.1 映射类型

RM[1:0]	地址：00000000h	地址：8000000h
00	Boot Code	Bank1（SRAM）
01	Bank1（SRAM）	Bank0（Flash）
10	Bank0（Flash）	Bank1（SRAM）
11	Bank1（SRAM）	Bank0（Flash）

说 明

CS（Chip Select，片选信号），选择不同的片选信号代表不同的意义，如\CS0 为选择 Flash，对应地址空间中的 Bank0；\CS1 选择 SRAM 或 PSRAM，对应地址空间中的 Bank1。

Memory Re-mapping Mechanism，即内存映射机制。它的目的是为了达到更弹性的系统配置。一般可映射到 3 种内存空间中，一是在芯片的 BOOTROM 中，二是在\CS0 片选后的 Flash 中，三是在\CS1 片选后的 SRAM 或 PSRAM 中。

（9）如果满足映射条件则设置两个单位的读等待状态的时间给\CS0 和\CS1 片选信号（即后面 EMI 时间设置中介绍的 RLT）。

提 示

一个读等待状态时间等于片选时间加上读取数据的时间。

接着复制异常处理器，用于处理发生的各种异常。再开始重新映射的过程，映射完成后检查是否有异常重置的过程。

（10）如果不满足映射条件则检查是否有异常重置的过程，检测使用了 WatchDog 机制。检查 WatchDog 时间重置寄存器 0x80040018，如果其值从 0xFFF 变为 0xFFA 则没有异常发生，否则有异常发生。

说 明

WatchDog，即看门狗，又叫 watchdog timer，是一个定时器电路，一般有一个输入，叫喂狗，一个输出到 MCU 的 RST 端。MCU 正常工作的时候，每隔一段时间输出一个信号到喂狗端，给 WDT 清零，如果超过规定的时间不喂狗，（一般在程序跑飞时），WDT 定时超过，就会给出一个复位信号到 MCU，使 MCU 复位，防止 MCU 死机。看门狗的作用就是防止程序发生死循环，或者说程序跑飞。

（11）开始检查是否有异常重置，如果有则利用异常处理器进行异常处理。

（12）开始检查是否有异常重置，如果没有则开始配置 EMI。详情见后面章节的描述。

说 明

EMI（External Memory Interface，外部内存接口），是 MCU 与外部存储设备之间通信的总线协议。

（13）配置好 EMI 后就将 Flash 中的数据复制到 SRAM 中。

（14）分配各种模式的栈顶指针。

（15）最后进行应用程序的初始化操作，启动完毕。

启动流程如图 16.3 所示。

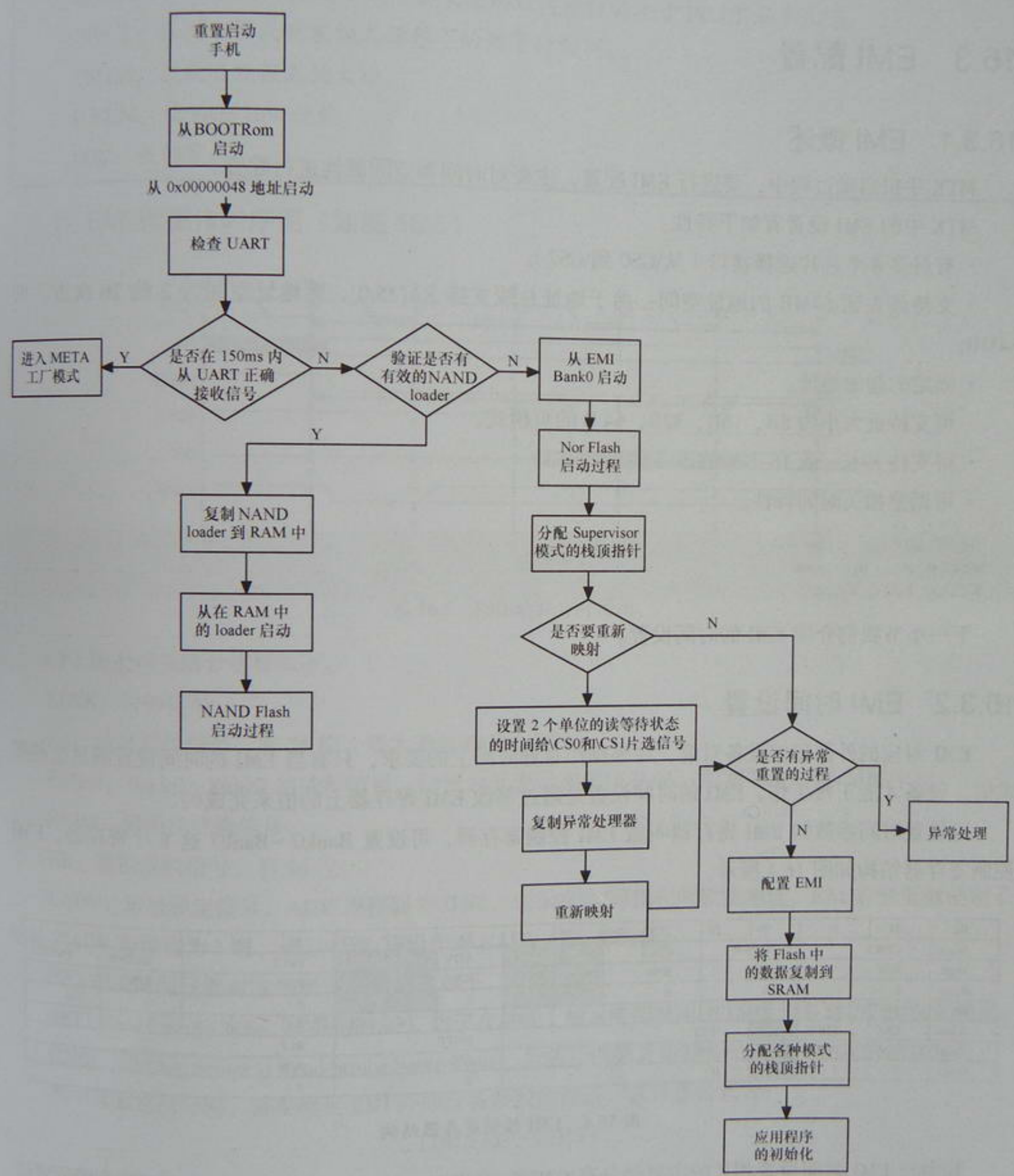


图 16.3 启动流程全图

启动过程主要用到 bootarm.s 和 init.c 两个文件中的代码，可打开查看详细的启动流程。其中 bootarm.s 是用 ARM 指令编写。

文件路径为：mcu\init\src\bootarm.s 和 mcu\init\src\init.c。

16.3 EMI 配置

16.3.1 EMI 概述

MTK 手机启动过程中，要进行 EMI 配置，主要对时间和空间属性进行配置。

MTK 中的 EMI 设置有如下特性。

- 有最多 8 个芯片选择接口（从\CS0 到\CS7）。
- 支持最大到 64MB 的地址空间：由于地址总线支持 EA[25:0]，则地址空间为 2 的 26 次方，即 64MB。
- 固定的起始地址。
- 可支持页大小为 8B、16B、32B、64 B 的页模式。
- 可支持 8-bits 或 16-bits 的设备。
- 可调整相关时间特性。

```
Read set-up time
Write set-up time
Write hold time
```

下一小节我们介绍 EMI 的时间设置。

16.3.2 EMI 时间设置

EMI 对应的外部存储设备对读、写等操作有着时间上的要求，只有当 EMI 的时间设置满足这些要求时，设备才能正常工作。EMI 的时间设置是通过修改 EMI 寄存器上的值来完成的。

可设置时间参数的 EMI 寄存器叫做 EMI 控制寄存器，可设置 Bank0～Bank7 这 8 个寄存器。EMI 控制寄存器结构如图 16.4 所示。

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Name	C2WS		C2WH		C2RS				ADV			PRLT			BMODE	PMODE
Type	R/W		R/W		R/W				R/W			R/W			R/W	R/W
Reset	0		0		0				1			0			0	0
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Name	DW	RBLN		WST					PSIZE			RLT				
Type	R/W	R/W		R/W					R/W			R/W				
Reset	0	1		0					0			7				

图 16.4 EMI 控制寄存器结构

其中与 EMI 时间设置相关的主要部分有 C2WS、C2WH、C2RS、WST 和 RLT。下面结合图示对这些与时间设置相关的部分进行解释。

说 明

其他部分的说明如下。

PMODE：是否使用页模式，如果使用则读操作取决于 PRLT 和 PSIZE。

BMODE：是否使用突发模式，如果使用则读操作取决于 PRLT 和 PSIZE。

PRLT：在同一页或突发模式操作下的读等待时间。

PSIZE：页或突发模式的大小。

RBLN：读 Byte lane 使能。

DW：数据宽度。

1. EMI 读操作时序图（如图 16.5）

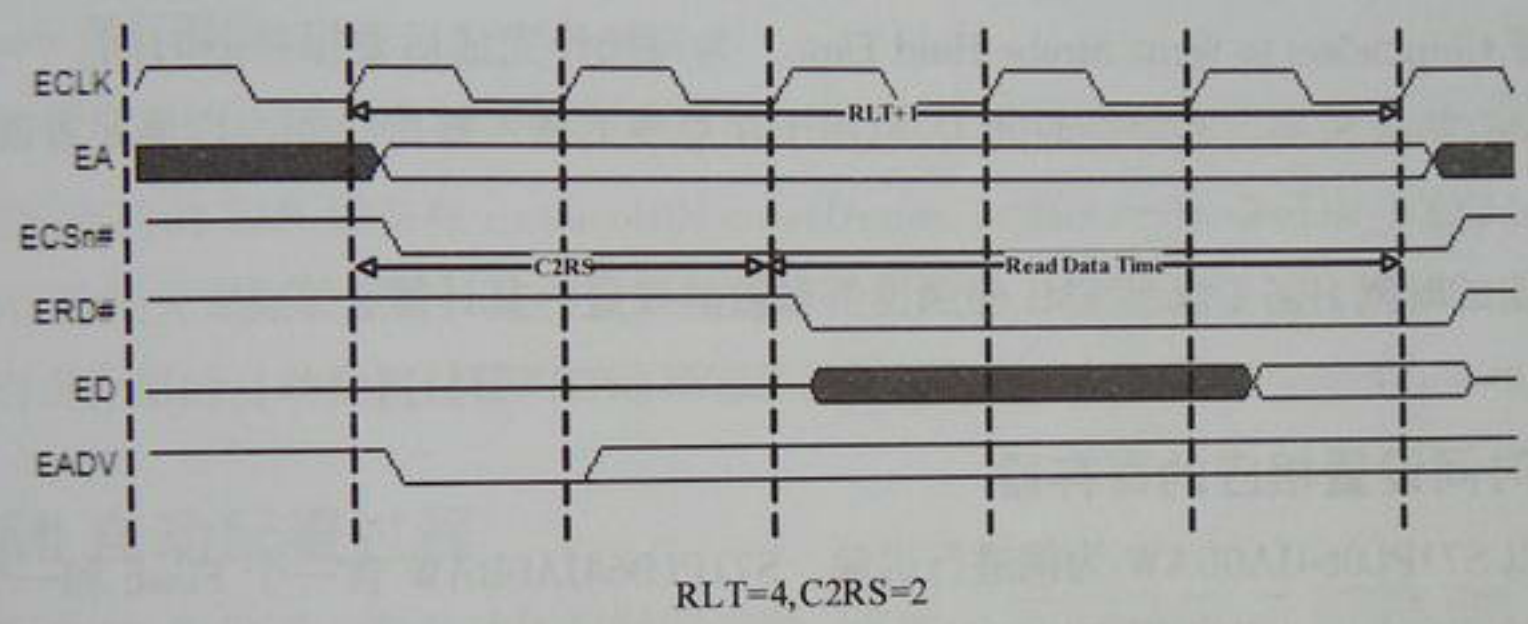


图 16.5 EMI 读操作时序图

（1）图上相关信号解释如下。

ECLK：时钟信号。

EA：地址总线信号，有 26 位，最大表示 64MB 空间。

ECSn#：Bank0～Bank7 的选择信号，信号为低表示选中该 Bank，可对其进行读/写操作。

ERD#：读操作使能信号。

ED：数据总线信号，有 16 位。

EADV：地址锁定信号，ADV 寄存器为 0 时，表示在读/写操作时锁定地址，EADV 变低就锁定了地址。ADV 寄存器为 1 时，表示在读/写操作时 EADV 信号总为低。

（2）相关时间设置对应控制寄存器部分如下。

RLT（Read Latency Time，读等待时间），用于在总线上插入等待时间，以满足设备对访问时间的要求。

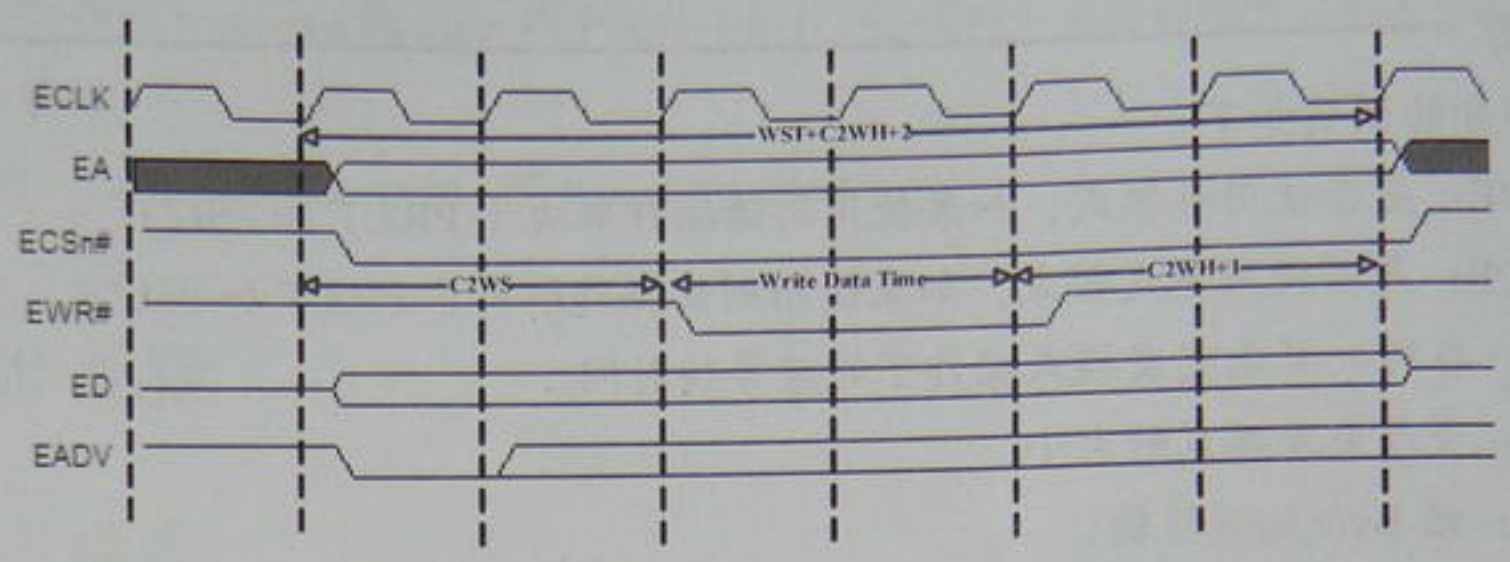
C2RS，即 Chip Select to Read Strobe Setup Time。为芯片选择开始到开始读操作之间的建立时间。

其中读数据的时间，需要满足 EMI 外部设备参数的规定，其计算公式为：

$$[(RLT+1) - C2RS]$$

2. EMI 写操作时序图（如图 16.6）

相关时间设置对应控制寄存器部分如下。



WST=3 C2WS=2 C2WH=1

图 16.6 EMI 写操作时序图

C2WS，即 Chip Select to Write Strobe Setup Time，为芯片选择开始到开始写操作之间的建立时间。

C2WH，即 Chip Select to Write Strobe Hold Time，为写操作完成后要保持的时间。

WST（Write Wait State，写等待时间），可用于在总线上插入等待时间，以满足外部设备对访问时间的要求。

其中写数据的时间，需要满足 EMI 外部设备参数的规定，其计算公式为：

$$[(WST+C2WH+2) - C2WS - (C2WH+1)]$$

3. 根据时间设置相应的寄存器

这里我们以 S71PL064JA0BAW 为例进行讲解。S71PL064JA0BAW 含一个 Flash 和一个 PSRAM，通过查找其相关的 datasheet，我们得到以下结果。

读操作总共等待的时间参数如下：

```
Random access: 60ns/70ns
```

读操作时 PSRAM 的 tOE 的时间参数如表 16.2 所示。

表 16.2 tOE 时间参数

参　数	符　号	最 小 值	最 大 值
OE# Access Time	Toe	–	40

写操作时 PSRAM 的 tWP 的时间参数如表 16.3 所示。

表 16.3 tWP 时间参数

参　数	符　号	最 小 值	最 大 值
WE# Write Pulse Width	Twp	45	–

所以外部设备与 EMI 对应参数要满足：

```
[ RLT + 1 ] > 70ns
[ (RLT+1) - C2RS ]>tOE=40ns
[ (WST+C2WH+2) - C2WS - (C2WH+1) ]>tWP=45ns
```

假设 RLT = 4T，C2RS = 1T，WST = 3T，C2WH = 0，C2WS = 1T。

说 明

1T = 1 / 时钟频率，即 1Ticket。例如当时钟频率为 52MHz 时，则 1T = 1 / 52MHz = 19.5ns。

```
[ RLT + 1 ]= (4T+1T) x 19.2ns = 96ns > 70ns
tOE = [ (RLT+1) - C2RS ] =5T - 1T = 76.8ns > 40ns ( tOE MAX )
tWP = [ (WST+C2WH+2) - C2WS - (C2WH+1) ] = 5T - 1T - 1T = 57.6ns > 45ns (tWP Min)
```

加上其他的参数如：ADV=1，PLRT=1T，Burst mode，PSIZE=16B。

可设置 PSRAM 寄存器为：

```
#define RAM_CS_Setting 0x44864324
```

参照 PSRAM 的设置可以设置 Flash 的寄存器。例如：

```
C2WS=1T, C2WH=0, C2RS=1T, ADV=1, PLRT=1T, Page mode,
WST=3T, PSIZE=16B, RLT=4T
```

可设置 Flash 寄存器为：

```
#define NOR_CS_1_Setting 0x44854324
```

大多数 MTK 支持的 EMI 设备在 mcu\tools\MemoryDeviceList\MemoryDeviceList_XXX.xls 文件中可以找到，包含了 EMI 的相关寄存器设置。这些设置都是最优化设置，不需要我们进行修改，当选中某个 EMI 设备后，使用后面介绍的自动配置过程，自动将这些设置复制到 mcu\custom\system\custom_EMI.h 文件中。

16.3.3 EMI 自动配置过程

对于 EMI，MTK 提供了一套自动化配置的过程，这可以大大减少驱动开发人员的编码量和降低出错的几率。

基本过程如下图 16.7 所示。

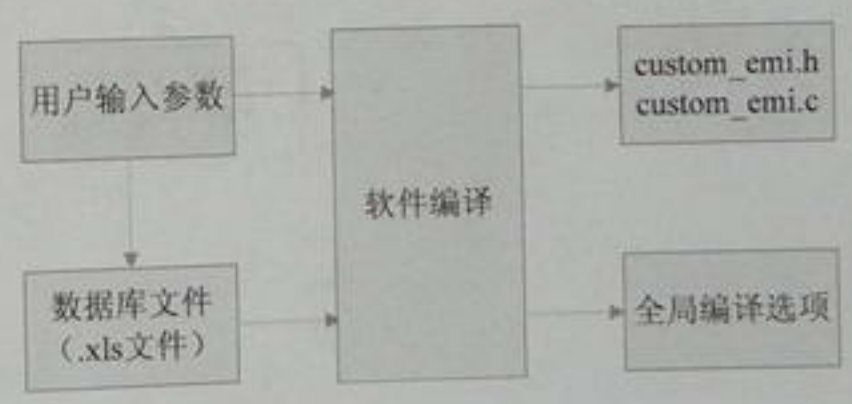


图 16.7 EMI 自动配置过程

1. 用户参考数据库文件（XLS 文件）输入 EMI 设置参数

数据库文件路径为：

ztenc26_06b_M431\tools\MemoryDeviceList\ MemoryDeviceList_XXX.xls。

文件中列举了 MTK 当前支持的各种外部存储设备（如 NOR Flash、RAM、LPSDRAM）及其相关参数（如寄存器设置、容量大小、区域划分）。

查找文件中是否有我们用到的 EMI 的类型型号。如果没找到则表明 MTK 目前对其还不支持；如果找到则写入 EMI 配置文件。

文件路径为：mcu\custom\system\ ProjectName \custom_MemoryDevice.h。

写入外部存储设备型号 ID 及类型，如下所示。

NOR-flash SRAM/PSRAM 类型：

```
#define MEMORY_DEVICE_TYPE NOR_RAM_MCP
#define CS0_PART_NUMBER LRS1828C
#define CS1_PART_NUMBER LRS1828C
```

LPSDRAM NFB 类型：

```
#define MEMORY_DEVICE_TYPE LPSDRAM
#define CS0_PART_NUMBER HYB18L256160BF_MT6228EVB
```

说　明

MEMORY_DEVICE_TYPE 是指存储设备的类型，主要有 NOR-flash SRAM/PSRAM MCP 和 LPSDRAM NFB 这两种类型。

NOR-flash SRAM/PSRAM MCP 类型的 CS0_PART_NUMBER 和 CS1_PART_NUMBER 要相同，LPSDRAM NFB 类型通常只用来制定 CS0_PART_NUMBER。

2. 软件编译过程

可以使用两种方法编译 EMI 设置。

（1）使用“New”方法编译工程。

EMI 检查过程如图 16.8 所示。

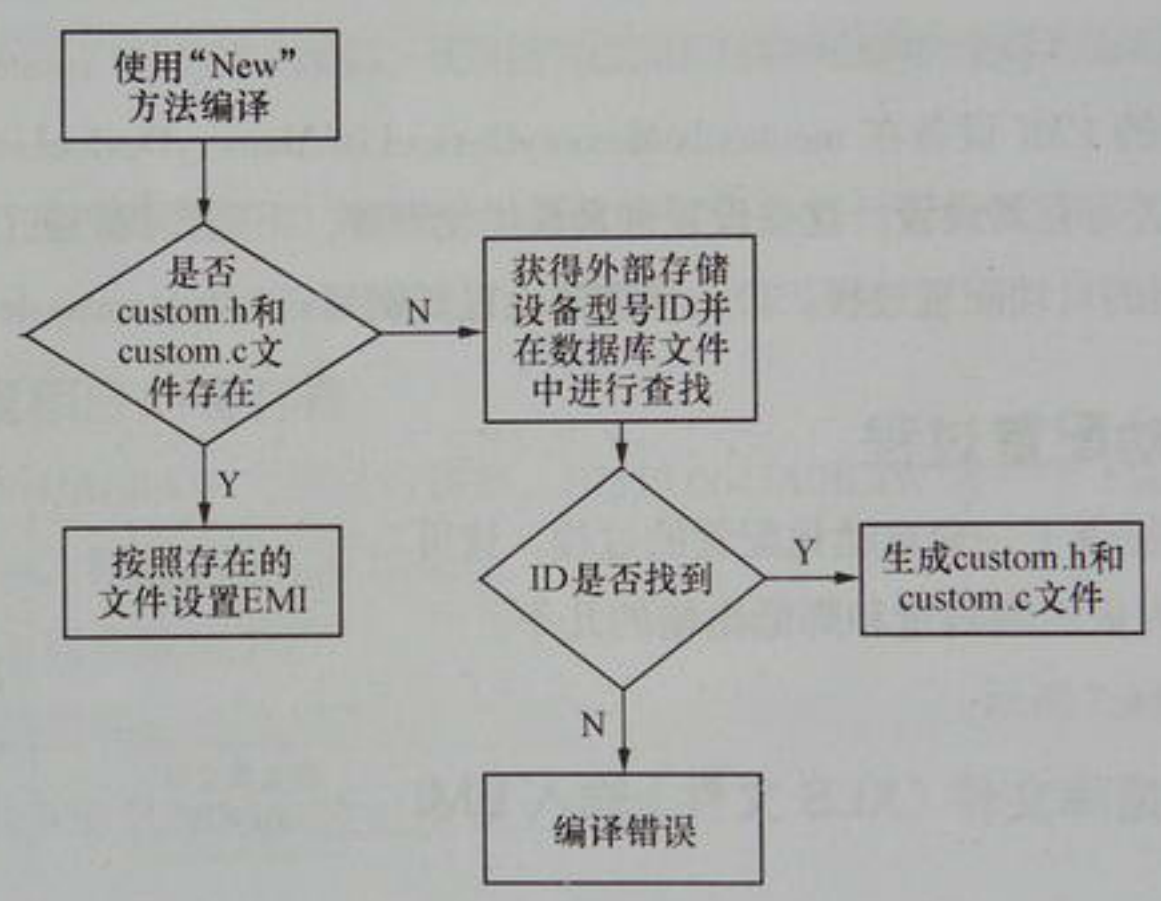


图 16.8　使用“New”编译时 EMI 检查过程

```
make ProjectName gprs new
```

- 检测是否有 custom_emi.h 和 custom_emi.c 文件。
- 如果有这两个文件则按照文件中的设置进行配置，这样用户可手动修改文件中的设置代码而不受重新编译影响。
- 如果没有这两个文件则获得外部存储设备型号 ID 并在数据库文件中进行查找。
- 没在数据库文件中找到此 ID，则报 Build 错误，表示不支持此类 EMI。
- 在数据库文件中找到此 ID，则生成这两个文件。

（2）使用 EMI 编译命令编译工程。

- 清除 custom_emi.h 和 custom_emi.c 文件：

```
make [project name] [gsm/gprs] emiclean
```

- 生成 custom_emi.h 和 custom_emi.c 文件：

```
make [project name] [gsm/gprs] emigen
```

> **提 示**
>
> 自定义与修改相应的 custom_emi.h 和 custom_emi.c 文件，修改后只用 remake 操作即可。custom_emi.h 和 custom_emi.c 文件在 ProjectDirectory\custom\system\ProjectName\目录下。

3. 生成 Custom_emi.h、Custom_emi.c 文件和全局编译选项

生成 Custom_emi.h 和 Custom_emi.c 文件后就可在文件上进行相关自定义设置的更改，例如 Flash 存储区域的分配操作，修改后只用 remake 编译就可以了。

```
make ProjectName gprs remake
```

文件路径为：

mcu\custom\system\ProjectName\custom_emi.h 和

mcu\custom\system\ProjectName\custom_emi.c。

16.3.4 EMI 存储区域划分

EMI 外部存储区域可分为 Flash 区域和 ROM 区域，Flash 区域分为代码区域和文件存储区，其中文件存储区又分为用户区域和系统区域。

假设我们使用的外部存储设备是由 128Mbit flash memory 和 32Mbit SRAM 组成，因为 1Byte 等于 8bit，所以它们就是我们通常所说的是 16MB ROM 和 4MB RAM。文件存储区占用 2MB，这 2MB 一般又被分为系统区域和用户区域，系统区域存储 NV 文件和 MMS 相关文件，对用户不可见，用户将其连上电脑就可以看到。但由于一些原因，有不少手机是不设用户区域的，用户区域过大，会导致彩信等一些模块不稳定，因此很多手机使用 NOR Flash 时不设用户区域，但 NAND Flash 一般都会设置一定的用户区域。如果需要设置，只要修改宏 PARTITION_SIZE 值就可以。所以我们能够使用的空间就只剩下 14MB。

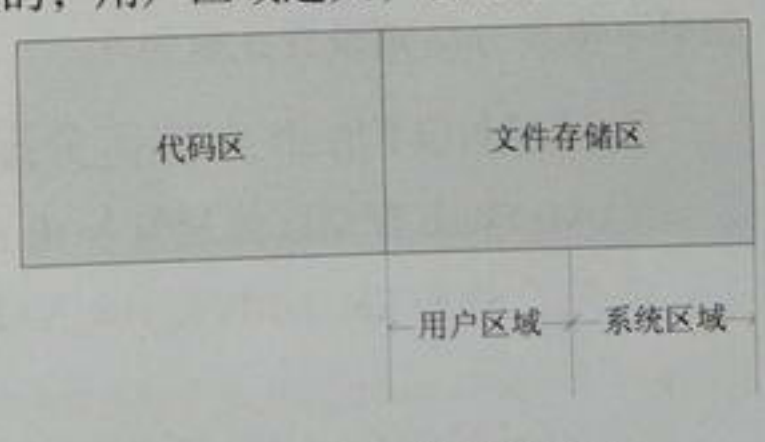


图 16.9 EMI 存储区域划分

EMI 存储区域划分如图 16.9 所示。

EMI 存储区域分配相关的步骤如下。

1. 定义文件存储区的起始位置

文件路径：mcu\custom\system\ProjectName\custom_EMI.h。

```
#define FLASH_BASE_ADDRESS    0x00E00000
```

这个宏变量的含义为文件存储区的起始地址，0～0x00e00000Bytes（0～14MB）为代码区，0x00e00000Byte（14MB）以后为文件存储区。

2. 定义文件存储区大小

文件路径：mcu\custom\system\ProjectName\custom_EMI.h。

```
#define ALLOCATED_FAT_SPACE    0x00200000
```

这个宏变量的含义为文件存储区大小，这里定义为 0x00200000Bytes（2MB）。

3. 划分文件存储区的各个子区域的大小和数量

文件路径：mcu\custom\system\ProjectName\custom_EMI.h。

```
FLASH_REGIONINFO_VAR_MODIFIER FlashRegionInfo RegionInfo[] =
{
    {0x40000, 7},
    {0x10000, 4},
    EndRegionInfo /* Don't modify this line */
};
```

RegionInfo 定义了文件存储区中子区域的信息表。一个子区域包含了连续的相同扇区大小的块，每个子区域的结构为：{子区域中扇区大小，子区域中扇区的数量}。RegionInfo 的值参考数据库文件（XLS 文件）中的值定义。上面代码中，可得文件存储区总大小为：0x40000 × 7 + 0x10000 × 4，即 2MB。如果要更改文件存储区大小为 4MB，要将{0x40000, 7}改为{0x40000, 15}，依此类推。

4. 定义文件存储区的区域总数

文件路径：mcu\custom\system\ProjectName\custom_EMI.h。

```
#define TOTAL_BLOCKS        11
```

定义了子区域中扇区数量的总和。

5. 定义用户区域大小

文件路径：mcu\custom\system\ProjectName\custom_MemoryDevice.h。

```
#define PARTITION_SECTORS  1024
```

定义了用户区域的大小，大小为：1024 × 512Byte，即 512KB。则系统区为：2MB – 512KB，即 1.5MB。如果不需要分区则设置此值为 0。

6. NAND Flash 存储区分配

NAND Flash 存储区分配与 NOR Flash 存储区分配有些不同。

定义了 SYSTEM_DRIVE_ON_NAND 或 NAND_FLASH_BOOTING 宏，代码如下：

```
#if ( defined(__FS_SYSDRV_ON_NAND__) || defined( _NAND_FLASH_BOOTING_) )
#define PARTITION_SIZE (180000)
#else
#define PARTITION_SIZE (0)
#endif
```

如果是 NAND Flash 启动，指定文件存储区的起始位置和定义文件存储区的代码如下：

```
#ifdef _NAND_FLASH_BOOTING_
#define FLASH_BASE_ADDRESS 0x02000000 // 32MB
#define ALLOCATED_FAT_SPACE 0x06000000 // 96MB
#endif /* _NAND_FLASH_BOOTING_ */
```

7. SCATXXXX.txt 文件

文件路径为：

mcu\build\ProjectName\SCATXXXX.txt。

在 SCAT 文件中，有一行是 SCHEME: external 14MB flash memory and 4MB SRAM，就是说 14MB ROM

和4MB RAM。由于用户需求不同，有些时候我们会修改 RAM 和 ROM 大小，一般就要修改相应的 SCAT 文件。在 SCAT 文件中，我们可以看到一行 ROM 0x00000000 0x00e00000，就是说可以使用的 ROM 从 0x00000000 开始，到 0x00e00000 结束，共计 0x00e00000 字节，在下面又可以看到这些 ROM 被分成 4 个 4MB 的段使用。在 SCAT 的行 EXTSRAM_LARGEPOOL_NORMAL 0x08000000 处我们可以看到 RAM 的使用情况，地址从 0x08000000 开始，到 0x08400000 结束，共计 0x00400000 BYTE，即 4MB。如果是 32MB ROM，8MB RAM，就要修改 ROM 0x00000000 0x00e00000 为 ROM 0x00000000 0x01c00000，修改 DUMMY_END 0x08400000 0x04 为 DUMMY_END 0x08800000 0x04。现在的 ROM 和 RAM 大小都为以前默认的两倍。

其实有时 RAM 空间紧张时不一定非要采用增加 RAM 来解决，这样成本较大，采用复用内存也可以节约大量内存。在 SCAT 文件中，很多时候，我们可以看到关键字 overlay，这是一些手机的应用中为节省内存使用的复合内存，如 INTSRAM_MULTIMEDIA 0x40000000 0xC000，声明了 MED 复用内存的起始地址，只要不冲突，这几乎是最好的解决内存紧张的方法。如果 ROM 空间紧张了，处理起来可能会比较麻烦一些，去掉不必要的图片、音乐和一些功能，减小图片的质量，把宏函数转为普通函数都可以节约一部分 ROM。

如果 RAM 或者 ROM 编译到最后出错，提示 ROM 空间不足或者 RAM 空间不足，这时就要精确计算超出部分的大小，然后再根据计算的结果值寻找解决办法。计算的方法是打开 LIS 文件，把 RAM 或者 ROM 加起来，减去 14 或者 4，超过的字节数，就是需要调整的内存大小。

关键代码如下：

```
;--------------------
;SCHEME   : external 15MB flash memory and 4MB SRAM
; PLATFORM : MT6226
;--------------------
ROM 0x08000000 0x00e00000
{
  ROM 0x08000000 0x00400000
  {
bootarm.obj (C$$code,+First)
        *.obj (LEADING_PART,+First)
        *plutommi.lib (+RO-CODE)
        *lcmmi.lib (+RO-CODE)
        *l4_classb.lib (+RO-CODE)
        *mtkapp.lib (+RO-CODE)
        *l1_classb.lib (+RO-CODE)
        *email.lib (+RO-CODE)
        *l4misc.lib (+RO-CODE)
  }
  ROM2 +0x0 0x00400000
  {
  …
  }
  …
  EXTSRAM_LARGEPOOL_NORMAL 0x00000140
  {
  …
  {
  * (+RW, +ZI)
```

```
}
DUMMY_END 0x00400000 0x04
{
bootarm.obj (DUMMY_POOL)
}
…
DYNAMIC_CODE6 0x40018800 OVERLAY 0x5000
{
 * (BCODE_DYNAMIC_CODE, BCODE_DYNAMIC_ZI)
 }
 INTSRAM_MULTIMEDIA 0x4001D800 0x00029800
 {
 med_main.obj (INTERNRW1, INTERNZI1)
 isp_comm.obj (INTERNRW1, INTERNZI1)
 }
}
```

16.4 驱动定制工具的使用

16.4.1 驱动定制工具简介

驱动定制工具是一个在 PC 端使用的工具，只需要简单地选择填入操作就可对 GPIO、GPO、EINT、ADC、KEYPAD 等的驱动参数进行设置，生成相关驱动代码，方便驱动程序的开发。

驱动定制工具的名字为：DrvTool。

可执行程序位置为：mcu\custom\drv\Drv_Tool\DrvGen.exe。

此工具生成的相关文件有这样的形式：xxx_drv.h、xxx_drv.c 和 xxx_var.c（xxx 表示驱动的组件名，例如 gpio、eint、adc、keypad、uem）。

工程文件：为了方便开发者保存设置，DrvTool 可生成工程文件，来存储在工具中进行的相关设置，用户关闭 DrvTool 后下次可通过工程文件继续使用和编辑设置信息。

工程文件的默认路径为 mcu\custom\drv\misc_drv\ProjectName\codegen\codegen.dws。

1. DrvTool 的基本使用流程（如图 16.10 所示）

（1）硬件工程师制作原理图。

（2）硬件工程师制作硬件配置表。

（3）驱动工程师分析硬件配置表。

（4）使用 DrvTool 配置 GPIO、GPO、EINT、ADC、KEYPAD 驱动参数。

（5）将配置保存为工具工程文件。

（6）生成相关文件。

（7）最后编译到手机 image 中。

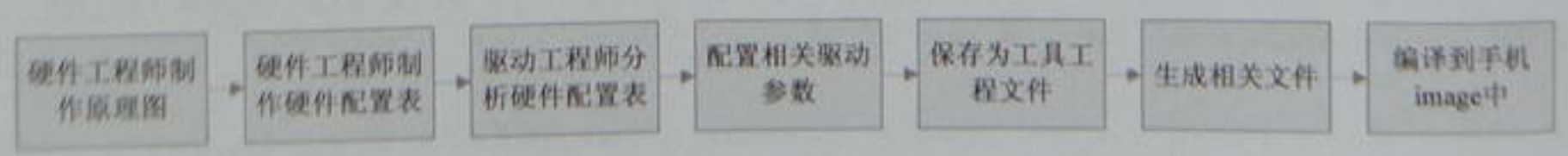


图 16.10 DrvTool 使用流程

2. 配置文件和描述文件

（1）配置文件

文件名的形式如 mt6xxx.fig，其中 mt6xxx 指所选 BB 的类型。此配置文件主要描述在不同的 BB 芯片上的可配置的相关选项。

（2）描述文件

文件名的形式如 xxx.CMP。

GPIO.CMP：描述 GPIO 变量名。

EINT.CMP：描述 EINT 变量名。

ADC.CMP：描述 ADC 变量名。

KEYPAD.CMP：描述逻辑键盘的定义。

UEM.CMP：描述所有 UEM 字符串的最大长度。

16.4.2 工具使用说明

下面讲述如何使用 DrvTool。

1. 打开 DrvTool

首先要保证在 makefile 中可使用 DrvTool。

文件路径：mcu\make\ProjectName.mak。

```
DRV_CUSTOM_TOOL_SUPPORT = TRUE # TRUE or FALSE
```

> **提 示**
>
> 当这个特性被应用后，编译时_CUST_NEW_会被定义，这个宏用于标识相关驱动是用 DrvTool 进行设置的，不需要在代码中重新设置。

运行工具，工具路径为：mcu\custom\drv\Drv_Tool\DrvGen.exe。

运行的界面如图 16.11 所示。

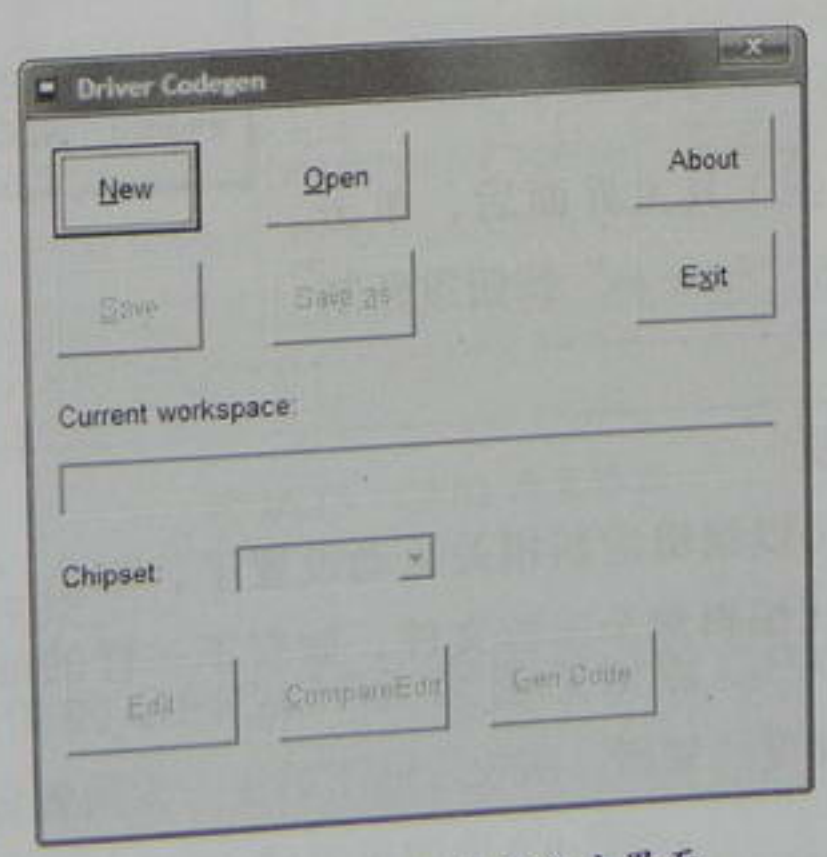


图 16.11 DrvTool 运行主界面

2. 创建新工程

在工具主界面单击“New”按钮就可以新建一个 DrvTool 工程。需要指定工程文件存放的路径和文件名，文件名以.dws 为后缀。MTK 工程中默认有一个路径来存放工程文件，如图 16.12 所示。

文件路径：mcu\custom\drv\misc_drv\ProjectName\codegen。

在 Current workspace 栏中显示了工程文件的路径和文件名，这时可以选择 BB 芯片类型（可选择的类型在 mt6xxx.fig 文件中有描述），然后就可以进行编辑操作了，如图 16.13 所示。

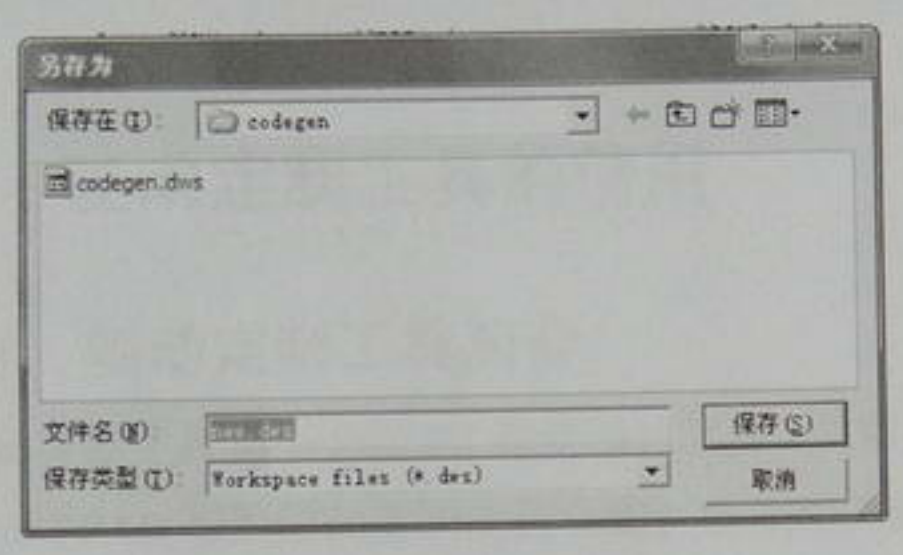


图 16.12　存储工程文件

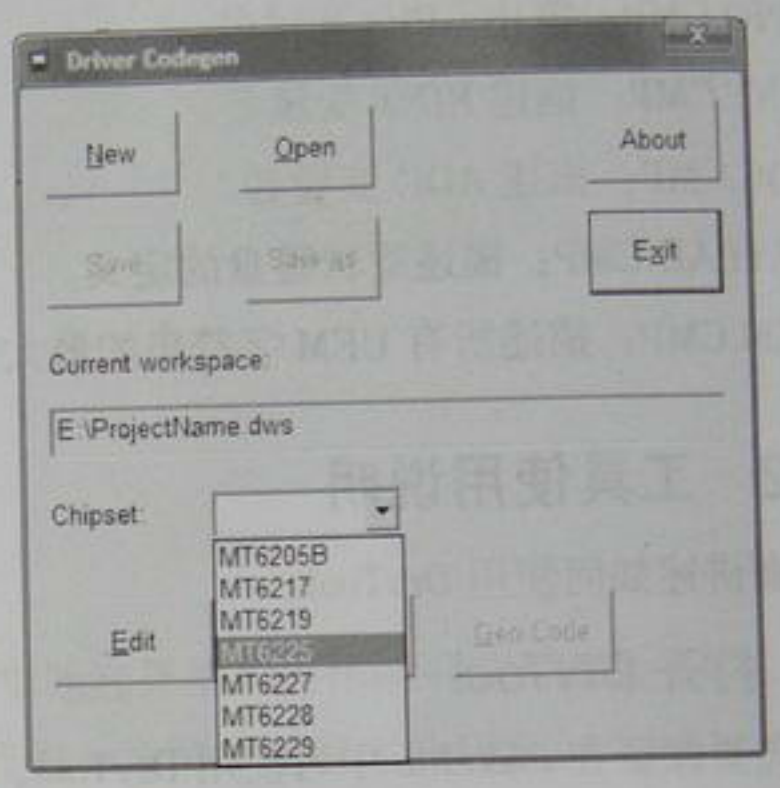


图 16.13　选择 BB 芯片类型

3. 打开工程

在工具主界面单击“Open”按钮可以打开一个已经存在的 DrvTool 工程。需要指定工程文件存放的路径和文件名，文件名以.dws 为后缀。MTK 工程中有一个默认的工程文件，可以以它为基础，打开并进行相关设置，然后就可以进行编辑操作了。如图 16.14 所示。

文件路径：mcu\custom\drv\misc_drv\ProjectName\codegen\codegen.dws。

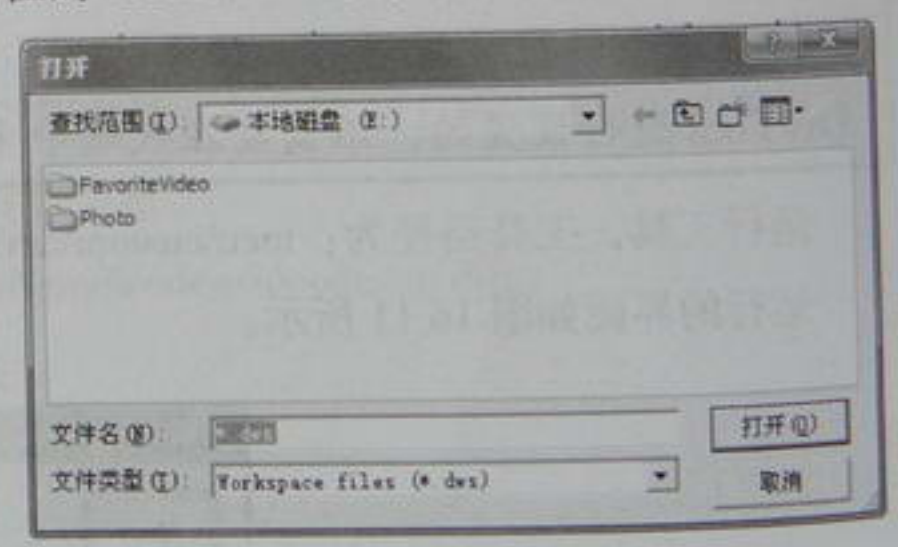


图 16.14　打开工程

4. 存储/另存为工程

当编辑完相关设置，返回到工具主界面后，单击“Save”按钮即可存储，也可单击“Save as”按钮重新制定路径和文件名进行存储。

5. 编辑/对比编辑设置项

创建或打开工程文件后就可以编辑定制相关驱动设置了，单击“Edit”按钮编辑相关驱动值，单击“CompareEdit”选项来对比编辑两个工程文件，如有不一样的地方工具会以显著的颜色标记，便于对比。

6. 生成驱动代码

编辑好驱动设置后，单击工具主界面上的“Gen Code”按钮生成代码。

16.4.3 GPIO/GPO 设置

GPIO（General Purpose Input Output，通用输入/输出），GPO（General Purpose Output，通用输出）。通过 GPIO/GPO 端口可输入或输出数据信号，与外部设备进行交互。我们以 MT6226 的 BB 为例，有 56 个 GPIO，5 个 GPO。

可设置的 GPIO/GPO 寄存器如下。

- GPIO_DIRn（方向控制寄存器）
- GPIO_PULLENn（上拉/下拉寄存器）
- GPIO_DINVn（数据倒置寄存器）
- GPIO_DOUTn（数据输出寄存器）
- GPIO_DINn（数据输入寄存器）
- GPIO_MODEn（模式设置寄存器）

现在 GPIO/GPO 寄存器基本上都用 DrvTool 工具进行设置，避免手动改写代码产生的问题。

使用 DrvTool 工具设置 GPIO 的步骤如下。

1. 打开 DrvTool 切换到 GPIO/GPO 设置的页面

GPIO 设置界面如图 16.15 所示。

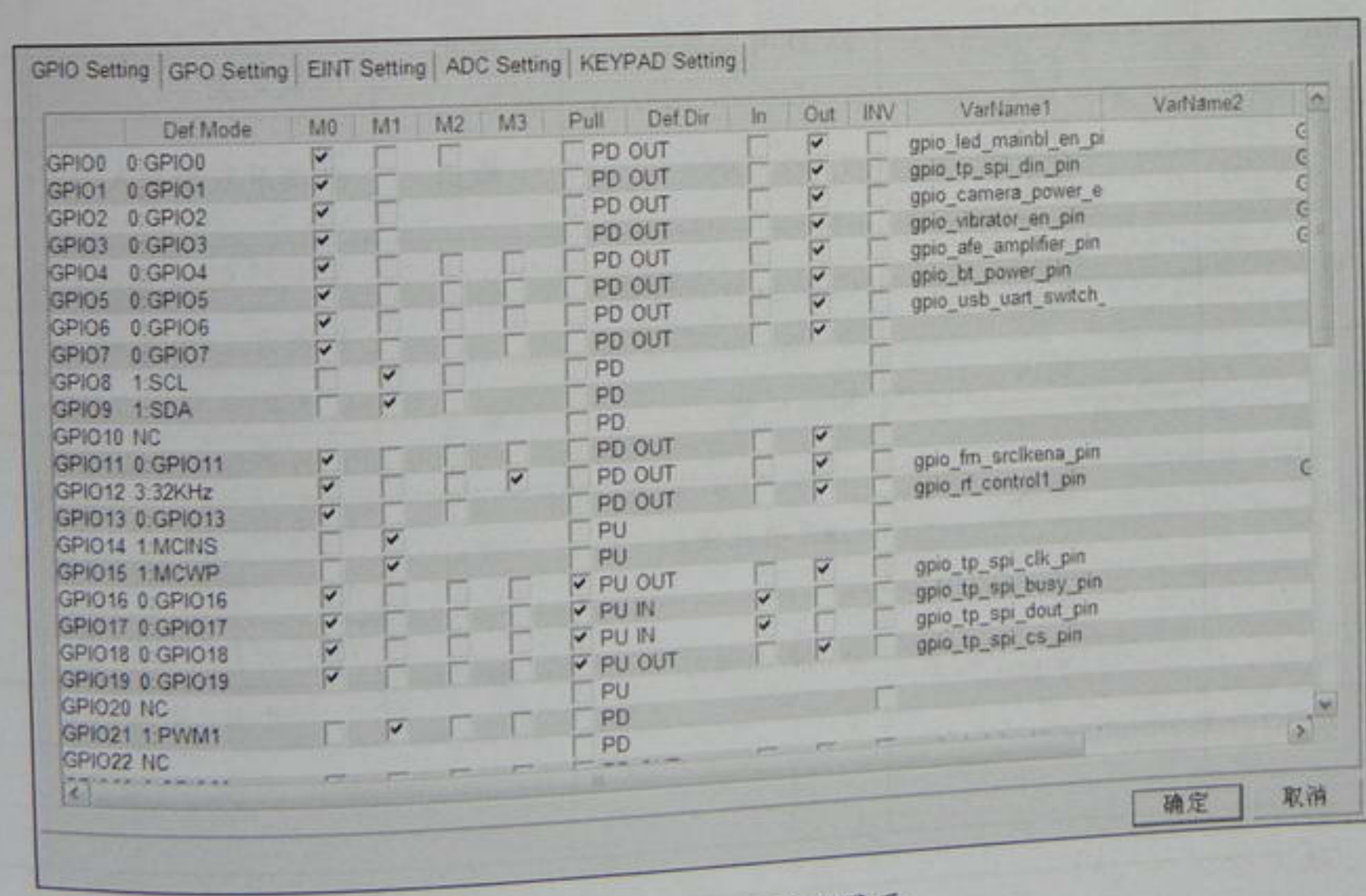


图 16.15 GPIO 设置界面

GPO 设置界面如图 16.16 所示。

2. 参考硬件工程师提供的 GPIO/GPO 配置表设置 GPIO/GPO 参数

如果是 GPIO 界面，可以设置模式、上拉下拉、方向、倒置、变量 1、变量 2。如果是 GPO 界面，可设置模式、变量 1、变量 2。GPIO/GPO 配置表如表 6.4 所示。

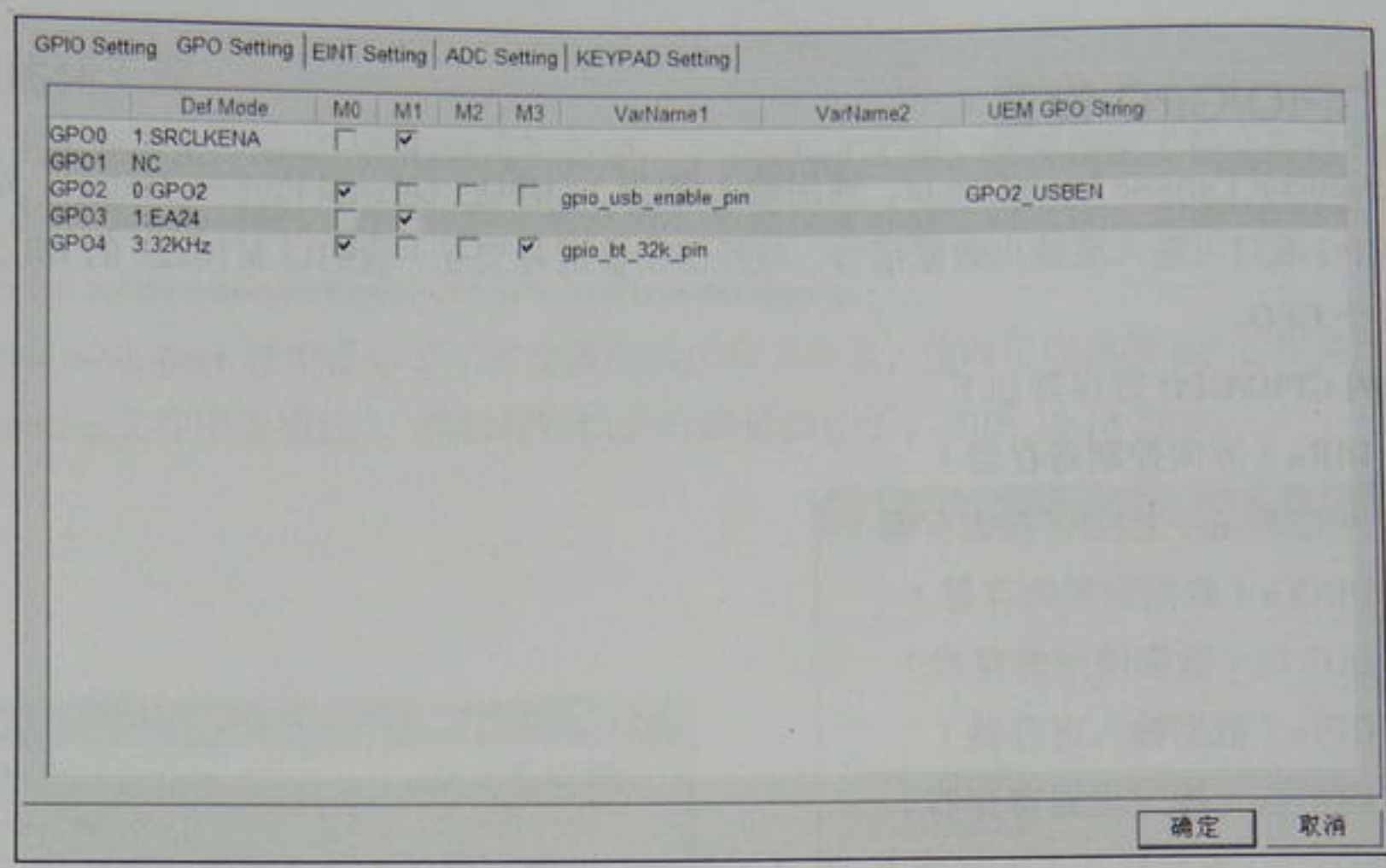


图 16.16 GPO 设置界面

表 16.4 GPIO/GPO 配置表

GPIO 编号	默认状态	用途	使用方式		备注
GPIO0	输入	LCD 背光使能	输出	0 背光灭	
	下拉			1 被光亮	
GPIO1	输入	TP-LSDN	输出	触摸屏数据输入	
	下拉				
GPIO2	输入	Camera 电源使能	输出	0 关闭 Cam 电源	
	下拉			1 打开 Cam 电源	
GPIO3	输入	振动使能	输出	0 关闭振动	
	下拉			1 打开振动	
GPIO4	输入	音频功放使能	输出	0 关闭喇叭	
	下拉			1 打开喇叭	
GPIO5	输入	BT 电源使能	输出	0 关闭 BT 电源	
	下拉			1 打开 BT 电源	
……					

注 意

GPIO/GPO 模式的设置要根据用途描述在模式下拉栏中进行选择，如果没有与之匹配的选项，则选择模式 0。

GPIO/GPO 变量名赋值：有些 GPIO/GPO 要赋予相关变量名，用以在软件模块中使用，工具中包含两列变量名的设置，即 VarName1 和 VarName2。并且只有默写 GPIO 模式才能选择这些变量，VarName2 用于当两个模块或应用程序共享一个 pin 脚时，进行区分。如图 16.17 所示。

VarName1	VarName2
gpio_led_status1_en_pin	gpio_flash_test_cmd_bit0_pin

图 16.17 GPIO/GPO 变量赋值

注 意

变量名可在 GPIO 和 GPO 中设置，不能重复，即在一个 GPIO/GPO 位置选择了的变量名，在其他 GPIO/GPO 位置就不能选择了。这些变量名都在 mcu\custom\drv\Drv_Tool\GPIO.cmp 文件中有定义。

文件路径：

mcu\custom\drv\misc_drv\ProjectName\gpio_drv.c。

GPIO/GPO 相关代码如下：

```
#ifdef __CUST_NEW__
extern void GPIO_setting_init(void);
#ifdef __PRODUCTION_RELEASE__
kal_bool   gpio_debug_enable = KAL_FALSE;
#else /* __PRODUCTION_RELEASE__ */
kal_bool   gpio_debug_enable = KAL_FALSE;
#endif /* __PRODUCTION_RELEASE__ */
#ifdef __USB_ENABLE__
extern const char gpio_usb_enable_pin;
#endif
```

其中如果设置 gpio_debug_enable 为 KAL_TRUE 则在进行 GPIO/GPO 操作前会先进行断言，可用于调试模式，便于对出现的异常进行调试。

```
void GPIO_init(void)
{
#ifdef __CUST_NEW__
GPIO_setting_init();
#ifdef __USB_ENABLE__
   GPO_WriteIO(0,gpio_usb_enable_pin);
 #endif   /*__USB_ENABLE__*/
#else /* __CUST_NEW__ */
    DRV_WriteReg(GPIO_MODE1,0x0000);
    DRV_WriteReg(GPIO_MODE2,0x5005);
    DRV_WriteReg(GPIO_MODE3,0x4500);
    DRV_WriteReg(GPIO_MODE4,0x1554);
    DRV_WriteReg(GPIO_MODE5,0x0295);
    DRV_WriteReg(GPIO_MODE6,0x5540);
    DRV_WriteReg(GPIO_MODE7,0x0055);
    DRV_WriteReg(GPO_MODE,0x0141);

    DRV_Reg(GPIO_DIR1) |= 0xffff;
    DRV_Reg(GPIO_DIR2) |= 0x80A0;
    DRV_Reg(GPIO_DIR3) |= 0xffff;
    DRV_Reg(GPIO_DIR4) |= 0xf0;

    #ifdef __USB_ENABLE__
    GPO_WriteIO(0,2);
```

```
    //DRV_WriteReg(GPIO_DIR1,0x0010);
    #endif /*__USB_ENABLE__*/
#endif
#endif /* __CUST_NEW__ */
}
```

如果使用 DrvTool 方式对 GPIO/GPO 进行设置，则进入 GPIO_setting_init 函数对 GPIO 的模式、方向、上下拉、是否倒置等参数进行设置，然后将 gpio_usb_enable_pin 端口设置为低。

GPIO 设置文件路径为：

mcu\custom\drv\misc_drv\common_drv\gpio_setting.c。

> *注　意*
>
> 这些对 GPIO/GPO 的设置等手机启动完成后就不能再进行改变了。

16.4.4 EINT 设置

EINT（External Interrupt，外部中断），主要处理由外部传来的中断请求。不同的 BB 有不同数量的 EINT 通道可以配置，以 MT6226 为例，有 8 个 EINT 通道可设置。

可设置 EINT 的通道号和 De-bounce 时间，使用 De-bounce 时间来保证稳定地获取 EINT。

EINT 通道号设置的相关文件路径：mcu\custom\drv\misc_drv\ProjectName\eint_def.c。

```
#ifdef __CUST_NEW__
#include "eint_drv.h"
extern const kal_uint8 AUX_EINT_NO;
extern const kal_uint8 CHRDET_EINT_NO;

#if defined(__PHONE_CLAMSHELL__) || defined(__PHONE_SLIDE__)
extern const kal_uint8 CLAMDET_EINT_NO;
#endif  /* __PHONE_CLAMSHELL__ || __PHONE_SLIDE__ */

#ifdef __USB_ENABLE__
extern const kal_uint8 USB_EINT_NO;
#else /* __USB_ENABLE__ */
extern const kal_uint8 USB_EINT_NO;
#endif  /* __USB_ENABLE__ */

#ifdef __BT_SUPPORT__
extern const kal_uint8 BT_EINT_NO;
#endif  /* __BT_SUPPORT__ */

#if defined(__CHARGER_USB_DETECT_WIHT_ONE_EINT__)
extern const kal_uint8 CHR_USB_EINT_NO;
#endif  /* __CHARGER_USB_DETECT_WIHT_ONE_EINT__ */

#ifdef __SWDBG_SUPPORT__
extern const kal_uint8 SWDBG_EINT_NO;
#endif  /* __SWDBG_SUPPORT__ */

#if defined(MOTION_SENSOR_SUPPORT)
extern const kal_uint8 MOTION_SENSOR_EINT_NO;
#endif  /* MOTION_SENSOR_SUPPORT */
```

```
#if defined(TOUCH_PANEL_SUPPORT) || defined(HAND_WRITING)
extern const kal_uint8 TOUCH_PANEL_EINT_NO;
#endif  /* TOUCH_PANEL_SUPPORT || HAND_WRITING */

#ifdef __WIFI_SUPPORT__
extern const kal_uint8 WIFI_EINT_NO;
#endif  /* __WIFI_SUPPORT__ */

#ifdef __SYNC_LCM_SUPPORT__
extern const kal_uint8 SYNC_LCM_EINT_NO;
#endif /* __SYNC_LCM_SUPPORT__ */

#else /* __CUST_NEW__ */

const kal_uint8 AUX_EINT_NO = 0;
const kal_uint8 CHRDET_EINT_NO = EINT_CHANNEL_NOT_EXIST;

#if defined(__PHONE_CLAMSHELL__) || defined(__PHONE_SLIDE__)
const kal_uint8 CLAMDET_EINT_NO = ?;
#endif  /* __PHONE_CLAMSHELL__ || __PHONE_SLIDE__ */

#ifdef __USB_ENABLE__
const kal_uint8 USB_EINT_NO = 2;
#else /* __USB_ENABLE__ */
const kal_uint8 USB_EINT_NO = EINT_CHANNEL_NOT_EXIST;
#endif  /* __USB_ENABLE__ */

#ifdef __BT_SUPPORT__
const kal_uint8 BT_EINT_NO = 3;
#endif  /* __BT_SUPPORT__ */

#if defined(__CHARGER_USB_DETECT_WIHT_ONE_EINT__)
const kal_uint8 CHR_USB_EINT_NO = 2;
#endif  /* __CHARGER_USB_DETECT_WIHT_ONE_EINT__ */

#ifdef __SWDBG_SUPPORT__
const kal_uint8 SWDBG_EINT_NO = EINT_CHANNEL_NOT_EXIST;
#endif  /* __SWDBG_SUPPORT__ */

#if defined(MOTION_SENSOR_SUPPORT)
const kal_uint8 MOTION_SENSOR_EINT_NO = ?;
#endif  /* MOTION_SENSOR_SUPPORT */

#if defined(TOUCH_PANEL_SUPPORT) || defined(HAND_WRITING)
const kal_uint8 TOUCH_PANEL_EINT_NO = 1;
#endif  /* TOUCH_PANEL_SUPPORT || HAND_WRITING */

#ifdef __WIFI_SUPPORT__
const kal_uint8 WIFI_EINT_NO = ?;
#endif  /* __WIFI_SUPPORT__ */

#ifdef __SYNC_LCM_SUPPORT__
const kal_uint8 SYNC_LCM_EINT_NO = 6;
#endif /* __SYNC_LCM_SUPPORT__ */

#endif /* __CUST_NEW__ */
```

De-bounce 时间设置的相关文件路径：mcu\custom\drv\misc_drv\ProjectName\eint_def.c。

```
/*Unit: 10ms*/
kal_uint8 custom_eint_sw_debounce_time_delay[EINT_MAX_CHANNEL] =
{
    50,   /*EINT 0,500ms*/
    0,   /*EINT 1,500ms*/
    50,   /*EINT 2,500ms*/
    50
};
```

1. 要响应 EINT 中断主要有以下步骤

（1）注册函数。

使用 EINT 注册函数注册响应函数：

```
void EINT_Registration(kal_uint8 eintno, kal_bool Dbounce_En, kal_bool ACT_Polarity, void
(reg_hisr)(void), kal_bool auto_umask)
```

其中 eintno 表示中断号，Dbounce_En 表示是否打开 Debounce 功能，ACT_Polarity 表示中断的极性，reg_hisr 表示中断响应函数，auto_umask 表示在终端响应函数完成后是否自动设置标志位。

例如函数：

```
EINT_Registration(CLAMDET_EINT_NO,KAL_TRUE,clam_state,CLAM_EINT_HISR, KAL_TRUE);
```

（2）实现中断响应函数。

```
void CLAM_EINT_HISR(void)
{
    ilm_struct          *clam_ilm;
    aux_id_struct       *aux_id_data;

    if (clam_state == LEVEL_LOW)
    {
        aux_id_data = (aux_id_struct*) construct_local_para(sizeof(aux_id_struct), TD_CTRL);
        aux_id_data->aux_id = AUX_ID_CLAM_CLOSE;
        DRV_BuildPrimitive(clam_ilm,
                           MOD_EINT_HISR,
                           MOD_UEM,
                           MSG_ID_AUX_ID,
                           aux_id_data);
    }
    else
    {
        aux_id_data = (aux_id_struct*)construct_local_para(sizeof(aux_id_struct), TD_CTRL);
        aux_id_data->aux_id = AUX_ID_CLAM_OPEN;
        DRV_BuildPrimitive(clam_ilm,
                           MOD_EINT_HISR,
                           MOD_UEM,
                           MSG_ID_AUX_ID,
                           aux_id_data);
    }
    msg_send_ext_queue(clam_ilm);
    ...
}
```

（3）最后开启/关闭 EINT 极性

```
void CLAM_EINT_HISR(void)
{
...
clam_state = !clam_state;
EINT_Set_Polarity(CLAMDET_EINT_NO,clam_state);
...
}
```

2. 使用 DrvTool 工具设置 EINT 步骤如下

（1）打开 DrvTool 工具切换到 EINT 设置的页面，如图 16.18 所示。

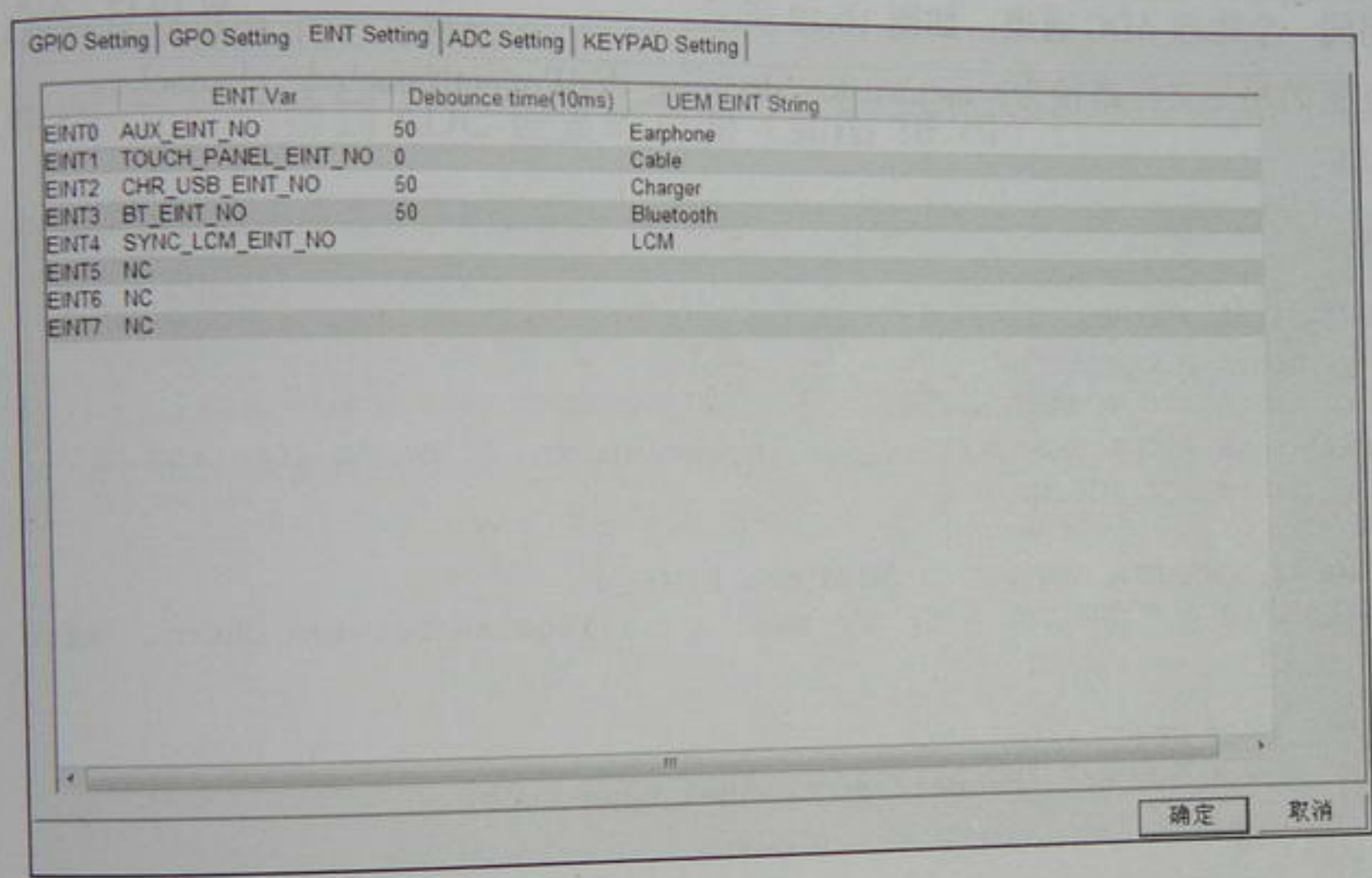


图 16.18　EINT 设置界面

（2）参考硬件工程师提供的 EINT 配置表设置 EINT。如表 16.5 所示。

表 16.5　ADC 配置表

通　道	解　释
EINT_0	耳机插入检测
EINT_1	触摸屏中断
EINT_2	充电中断
EINT_3	蓝牙中断
EINT_4	NC
EINT_5	NC
EINT_6	NC
EINT_7	NC

16.4.5　ADC 设置

ADC（Auxiliary Device Channel，辅助设备通道），主要用于外部设备的检测、电池电量检测、充电

检测、温度测量等。ADC 的实质就是对通道上的电压进行检测，不同的 BB 有不同数量的 ADC 通道可以配置，以 MT6226 为例，有 7 个 ADC 通道可设置。

ADC 有两种操作模式：立即模式和时间触发模式。立即模式就是一旦 ADC 检测功能被启用立即开始检测，时间触发模式就是在制定时间后开始检测。也可以通过设置成自动采样的模式，当本次采样完成后开始下一次的采样。

ADC 的一个物理的通道可有许多逻辑的拥有者，即多个 Task 可以同时访问同一个物理 ADC 通道。如图 16.19 所示。

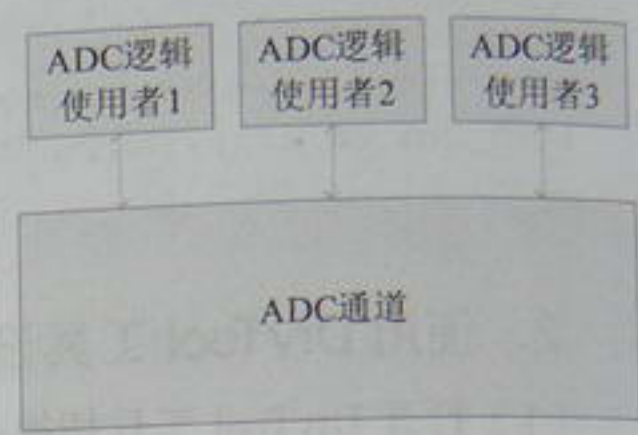


图 16.19 ADC 通道

ADC 配置的相关文件路径为：mcu\custom\drv\misc_drv\ProjectName\adc_channel.c。

代码如下：

```
#ifndef __CUST_NEW__
const kal_uint8 ADC_VBAT = 0;
const kal_uint8 ADC_VISENSE = 1;
const kal_uint8 ADC_VBATTMP = 2;
const kal_uint8 ADC_VCHARGER = 3;
const kal_uint8 ADC_ACCESSORYID = 5;//tmp solution, maybe SA will add this
const kal_uint8 ADC_PCBTMP = 6;

#if defined(__CHARGER_USB_DETECT_WIHT_ONE_EINT__)
const kal_uint8 ADC_CHR_USB = 6;  // adc to distinguish between charger with usb
#endif

#endif /* __CUST_NEW__ */
kal_uint8 custom_adc_get_channel(adc_channel_type type)
{
switch(type)
{
    case vbat_adc_channel:
        return ((kal_uint8)ADC_VBAT);
    case visense_adc_channel:
        return ((kal_uint8)ADC_VISENSE);
    case vbattmp_adc_channel:
        return ((kal_uint8)ADC_VBATTMP);
    case aux_adc_channel:
        return ((kal_uint8)ADC_ACCESSORYID);
    case vcharger_adc_channel:
        return ((kal_uint8)ADC_VCHARGER);
    case pcbtmp_adc_channel:
        return ((kal_uint8)ADC_PCBTMP);
#if defined(__CHARGER_USB_DETECT_WIHT_ONE_EINT__)
        case chr_usb_adc_channel:
            return ((kal_uint8)ADC_CHR_USB);
#endif
        default:
            ASSERT(0);
            return 100;
    }
}
```

可设置的主要 ADC 类型如表 16.6 所示。

表 16.6　可设置的主要 ADC 类型

类　型	解　释
ADC_VBAT	电池电压通道
ADC_VBATTMP	电池温度电压通道
ADC_ACCESSORYID	辅助设备电压通道
ADC_VISENSE	充电电流感应电压通道
ADC_CHARGER	充电电压通道

使用 DrvTool 工具设置 ADC 步骤如下。

1. 打开 DrvTool 切换到 ADC 设置的页面（如图 16.20）

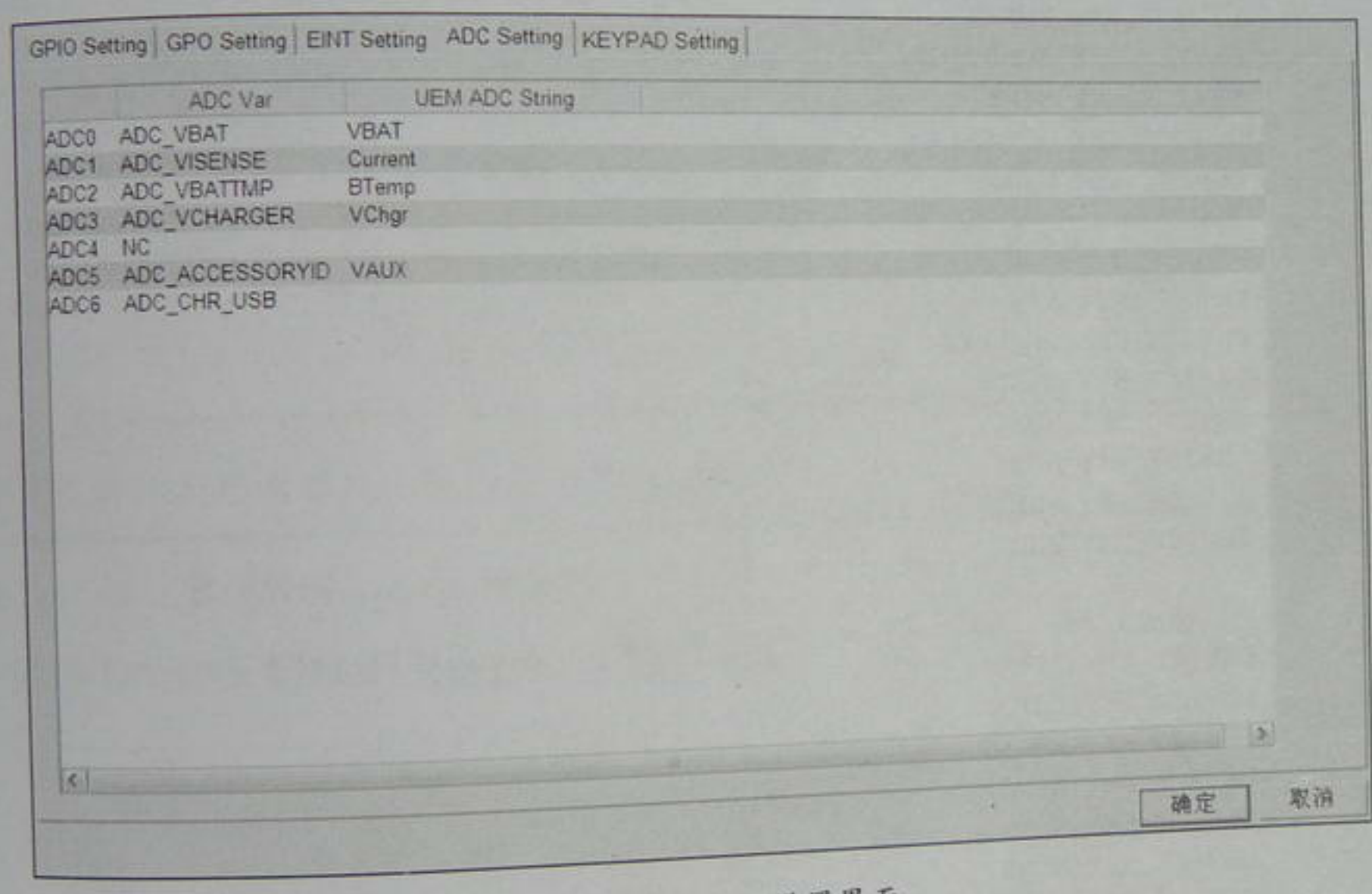


图 16.20　ADC 设置界面

2. 参考硬件工程师提供的 ADC 配置表设置 ADC 的类型（如表 16.7）

表 16.7　ADC 配置表

通　道	解　释
ADC_0	电池电压通道
ADC_1	充电电流感应电压通道
ADC_2	NC
ADC_3	充电电压通道
ADC_4	NC
ADC_5	辅助设备电压通道
ADC_6	USB 检测通道

16.4.6 Keypad 设置

Keypad 的硬件实现是一个二维矩阵，交叉的点就代表一个按键，所以设置 Keypad 就是对这个矩阵赋予有意义的键值。另外还可定义开机键和设置长按开机键时间，以 MT6226 为例，是 7×6 的按键矩阵。

Keypad 定义的相关文件路径：mcu\custom\drv\misc_drv\ProjectName\keypad_def.c。

```
const keypad_struct  keypad_custom_def = {
        /*keypad mapping*/
        {
        #ifndef __CUST_NEW__
            /* row 0 */
            DEVICE_KEY_SEND,
            DEVICE_KEY_SK_RIGHT,
            DEVICE_KEY_NONE,
            DEVICE_KEY_FUNCTION,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_END,

            /* row 1 */
            DEVICE_KEY_UP,
            DEVICE_KEY_SK_LEFT,
            DEVICE_KEY_MENU,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_END,

            /* row 2 */
            DEVICE_KEY_DOWN,
            DEVICE_KEY_LEFT,
            DEVICE_KEY_RIGHT,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_END,

            /* row 3 */
            DEVICE_KEY_NONE,
            DEVICE_KEY_VOL_UP,
            DEVICE_KEY_NONE,
            DEVICE_KEY_VOL_DOWN,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_END,

            /* row 4 */
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
            DEVICE_KEY_NONE,
```

```
        DEVICE_KEY_END,

        /* row 5 */
        DEVICE_KEY_NONE,
        DEVICE_KEY_NONE,
        DEVICE_KEY_NONE,
        DEVICE_KEY_NONE,
        DEVICE_KEY_NONE,
        DEVICE_KEY_NONE,
        DEVICE_KEY_END,
#else /* __CUST_NEW__ */
        KEYPAD_MAPPING
#endif /* __CUST_NEW__ */
},
/*power on period*/
Custom_Keypress_Period,     //开机键长按的时间
/*powerkey position*/
#ifndef __CUST_NEW__
/*powerkey position*/
DEVICE_KEY_END              //开机键
#else /* __CUST_NEW__ */
POWERKEY_POSITION
#endif /* __CUST_NEW__ */
};
```

注 意

一个逻辑按键可能对应矩阵上多个物理按键。

使用 DrvTool 工具设置 Keypad 步骤如下。

1. 打开 DrvTool 切换到 Keypad 设置的页面（如图 16.21）

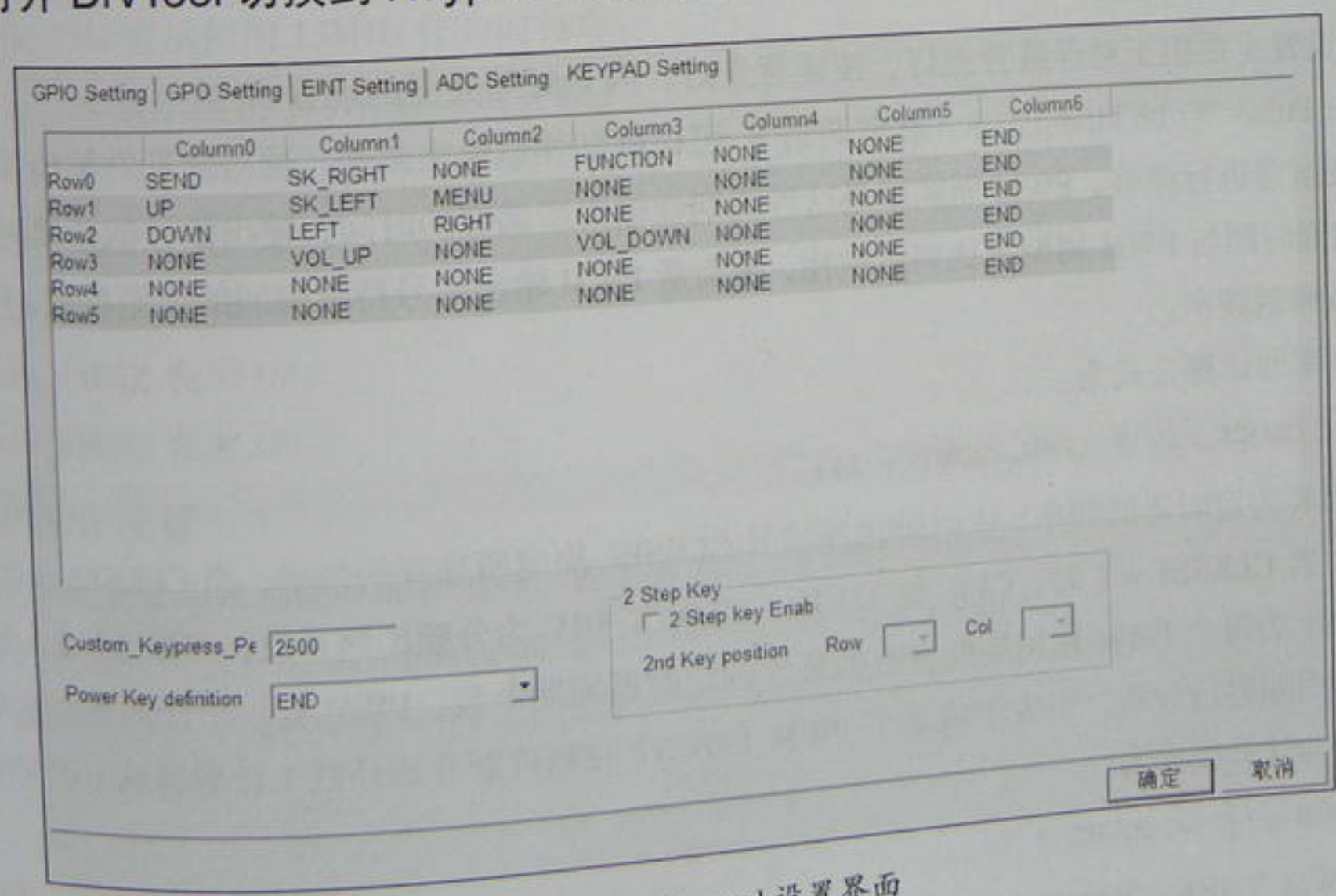


图 16.21 Keypad 设置界面

2. 参考硬件工程师提供的 Keypad 矩阵配置表设置（如表 16.8）

表 16.8 Keypad 配置表

	KCOL0	KCOL1	KCOL2	KCOL3	KCOL4
KROW0	多功能键	0	返回	上	
KROW1	左	8	右	OK	
KROW2	拨号	*	6	下	
KROW3	1	7	3	2	
KROW4	4	#	9	5	
KROW5					
GND					PWR KEY

16.5 设备默认值的设置

有些设备的默认参数可作为一个逻辑数据项存储在 NVRAM 系统中，在设备启动的时候被读取，然后进行设置。可设置默认值的设备主要有如下几类：

- PWM 设置；
- LCD 对比度设置；
- 电池电量等级设置；
- 声音模块设置。

16.5.1 PWM 设置

PWM 设置主要用于对屏幕背光灯、按键背光灯、闪光灯等亮度的设置调整。

PWM（Pulse-Width Modulation，脉宽调制），可以周期性地输出信号，通过周期性输出信号的持续时间可对亮度等进行调整。PWM 周期就是这种周期性输出所需要的时间，输出信号的持续时间又叫接通时间，接通时间与 PWM 周期之比叫占空比，是衡量 PWM 输出信号持续时间的重要参数。PWM 周期的倒数叫做调制频率。

调制频率的计算公式为：

```
CLK / ( CLOCK_DIV * (PWM_COUNT + 1))
```

其中 CLK 为定时器的频率，由时钟选择信号 CLKSEL 决定所选择的频率，当 CLKSEL = 0 时，CLK 为 13MHz；当 CLKSEL = 1 时，CLK 为 32KHz。CLOCK_DIV 为分频比例系数，可取 1、2、4、8 值。PWM_COUNT 为每个 PWM 输出周期含有的最大的定时器周期个数，PWM 有一个计数器用来对到达第几个定时器周期进行计数，当大于或等于 PWM_COUNT 时则重新开始计数（计数器从 0 开始计数）。

占空比的计算公式为：

```
PWM_THRES / (PWM_COUNT + 1)
```

其中 PWM_THRES 为输出“门槛”标识，当计数器大于或等于 PWM_THRES 时，PWM 输出信号 1，

当计数器小于 PWM_THRES 时，PWM 输出信号 0。

这些公式中的参数都可由改变 PWM 寄存器来控制。PWM 信号时序图如图 16.22 所示。

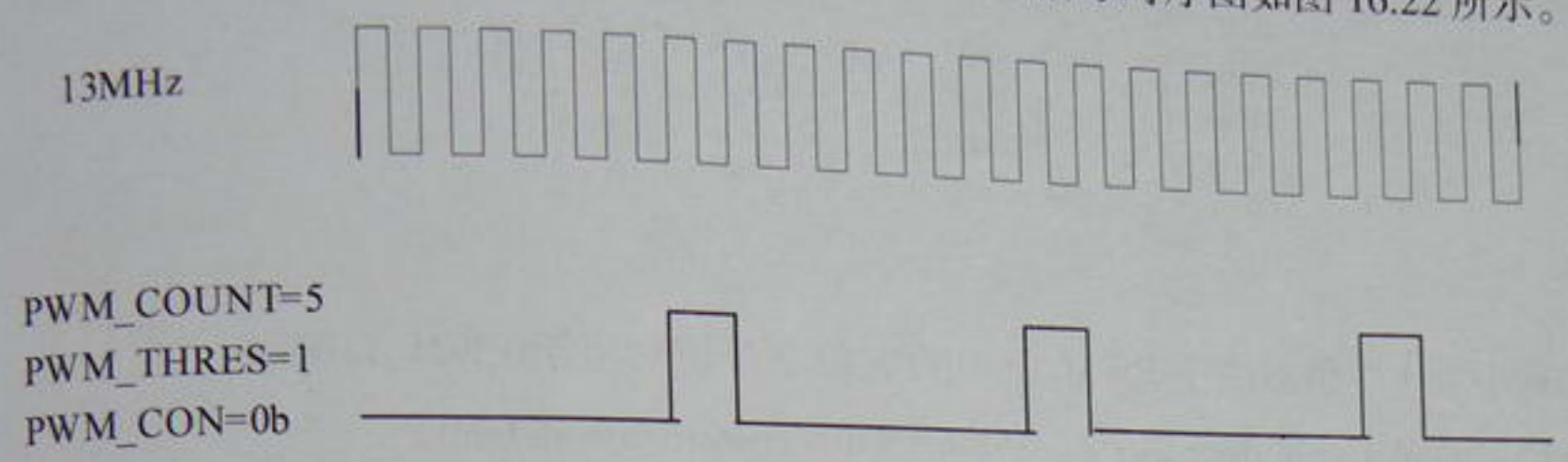


图 16.22 PWM 信号时序图

如图所示，PWM_CON = 0b 表示输入 CLK 为 13MHz；PWM_COUNT = 5 表示计数器从第 0 个定时器周期开始计数，当到第 5 个定时器周期时，就重新从 0 开始计数；PWM_THRES = 1 表示计数器为 1~5 时输出 PWM 数据为 0，当计数器为 0 时输出 PWM 数据为 1。

1. PWM 初始化

PWM 初始化相关文件的路径为：mcu\custom\drv\misc_drv\ProjectName\PWMdrv.c。

```
void PWM_initialize(void)
{
   PWM_Init(pwmclk_32k,pwmclk_8MHZ);   /*clok = 32k/8*/
   PWM2_Init(pwmclk_32k,pwmclk_8MHZ);  /*clok = 32k/8*/
}
```

在此处可设置 PWM1 和 PWM2 的初始值。

（1）可以使用 13MHz 或 32kHz 的时钟源，对应公式中的 CLK。

pwmclk_13M 表示使用 13MHz 作为时钟源。

pwmclk_32k 表示使用 32kHz 作为时钟源。

（2）可以使用 4 种分频比例系数，对应公式中的 PWM_COUNT。

pwmclk_1MHZ 表示 1。

pwmclk_2MHZ 表示 1/2。

pwmclk_4MHZ 表示 1/4。

pwmclk_8MHZ 表示 1/8。

2. PWM 设置

PWM 设置主要用来调整屏幕背光灯、键盘背光灯、闪光灯亮度等。

设置的相关文件为：mcu\custom\app\ProjectName\nvram_user_config.c。

```
kal_uint32 const NVRAM_EF_CUST_HW_LEVEL_TBL_DEFAULT[] = {
                200, 60,  /* PWM 1  Level 1 */
                200, 50,  /* PWM 1  Level 2*/
                200, 40,  /* PWM 1  Level 3*/
                200, 30,  /* PWM 1  Level 4*/
                200, 20,  /* PWM 1 Level 5 */
                200, 80,  /* PWM 2  Level 1 */
                200, 60,  /* PWM 2  Level 2*/
```

```
                    200, 40,  /* PWM 2  Level 3*/
                    200, 20,  /* PWM 2  Level 4*/
                    200, 0,   /* PWM 2 Level 5 */
                    200, 80,  /* PWM 3  Level 1 */
                    200, 60,  /* PWM 3  Level 2*/
                    200, 40,  /* PWM 3  Level 3*/
                    200, 20,  /* PWM 3  Level 4*/
                    200, 0,   /* PWM 3 Level 5 */
...
};
```

每组结构中第 1 个值对应于 PWM_COUNT，第 2 个值对应于 PWM_THRES。

文件路径：mcu\custom\drv\misc_drv\ProjectName\custom_hw_default.c。

```
kal_uint32 PWM1_Level_Info[PWM_MAX_LEVEL][2] =
{
   /*Freq,duty*/
   {160,80},    /*Level 1*///yangpeng
   {160,60},    /*Level 2*/
   {160,40},    /*Level 3*/
   {160,20},    /*Level 4*/
   {160,0}    /*Level 5*/
};

kal_uint32 PWM2_Level_Info[PWM_MAX_LEVEL][2] =
{
   /*Freq,duty*/
   {200,20},    /*Level 1*/
   {200,40},    /*Level 2*/
   {200,60},    /*Level 3*/
   {200,80},    /*Level 4*/
   {200,100}    /*Level 5*/
};

kal_uint32 Alter_Level_Info[PWM_MAX_LEVEL][2] =
{
   /*Freq,duty*/
   {200,80},    /*Level 1*/
   {200,60},   /*Level 2*/
   {200,40},   /*Level 3*/
   {200,20},   /*Level 4*/
   {200,0}  /*Level 5*/
};
```

提 示

Alert 主要用于输出警告声音，其设置原理与 PWM 设置基本一致，故不再专门介绍，有兴趣的读者可参考相关资料。

GPIO 端口中的 21、22、23 号端口都有专门的模式用于 PWM 或 Alert，如果硬件使用了这些模式，则代码中设置的 PWM 值才可生效。

16.5.2 LCD 对比度设置

相关文件路径为：mcu\custom\app\ProjectName\nvram_user_config.c。

```
kal_uint32 const NVRAM_EF_CUST_HW_LEVEL_TBL_DEFAULT[] = {
...
```

```
#if (defined MTKLCM)
        34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
        44, 45, 46, 47, 48,
#elif (defined MTKLCM_COLOR)
        /* Main LCD contrast level 1 ~ 15 */
        148, 149, 150, 151, 152, 153, 154, 155, 156, 158,
        160, 162, 164, 166, 168,
#else
        /* Main LCD contrast level 1 ~ 15 */
        120, 122, 124, 126, 128, 130, 132, 134, 136, 138,
        140, 142, 144, 146, 148,
#endif
        /* Main LCD Bias Param (Reserved) */
        0, 0, 0, 0, 0,
        /* Main LCD Linerate Param (Reserved) */
        0, 0, 0, 0, 0,
        /* Main LCD Temperature Param (Reserved) */
        0, 0, 0, 0, 0,
        /* Sub LCD contrast level 1 ~ 15 */
        20, 22, 24, 26, 28, 30, 32, 34, 36, 38,
        40, 42, 44, 46, 48,
        /* Sub LCD Bias Param (Reserved) */
        0, 0, 0, 0, 0,
        /* Sub LCD Linerate Param (Reserved) */
        0, 0, 0, 0, 0,
        /* Sub LCD Temperature Param (Reserved) */
        0, 0, 0, 0, 0,
…
};
```

16.5.3 电池电量等级设置

在NVRAM中设置电量相关文件路径为：mcu\custom\app\ProjectName\nvram_user_config.c。

```
kal_uint32 const NVRAM_EF_CUST_HW_LEVEL_TBL_DEFAULT[] = {
…
        /* Battery voltage Level */
        3300000, /* Low Battery Power off */
        3400000, /* Low Battery TX PROHIBIT*/
        3500000, /* Low Battery Warring */
        /* battery level 1 ~ 8 */
        3600000, 3740000, 3880000, 9999999,
        9999999, 9999999, 9999999,
…
};
```

如上面代码所示，3.30V 或以下为低电量自动关机；3.40V 或以下为无法呼出电话；3.50V 或以下为低电量报警；3.60V 以下为电量等级 0；3.60V～3.74V 为电量等级 1；3.74V～3.88V 为电量等级 2；3.88V 以上为电量等级 3。

电量等级提供给应用层是用一个枚举型结构表示的，对应于 NVRAM 中设置的电量等级。

文件路径为：mcu\ps\l4\include\device.h。

```
typedef enum
{
    BATTERY_LOW_POWEROFF = 0,
    BATTERY_LOW_TX_PROHIBIT,
    BATTERY_LOW_WARNING,
```

```
    BATTERY_LEVEL_0,
    BATTERY_LEVEL_1,
    BATTERY_LEVEL_2,
    BATTERY_LEVEL_3,
    BATTERY_LEVEL_4 = BATTERY_LEVEL_3, /* BATTERY_LEVEL_4 */
    BATTERY_LEVEL_5 = BATTERY_LEVEL_3, /* BATTERY_LEVEL_5 */
    BATTERY_LEVEL_6 = BATTERY_LEVEL_3, /* BATTERY_LEVEL_6 */
    BATTERY_LEVEL_LAST = 9
} battery_level_enum;
```

16.5.4 声音模块设置

MTK 的声音模块的作用是使输入的音频数据可以放大输出，这样用户就可以通过手机听到声音和音乐了；或者用户输入语音数据，语音模块传到 MTK 芯片内部进行处理。我们可以看出声音传递主要涉及 3 方面：BB 或 FM 端、声音模块、用户。

声音模块的框架图如下图 16.23 所示。从 BB 或 FM 输入到声音模块的数据有左声道声音、右声道声音、语音输出数据；从声音模块输出到 BB 或 FM 的数据有语音输入数据；从音乐模块输出到用户的有耳机左声道声音、耳机右声道声音、耳机语音数据、手机声音；从用户输入到音乐模块的有耳机麦克风数据、手机麦克风数据。

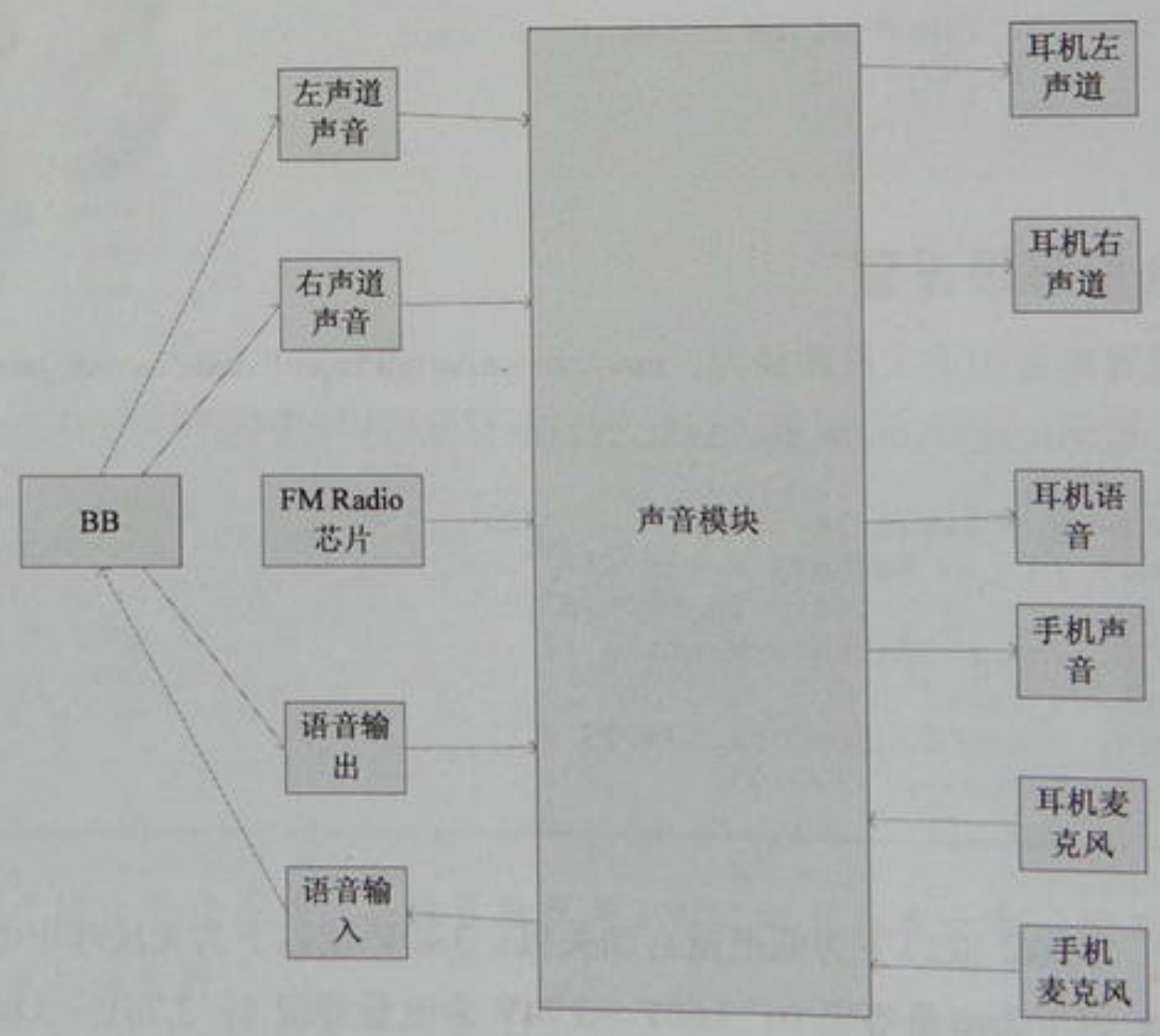


图 16.23 声音模块

1. 声音模块与用户之间的数据传输通道可用一些宏标志来代替，用于代码编程

耳机左声道声音、耳机右声道声音：L1SP_BUFFER_ST。

耳机语音数据：L1SP_BUFFER_0。

手机声音：L1SP_BUFFER_1。

手机麦克风数据：L1SP_LNA_0。

耳机麦克风数据：L1SP_LNA_1。

这些宏标志定义的文件路径为：mcu\interface\l1audio\L1audio.h。

```
/* ---------------------------------- */
/* Audio Output Device Definition  */
/* ---------------------------------- */
#define  L1SP_BUFFER_0              0x01   /*NOTE: Don't use buffer definition directly  */
#define  L1SP_BUFFER_1              0x02   /*   Use speaker definition below          */
#define  L1SP_BUFFER_ST       0x04
#define  L1SP_BUFFER_EXT      0x08
#define  L1SP_BUFFER_EXT_G    0x10
#define  L1SP_STEREO2MONO     0x20   /* Do not use this term for speaker definition */
#define  L1SP_BUFFER_ST_M     (L1SP_BUFFER_ST|L1SP_STEREO2MONO)
#define  L1SP_EXT_DAC_BUF0    0x40
#define  L1SP_EXT_DAC_BUF1    0x80

#define  L1SP_LNA_0           0
#define  L1SP_LNA_1           1
#define  L1SP_LNA_FMRR        3
```

2. 设置音乐通道

通过设置宏标志给音乐通道，使上层应用更方便调用这些音乐通道。

普通模式麦克风通道：L1SP_MICROPHONE1。

耳机模式麦克风通道：L1SP_MICROPHONE2。

普通模式听筒通道：L1SP_SPEAKER1。

耳机模式听筒通道：L1SP_BUFFER_ST。

音量放大器输出通道：L1SP_LOUD_SPEAKER。

文件路径：mcu\custom\audio\ProjectName\audcoeff.c。

```
const unsigned char  L1SP_MICROPHONE1  = L1SP_LNA_0;
const unsigned char  L1SP_MICROPHONE2  = L1SP_LNA_1;
const unsigned char  L1SP_SPEAKER1     = L1SP_BUFFER_0;
const unsigned char  L1SP_SPEAKER2     = L1SP_BUFFER_ST;
const unsigned char  L1SP_LOUD_SPEAKER =
                                L1SP_BUFFER_ST_M|L1SP_BUFFER_EXT;
```

3. 初始化音量放大器

要设置好音乐放大器的 GPIO 和相关寄存器，音量放大器才能正常工作。AFE_SwitchExtAmplifier 函数的功能是打开或关闭喇叭，一般由一个 GPIO 或 GPO 脚控制。

文件路径：mcu\custom\audio\ProjectName\afe.c。

```
/*****************************************************************************
* FUNCTION
*  AFE_SwitchExtAmplifier
* DESCRIPTION
*   This function is to turn on/off external amplifier.
*****************************************************************************/
void AFE_SwitchExtAmplifier( char sw_on )
{
    void GPO_WriteIO(char data,char port);
```

```
#ifdef __CUST_NEW__
    #if defined(MT6318)
    if( sw_on )
        pmic_speaker_enable(KAL_TRUE);
    else
        pmic_speaker_enable(KAL_FALSE);
    #else/*MT6305*/
    extern const char gpio_afe_amplifier_pin;
    if( sw_on )
        GPIO_WriteIO( 1, gpio_afe_amplifier_pin);/*wushengneng modify 2008.07.17*/
    else
        GPIO_WriteIO( 0, gpio_afe_amplifier_pin);/*wushengneng modify 2008.07.17*/
    #endif
#else /* __CUST_NEW__ */
    #if defined(MT6318)
    if( sw_on )
        pmic_speaker_enable(KAL_TRUE);
    else
        pmic_speaker_enable(KAL_FALSE);
    #else/*MT6305*/
    if( sw_on )
        GPO_WriteIO( 1, 1 );
    else
        GPO_WriteIO( 0, 1 );
    #endif
#endif /* __CUST_NEW__ */
}
/*****************************************************************************
* FUNCTION
*  AFE_Initialize
* DESCRIPTION
*   This function is to set the initial value of AFE HW.
*****************************************************************************/
void AFE_Initialize( void )
{
    // AFE_VAC_CON0::VCALI is chip dependent, will be set in l1audio\afe2.c
    // immediately after this function is called
    *AFE_VAC_CON0  = 0x40;
    *AFE_VAC_DCON1 = 0x80;
}
```

4. 音量设置

文件路径：mcu\custom\audio\ProjectName\nvram_default_audio.c。

（1）类型 1

音量设置要指定音量模式、音量类型、音量大小。

音量模式有 3 种：普通模式、耳机模式、免提模式。

音量类型有 7 种：Call Tone、Keypad Tone、Microphone、FM radio、Speech Sound、Side Tone、Media。

音量大小有 7 个等级：每个等级的范围为 0～255。

这些音量值可在代码中设置，也可以在工程模式中进行设置。

```
#define GAIN_NOR_KEY_VOL    3
#define GAIN_NOR_KEY_VOL0          67
#define GAIN_NOR_KEY_VOL1          87
```

```
#define GAIN_NOR_KEY_VOL2          107
#define GAIN_NOR_KEY_VOL3          127
#define GAIN_NOR_KEY_VOL4          147
#define GAIN_NOR_KEY_VOL5          167
#define GAIN_NOR_KEY_VOL6          187
```

表示普通模式，按键音，音量大小默认为第 3 个等级，即 127。

类型 1 可设置的音量类型列举如下。

· 普通模式：

```
- GAIN_NOR_CTN_VOL0-7 (Normal mode Call tone)
- GAIN_NOR_KEY_VOL0-7 (Normal mode Keypad tone)
- GAIN_NOR_MIC_VOL0-7 (Normal mode Microphone)
- GAIN_NOR_GMI_VOL0-7 (Normal mode GMI)
- GAIN_NOR_SPH_VOL0-7 (Normal mode Speech tone)
- GAIN_NOR_SID_VOL0-7 (Normal mode Side tone)
- GAIN_NOR_MED_VOL0-7 (Normal mode melody)
```

· 耳机模式：

```
- GAIN_HED_CTN_VOL0-7 (Headset mode Call tone)
- GAIN_HED_KEY_VOL0-7 (Headset mode Keypad tone)
- GAIN_HED_MIC_VOL0-7 (Headset mode Microphone)
- GAIN_HED_GMI_VOL0-7 (Headset mode GMI)
- GAIN_HED_SPH_VOL0-7 (Headset mode Speech tone)
- GAIN_HED_SID_VOL0-7 (Headset mode Side tone)
- GAIN_HED_MED_VOL0-7 (Headset mode melody)
```

· 免提模式：

```
- GAIN_HND_CTN_VOL0-7 (Handfree mode Call tone)
- GAIN_HND_KEY_VOL0-7 (Handfree mode Keypad tone)
- GAIN_HND_MIC_VOL0-7 (Handfree mode Microphone)
- GAIN_HND_GMI_VOL0-7 (Handfree mode GMI)
- GAIN_HND_SPH_VOL0-7 (Handfree mode Speech tone)
- GAIN_HND_SID_VOL0-7 (Handfree mode Side tone)
- GAIN_HND_MED_VOL0-7 (Handfree mode melody)
```

（2）类型 2

16 个等级的音量设置，例如音乐播放器和视频播放器，这些音量可在代码中设置也可以在工程模式中进行设置。

代码如下所示：

```
#define GAIN_NOR_MED_VOL_MAX 176
#define GAIN_HED_MED_VOL_MAX 208  /*Headset*/
#define GAIN_HND_MED_VOL_MAX 176  /* Handfree*/
#define GAIN_TVO_VOL_MAX 176

#define GAIN_NOR_MED_VOL_STEP 4 /* 1=0.5dB as unit */
#define GAIN_HED_MED_VOL_STEP 4 /* 1=0.5dB as unit */
#define GAIN_HND_MED_VOL_STEP 4 /* 1=0.5dB as unit */
#define GAIN_TVO_VOL_STEP 4 /* 1=0.5dB as unit */
```

XXX_MAX 表示最大的音量值，XXX_STEP 表示步进值和等级。

5. 音量设置的相关 API

在应用层设置音量的 API 主要包括：

```
Tone_SetOutputVolume
KT_SetOutputVolume
L1SP_SetMicrophoneVolume
FMR_SetOutputVolume
L1SP_SetSpeechVolume
L1SP_SetSidetoneVolume
Media_SetOutputVolume
```

16.6 可插入设备的配置

可插入设备包括充电线、辅助设备（如耳机）、USB 线和 UART 线。

16.6.1 充电参数设置

充电模块是一个独立的 Task，外界与 MMI Task、AUX Task、NVRAM 系统、ADC 驱动、PMIC 驱动相互协调工作，内部主要由 ADC 调度器和充电算法逻辑组成，框架如下图 16.24 所示。

一般手机上使用的电池是锂电池，锂电池充电过程主要有预充电阶段、恒流充电阶段、恒压充电阶段，最后直到充电完成，充电过程如图 16.25 所示。

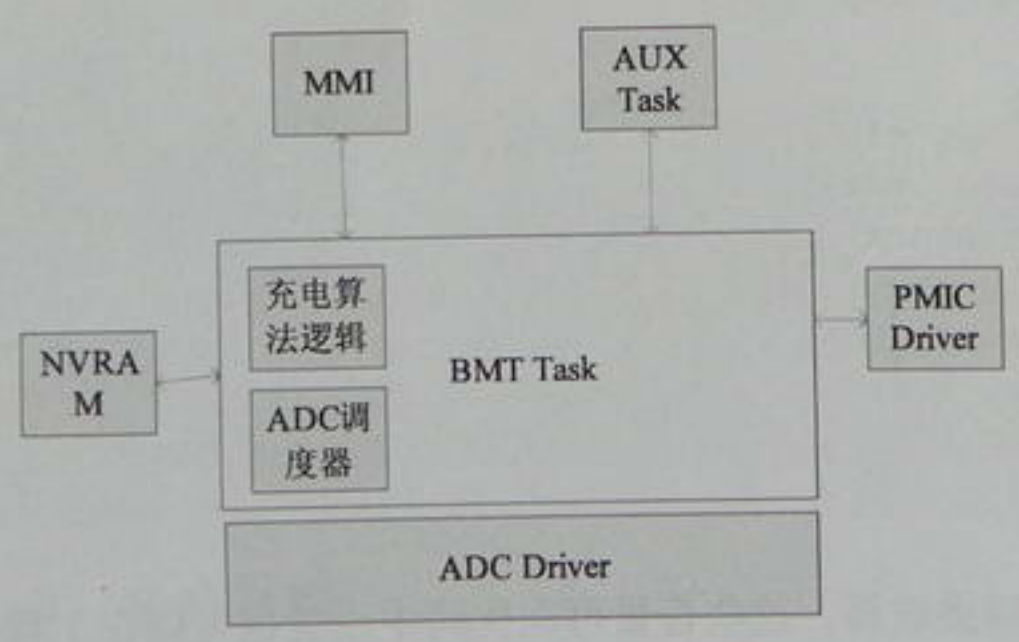


图 16.24 充电模块框架图

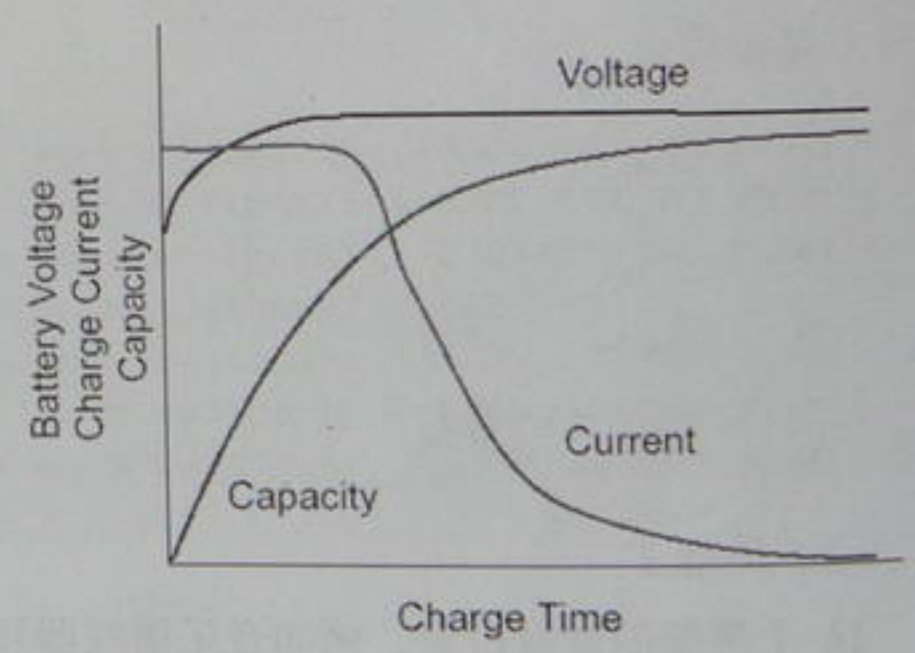


图 16.25 锂电池充电过程图

充电模块中的充电算法逻辑实质上就是一个状态迁移图。当插入充电器后进入初始状态，在等待一定的安全时间后进入预充电状态，当充电电压大于 3.4V 时进入快速充电状态（即锂电池的恒流充电阶段），然后当充电电压大于 4.1V 时进入充电结束状态（即锂电池的恒压充电阶段），最后当充电电流小于 60mA 时进入充电完成状态。当处于预充电、快速充电、充电结束、充电完成状态时，遇到有电话通信并且充电电压大于 4.05V 时，就进入通话/暂停状态。在通话/暂停状态当充电电压小于 3.8V 时又进入快速充电状态；在充电完成状态当充电电压小于 4.1V 又进入快速充电状态。

充电过程完整状态图如图 16.26 所示。

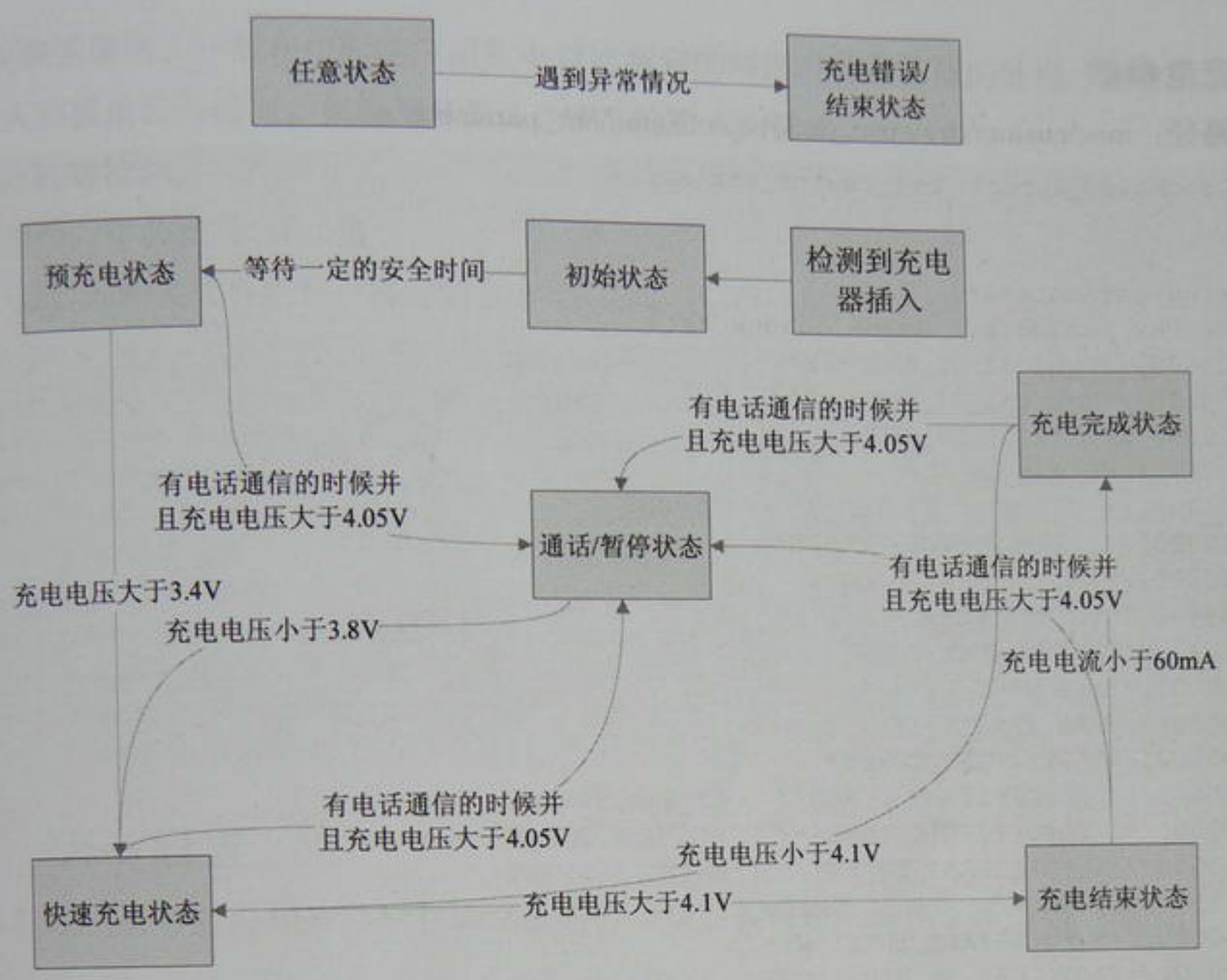


图 16.26 充电过程完整状态图

在任意状态下，当遇到异常情况时会进入充电错误/结束状态。

异常情况主要有：

- ADC 检测电压异常。
- 等待安全时间超时。
- 充电器拔出。

与充电模块相关的参数主要有以下几种。

1. ADC 校准值

文件路径：mcu\custom\drv\misc_drv\ProjectName\chr_parameter.c。

```
bmt_customized_struct  bmt_custom_chr_def =
{
…
/*ratio = adc_volt_factor/100*/
//const kal_uint16 adc_volt_factor[ADC_MAX_CHANNEL]
{
   100,
   100,
    50,
   250,
   50,
    50,/*NC*/
    50 /*NC*/
 },
…
 };
```

2. 充电参数

文件路径：mcu\custom\drv\misc_drv\ProjectName\chr_parameter.c。

```
bmt_customized_struct  bmt_custom_chr_def =
{
…
/*charing parameters*/
/*Check Phy parameters,Maybe changed*/
   1100000,/*Typical_LI_BATTYPE*/
   1100000,/*Typical_NI_BATTYPE*/
   1000000,/*ICHARGE_ON_HIGH*/
   20000,/*ICHARGE_ON_LOW*/
   1000000,/*ICHARGE_OFF_HIGH*/  //??50000
   4050000,/*V_FAST2TOPOFF_THRES*/
   2500000,/*BATTMP_MINUS_40C*/
   1469409,/*BATTMP_0C*/
   520042,/*BATTMP_45C*/
   5000000,/*MAX_VBAT_LI*/
   5500000,/*MAX_VBAT_NI*/
   3400000,/*V_PRE2FAST_THRES*/
   250000,/*I_TOPOFF2FAST_THRES*/  /*250ma,TOPOFF->FAST*/
   120000,/*I_TOPOFF2FULL_THRES*/   /*120ma,TOPOFF->BATFULL*/
   4110000,/*V_FULL2FAST_THRES*/  /*BATFULL->FAST*/
   414557,/*V_TEMP_FAST2FULL_THRES_NI*/   /*50oC,FAST->BATFULL*/
   4050000,/*V_FULL2FAST_THRES_NI*/
   600000,/*FAST_ICHARGE_HIGHLEVEL*/  /*600ma,for table search*/
   400000,/*FAST_ICHARGE_LOWLEVEL*/   /*400ma,for table search*/
   4050000,/*V_PROTECT_HIGH_LI*/
   3800000,/*V_PROTECT_LOW_LI*/
   6500000,/*VCHARGER_HIGH*/
   0,/*VCHARGER_LOW*/
/*Time delay*/
/* PRE CHARGE ,search table*/
/*TON = 3s,TOFF=2s*/
   3,/*PRE_TON*/
   2,/*PRE_TOFF*/

/* FAST CHARGE ,search table*/
/*TON = 3s,TOFF=0s*/
   3,/*TOPOFF_TON*/
   0,/*TOPOFF_TOFF*/

   6,/*BATFULL_TON_LI*/   /*unit : second*/
   0,/*BATFULL_TOFF_LI*/
   1,/*BATFULL_TON_NI*/   /*unit : second*/
   9,/*BATFULL_TOFF_NI*/
   6,/*BATFULL_TOFF*/
   10,/*BATHOLD_OFF*/

   2.5,/*ADC_ISENSE_RESISTANCE_FACTOR*/        /*1/0.4*/
   24, /*bmt_measure_discard_time*/ /*24 ticks*/
…
};
```

16.6.2 辅助设备检测设置

AUX，即辅助设备模块，是一个独立的 Task，主要对辅助设备的电压进行检测加以区分处理。与

EINT 模块联系紧密，一般在中断响应函数中对检测到的辅助设备做相应的处理。包含的功能有：

- 插入和拔出设备检测。如检测耳机，UART 的插入和拔出。
- Send 按键检测。
- 工程模式中的获得 ADC 值。

AUX 设置的相关文件路径：mcu\\custom\drv\misc_drv\ProjectName\Auxmain.c。

```
#define UART_ADC                    9999999
#define NORMAL_EARPHONE_ADC_HIGH     2800000
#define NORMAL_EARPHONE_ADC_LOW       500000/*0.5~2.8*/
#define SINGLE_EARPHONE_ADC_HIGH     2800000
#define SINGLE_EARPHONE_ADC_LOW       500000/*0.5~2.8*/
#define SENDKEY_ADC                  300000
#define POLLING_INTERVAL            50 /*polling interval*/
#define TURN_ON_BIAS_INTERVAL        10 /*measure adc interval*/
#define PLUGIN_DEBOUNCE_TIME        100 /*unit=10ms*/
#define PLUGOUT_DEBOUNCE_TIME       100
#define VOLTAGE_MEASURE_TIME        450
```

16.6.3 USB 设置

USB 模块主要对插入 USB 后的存储功能进行设置，其软件架构如下图 16.27 所示。

图 16.27　USB 模式框架图

当有 USB 插入时，EINT 先通知 USB 模块，再通过 MMI 模块选择 Mass Storage 存储模式，则进入 USB 存储模式，这时手机就可当优盘使用了。

USB 模块相关设置有如下几种。

1. 设备描述参数

可将厂商和产品的描述写入 USB 参数，当插入 USB 后，在 PC 端可读取这些信息。

文件路径：mcu\\custom\drv\misc_drv\ProjectName\Usb_custom.c。

```
static const kal_uint16 USB_MANUFACTURER_STRING[] =
{
  0x0316,
  'M',
  'y',
  ' ',
  'C',
  'o',
  'm',
  'p'
  'a'
  'n'
  'y'
};
static const kal_uint16 USB_PRODUCT_STRING[] =
{
  0x030e,
  'V',
  'T',
  '6',
  '2',
  '2',
  '6',
};
```

说　明

数组的第一个 16bit 中 03 后面的数字表示字符串长度。

长度=（字符串个数+1）×2。

加 1 用于表明字符串最后的空字符，乘 2 表示 Unicode。

例如，USB_MANUFACTURER_STRING 的字符串“My Company”的长度为（10+1）×2＝22＝0×16。

2. GPIO 配置

配置 USB 使能控制的 GPIO 或 GPO 管脚，gpio_usb_enable_pin 应该在 DrvTool 中配置好。

文件路径：mcu\\custom\drv\misc_drv\ProjectName\usb_custom.c。

```
void USB_PowerControl(kal_bool enable)
{
#ifdef __USB_ENABLE__
 #ifdef MT6318
      pmic_usb_regulator_enable(enable);
 #else
      #ifdef __CUST_NEW__
      if (enable == KAL_TRUE)
            GPO_WriteIO(1,gpio_usb_enable_pin);
      else
            GPO_WriteIO(0,gpio_usb_enable_pin);
            #else /* __CUST_NEW__ */
      if (enable == KAL_TRUE)
            GPO_WriteIO(1,2);
      else
            GPO_WriteIO(0,2);
            #endif /* __CUST_NEW__ */
 #endif    /* MT6318 */
#endif   /*__USB_ENABLE__*/
```

3. Mass Storage 存储模式设置

可设置 MSDC、NOR、NAND 等 Mass Storage 存储方式。相关的参数分别为：USB_MSDC_drv、USB_NOR_drv 和 USB_NAND_drv。

文件路径：mcu\\custom\drv\misc_drv\ProjectName\custom_drv_init.c。

```
void custom_drv_init(void)
{
   GPIO_init(); /* configure GPIO for debugging */
   spi_ini(); /* For LCD module */
   LCD_FunConfig();
   Alter_init();
#ifndef __L1_STANDALONE__
   PWM_initialize();
#endif

#ifdef __USB_ENABLE__
   #if ( (defined(__MSDC_MS__))||(defined(__MSDC_MSPRO__))||(defined(__MSDC_SD_MMC__)) )
      USB_Ms_Register_DiskDriver(&USB_MSDC_drv);
   #endif
   #ifdef __USB_RAMDISK__
      USB_Ms_Register_DiskDriver(&USB_RAM_drv);
   #endif  /*__USB_RAMDISK__*/
   USB_Ms_Register_DiskDriver(&USB_NOR_drv);
   #ifdef NAND_SUPPORT
   USB_Ms_Register_DiskDriver(&USB_NAND_drv);
   #endif
#endif  /*__USB_ENABLE__*/
}
```

16.7 其他设备的配置

16.7.1 MSDC 设置

MSDC（Mass Storage Device Controller，存储设备控制器），用于对存储卡的管理和控制。

MSDC 模块的框架如图 16.28 所示。

MSDC 的相关配置如下。

1. 配置热插拔特性

文件路径：mcu\make\ProjectName.mak。

通过定义__MSDC_NOT_SUPPORT_HOT_PLUG__，决定是否可以热插拔。

2. 配置是否写保护

文件路径：mcu\make\ProjectName.mak。

通过定义__MSDC_NO_WRITE_PROTECT__，决定是否写保护。

3. 配置驱动电流

文件路径：mcu\custom\drv\misc_drv\[PRJ]\msdc_custom.c。

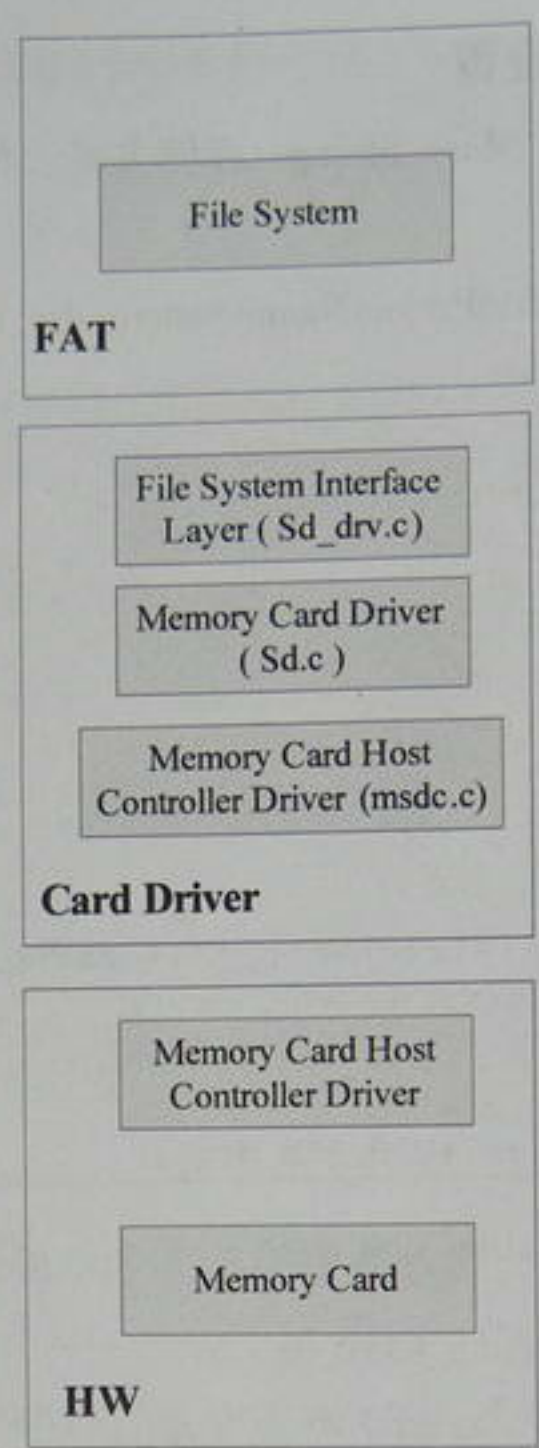


图 16.28　MSDC 框架图

```
// it only apply to the SD card io control
static const msdc_io_ctrl_struct msdc_io_ctrl_sd_custom =
{
  MSDC_ODC_8MA, // driving capacity of CMD/BS and SCLK
  MSDC_ODC_2MA, // driving capacity of data lines
  MSDC_ODC_SLEW_FAST, // slew rate of CMD/BS and SCLK
  MSDC_ODC_SLEW_FAST // slew rate of data lines
};
```

16.7.2　触摸屏设置

触摸屏设置的步骤如下。

1．定义触摸屏的 EINT

触摸屏事件也是 EINT 的一种，所以先要用 DrvTool 配置好 EINT。

文件路径：mcu\custom\drv\misc_drv\ProjectName\eint_def.c。

```
#if defined(TOUCH_PANEL_SUPPORT) || defined(HAND_WRITING)
extern const kal_uint8 TOUCH_PANEL_EINT_NO;
#endif   /* TOUCH_PANEL_SUPPORT || HAND_WRITING */
```

2．定义触摸屏相关的 GPIO

触摸屏需要使用到 GPIO 端口，所以先要用 DrvTool 配置好 GPIO，然后用于触摸屏配置的相关

文件。

文件路径：mcu\custom\drv\misc_drv\ProjectName\ touch_panel_spi.h。

```
#ifdef __CUST_NEW__
   extern const char gpio_tp_spi_din_pin;
   extern const char gpio_tp_spi_clk_pin;
   extern const char gpio_tp_spi_dout_pin;
   extern const char gpio_tp_spi_cs_pin;
   extern const char gpio_tp_spi_busy_pin;

   #define SPI_DIN_PIN     gpio_tp_spi_din_pin  /*GPO*/
   #define SPI_CLK_PIN     gpio_tp_spi_clk_pin /*GPO*/
   #define SPI_DOUT_PIN    gpio_tp_spi_dout_pin /*GPI*/
   #define SPI_CS_PIN      gpio_tp_spi_cs_pin /*GPO*/
   #define SPI_BUSY_PIN    gpio_tp_spi_busy_pin /*GPI*/
#else /* __CUST_NEW__ */
   #define SPI_DIN_PIN     40  /*GPO*/
   #define SPI_CLK_PIN     16 /*GPO*/
   #define SPI_DOUT_PIN    18 /*GPI*/
   #define SPI_CS_PIN      19 /*GPO*/
   #define SPI_BUSY_PIN    17 /*GPI*/
#endif /* __CUST_NEW__ */
```

3. 实现相关接口，基本操作

文件路径：mcu\custom\drv\misc_drv\ProjectName\ touch_panel_spi.c。

```
#define SET_CLK_HIGH()          (GPIO_WriteIO(1,SPI_CLK_PIN))
#define SET_CLK_LOW()           (GPIO_WriteIO(0,SPI_CLK_PIN))
#define SET_DATA_HIGH()         GPIO_WriteIO(1,SPI_DIN_PIN)
#define SET_DATA_LOW()          GPIO_WriteIO(0,SPI_DIN_PIN)
#define GET_DATA_BIT()          GPIO_ReadIO(SPI_DOUT_PIN)
#define GET_BUSY_BIT()          GPIO_ReadIO(SPI_BUSY_PIN)

void serial_write_bit_high(void);
void serial_write_bit_low(void);
void serial_delay(void);
void serial_write_data(kal_uint8 data);
kal_uint16 serial_read_data(void);
void serial_init(void);
```

4. 设置 ADC 和坐标校准

相关文件路径：mcu\custom\drv\misc_drv\ProjectName\touch_panel_custom.c。

```
TouchPanel_custom_data_struct  tp_custom_data_def =
{
 /*ADC*/
 TOUCH_PANEL_ADC_X_START,
 TOUCH_PANEL_ADC_X_END,
 TOUCH_PANEL_ADC_Y_START,
 TOUCH_PANEL_ADC_Y_END,
 /*Coord.*/
 TOUCH_PANEL_COORD_X_START,
 TOUCH_PANEL_COORD_X_END,
```

```
    TOUCH_PANEL_COORD_Y_START,
    TOUCH_PANEL_COORD_Y_END,
    /*eint level*/
    TOUCH_PANEL_EINT_DOWN_LEVEL
   };
   void tp_read_adc(kal_int16 *x, kal_int16 *y)
   {
       kal_uint8 command=0;
       /*YADC*/
       command=TOUCH_PANEL_START_BIT|TOUCH_PANEL_X_PLUS| TOUCH_PANEL_12BIT_SAMPLE| TOUCH_PANEL_
DIFF_MODE;
       serial_write_data(command);
       *y=serial_read_data();
       /*X ADC*/
       command=TOUCH_PANEL_START_BIT|TOUCH_PANEL_Y_PLUS| TOUCH_PANEL_12BIT_SAMPLE|TOUCH_ PANEL_
DIFF_MODE;
       serial_write_data(command);
       *x=serial_read_data();
   }
```

5. 修改相关坐标

有时不同的触摸屏分辨率不同，刷新顺序不同，要设置好相关参数。

文件路径：mcu\custom\drv\misc_drv\ProjectName\touch_panel_custom.h。

```
#define TOUCH_PANEL_COORD_X_START 0
#define TOUCH_PANEL_COORD_X_END   175
#define TOUCH_PANEL_COORD_Y_START 0
#define TOUCH_PANEL_COORD_Y_END   219
```

16.7.3 LCD 屏幕移植

LCD 负责数据的显示，一般来说都有一个主屏幕，还有可选的次屏幕。BB 与 LCD 的接口分为并行接口和串行接口，针对这两种接口 BB 内部有并行加速器和串行加速器，另外还有 LCD DMA 控制器，专门用于 LCD 数据的传输，如图 16.29 所示。

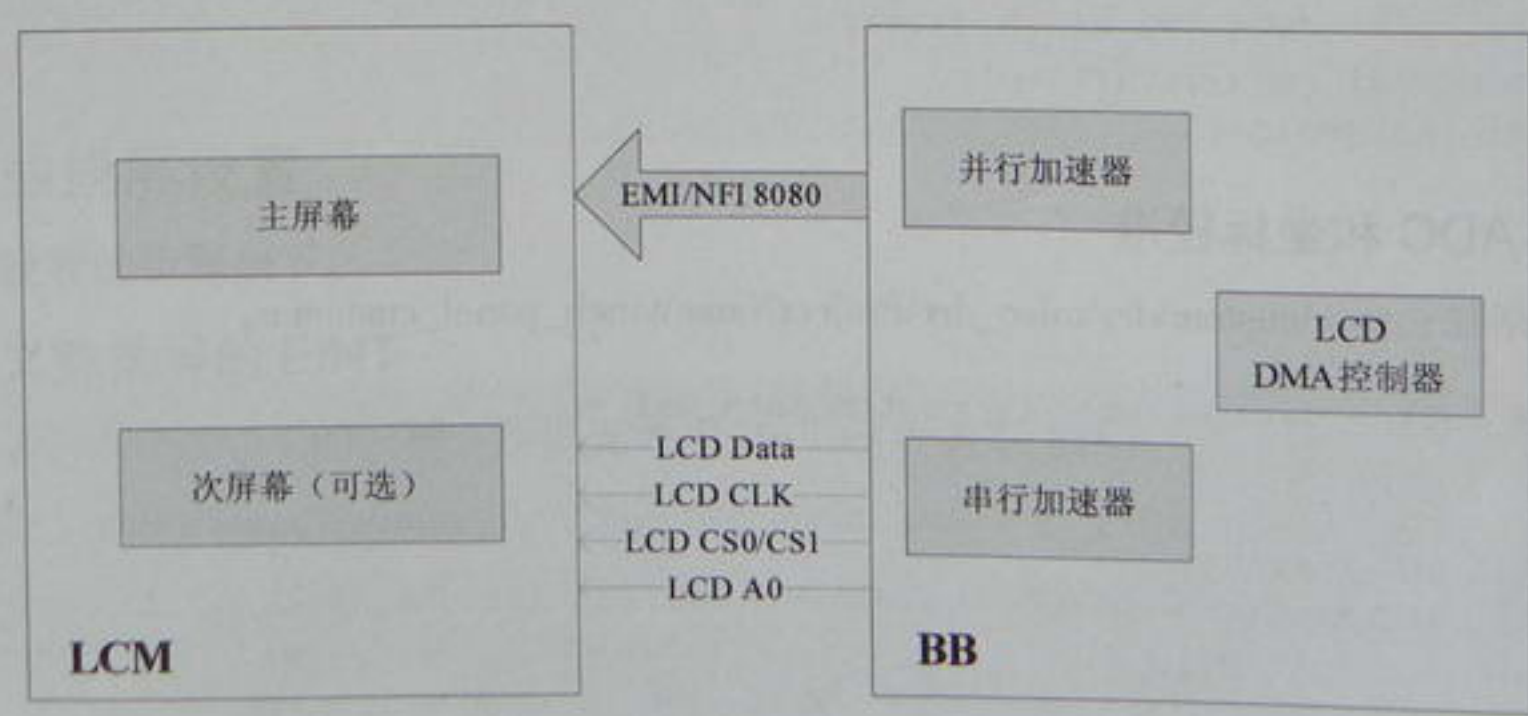


图 16.29 LCM 与 BB 交互图

图 16.30 列举了 8080 系列并行接口数据传输的时序图。

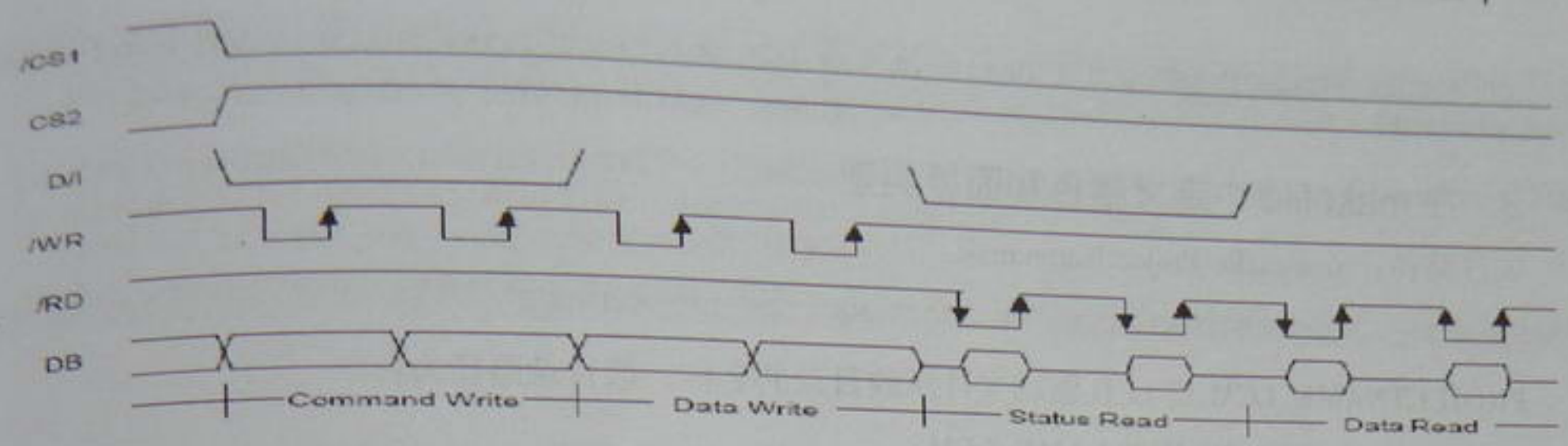


图 16.30　并行传输时序图

以 80 系统 18bit 总线并行接口为例，MPU 和 LCM 之间的连接如下图 16.31 所示。

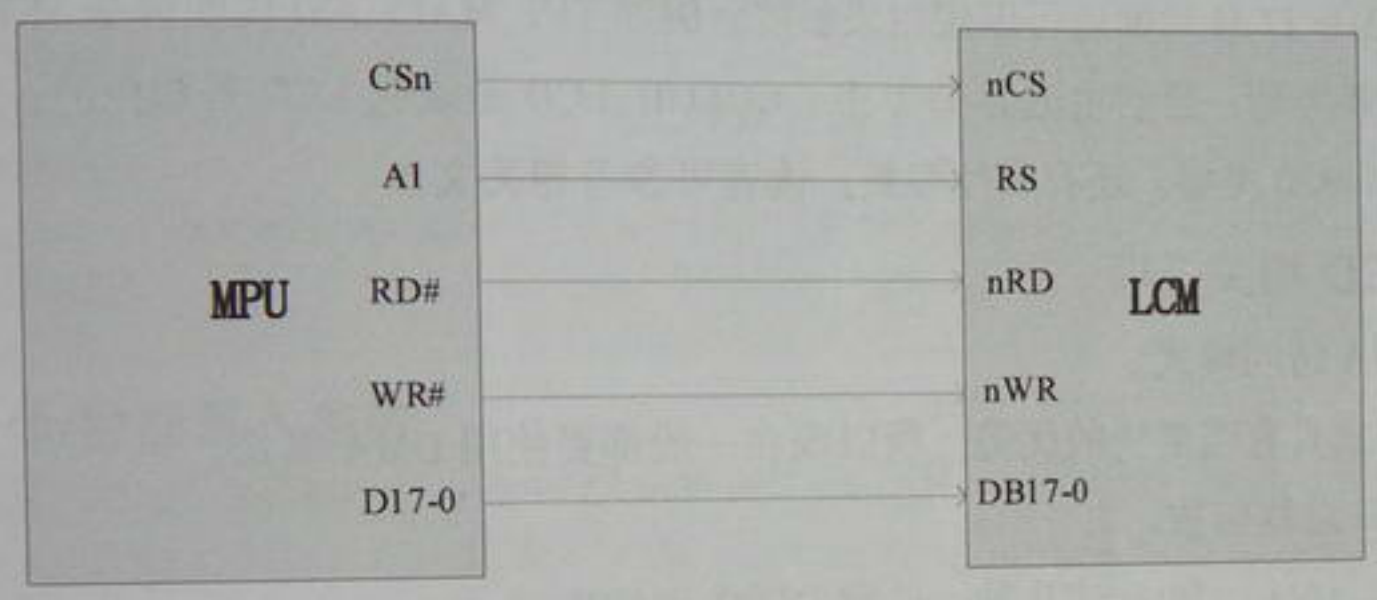


图 16.31　MPU 与 LCM 连接

LCD 移植步骤如下。

1. 定义 LCD 图像大小

文件路径：mcu\make\ProjectName.mak。

```
MAIN_LCD_SIZE = 240 × 320
```

文件路径：mcu\custom\drv\lcd\ProjectName_lcm\lcd_sw_inc.h。

根据 makefile 所定义的大小定义 LCD 的长和宽。

```
#ifdef __MMI_MAINLCD_128X128__
#define LCD_WIDTH   (128)
#define LCD_HEIGHT  (128)
#elif (defined(__MMI_MAINLCD_128X160__))
#define LCD_WIDTH   (128)
#define LCD_HEIGHT  (160)
#elif (defined(__MMI_MAINLCD_176X220__))
#define LCD_WIDTH   (176)
#define LCD_HEIGHT  (220)
#elif (defined(__MMI_MAINLCD_240X320__))
#define LCD_WIDTH   (240)
#define LCD_HEIGHT  (320)
#endif
```

2. 定义 LCD 模块名

文件路径：mcu\make\ProjectName.mak。

```
LCD_MODULE=PROJECTNAME_LCM # Based on the LCM solutions (even multiple LCM modules for this project)
```

3. 在 makefile 中定义颜色和面板类型

文件路径：mcu\make\ProjectName.mak。

```
COM_DEFS_FOR_PROJECTNAME_LCM= PROJECTNAME _LCM TFT_MAINLCD
```

PROJECTNAME_LCM 是 LCD 驱动文件所放目录的名称，该目录路径为：

mcu\custom\drv\lcd\PROJECTNAME_LCM。

目录下有 5 个文件：lcd.c、lcd_hw.h、lcd_sw.h、lcd_sw_inc.h、lcd_sw_rnd.h。如果没有则需要从原始工程中将这 5 个文件复制过来。

PROJECTNAME_LCM 后面还可以接相关参数。例如 TFT_MAINLCD 说明屏是 TFT 类型的，还可以是 CSTN、OLED 等类型，目前主流为 TFT 型；COLOR_LCD 是说明 LCD 是彩色的；QVGA_MAINLCD 说明分辨率为 240×320 类型。还有其他参数，读者可参考相关文档。

4. 定义 LCD 相关选项

（1）使用 DMA 访问模式。

使用 DMA 模式具有速度快的优势，所以现在一般都要使用 DMA 模式。

（2）定义 LCD 总线位数。

有 8bit、9bit、16bit、18bit 这几种，位数主要由硬件决定。

（3）定义 LCD 输出格式。

可输出的格式详见 mcu\custom\interface\hwdrv\Lcd_if.h 文件。

例如 LCM_16BIT_16_BPP_RGB565_1 表示总线位数为 16 位（与定义的总线位数一致），每个像素占 16 位，RGB 顺序为 RRRRRGGGGGGBBBBB。

（4）设置主 LCD 命令/数据地址。

根据所选 LCD 的 datasheet 中的 RS 和 RW 寄存器的说明来设置 LCD 的命令/数据地址标识，某 LCD 参数如下表 16.9 所示。

表 16.9　LCD 寄存器 RS/RW 参数表

RS	RW	Function
0	0	Set index register
0	1	Read status
1	0	Write instruction or RAM data
1	1	Read instruction or RAM data

可以看出 RS 为 0 代表对命令的操作，RS 为 1 代表对数据的操作。所以 LCD_PARALLEL0_A0_LOW_ADDR 表示命令地址，LCD_PARALLEL0_A0_HIGH_ADDR 表示数据地址。

（5）设置次 LCD 命令/数据地址。

方法同主 LCD 的设置。其余改动为将 XXX_PARALLEL0_XXX 改为 XXX_PARALLEL1_XXX，表示

次 LCD。

文件路径：mcu\custom\drv\lcd\ProjectName_lcm\lcd_sw.h。

```
#if (defined(MT6218B) || defined(MT6219) || defined(MT6217) ||defined(MT6226) ||defined(MT6226M)
||defined(MT6227)||defined(MT6227D)||defined(MT6226D))
  #define LCD_CMD_DMA_MODE //使用 DMA 访问模式
  #define LCD_16BIT_MODE //定义 LCD 总线位数

  #define LCD_DUMMYADDR                          0x90000000

  #define LCD_HX8306A_CTRL_ADDR        LCD_PARALLEL0_A0_LOW_ADDR
// LCD HX8306A 设置命令地址
  #define LCD_HX8306A_DATA_ADDR        LCD_PARALLEL0_A0_HIGH_ADDR
//LCD HX8306A 设置数据地址
  #define MAIN_LCD_CMD_ADDR            LCD_HX8306A_CTRL_ADDR
//主 LCD 设置命令地址
  #define MAIN_LCD_DATA_ADDR           LCD_HX8306A_DATA_ADDR
//主 LCD 设置数据地址
  #define MAIN_LCD_OUTPUT_FORMAT    LCM_16BIT_16_BPP_RGB565_1
//定义 LCD 输出格式
#endif
```

5. 实现命令/数据写入函数

```
LCD_CtrlWrite_ XXXX(_data)     //命令写入函数
LCD_DataWrite_ XXXX(_data)    //数据写入函数
LCD_RAMWrite_XXXX (_data)  //RAM 写入函数（可选）
```

下面以两个例子来说明函数的实现。

（1）例子 1：8bit 总线接口，DMA 模式，传输 16bit/pixel 的数据。

80 系统 8-bit 总线/串行数据传输，一般一条命令或数据分 2 或 3 次传输，如图 16.32 所示。

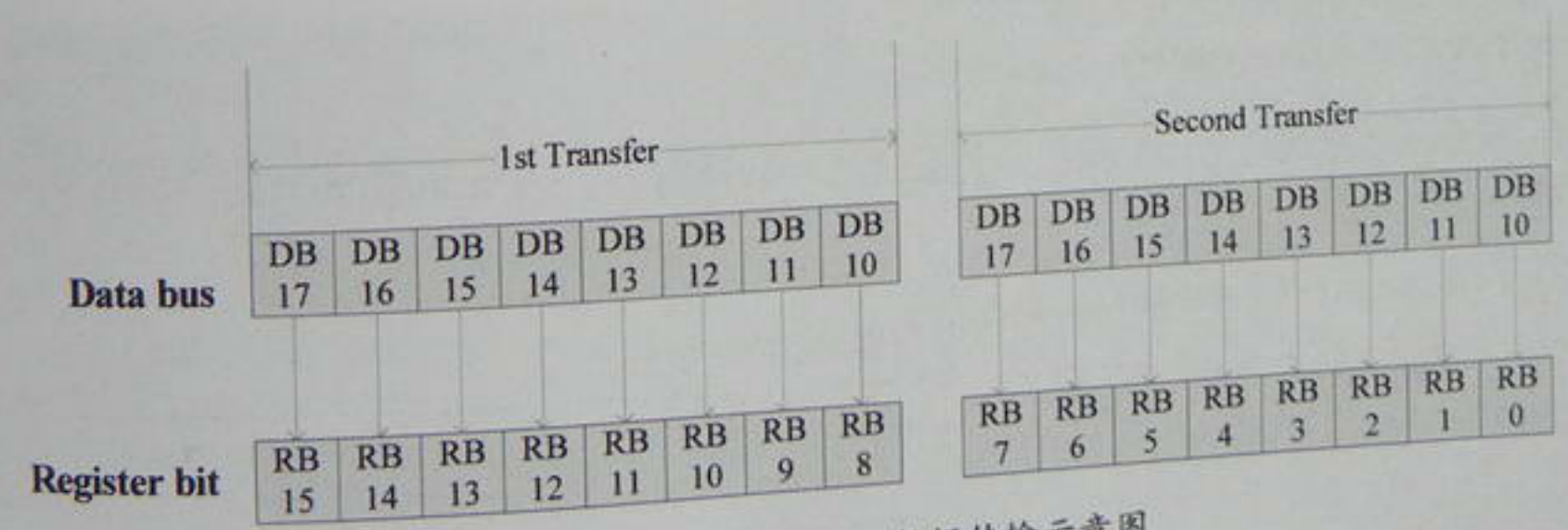


图 16.32　8bit 总线接口数据传输示意图

命令/数据为 16 位存在寄存器中。第 1 次传输命令/数据先取高 8 位，寄存器右移 8 位，再传入总线中 DB17～DB10。第 2 次传输命令/数据取后 8 位，再传入总线中 DB17～DB10。SET_LCD_CMD_PARAMETER 宏第 1 个参数表示在本次 DMA 传输中的序号；第 2 个参数表示是传输命令还是传输数据，LCD_CMD 为传输命令类型，LCD_DATA 为传输数据类型；第 3 个参数表示传输的参数值。LCD_SEND_DMA_CMD(X) 表示本次传输了 X 次命令/数据。

文件路径：mcu\custom\drv\lcd\ProjectName_lcm\lcd_sw.h。

```
#if (defined(LCD_8BIT_MODE))
#define LCD_CtrlWrite_HX8306A(_data) \
{\
 SET_LCD_CMD_PARAMETER(0,LCD_CMD,((_data & 0xFF00)>>8));\
 SET_LCD_CMD_PARAMETER(1,LCD_CMD,(_data & 0xFF));\
 LCD_SEND_DMA_CMD(2);\
}
#define LCD_DataWrite_HX8306A(_data) \
{\
 SET_LCD_CMD_PARAMETER(0,LCD_ DATA,((_data & 0xFF00)>>8));\
 SET_LCD_CMD_PARAMETER(1,LCD_ DATA,(_data & 0xFF));\
 LCD_SEND_DMA_CMD(2);\
}
```

（2）例子 2：18bit 总线接口，DMA 模式，传输 16bit/pixel 的数据。

80 系统 18-bit 总线，一次传输一条命令或数据，如图 16.33 所示。

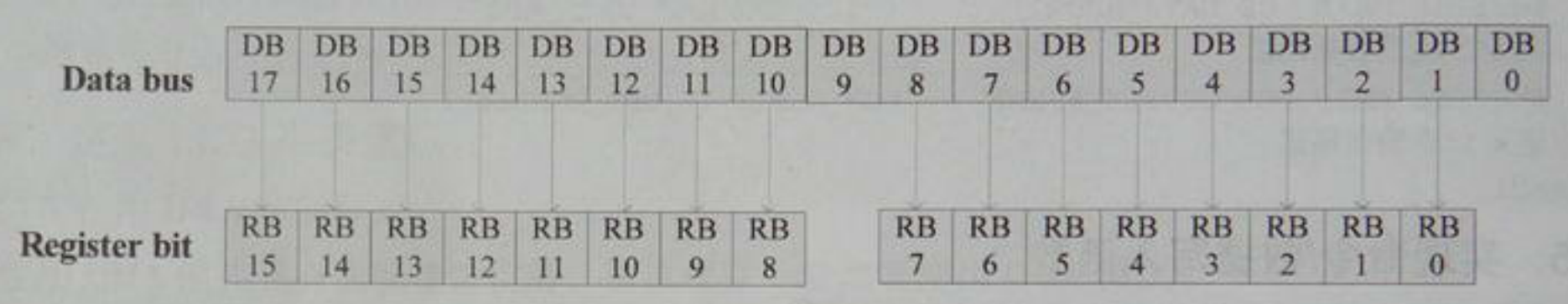


图 16.33　18Bit 总线接口数据传输示意图

命令/数据为 16 位存在寄存器中。由于是 18bit 总线，所以命令/数据高 8 位要左移 2 位，与 DB17～DB10 匹配。命令/数据低 8 位要左移 1 位，要与 DB8～DB1 匹配。整合好 18 位命令/数据后，再传到数据总线 DB17～DB0 中。

文件路径：mcu\custom\drv\lcd\ProjectName_lcm\lcd_sw.h。

```
#elif (defined(LCD_18BIT_MODE))
#define LCD_CtrlWrite_HX8306A(_data) \
{\
SET_LCD_CMD_PARAMETER(0,LCD_CMD,(((_data & 0xFF00)<<2)|(_data&0xFF)<<1 ));\
 LCD_SEND_DMA_CMD(1);\
}
#define LCD_DataWrite_HX8306A(_data) \
{\
SET_LCD_CMD_PARAMETER(0,LCD_DATA,(((_data & 0xFF00)<<2)|(_data&0xFF)<<1 ));\
 LCD_SEND_DMA_CMD(1);\
}
#endif
```

6. 定义主/次 LCD 的函数变量

LCD_Funcs 的结构定义了在普通模式和工程模式下 LCD 的一些基本操作。

文件路径：mcu\interface\hwdrv\Lcd_if.h。

```
typedef struct
{
    void (* Init)(kal_uint32 background, void **buf_addr);// LCD initialize function
    void (* PWRON)(kal_bool on);// LCD power on/off function
```

```
    void (* BrightLevel)(kal_uint8 level);// Adjust LCD brightness
    void (* SCREENON)(kal_bool on);// LCD screen on
    void (* BlockWrite)(kal_uint16 startx,kal_uint16 starty,kal_uint16 endx,kal_uint16
endy);
                        // Block data write to LCD driver IC
    void (* GetSize)(kal_uint16 *out_LCD_width,kal_uint16 *out_LCD_height);
  // Obtain the size of the LCD panel
    void (* SleepIn)(void);// LCD enter sleep mode
    void (* SleepOut)(void);// LCD exit sleep mode
    void (* PartialOn) (kal_uint16 start_page,kal_uint16 end_page);
    void (* PartialOff) (void);
  kal_uint8 (*partial_display_align) (void);
    /*Engineering mode*/
    kal_uint8 (* get_param_number)(lcd_func_type type);
  // Obtain the parameter number of the function
    void (* set_bias)(kal_uint8 *bias);// Configure LCD bais
    void (* set_contrast)(kal_uint8 *contrast); // Configure LCD constrast
    void (* set_linerate)(kal_uint8 *linerate);// Configure LCD frame-refresh rate
    void (* set_temp_compensate)(kal_uint8 *compensate);
  // Configure LCD temperature compensation parameter
  #ifdef __LCD_ESD_RECOVERY__
    kal_bool (* esd_check)(void);
  #endif
  #ifdef LCM_ROTATE_SUPPORT
   void (* set_rotate)(kal_uint8 rotate_value);
  #endif
  }LCD_Funcs;
```

通过定义 LCD 的函数变量，上层就可以调用 LCD 驱动相关函数了。一般来说，其中常用的有 Init、SleepIn、SleepOut、BlockWrite 函数。

文件路径：mcu\custom\drv\Lcd\ProjectName_lcm\lcd.c。

```
LCD_Funcs LCD_func_HX8306A = {
 LCD_Init_HX8306A,
 LCD_PWRON_HX8306A,
 LCD_SetContrast_HX8306A,
 LCD_ON_HX8306A,
 LCD_BlockWrite_HX8306A,
 LCD_Size_HX8306A,
 LCD_EnterSleep_HX8306A,
 LCD_ExitSleep_HX8306A,
 LCD_Partial_On_HX8306A,
 LCD_Partial_Off_HX8306A,
 LCD_Partial_line_HX8306A,
 /*Engineering mode*/
 LCD_GetParm_HX8306A,
 LCD_SetBias_HX8306A,
 LCD_Contrast_HX8306A,
 LCD_LineRate_HX8306A,
 LCD_Temp_Compensate_HX8306A
 #ifdef __LCD_ESD_RECOVERY_
 ,LCD_ESD_check_HX8306A
 #endif
};
```

7. 配置 LCD 函数变量

文件路径：mcu\custom\drv\lcd\ProjectName_lcm\lcd.c。

```
void LCD_FunConfig(void)
{
    MainLCD = &LCD_func_HX8306A;
}
```

> **提 示**
>
> 如果有次 LCD，也要声明次 LCD 函数变量，并配置。

8. 设置 LCD 数据总线类型

这个值要与上面设置的总线位数一致。可设置的总线位数与芯片类型相关，例如 MT6219 可设置 8bit，MT6217 可设置 8bit/16bit，MT6226 及以上版本可设置 8bit/9bit/16bit/18bit。

文件路径：mcu\custom\drv\lcd\ProjectName_lcm\lcd.c。

```
void init_lcd_interface(void)
{
…
#if (defined(MT6226) || defined(MT6226M) || defined(MT6227) || defined(MT6227D) || defined
(MT6226D))
#if (defined(LCD_16BIT_MODE))
SET_LCD_PARALLEL_16BIT_DATA_BUS
…
#endif
#endif
…
}
```

9. 设置 LCD 时间特性

文件路径：mcu\custom\drv\lcd\ProjectName_lcm\lcd.c。

```
void init_lcd_interface(void)
{
…
SET_LCD_PARALLEL_CE2WR_SETUP_TIME((kal_uint32)1);
SET_LCD_PARALLEL_CE2WR_HOLD_TIME(1);
SET_LCD_PARALLEL_CE2RD_SETUP_TIME(1);
SET_LCD_PARALLEL_WRITE_WAIT_STATE(2);
SET_LCD_PARALLEL_READ_LATENCY_TIME(0);
SET_LCD_ROI_CTRL_CMD_LATENCY(2);
…
}
```

SET_LCD_PARALLEL_CE2WR_SETUP_TIME：设置 C2WS 参数。

SET_LCD_PARALLEL_CE2WR_HOLD_TIME：设置 C2WH 参数。

SET_LCD_PARALLEL_CE2RD_SETUP_TIME：设置 C2RS 参数。

SET_LCD_PARALLEL_WRITE_WAIT_STATE：设置 WST 参数。

SET_LCD_PARALLEL_READ_LATENCY_TIME：设置 RLT 参数。

SET_LCD_ROI_CTRL_CMD_LATENCY：设置通过 DMA 发送 command 的时候，两条命令之间的间隔时间。

根据 LCD datasheet 上的时间参数说明和时序特性来计算这些参数，获得单位为纳秒的值。再根据是否设置 SET_LCD_PARALLEL_CLOCK_X 参数，计算出每个 Ticket 的时间。例如 SET_LCD_PARALLEL_CLOCK_52M，说明每个 Ticket 为 1/52M，约为 20ns。最后计算出需要多少个 Ticket，即为最终设置的参数。

设置的过程类似于 EMI 时间设置，可参考进行。

10. 实现基本显示函数

一般所选用的 LCD 的模组厂商会提供 LCD 的初始化代码、进入睡眠模式代码、退出睡眠模式代码等。

文件路径：mcu\custom\drv\lcd\ProjectName_lcm\lcd.c。

LCD_Init_XXXX()：LCD 初始化。

LCD_EnterSleep_ XXXX ()：LCD 进入睡眠模式。

LCD_ExitSleep_XXXX ()：LCD 退出睡眠模式。

下面以 LCD_Init_XXXX()函数的修改为例来进行说明，睡眠模式代码修改类似。

LCD 模组厂商给的初始化代码如下：

```
#if 1 //cmo2.8+hx8347
// Gamma for CMO 2.8"
WMLCDCOM(0x0046);WMLCDDATA(0x0091);
WMLCDCOM(0x0047);WMLCDDATA(0x0011);
WMLCDCOM(0x0048);WMLCDDATA(0x0000);
WMLCDCOM(0x0049);WMLCDDATA(0x0066);
WMLCDCOM(0x004A);WMLCDDATA(0x0037);
WMLCDCOM(0x004B);WMLCDDATA(0x0004);
WMLCDCOM(0x004C);WMLCDDATA(0x0011);
WMLCDCOM(0x004D);WMLCDDATA(0x0077);
WMLCDCOM(0x004E);WMLCDDATA(0x0000);
WMLCDCOM(0x004F);WMLCDDATA(0x001F);
WMLCDCOM(0x0050);WMLCDDATA(0x000F);
WMLCDCOM(0x0051);WMLCDDATA(0x0000);
//240x320 window setting
WMLCDCOM(0x0002);WMLCDDATA(0x0000); // Column address start2
WMLCDCOM(0x0003);WMLCDDATA(0x0000); // Column address start1
WMLCDCOM(0x0004);WMLCDDATA(0x0000); // Column address end2
WMLCDCOM(0x0005);WMLCDDATA(0x00EF); // Column address end1
WMLCDCOM(0x0006);WMLCDDATA(0x0000); // Row address start2
WMLCDCOM(0x0007);WMLCDDATA(0x0000); // Row address start1
WMLCDCOM(0x0008);WMLCDDATA(0x0001); // Row address end2
WMLCDCOM(0x0009);WMLCDDATA(0x003F); // Row address end1
…
// Display ON Setting
WMLCDCOM(0x0090);WMLCDDATA(0x007F); // SAP=0111 1111
WMLCDCOM(0x0026);WMLCDDATA(0x0004); //GON=0, DTE=0, D=01
Delayms(50);
WMLCDCOM(0x0026);WMLCDDATA(0x0024); //GON=1, DTE=0, D=01
Delayms(50);
```

```
WMLCDCOM(0x0026);WMLCDDATA(0x002C); //GON=1, DTE=0, D=11
Delayms(50);
WMLCDCOM(0x0026);WMLCDDATA(0x003C); //GON=1, DTE=1, D=11
Delayms(50);
WMLCDCOM(0x0022);                   //WRITE DATA TO GRAM
#endif
```

其中 WMLCDCOM 设置写入数据的寄存器，WMLCDDATA 设置写入的数据值。所以在 LCD_Init_XXXX 函数中的修改如下：

```
void LCD_Init_HX8306A(kal_uint32 bkground, void **buf_addr)
{
// Gamma for CMO 2.8"
LCD_set_reg(0x0046,0x0091);
LCD_set_reg(0x0047,0x0011);
LCD_set_reg(0x0048,0x0000);
LCD_set_reg(0x0049,0x0066);
LCD_set_reg(0x004A,0x0037);
LCD_set_reg(0x004B,0x0004);
LCD_set_reg(0x004C,0x0011);
LCD_set_reg(0x004D,0x0077);
LCD_set_reg(0x004E,0x0000);
LCD_set_reg(0x004F,0x001F);
LCD_set_reg(0x0050,0x000F);
LCD_set_reg(0x0051,0x0000);
//240×320 window setting
LCD_set_reg(0x0002,0x0000);// Column address start2
LCD_set_reg(0x0003,0x0000);// Column address start1
LCD_set_reg(0x0004,0x0000);// Column address end2
LCD_set_reg(0x0005,0x00EF);// Column address end1
LCD_set_reg(0x0006,0x0000);// Row address start2
LCD_set_reg(0x0007,0x0000);// Row address start1
LCD_set_reg(0x0008,0x0001);// Row address end2
LCD_set_reg(0x0009,0x003F);// Row address end1
…
// Display ON Setting
LCD_set_reg(0x0090,0x007F); // SAP=0111 1111
LCD_set_reg(0x0026,0x0004); //GON=0, DTE=0, D=01
Delayms(50);
LCD_set_reg(0x0026,0x0024); //GON=1, DTE=0, D=01
Delayms(50);
LCD_set_reg(0x0026,0x002C); //GON=1, DTE=0, D=11
Delayms(50);
LCD_set_reg(0x0026,0x003C); //GON=1, DTE=1, D=11
Delayms(50);
//LCD_CtrlWrite_HX8306A(0x0022);    //WRITE DATA TO GRAM  Delete Here!
}
```

其中 LCD_set_reg 为设置寄存器的值，Delayms 为延迟时间。

注 意

一般初始化函数中不含 LCD 厂商提供驱动代码的最后一行，即写数据到 GRAM 中，因为在 LCD_BlockWrite_XXXX 函数实现中才写入到 GRAM 中。

LCD_BlockWrite_XXXX(kal_uint16 startx, kal_uint16 starty, kal_uint16 endx, kal_uint16 endy)：用于将一块特定区域的数据显示到 LCD 上，需要扫描 LCD，扫描方式是从（startx，starty）坐标开始逐行扫描到（endx, endy）坐标。

扫描方式如图 16.34 所示。

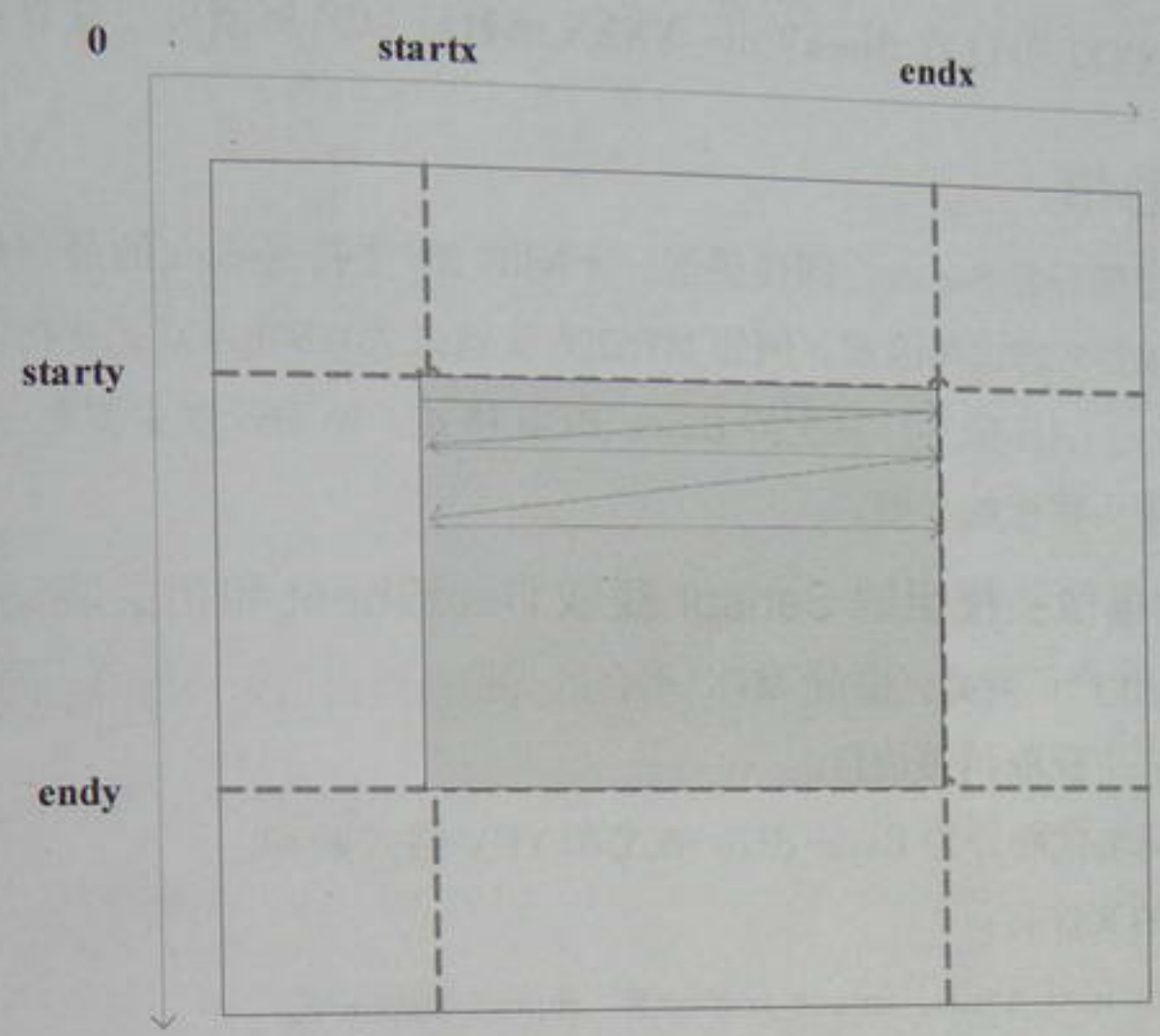


图 16.34　扫描方式

实现这个函数需要查找 LCD 的 datasheet。了解对应 startx，starty，endx，endy 的 LCD 中的寄存器，了解写入 GRAM 的寄存器。例如下面代码：0x0002 寄存器、0x0003 寄存器对应 startx；0x0022 代表 GRAM 寄存器。

```
void LCD_BlockWrite_HX8306A(kal_uint16 startx,kal_uint16 starty,kal_uint16 endx,kal_uint16
endy)
{
SET_LCD_CMD_PARAMETER(0,LCD_CMD,((0x0002&0xFF00) )|((0x0002&0xFF) ));
SET_LCD_CMD_PARAMETER(1,LCD_DATA,((startx&0xff00)>>8));
SET_LCD_CMD_PARAMETER(2,LCD_CMD,((0x0003&0xFF00) )|((0x0003&0xFF) ));
SET_LCD_CMD_PARAMETER(3,LCD_DATA,(startx&0x00ff));
SET_LCD_CMD_PARAMETER(4,LCD_CMD,((0x0004&0xFF00) )|((0x0004&0xFF) ));
SET_LCD_CMD_PARAMETER(5,LCD_DATA,((endx&0xff00)>>8));
SET_LCD_CMD_PARAMETER(6,LCD_CMD,((0x0005&0xFF00) )|((0x0005&0xFF) ));
SET_LCD_CMD_PARAMETER(7,LCD_DATA,(endx&0x00ff)); SET_LCD_CMD_PARAMETER(8,LCD_CMD,((0x0006&
0xFF00) )|((0x0006&0xFF) ));
SET_LCD_CMD_PARAMETER(9,LCD_DATA,((starty&0xff00)>>8));
SET_LCD_CMD_PARAMETER(10,LCD_CMD,((0x0007&0xFF00) )|((0x0007&0xFF) ));
SET_LCD_CMD_PARAMETER(11,LCD_DATA,(starty&0x00ff));
SET_LCD_CMD_PARAMETER(12,LCD_CMD,((0x0008&0xFF00) )|((0x0008&0xFF) ));
SET_LCD_CMD_PARAMETER(13,LCD_DATA,((endy&0xff00)>>8));
SET_LCD_CMD_PARAMETER(14,LCD_CMD,((0x0009&0xFF00) )|((0x0009&0xFF) ));
SET_LCD_CMD_PARAMETER(15,LCD_DATA,(endy&0x00ff));
SET_LCD_CMD_PARAMETER(16,LCD_CMD,((0x0022&0xFF00) )|((0x0022&0xFF) ));
```

```
    ENABLE_LCD_TRANSFER_COMPLETE_INT;     //数据传输完成，产生中断
    ENABLE_LCD_ROI_CTRL_CMD_FIRST;
    SET_LCD_ROI_CTRL_NUMBER_OF_CMD(17); //发送的控制命令总数
}
```

另外如果要改变 LCD 图像的旋转方式，可设置 LCD 相关寄存器。

实现 LCD_Init_XXXX()和 LCD_BlockWrite_XXXX 函数后 LCD 屏就可正常显示了。

16.7.4 摄像头移植

摄像头模块的关键部分是 Sensor，即传感器。不同的 BB 支持 Sensor 的最高像素和格式都不相同，所以选取摄像头的时候要考虑这些因素。例如 MT6226 支持最大 0.3M 像素，支持 YUV 和 Bayer RGB 的格式；MT6219 支持最大 1.3M 像素，只支持 Bayer RGB 格式。驱动开发主要对 Sensor 部分代码进行修改，下面讲解一下摄像头移植的步骤。

1. 根据选择的摄像头模组的 Sensor 获取 DataSheet 和相关驱动代码

现在主要的摄像头生产厂商都提供对 MTK 平台的支持并可提供驱动代码，同时我们也可以在 MTK 通过的 Sensor 支持列表中获取相关信息。

根据支持的格式驱动代码分为 Bayer RGB 格式和 YUV 格式两种。

RGB 格式的包括的文件有：

camera_para.c、usbvideo_attr.c、image_sensor.h、image_sensor.c。

YUV 格式的包括的文件有：

camera_yuv_para.c、usbvideo_attr.c、image_sensor.h、image_sensor.c。

2. 工程 makefile 中修改

文件路径：mcu\make\ProjectName.mak。

```
CMOS_SENSOR            = OV76XX    #摄像头型号
YUV_SENSOR_SUPPORT     = TRUE      #是否使用 YUV 格式
HORIZONTAL_CAMERA      = FALSE     #主要为了避免垂直显示摄像头的景物
WEBCAM_SUPPORT         = TRUE      #网络摄像头功能
```

3. 驱动代码移植

（1）Bayer RGB 格式

建立 mcu\custom\drv\image_sensor 目录，并将 image_sensor.h、image_sensor.c 文件放在目录下。将 camera_para.c, usbvideo_attr.c 文件放在 mcu\custom\drv\camera 目录下，覆盖原有文件。

（2）YUV 格式

建立 mcu\custom\drv\yuv_sensor 目录，并将 image_sensor.h、image_sensor.c 文件放在目录下。将 camera_yuv_para.c、usbvideo_attr.c 文件放在 mcu\custom\drv\camera 目录下，覆盖原有文件。

4. 摄像头开启/关闭设置

文件路径：mcu\custom\drv\camera\ProjectName\Camera_hw.c。

```
void cis_module_power_on(kal_bool on)
{
    if(on==KAL_TRUE)
    {
        sccb_setDelay(0);
        sccb_config(SCCB_SW_8BIT, SIV100B_I2C_WRITE_ID, SIV100B_I2C_READ_ID, NULL);
        GPIO_InitIO(1, MODULE_CMPDN_PIN);
        GPIO_ModeSetup(MODULE_CMPDN_PIN, 0);
        GPIO_WriteIO(0, MODULE_CMPDN_PIN);

        GPIO_InitIO(1, MODULE_POWER_PIN);
        GPIO_ModeSetup(MODULE_POWER_PIN, 0);
        GPIO_WriteIO(1, MODULE_POWER_PIN);
    }
    else
    {
        // Sensor Module Power

        GPIO_InitIO(1, MODULE_POWER_PIN);
        GPIO_ModeSetup(MODULE_POWER_PIN, 0);

        GPIO_WriteIO(0, MODULE_POWER_PIN);

        // Sensor Power, CMOS Sensor Power Down Signal Output
        GPIO_ModeSetup(MODULE_CMPDN_PIN, 0);
        GPIO_ModeSetup(MODULE_RESET_PIN, 0);

        GPIO_WriteIO(0, MODULE_CMPDN_PIN);
        GPIO_WriteIO(0, MODULE_RESET_PIN);

        GPIO_InitIO(1, MODULE_CMPDN_PIN);
        GPIO_InitIO(1, MODULE_RESET_PIN);

        // SCCB Low
        GPIO_ModeSetup(SCCB_SERIAL_CLK_PIN,0);
        GPIO_ModeSetup(SCCB_SERIAL_DATA_PIN,0);
        GPIO_WriteIO(0, SCCB_SERIAL_CLK_PIN);
        GPIO_WriteIO(0, SCCB_SERIAL_DATA_PIN);
        GPIO_InitIO(1, SCCB_SERIAL_CLK_PIN);
        GPIO_InitIO(1, SCCB_SERIAL_DATA_PIN);
    }
}
```

主要修改的地方为总线控制方式和 GPIO 口的设置。

总线控制方式使用 sccb_config 函数设置使用软件模式还是硬件模式以及传输位数。其中 Sensor 的读/写 ID 操作要参照 Sensor 的 datasheet 设置正确。

5. 效果调整

目前以 YUV 格式居多，YUV 格式后期效果的调整主要与 Sensor 厂商联系进行调整，而 Bayer RGB 由于内部实现使用了 BB 中的 ISP 功能，所以后期效果的调整要与 MTK 进行联系。也可以自己查阅 datasheet 进行调整。

结语：

本章对 MTK 驱动开发进行了大致的介绍，可作为驱动开发进阶的基础。读者学完本章应该对 MTK 驱动框架、MTK 手机启动流程、涉及的各个主要模块的驱动开发有一个清晰的认识。读者应该能够在本章的指导下对硬件工程师设计出的 MTK 手机硬件做出配置，完成基本的功能。

MTK 音乐播放过程中

aud_main.c中

```
aud_main( ){
        :
        case MSG_ID_AUD_MEDIA_PLAY_STREAM_EVENT_IND
                aud_media_process_file_stream(ilm_ptr);
        :
}
```

征募！

移动开发

图书作者
业内专家
技术团队

希望你的团队像他们一样强大？
别犹豫，快加入我们！

- 提出富有创意的选题建议
- 参与已有图书的技术审校
- 实现自己的技术图书创作
- 发表对产业链的真知灼见
- 获得多方的品牌合作机会

移动开发产业链上“实力雄厚”、“蓄势待发”、“逐步成长”的团队和“观察者”。

- 富有无限的创意
- 具有旺盛的精力和耐心
- 热爱写作
- 乐于分享

联系邮箱：
jiangjia@ptpress.com.cn

走出山寨。重拾某种我们日渐丧失的精神：创新精神！

《中国企业家》

山寨机为我们指明了未来的三种创新策略：一是坚定地站在用户的一边；二是将非技术的创新系统化；三是捕捉本地的机会，但在全球整合创意与资源。从某种意义上说，这是有中国特色的创新之道，能够在花费较少的情况下取得最大成效。山寨机的成功，告诉我们一个重要的道理：企业的发展并不完全在于核心技术的掌握，更重要的在于对市场的感应和这种感应之后的反应速度。

《21世纪商业评论》

山寨是否能像“沙盒创新”一样，成为新兴市场为商业世界贡献的另一颗明星，取决于山寨的低成本到底来源于创造性的低成本能力，冗余功能，还是只不过是功能多样的廉价方案。

《中国青年报》

“山寨产品”是市场处于无序状态时才会盛行的，它们只能寄生于正常生产和销售所照顾不到的环节，在不影响大局的前提下，它是对市场的一种补充。现在它却大有颠覆所有制造业的发展趋势，而在讲究创意价值、重视创造力的时代，“山寨产品”的大行其道，完全不可思议。

《中外管理》

是时候摘掉有色眼镜，重新审视“山寨”问题了，巴掌大小的手机背后隐藏着一条极有可能代表中国制造业希望的产业链。这股野蛮生长的力量，以目前引人注目的生产能力和市场占有率，已经从产业的谷底爬升到了半坡之上。山寨精神，正在为中国制造注入新的活力。

《南都周刊》

保卫山寨，是在保卫某种我们失落的精神，保卫这种精神是在反抗现代社会固有的势利。当然保卫山寨，并不是要认可某种侵犯知识产权的高模仿行为，也不是要赞同企业偷税漏税的做法。保卫山寨，保卫的是一种日渐丧失的自由精神。

封面设计：黄 凯

分类建议：计算机/程序设计/移动开发

人民邮电出版社网址 www.ptpress.com.cn

ISBN 978-7-115-21668-7

定价：49.00 元