

前言	9
认识手机的存储区间.....	9
手机系统的组成.....	9
NOR.....	9
RAM.....	10
NAND	10
什么是 BootLoader?	10
BootLoader	10
引导系统启动.....	11
下载 BIN 文件.....	11
关机充电.....	11
如何下载 BootLoader	11
应用 BIN 数据区存在哪里?	11
BIN 文件数据区.....	11
一般文件数据保存在哪里?	11
EFS 文件系统数据。	11
NAND 的数据存储区	11
USB 盘区.....	12
软件开发人员需要做的工作.....	12
开发人员的工作流程.....	12
安装开发环境.....	12
安装 VC6++开发工具	12
安装 BREW3.15 的开发环境	13
安装 BUIW 开发包	13
设备文件	13
环境变量	14
安装 ARM 编译器.....	14
安装调试工具.....	14
安装其它编译工具.....	14
安装 USB 驱动程序.....	14
应用基本规范.....	14
应用名称规范.....	15
应用名称.....	15
现在的目录状况.....	15
提示.....	15
模拟器的 dll	16
dll 应该小写	16
提示.....	16
mif 文件和资源文件名称.....	16
mif 文件.....	16
提示.....	16
应用的目录规范.....	16
一级目录.....	16
二级目录.....	16
注意.....	17
应用的.c 和.h 文件.....	17
文件名称格式.....	17
文件名称长度.....	17
文件内容.....	18
项目文件.....	18
开发环境的目录宏定义和环境变量定义.....	18
开发环境.....	18
错误的开发方式.....	18
如何设置开发环境.....	19
应用中的测试窗口.....	20
功能测试窗口.....	20
不显示功能窗口.....	20

开始应用开发.....	21
新建应用	21
使用 VC6++向导	21
手工修改项目文件.....	21
BID 和 MIF 文件.....	22
创建 BID 文件.....	22
定义宏名称和 CLSID 值	22
模拟器 mif 文件的作用	22
创建应用的 mif 文件.....	23
创建扩展对象的 mif 文件.....	24
通过 mif 文件设置应用或者对象是否可见.....	25
编译 mif 文件.....	26
VC 编译应用.....	26
必须去掉警告信息.....	26
区分调试环境和手机环境.....	26
代码检查.....	26
设置模拟器应用.....	26
设置设备文件.....	27
模拟器调试应用.....	29
应用开发基本问题（初学者问答）	30
为什么启动不了应用.....	30
为什么创建对象总是失败.....	30
程序架构基本规范.....	30
程序结构标准化的需要.....	30
主程序结构不合理.....	30
窗口参数结构传递不合理.....	31
应用释放所有窗口过程不合理.....	32
主程序数据结构.....	33
窗口独立数据结构.....	33
窗口间参数传递.....	33
程序结构图示.....	34
数据结构规范.....	35
数据结构名称定义.....	35
结构的名称.....	35
公共的数据结构.....	36
曾经的问题.....	36
数据结构中内存注意事项.....	36
中英文版本内存不一样。.....	36
大数据量时内存重复使用问题.....	37
代码编码规范（简要）	37
编码基本事项.....	37
示例	37
调试信息问题.....	40
DBGPRINTF 调试信息	40
写文件调试信息.....	40
调试信息不应该放的地方.....	40
程序内存和堆栈.....	41
程序总的可用内存.....	41
总的内存.....	41
误区.....	41
内存需求空间检查.....	41
应用需求内存检查.....	41
接口需求内存检查.....	41
函数内栈空间问题和错误 rex.c 841	42
著名的 841 错误.....	42
使用数组的情况.....	42
数组改用指针.....	42

使用异步消息.....	42
入参使用指针.....	42
参数错误例子.....	43
中英文版本资源规范.....	43
版本目录和资源 ID.....	43
版本资源文件.....	43
资源 ID.....	43
载入过程.....	43
加速载入过程.....	44
调试环境和手机环境的资源.....	44
应用直接替换.....	44
OEM 层替换.....	44
编译到 BIN 文件.....	44
下载到手机目录.....	45
优缺点.....	45
资源不可采用的方式.....	45
错误的资源处理方式.....	45
资源载入失败（ISHELL_LoadResString）现象。.....	46
文件路径错误。.....	46
数据缓冲区内内存分配太小。.....	46
系统内部解析错误。.....	47
程序 CLSID 规范.....	48
CLSID 是什么？.....	48
CLSID 的定义.....	48
CLSID 和 BID 文件的位置.....	49
应用引用 CLSID.....	49
CLSID 错误的做法.....	49
窗口和事件处理.....	49
应用程序组成和事件处理.....	49
基本组成.....	49
窗口消息和事件.....	50
事件传递过程.....	50
创建根窗口（ROOTFORM）.....	51
创建根窗口.....	51
释放根窗口.....	51
释放根窗口和注意事项.....	51
建议.....	51
白屏问题.....	51
白屏闪现问题.....	51
解决白屏问题.....	52
错误的解决方式.....	52
创建窗口.....	52
窗口和根窗口的关系.....	52
创建窗口对象.....	53
释放窗口.....	53
窗口处理.....	54
设置窗口处理函数（XXX_HandleEvent）和关闭窗口处理函数（XXX_FormDelete）.....	54
把窗口（FORM）加入根窗口（ROOTFORM）.....	54
把窗口从根窗口（ROOTFORM）移出来.....	55
窗口函数处理规范.....	56
窗口事件处理示例.....	57
窗口的其它事件.....	57
控件和事件处理.....	58
控件列表.....	58
控件、容器和窗口关系.....	59
与 WINDOWS 同类控件的区别.....	60

控件的基本属性.....	60
创建控件和使用控件.....	60
列表控件（LIST）使用和示例.....	60
容器控件（IXYCONTAINER）	66
比例容器.....	67
VIEWPORT 控件.....	68
网格控件（GRID）	68
按钮（非标准）	68
显示图片控件.....	70
静态文本控件.....	70
TEXT 控件.....	71
滚动条控件.....	73
菜单控件.....	73
CheckBox 控件	73
TAB 控件.....	73
Radio 控件	73
如何把 CheckBox 和 Static 控件捆绑一起.....	75
引用计数问题。.....	75
认识引用计数.....	75
为什么这么强调引用计数.....	76
哪些操作增加了引用计数.....	76
哪些窗口减少了引用计数.....	80
替换控件默认函数.....	82
如何让静态控件响应焦点事件.....	82
控件响应点击事件的前提.....	82
如何修改默认函数.....	82
处理事件.....	83
焦点和 5 向键顺序.....	84
控件的焦点.....	84
键盘操作规则.....	84
5 向键顺序.....	84
如何创建一个自定义控件.....	84
应用窗口规范.....	84
正常窗口	84
大小.....	84
按钮位置.....	84
应用菜单.....	84
编辑菜单.....	84
进度条窗口.....	84
进度条窗口的关闭和任务取消.....	84
进度条标题.....	85
内容或者进度显示.....	85
窗口大小.....	85
进度条按钮大小.....	85
进度条窗口的错误现象.....	85
半屏幕窗口.....	86
位置.....	86
按钮位置.....	86
全屏窗口	86
哪些应用使用了全屏窗口.....	86
设置全屏窗口.....	86
全屏窗口规范.....	86
应用内部.....	86
协同应用.....	87
程序<关于>版本号管理规范	87
程序版权规范.....	87
版权	87

作者和修改内容.....	87
程序划屏处理规范.....	87
135 度斜线线划屏.....	87
90 度斜线线划屏（改变私密状态）.....	88
应用该如何处理收到的私密消息.....	88
程序异常处理.....	89
程序异常意识.....	89
内存没释放.....	89
内存不足的异常.....	89
用户强制关闭应用的异常（AVK_END）.....	89
资源数据错误的异常.....	89
用户数据错误的异常.....	89
使用 goto 处理异常.....	89
正确使用 goto 语句.....	89
不要滥用 goto 语句.....	90
异常的提示信息.....	90
准确标题信息.....	90
准确的内容提示.....	90
准确的图标.....	90
程序互斥规范.....	90
程序自动化编译规范.....	90
程序宏定义规范.....	91
应用内部的宏定义.....	91
应用间的宏定义.....	91
宏定义的名称.....	92
调试信息规范.....	92
日志文件.....	92
日志文件的目录和大小。.....	92
否写日志.....	92
正式版本.....	92
QDXM 调试信息.....	92
不要频繁打印调试信息.....	92
使用中文.....	92
内容准确不罗嗦.....	92
全局变量和 inline 函数.....	93
全局变量.....	93
慎用全局变量.....	93
命名全局变量.....	93
修改全局变量.....	93
inline 函数.....	93
优缺点.....	93
哪些函数建议使用 inline.....	93
大数据量处理 CPU 时间限制.....	93
为什么不能使用 FOR、WHILE 连续处理大数据量.....	94
CPU 时间限制.....	94
提示.....	94
使用 ISHELL_POSTEVENT 消息处理。.....	94
消息机制.....	94
处理消息位置.....	94
消息丢失问题.....	94
性能问题.....	94
休眠状态.....	94
使用 ISHELL_SETTIMER.....	94
休眠挂起状态.....	95
解决系统休眠.....	95
取消定时器.....	95
定时器间隔周期.....	95

周期和暴力测试问题.....	95
ISHELL Resume 函数处理重复执行的过程.....	95
效率问题.....	95
休眠问题.....	95
取消回调.....	95
暴力测试问题.....	96
如何使用.....	96
带窗口的接口或对象规范.....	96
单实例对象.....	96
单实例对象优缺点.....	97
优点.....	97
缺点.....	97
多实例对象.....	98
多实例对象优缺点.....	98
数据安全.....	98
内存需求较大.....	98
应用程序更为复杂.....	98
更多的异常处理.....	98
释放更为麻烦.....	99
多实例对象的标准规范:	99
标准创建接口.....	99
标准 Release 接口.....	99
能被动移出 (_REMOVE)	99
能动态移出所有窗口.....	99
不能只移出一个顶部窗口.....	99
多窗口 Release.....	99
对象内部有多个窗口.....	99
对象中还创建其它对象.....	99
内存	100
应用检查内存.....	100
对象需要检查内存.....	100
对象示例	100
应用程序和对象创建前.....	101
应用创建了对象后.....	101
把所有窗口都释放.....	101
先释放对象.....	101
释放函数示例.....	102
释放函数该处理过程.....	102
对象的数据.....	103
误区.....	103
ARM 编译项.....	103
如何在把应用编译入手机 BIN 文件.....	103
把应用放到编译目录下.....	103
在 OEMModTableExt.c 文件中增加	104
在 incpath.min 文件中增加	104
在 dmss_qsc60x0.mak 文件中增加:	104
在 dmss_objects.min 文件中增加:	104
在 dmss_rules.min 中增加,	104
如何修改 min 文件.....	104
min 文件的意义.....	104
增加 C 文件.....	104
注意事项.....	104
nand 和 nor 的区别.....	104
设置文件系统区.....	105
性能优化	105
性能优化的需求.....	105
显示过程的优化.....	105

资源载入的优化.....	105
开发注意事项.....	105
如何在模拟上调试唤醒挂起.....	105
如何让系统不进入休眠状态.....	105
获取当前系统的背光值.....	105
取消背光.....	106
图示.....	106
UI 界面应用和底层应用交互的过程.....	106
向底层注册回调函数.....	106
开始向底层写入数据.....	106
底层调用回调函数.....	106
更新数据和相关模块数据.....	107
从底层取消.....	107
图示.....	107
ClearCase 上应该保存哪些文件.....	108
应用的源代码.....	108
应用完整的资源文件.....	108
应用的批处理文件.....	108
应用配置文件.....	108
完整的测试代码.....	108
误区.....	108
RELEASEIF 和 IWIDGET_Release 的异同.....	108
共同点.....	108
区别.....	109
ModelListener 的取消问题。.....	109
使用监听对象（ModelListener）.....	109
不取消监听对象可能产生的结果.....	109
注意.....	109
BPOINT1 和 BPOINT 3 的错误。.....	109
内存泄露（BPOINT1）.....	109
内存重复释放（BPOINT3）.....	110
内存越界.....	110
内存问题的建议.....	110
采取的措施.....	110
OEM 层不应该处理 UI 的事情.....	111
文件操作注意.....	111
不能同时对一个文件进行操作.....	111
树型文件夹问题.....	111
系统 USB 文件目录.....	111
T 卡文件目录.....	111
编译环境下不应该有垃圾文件.....	112
mif 文件中的项意义.....	112
提交版本前测试项.....	112
应用启动测试.....	112
干净环境的启动测试.....	112
丢失配置文件的启动测试.....	112
安全模式下的启动.....	112
编译应用和功能测试.....	113
提交版本前.....	113
自动化编译问题.....	113
修改注意的问题.....	113
启动速度测试.....	113
空记录启动的时间.....	113
满记录启动的时间.....	114
大数据量操作测试.....	114
载入数据需要的时间.....	114
删除所有数据需要的时间.....	114

满数据量下所有可能进行的操作.....	114
系统极度繁忙测试（暴力测试）.....	115
应用的暴力测试（单个应用）.....	115
系统繁忙的暴力测试（多个应用）.....	115
暴力测试的提示（更高的品质）.....	115
应用互斥测试.....	116
同时对 T 卡的写文件.....	116
同时对数据库的操作.....	116
占用内存测试（启动内存和最大内存）.....	116
内存稳定情况.....	116
空记录启动后占用的内存.....	116
满记录后启动占用的内存.....	116
所有窗口打开后占用的内存.....	116
使用过程是否有内存泄露.....	116
是否有 BPOINT1 和 BPOINT3 产生内存问题.....	116
操作响应速度.....	117
操作响应的速度.....	117
对数据库的操作响应速度（批删除、批增加）.....	117
对底层任务处理的响应速度.....	117
批删除文件，COPY 文件的响应速度.....	117
系统时间测试.....	117
当前时间下正常情况.....	117
网络更换（如启动 C 网同步时间）.....	117
修改为 1980 年前情况.....	117
修改为 2050 年后的情况.....	117
时区改边的情况.....	117
待机测试.....	117
正常待机.....	117
强制待机.....	117
待机后来电和短信.....	118
挂起和唤醒测试.....	118
正常挂起和唤醒后.....	118
T 卡插拔测试.....	118
是否产生系统崩溃.....	118
是否产生内存泄露等.....	118
是否文件丢失.....	118
是否应用执行失败.....	118
系统极度繁忙的时候是否更大几率产生问题.....	118
断网测试.....	118
强制关闭网络的测试.....	118
反复打开关闭应用的情况.....	119
新建默认项测试.....	119
默认标题.....	119
默认日期时间.....	119

QUALCOMM 平台 BUIW 开发文档

编写：林 树 春

前言

从 268 项目开始，到现在 2938 基本开发完成，在开发过程中遇到了很多问题，我们都非常艰难地走过来了；因此我们把过去开发中产生的问题进行总结，并将错误产生的原因显示给后来的开发者，将是这文档的主要目的。

我们从开发者的角度，把过去开发过程中产生的问题和 BUG，都罗列出来，并且把一些代码和示例也列出来，大家一起参考；文档中可能列出了我们曾经不够完美的代码和一些开发的规范。

在文档中我更多注意的是开发过程中，提到开发人员的意识问题，以及时常需要关注的问题和要点。对很多普通的问题，文档中都不仔细提及这些问题。

如文档中内容需要修改或者存在缺陷，可直接联系（林树春，邮件 linshuchun@yulong.com），我将在继续修改和维护该文档。

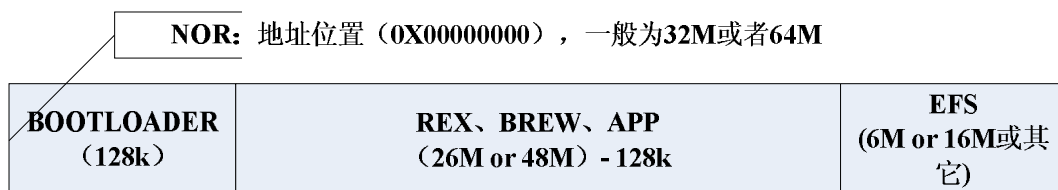
认识手机的存储区间

➤ 手机系统的组成

手机存储介质一般分为 3 块，它们分别功能是存储启动文件和系统 BIN 需要的 NOR、系统和程序运行需要的内存空间 RAM、系统配置文件及普通文件和作为 USB 使用需要的 NAND，前 2 个存储区是必须的，有些手机可能没有 NAND 这块存储区；这些区块的主要作用，在下面对它们进行简要的说明：

◆ NOR

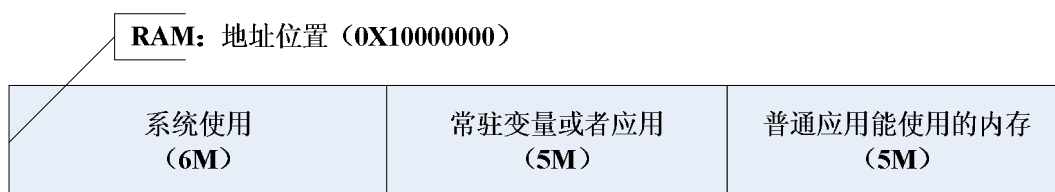
该块区域保存了系统启动必须需要的数据，大小一般为 32M 或者 64M，其中又分为 3 部分存储数据（如下图）；该块系统数据区主要特点是该区的数据不会断电而丢失、可以直接启动应用以及读快写慢的特点（写的速度一般为为 NAND 的七分之一到十分之一之间，这也说明了为什么 CP828 下载 32M 的 BIN 文件只需要 53 秒左右，而 CP268 却需要 7 分钟的原因），这里以 32M 和 64M 说明，结构如下图（图中第 2 区块和第 3 区块可以根据实际情况做相应调整）：



NOR 结构

◆ RAM

内存区和 WINDOWS 的内存区一样，一般使用 8M、16M、32M 或者更大，分系统使用、常驻应用使用和普通应用使用的区，大部分情况下（如 CP2938 机器启动后）内存使用的分布情况大概如下图。

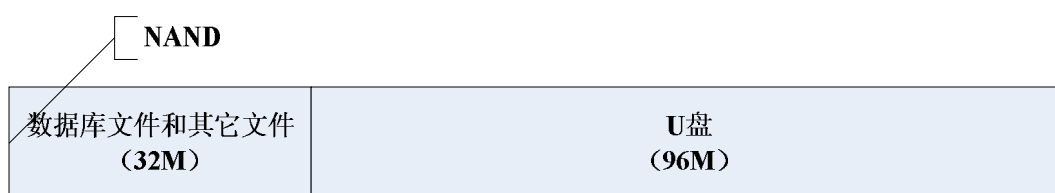


RAM 结构

从上图可以看出，在系统启动完成后，普通应用能使用的内存非常少（一般为 5-6M），应用开发就应该时时刻刻特别注意内存问题。

◆ NAND

NAND 是用户数据存储区和 U 盘区，现在一般为 128M 大小；这里一般在前 32M 中保存了应用的数据库文件和应用需要的配置文件等，而后 96M 当本机 U 盘使用，即应用中《本机当 U 盘》文件存储的区域就是在该块中，有时应用的大部分日志文件也是保存在该块中的，NAND 存储区的特点是写快读也快，但成本价格高。



这块区域（NAND）不是必须的，大部分情况下为了节约成本，该块可能被去掉，而把相应的数据放到 NOR 的 EFS 中，但 NOR 的特点是写文件慢（慢 7-10 倍），可能会影响写文件的操作速度，因此有写日志文件的应用就需要考虑这些日志可能速度影响操作的问题了。

➤ 什么是 BootLoader?

◆ BootLoader

BootLoader 是引导系统启动的小应用（一般小于 128K），手机启动后，首先载入就是该应用。

该应用主要有 3 个功能：

◆ 引导系统启动

该BOOTLOADER应用是手机开始启动时必须首先被载入的第一个应用，由该应用把 BREW 系统启动起来，在 BREW 系统启动后，该应用就完成功能退出了。

◆ 下载 BIN 文件

静态应用经过 ARM 编译后，最后需要把整个 BIN 文件（24M 到 48M 之间）下载到手机上，而在这个过程中系统并没有启动起来；BootLoader 的功能就是把系统的 BIN 文件下载到手机上，下载完成后，重新启动时由 BootLoader 引导系统跑起来。

◆ 关机充电

关机后系统并没执行，因此充电过程由该应用执行。

◆ 如何下载 BootLoader

刚生产的机器，没有任何数据，因此需要使用硬件调试器，下载该应用。

➤ 应用 BIN 数据区存在哪里？

◆ BIN 文件数据区

ARM 编译完成后的 BING，该 BIN 文件中包含 3 块数据，包括实时操作系统数据、BREW 系统需要的数据、普通 PDA 应用需要的配置数据（如资源文件 BAR 的数据）。

实时操作系统的数据库、BREW 系统数据以及普通 PDA 应用的 BIN 文件都放在该区中，如果整个 NOR 为 32M，那该区一般分为（32M-6M-128K）大小，可以根据实际情况加以调整。

所有最后 ARM 编译产生的 BIN 文件，不应该超过这个值，不过应用开发人员一般不必关系这个问题的存在；如果 BIN 文件确实大，可以使用 64M 的 NOR，将把这里调整为 48M，那是足够的。

➤ 一般文件数据保存在哪里？

普通文件在以下 3 个位置都可以存储。

◆ EFS 文件系统数据。

应用的一些基本数据，或者 BREW 的基本文件数据，都将保存在这个区域，如果是 32M 的 NOR 一般该区为 6M；如果使用 64M 的 NOR，可能调整为 16M，把另外有些数据（如 XBASE 数据库文件）放到该区域。

◆ NAND 的数据存储区

现在我们的数据库文件都是放在该区，在 NAND 区中的前 32M 中存储；这个数据库文件也有可能放到 NOR 上。

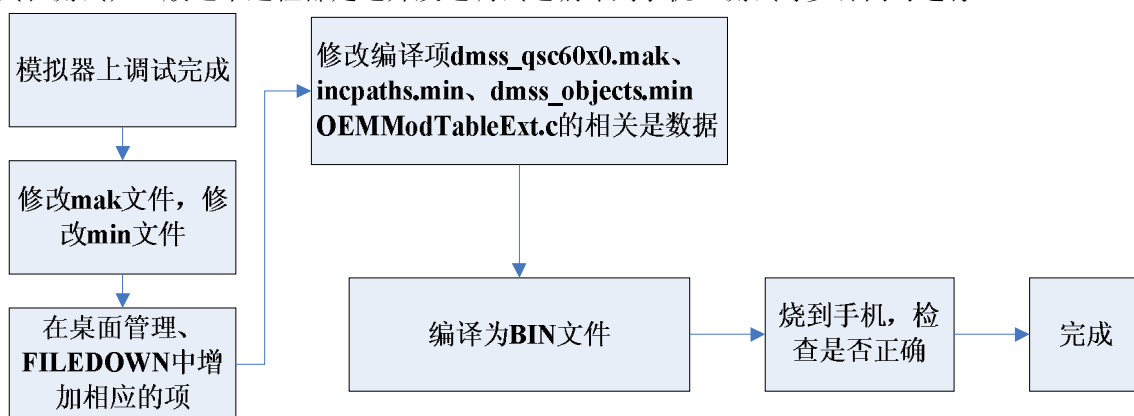
◆ USB 盘区

用户的文件或者有时有些日志信息，都放在该区中，这里的 U 盘是指本机当 U 盘使用；该区的数据文件，用户是可以看到的，并且一般也是可以直接操作的。

➤ 软件开发人员需要做的工作

◆ 开发人员的工作流程

开发人员的主要工作是按需求在模拟器上，开发需要的程序，实现完整的功能，并且把程序的功能结构调试完成、考虑各种可能的情况把异常都处理好，然后编译到手机上进行真实环境调试和测试；一般这个过程都是边开发边调试边编译到手机上测试等步骤同时进行。



开发人员的工作流程

安装开发环境

安装开发环境的过程，这里不做非常具体的描述，点到为止，如果安装过程中有问题，可直接和其他开发人员交流，大概过程如下步骤：

➤ 安装 VC6++开发工具

先安装 VC++开发工具和相应开发补丁（必须统一使用 VC6++版本），可再安装 VC 助手或者其它辅助开发工具。注意安装顺序，如果先下载安装 BREW 开发环境，再安装 VC6 和补丁，有可能导致 VC6++中没有 BREW 项目向导。

提示：

应用开发必须统一使用 VC6++开发工具，规规矩矩使用 VC6++开发，有些开发人员使用 VC2005 导致程序移交给其它开发人员时，需要重新调整开发环境。

CC 管理人员也需要检查这些代码是否在 VC6++上完成，否则将给以警告。

➤ 安装 BREW3.15 的开发环境

从 QUALCOMM 开发网站下载 BREW3.15 的开发环境，包括 BUIW 开发环境和开发调试工具等，都可以直接在网上安装，一般可按默认目录安装。也可以从其它已经安装好的机器上直接 COPY 开发环境（这种情况 VC6 中不会有项目向导）。

➤ 安装 BUIW 开发包

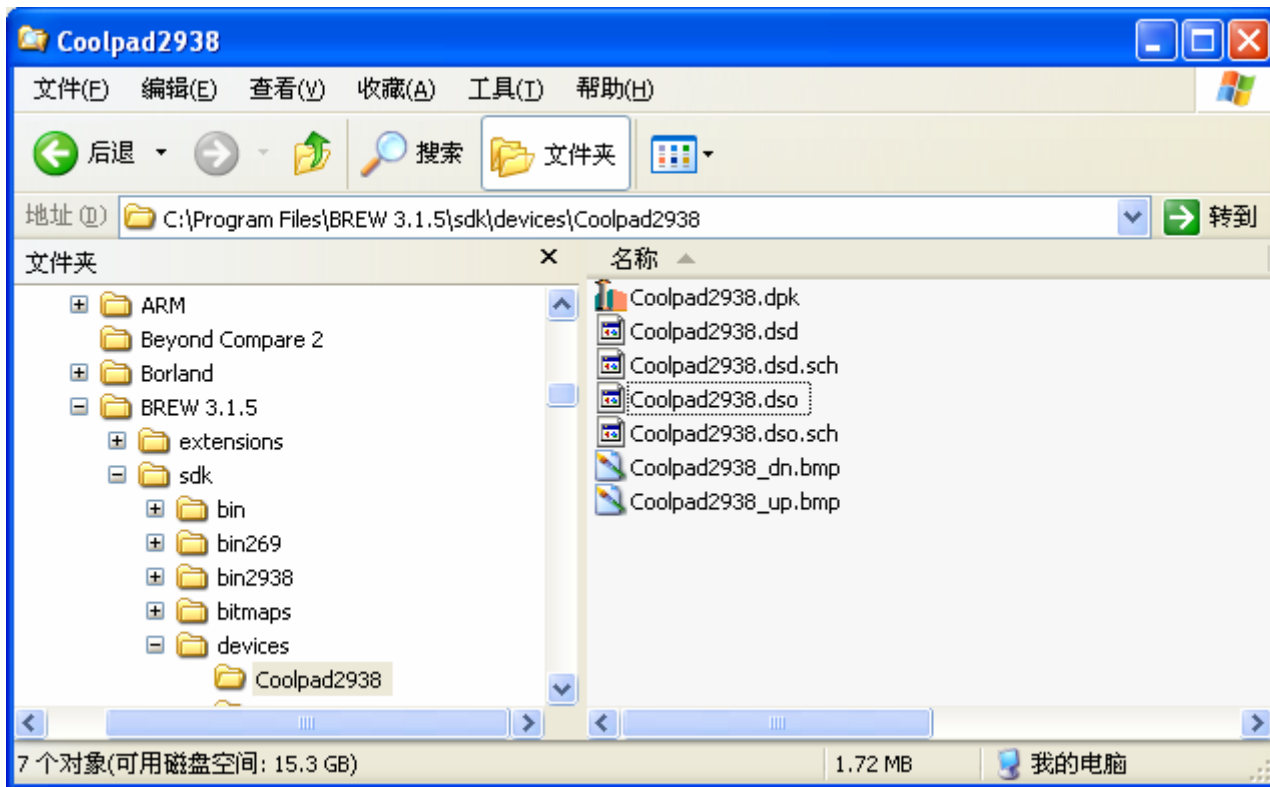
新建一个开发目录，下载公司最新的 BUIW 开发包到该目录，把该开发包中的文件放到对应位置，注意如果开发环境 inc 位置和开发包中 inc 位置不一样，可能需要更改开发包中 inc 目录的位置，可参考开发包中说明文档。

提示：

有些静态编译应用（如联系人的数据库对象），需要安装（BREWpk.rar）目录下的所有开发组件，有需要的应用再安装这些。

➤ 设备文件

下载最新的设备文件，一般放到按默认位置，如（C:\Program Files\BREW 3.1.5\sdk\devices）目录下，在模拟器里需要指定该目录位置，如果放在其它目录，模拟器调整下该目录也可以的。



模拟器设备文件

提示：

部分应用可能需要有共享内存的模拟器，如联系人，如果没有共享内存会对象创建失败。

➤ 环境变量

安装后需要设置开发环境的环境变量，如果按默认安装，一般都会自动注册需要的环境变量，如没有可以按具体的项自己定义。

➤ 安装 ARM 编译器

安装 ARM 编译器到 C 盘根目录（选择定制安装，把 example 项去掉），安装 ARM 的 3771 补丁，先删除 C:\ADSV1_2\Lib\armlib 目录下的所有文件，用 ARM 编译器中的 armlib 目录替换 C:\ADSV1_2\Lib\armlib 目录下的内容。

提示：

使用正版 ARM 编译器。

➤ 安装调试工具

安装 QXDM 查看调试信息工具，并且注册。

➤ 安装其它编译工具

安装 ActivePerl 程序，同时安装 Cygwin，注意一定要选择全部安装，否则编译不了应用；增加环境变量<Cygwin 安装目录>/bin 到 path 项中。

➤ 安装 USB 驱动程序

安装 1.1 版本 USB 驱动。

应用基本规范

我们将对应用开发的基本构架做一个规范和说明，尽量使用开发人员能回避那些曾经出现过的麻烦问题，使初学者能避免或者不需要特别注意 BREW 开发环境本身存在的一些 BUG。

BREW 并不完美，应该说是错误百出，真是非常不人性化一个开发环境，因此我们定义部分规范，建议开发人员都按这些基本规范构建应用和目录结构。

➤ 应用名称规范

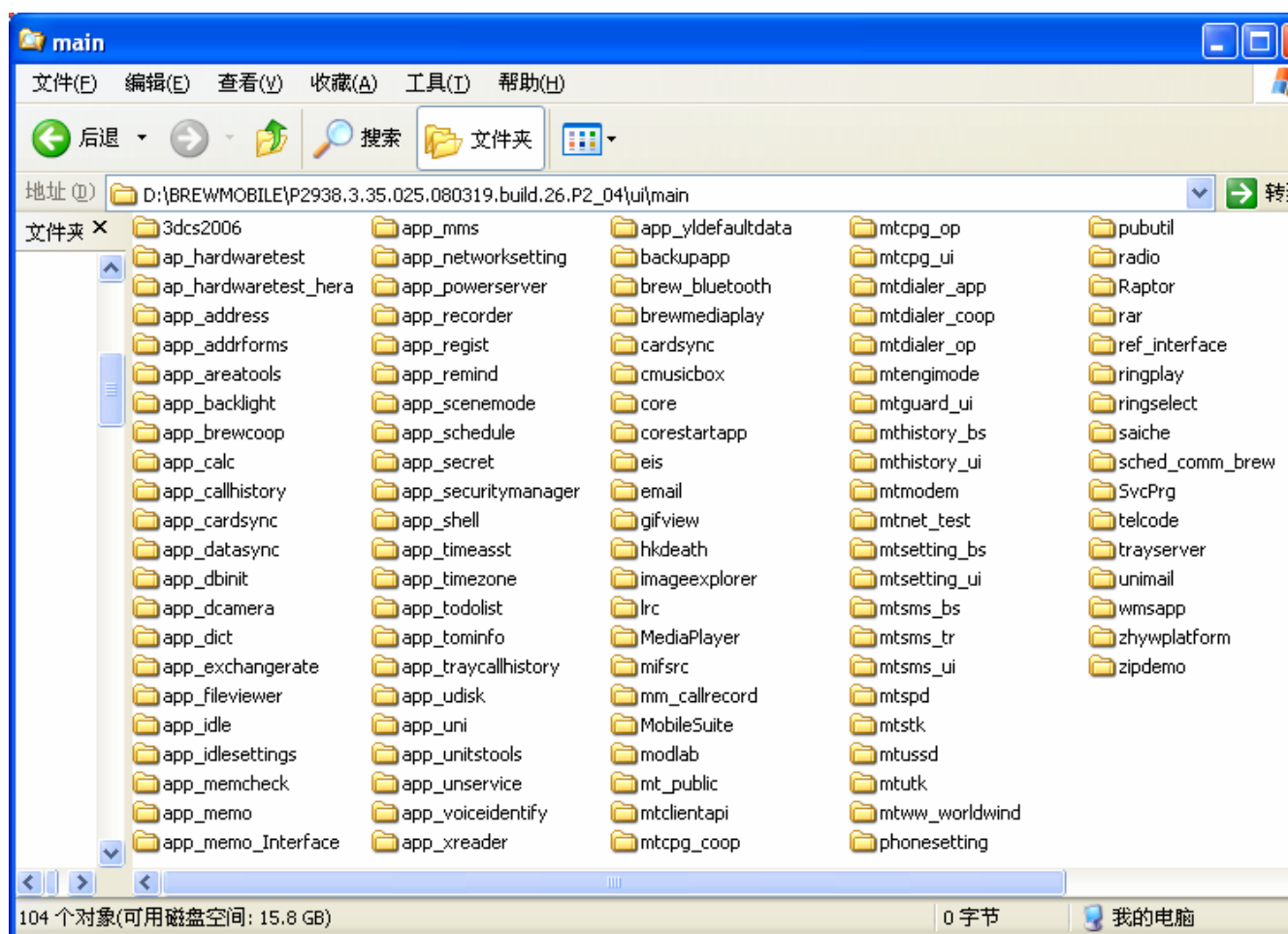
◆ 应用名称

应用分 3 类，PDA 应用名称**必须使用**（**app_**xxx）的格式、通讯的应用**必须使用**（**mt_**xxx）的格式、多媒体的应用名称**必须使用**（**mm_**xxx）的格式来命名，并且**名称都必须使用小写**，名称长度应该在 8-16 个字符之间比较合适，当然，使用更长或者更短的名称也是可以的，但不推荐。

特别强调，应用名称一定不要大小写混合，因为模拟器上编译的 DLL 可能会因为大写而找不到。

◆ 现在的目录状况

从下图可以看出，现在应用的名称是各有各的特点和特色；希望今后能慢慢改善状况。



现在的目录状况

◆ 提示

如果存在大写，有可能 BREW 模拟找不到该应用的 DLL，因而会花很长时间去找这样不该出现并且没有任何意义的问题。

➤ 模拟器的 dll

◆ dll 应该小写

VC6++编译产生的模拟器调试 dll 名称必须是小写。

◆ 提示

模拟器是通过 mif 文件找对应目录下的应用 dll 文件；如果是目录或者 dll 有大写字符，可能产生找不到的情况。

一个目录下只能有一个 dll 文件，如果有多个，模拟器存在 bug，可能找到另外一个 dll 而无法启动应用。

所以为了解决这些基本的问题，应该在程序的入口，就开始使用 DBGPRINTF 打印调试信息，查看定位是否进入应用。

➤ mif 文件和资源文件名称

◆ mif 文件

文件名称都必须是小写。

◆ 提示

1: 文件不能包含汉字，应该使用标准名称。

2: 名称不要太长。

3: 如果模拟器找不到该应用默认图标，可以先看看 mif 文件的 Applets->Applet Type 是否被设置为不可见项了

➤ 应用的目录规范

◆ 一级目录

一个应用只能许可最多 2 个一级目录。

一般来说，如果应用是个独立的应用，不对外提供接口，1 个目录足够了；如果是需要对外提供很多个接口，或者本身就是个对象，2 个一级目录也是足够的。

建议目录名称长度限制 16 个字节以内。

太多的一级目录，影响开发环境的目录结构，使目录树看起来非常紊乱。

◆ 二级目录

二级目录主要保存当前应用的代码文件，建议把代码分为.c 和.h 各自的目录保存，即保存在 \src 和 \inc 目录下；现在我们的 BREW 系列手机，都可能有中英文版本，应用二级目录下也应该包含英文版本的目录；建议一般情况下都需要包含以下目录

《1》 \const 常量资源文件的目录，一般应用都有些资源，是中英文共用的，建议都放到这个目录。

《2》 \en 英文版本的资源文件，如果没有英文版本，可以不需要。

《3》 \zhcn 中文版本的资源文件。

《4》 \inc 应用的所有头文件都必须放到该目录下。

- 《5》 \src 应用的所有 C 文件都必须放到该目录下。
- 《6》 \imgzhcn 中文 图片资源文件都放到该目录下目录
- 《7》 \imgen 英文图片资源文件都放到该目录下目录，如果没有英文版本，可以不需要。
- 《8》 \mif 应用的 mif 文件
- 《9》 \bid 应用的 bid 文件

当然这些目录并不是必须存在的，比如没英文版本，和\en 有关的目录就可以不必存在。

◆ 注意

现在有些 UI 应用，所有文件都放在一级根目录下，包含了很多文件，有的多到 50 多个文件，甚至图片文件都放在根目录下，这种结构很不合理。

文件结构应该让人一目了然，知道该到哪个目录下找需要的文件，这应该是每个应用应该做到的基本功能。

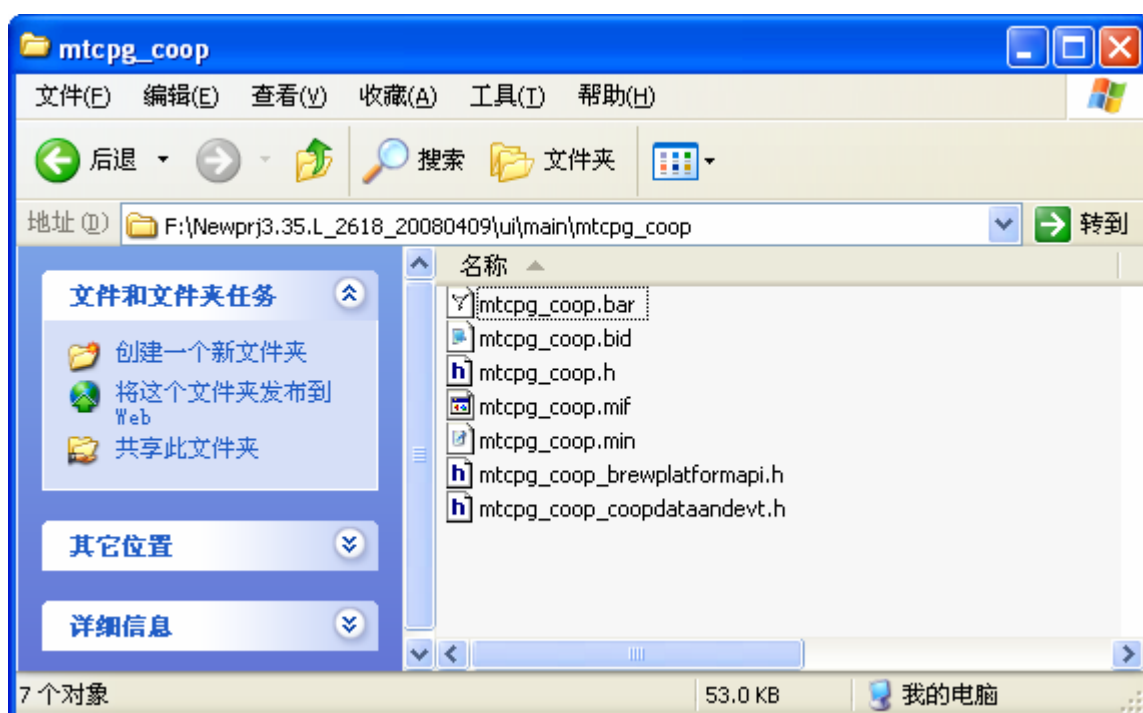
➤ 应用的.c 和.h 文件

◆ 文件名称格式

为了区别文件是哪个应用的，建议代码文件名称以应用名称前 4 个字符开头，如果联系人的应用文件，全部都加有 addr 开头。

◆ 文件名称长度

建议.c 和.h 文件，建议名称不要超过 16 个字节,但也不要太短；太长显得罗嗦，太短不能表明是哪个应用的文件。



长达 27 个字符的文件名称

◆ 文件内容

建议 C 文件内容不要超过 5000 行代码，如果超过，建议分长多个文件，把部分公用函数提取出来；另外也不要几行代码单独就放一个文件，H 文件不限制这些。

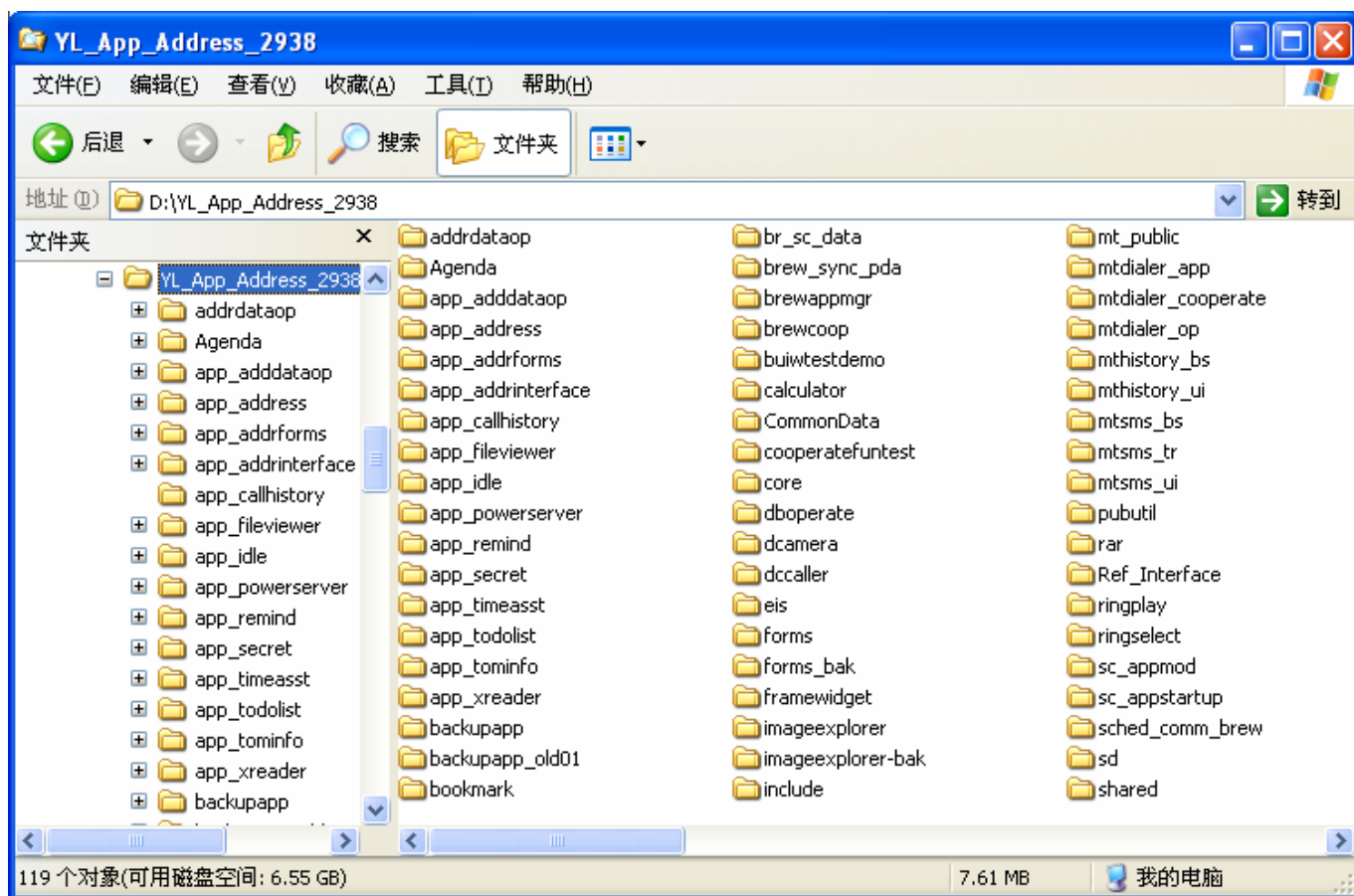
◆ 项目文件

项目文件（.dsp 和 .dsw 文件）应该和 C 文件等放在一起，理论上应该从 CC 上下载下来的程序代码，就能直接编译。

➤ 开发环境的目录宏定义和环境变量定义

◆ 开发环境

一般情况下，开始一个应用后，建议把这个应用的相关应用和 widges 以及 forms 都放在同一个目录下；主要为了编码和调试方便，基本的目录结构如下图（一目了然，我们的应用名称很不规范）。



开发环境目录结构

该图是联系人应用的开发目录，所有和该应用有关的其它应用，都放在该目录下。

◆ 错误的开发方式

曾经看到 2 种开发方式，认为非常不可取的方式：

- 《1》 在编译环境下开发。虽然开发也是可以的，但每次新的编译环境出来，就需要重新调整 1 次开发环境，显然是非常麻烦的事情。并且开发环境下，很多相关文件（如 DLL 文件）是不全的或者没有的，开发过程中产生的问题也都是不可预料的，显然，麻烦不是多了一点点。
- 《2》 把应用文件都映射到 A 盘，然后在 A 盘进行开发。虽然很有创新精神，但这样做很麻烦。特别是如果应用需要换人开发，接手的人很大可能是需要重新进行调整开发环境。

二、环境设置

- (1) 把工作目录映射成 A 盘：

```
c:\windows\system32\subst A: /D
```

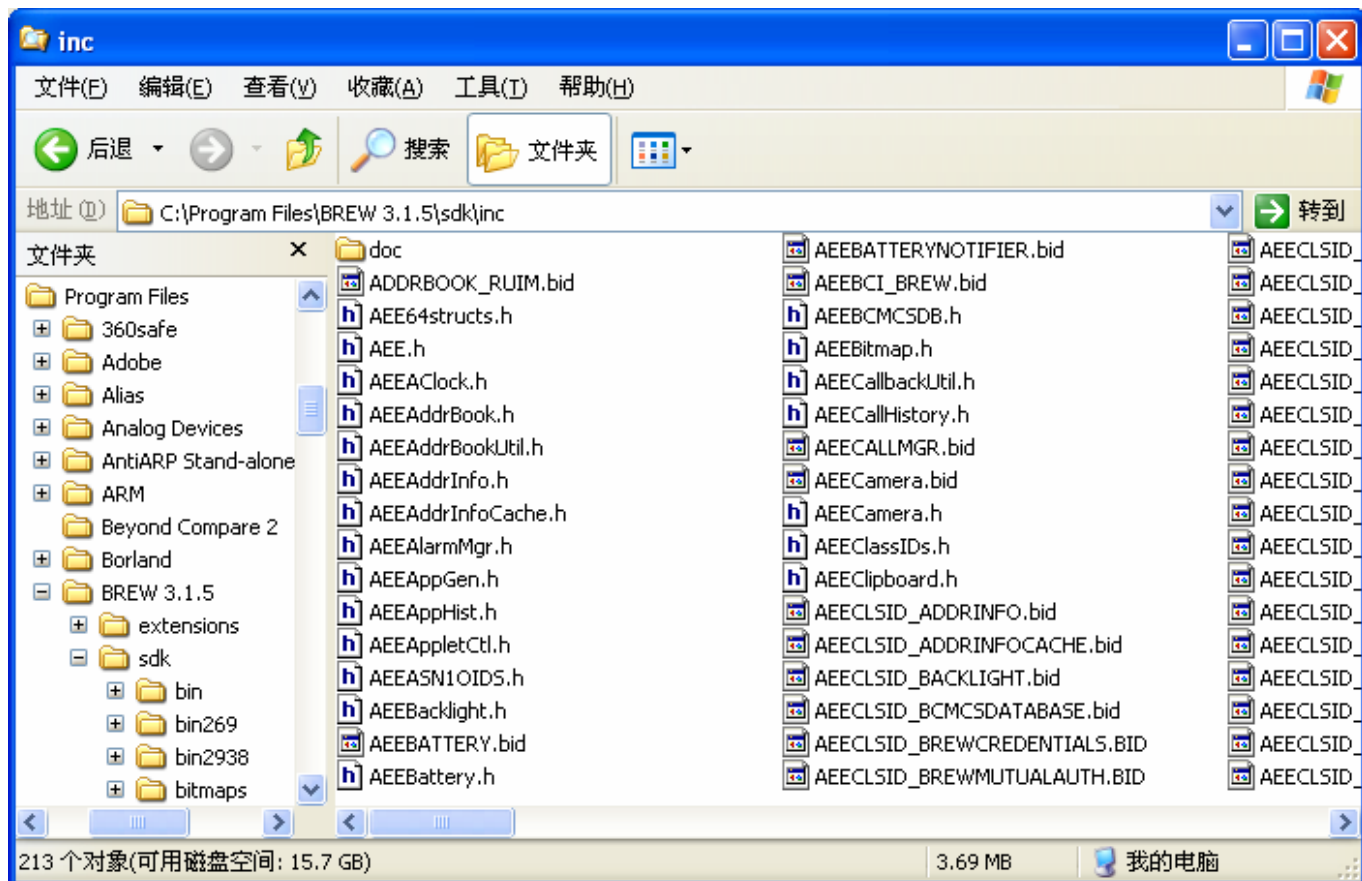
```
c:\windows\system32\subst A: E:\宇龙项目\2718
```

错误的开发方式

◆ 如何设置开发环境

调试环境下应用中可能使用了很多其它应用的接口或引用目录下的头文件，以及考虑到应用可能转给其他人开发，所以希望通过宏定义来引用目录，快捷更换开发环境。

- 《1》 AEE 目录的宏定义，应用程序都包含该目录下的头文件，如下图，建议应用都按 <BREWDIR> 定义该环境变量。



系统的 INC 目录

《2》 应用工程的目录宏定义，就是应用开发环境的目录，建议按 <BREWDIR_APP_XXXX> 这样的格式来定义。如联系人开发环境为 D:\YL_App_Address_2938, 环境变量定义为 <BREWDIR_APP_2938>。开发时，在 VC6++ 的菜单 project>settings 的 C/C++ 页中 Project options 项进行设置

如果按这 2 种方式定义开发环境和工程，更换目录环境或者更换开发人员，重新设置开发环境将是非常容易的事情。

➤ 应用中的测试窗口

◆ 功能测试窗口

建议每个复杂应用程序应该有个自己的测试功能窗口，所有的功能都应该有测试项，理论是该窗口应该是个完整的功能窗口。

对外提供接口的应用，建议需要提供测试应用。如果别人调用过程失败，可以通过测试程序查找问题所在。

◆ 不显示功能窗口

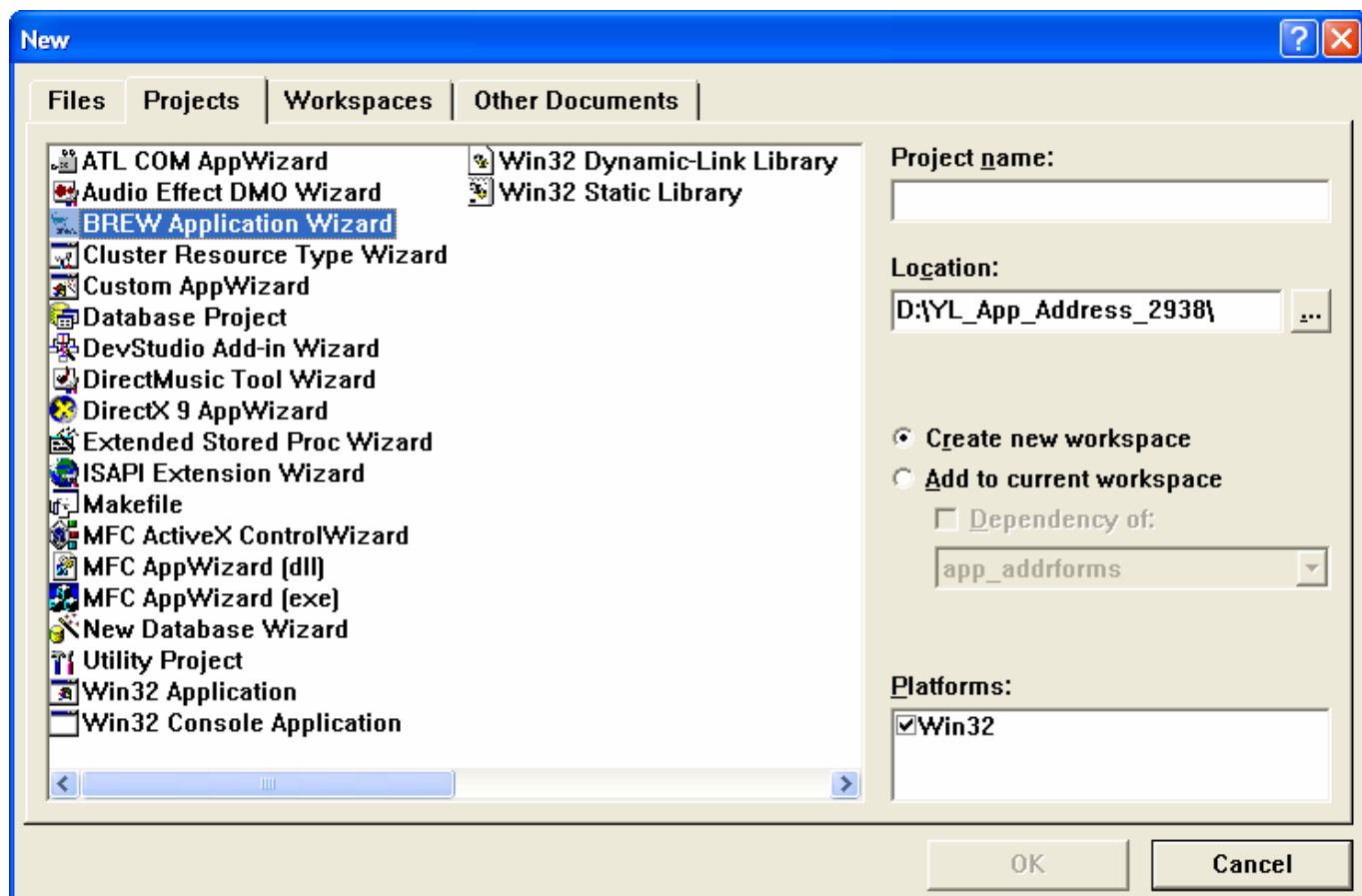
该窗口最后发布一定不能显示出来，通过宏定义来显示隐藏该窗口。

开始应用开发

➤ 新建应用

◆ 使用 VC6++向导

一般情况，在安装 QUALCOMM 的开发环境后，VC6++的新建页中会增加一项（BREW Application Wizard），如下图；新建应用，按默认项就可以新建一个应用。



BREW 应用向导

◆ 手工修改项目文件

如果安装过程中产生问题，并没有 BREW 应用向导，也可以直接到安装时的默认目录下，找 examples 下的一个应用（如 expensetracker 应用，或者 BREW 的例子程序 calculator），把相对应的名称修改成新应用的名称。修改的文件包括 a.dsp 和 a.dsw 文件中的名称。

➤ BID 和 MIF 文件

◆ 创建 BID 文件

打开记事本创建一个新文件，名称按 AEEID_XXXXX.bid 命名（这里名称大小写没关系），如下图。

◆ 定义宏名称和 CLSID 值

按下图格式定义宏名称和 ID 值，这里的 CLSID 值需要向应用管理组申请。如下图应用的 CLSID 为 0X01500013



```
#ifndef ADDRFORM_BID
#define ADDRFORM_BID

#define AEECLSID_ADDRFORM      0x01500013

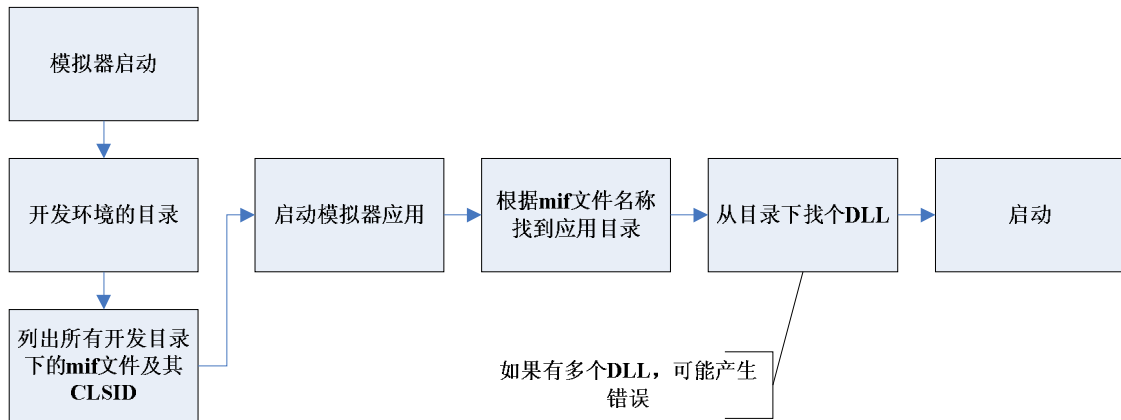
#endif //ADDRFORM_BID
```

BID 格式

注意：

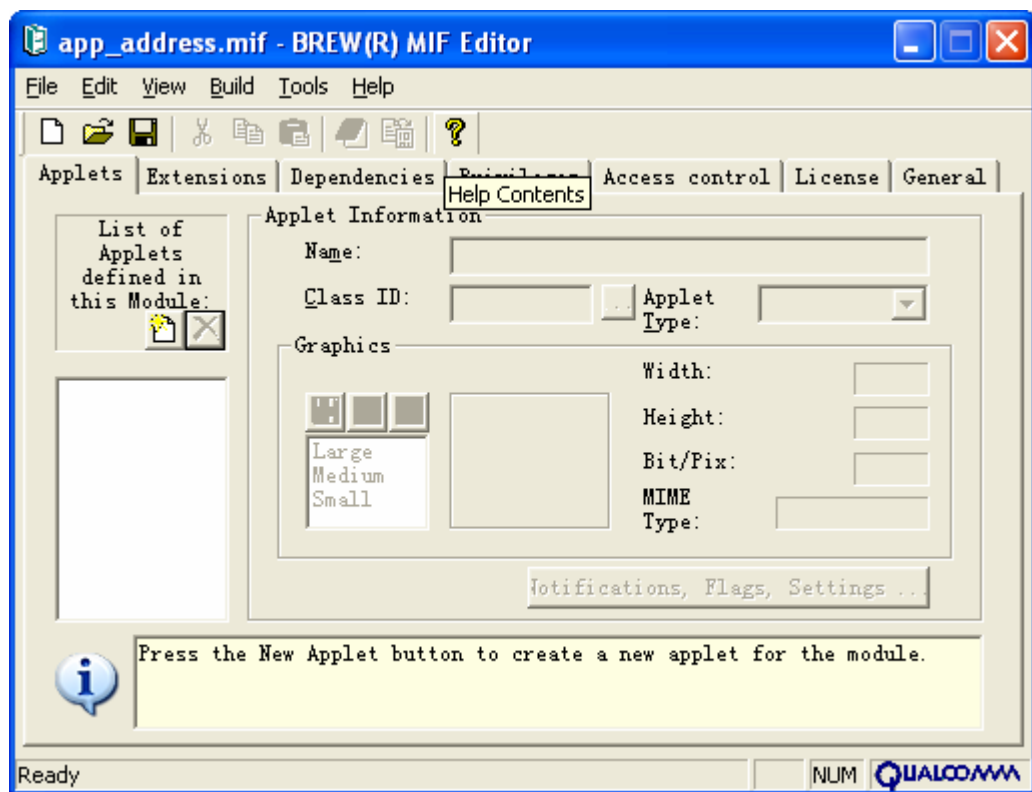
如果使用相同的 CLSID，可能产生不可预知的问题。

◆ 模拟器 mif 文件的作用



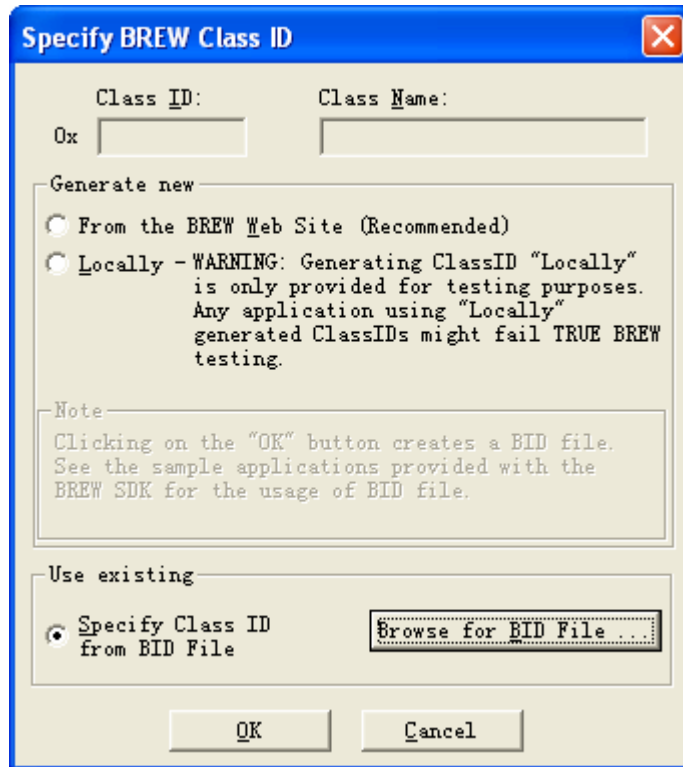
◆ 创建应用的 mif 文件

从 BREW SDK TOOLS 菜单中找到 mif 编辑程序，新建 mif 文件。可见应用的 mif 文件中增加 BID 文件过程如下。



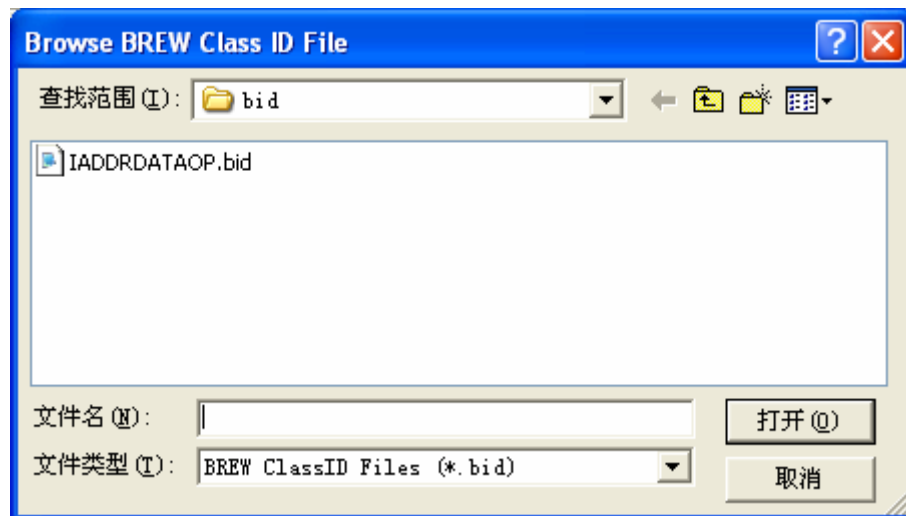
mif 编辑窗口

指定 BID 文件：



从指定的 BID 文件中设置 CLSID 值

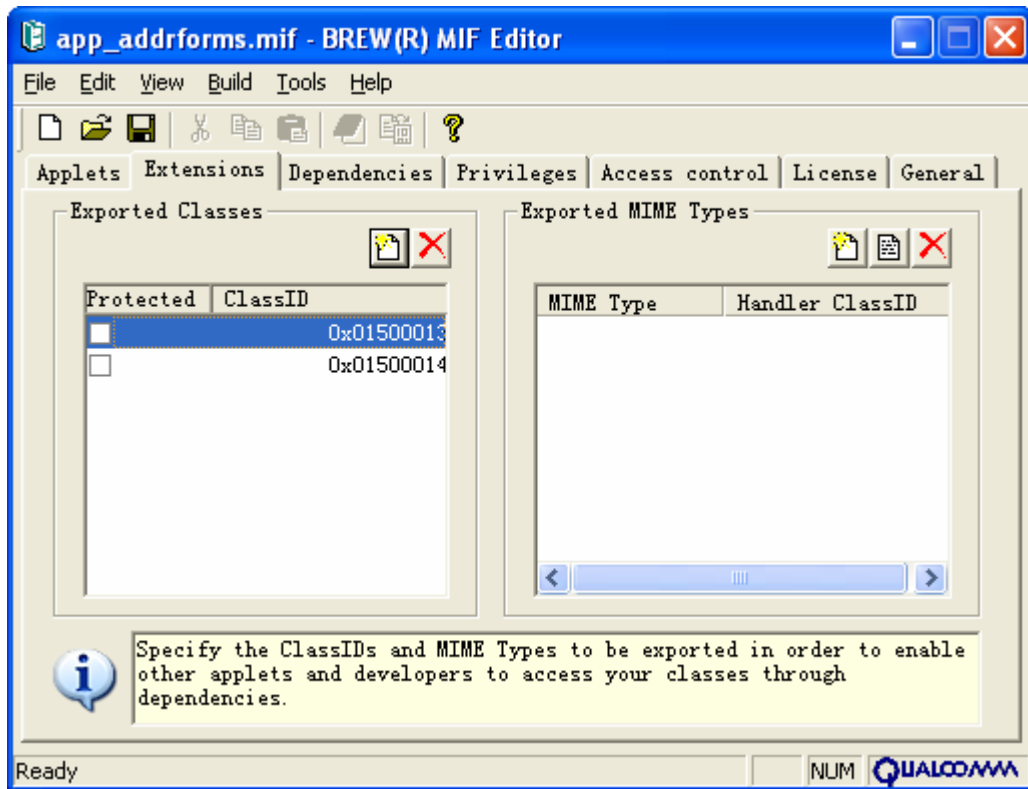
现在 BID 文件



选择 BID 文件

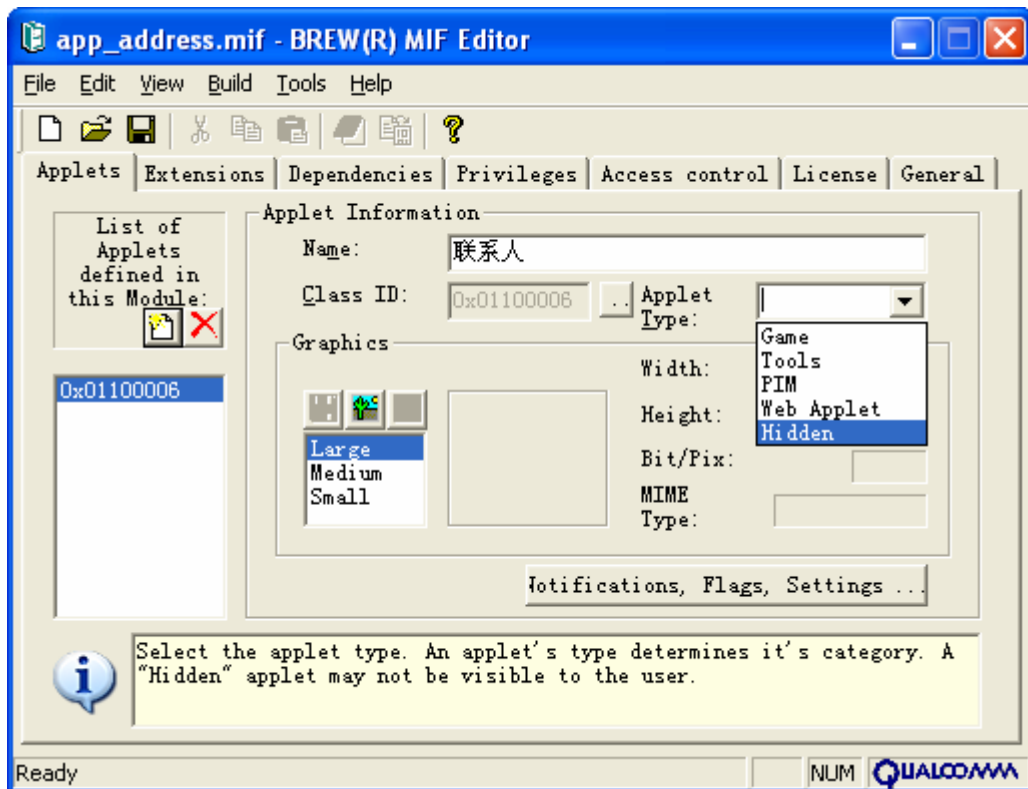
◆ 创建扩展对象的 mif 文件

扩展对象的 mif 文件指定位置不同，在第 2 个页面中，其它相同。



扩展对象的 mif 文件

- ◆ 通过 mif 文件设置应用或者对象是否可见



设置是否可见

注意：

由于 mif 文件决定应用是否可见，所以在模拟器上，应该设置为可见；而在手机上，由于应用都是静态应用，所以应该设置不可见，让应用桌面管理来启动才显示可见。

◆ **编译 mif 文件**

保存到项目下的 mif 目录后，编译（F5 编译或者菜单编译），产生.mif 文件。

➤ **VC 编译应用**

◆ **必须去掉警告信息**

使用的是 VC6++ 开发工具，编译过程和 WINDOWS 桌面应用的编译过程是一样的。如果应用开发过程中编译有警告信息，建议都解决这些信息。警告信息可能在影响最后 ARM 编译文件和手机上执行的效率。

最后用 ARM 进行编译的时候，一般都不希望还有警告信息。

◆ **区分调试环境和手机环境**

如果应用需要定义在手机上才执行的过程，建议使用

```
#ifdef AEE_STATIC
```

```
//只能在手机环境处理的任务
```

```
#endif
```

的格式把手机环境和模拟环境分开，当然，使用

```
#ifdef _WIN32
```

```
//只能在 WINDOWS 上调试处理的任务
```

```
#endif
```

也是可以的。

◆ **代码检查**

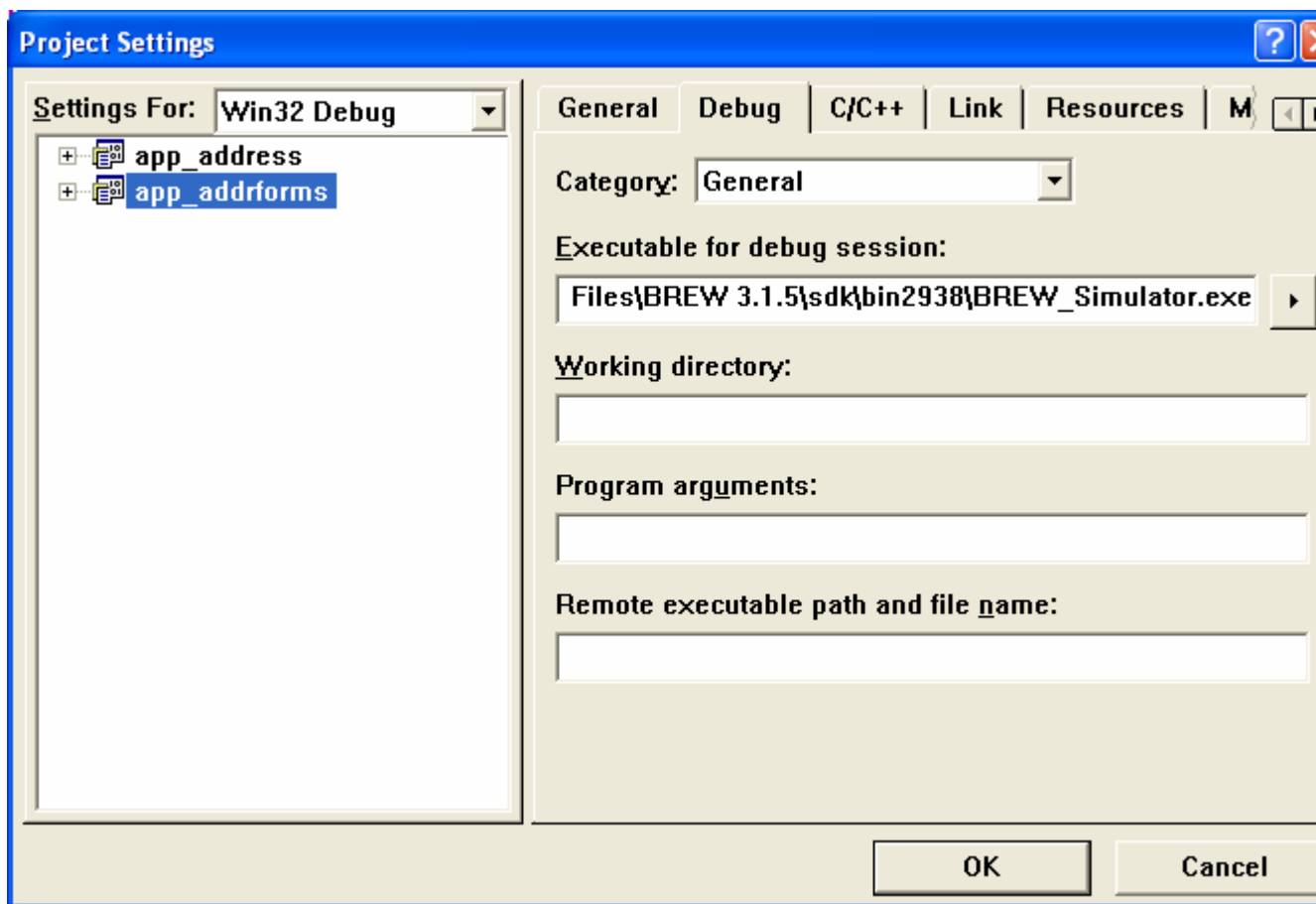
推荐使用 PC_LINT 工具检查应用代码，看是否有存在不合理或者可能产生问题的代码。这个工具对代码的检查非常严格，一般使用它进行检查可能把潜在的问题找出来。

◆ **设置模拟器应用**

VC6++ 编译产生的 dll 文件，需要使用模拟器调试模拟手机环境，项目中设置该程序的过程如下图。如果按默认的安装，那应该能在

C:\Program Files\BREW 3.1.5\sdk\bin\BREW_Simulator.exe

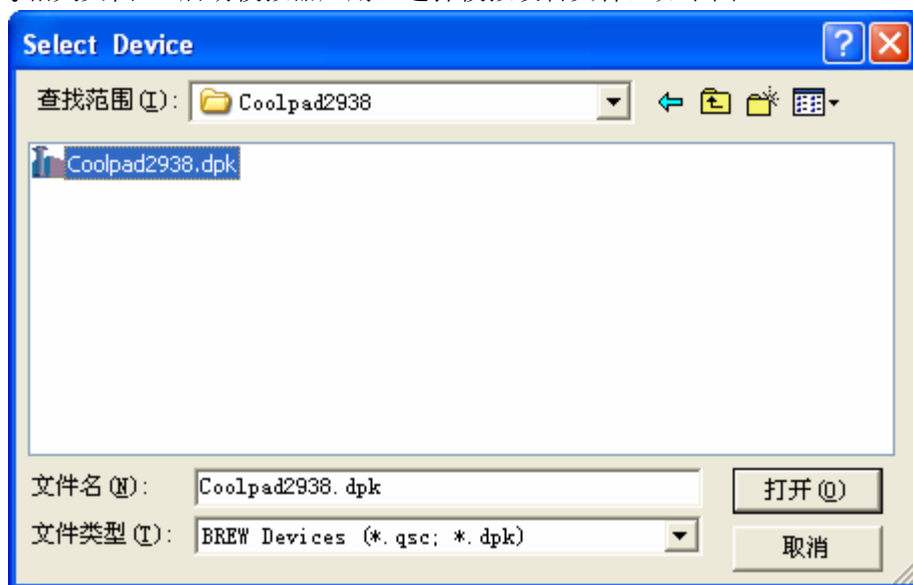
下找到该模拟器，如果不是默认安装，可以到相关目录找该模拟器应用程序。



设置 VC6++的模拟器调试应用

◆ 设置设备文件

模拟器显示屏幕大小，手机形状如何，都在设备文件中设置（如何设置或者产生设备文件，参考相关文档）。启动模拟器应用，选择模拟设备文件，如下图：



选择模拟器设备文件





默认的设备文件和 2938 手机的设备文件

◆ 模拟器调试应用

一般模拟器可以调试大部分手机应用内容，所以能在模拟上执行的代码，都需要在模拟器上执行，确保正确后再编译到手机上。

如果要调试一个应用，先在开发环境中把该应用的 mif 文件放到开发目录下，设置 mif 中该

应用的可见项；从 VC++ 中编译该应用并且直接启动模拟器，从模拟器菜单 (File->Change Applets Dir) 载入需要调试的环境目录，就能从调试器中看到这些调试的应用。结果如上图（如图 2938 手机的设备文件）。

如果全部设置正确，点击应用图标，就能启动应用，并且在 VC++ 中设置断点和调试信息，具体过程和 WINDOWS 桌面开发一样。

➤ 应用开发基本问题（初学者问答）

◆ 为什么启动不了应用

程序已经编译过了，dll 也已经产生，模拟器启动后，可找不到图标或者启动不了应用。

- 1: 检查应用目录大小写。
- 2: 开发环境目录下，检查 mif 文件是否存在、是否小写、是否设置了可见项。
- 3: 编译的 dll 是否小写、是否目录下有多个 dll
- 4: CLSID 是否正确、是否和其它应用 CLSID 重复。

◆ 为什么创建对象总是失败

应用已经启动，能进入应用程序入口，可创建 FORM 或者其它对象，总是失败。

- 1: 开发环境是否使用的是最新的开发包。
- 2: 开发环境下，是否存在 \forms 和 \widgets 目录（窗口和控件都在这 2 个目录下）。
- 3: 是否使用和开发包的 AEE.h 头文件、位置是否正确。
- 4: 开发环境的环境变量是否指到其它目录下。
- 5: 部分应用需要共享内存，是否该模拟器提供这个功能。

程序架构基本规范

➤ 程序结构标准化的需要

在 268 到 2938 的开发过程中，我们碰到了很多问题，特别是应用之间协同处理任务的时候，可能各个应用互相调用，彼此创建和释放的方式都不相同，在处理任务可能产生未知的问题，这些问题曾经很长时间都困扰着我们，因此建议今后应用都采用一种统一的程序架构，来解决这些问题。

现在各个应用结构和处理过程各有各的方式，我们列举程序中三个主要方面来说明现在存在的问题。

◆ 主程序结构不合理

有些已经在 2 系列中完成的应用，主程序数据结构不包含下级窗口的数据结构指针，而是包

含下级窗口的窗口指针，这在处理协同应用释放的过程，可能造成困难，因为有时异常中断处理事务的时候，需要能主动关闭所有已经创建的对象和窗口，包括几层窗口下的所有对象，而这个时候只能拿到下级窗口的指针。

下图是应用的一个数据结构：

```
typedef struct _Explore
{
    AEEApplet      a ;
    AEEDeviceInfo DeviceInfo;
    IRootForm*     rootForm;

    IForm*          mainForm;           //主窗体，目录窗体
    HandlerDesc     mainFormHandler;
    IForm*          inputForm;
    HandlerDesc     inputFormHandler;
    IForm*          renameForm;
    IForm*          propForm;
    IForm*          copytipForm;       //复制移动提示窗体
    HandlerDesc     copytipFormHandler;
    IForm*          selectdiskForm;    //选择目录窗体
    HandlerDesc     selectdiskFormHandler;

    MainForm*       MainWnd;
    InputForm*      InputWnd;
    RenameForm*     RenameWnd;
    PropForm*       PropWnd;
    TipForm*        TipWnd;
    CopyTipForm*    CopyTipWnd;
    SelectDiskForm* SelectDiskWnd;
}
```

一个不合理的数据结构

这个结构中包含了 IForm 的指针，同时也包含了该子窗口的数据结构（如 MainForm、RenameForm 等），显然包含 IForm 指针是多余的。有些应用的程序结构也是这样，并且使用过程不是保存这些子窗口的数据结构，而是保存这些 IForm 的指针，导致最后 AVK_END 的时候，处理麻烦。

◆ 窗口参数结构传递不合理

当每创建下一层窗口的时候，传入的参数都该是整个应用的数据结构，而不该是该窗口的结构；现在的 BREW 应用，因为可能从当前窗口中需要使用到其它窗口的数据；如果是传入整个应用对象，数据处理就容易多。

看过了很多 2 系列手机应用的代码，发现现在已经完成的很多应用，对象或者窗口数据结构的参数，很多是使用 pMe 来命名（估计都是受到 Demos 应用的影响），这本很正常，问题在于发现很多的应用中，每个窗口的数据结构都是 pMe，却每个地方表示的意义都不一样，虽然不会造成问题，但读起来比较困难。

下图是个明显的例子。

```
static void MainForm_Delete(MainForm * pMe) ;
static void SelectDiskForm_Delete(SelectDiskForm* pMe);
static void InputForm_Delete(InputForm * pMe) ;
static void PropForm_Delete(PropForm * pMe);
```

每个窗口数据结构都是 pMe，阅读有困难

写程序的开发人员一定能明白该代码，可如果转手 2 到 3 个人后，读起来就麻烦了很多。参数明确简单化，是我们应该提倡的。

下面是应用的一段代码：

```
int CDeleteForms_RunDeleteForm(CAddress *pMe);
int CDeleteForms_RunSelectForm(CAddress *pMe);
int CDeleteForms_RunSelfFormExt(CAddress *pMe);
```

统一 pMe 参数

显然，任何时候，这个 pMe 都代表一样的数据结构，读起来也就容易些。

◆ 应用释放所有窗口过程不合理

有些应用，窗口不多，所以就采用判断每个窗口是否存在，然后处理释放，这是通常的做法，但一般认为不够规范。

下图是一个应用释放窗口的过程，该过程主要产生在用户按下了 AVK_END 键时，把所有已经创建的窗口释放的过程。

```
if(pMe->renameForm != NULL)
{
    IFORM_Release(pMe->renameForm);
    pMe->renameForm = NULL;
}

if(pMe->selectdiskForm != NULL)
{
    IFORM_Release(pMe->selectdiskForm);
    pMe->selectdiskForm = NULL;
}

if(pMe->propForm != NULL)
{
    IFORM_Release(pMe->propForm);
    pMe->propForm = NULL;
}

if(pMe->rootForm != NULL)
{
    IROOTFORM_Release(pMe->rootForm);
    //pMe->rootForm = NULL;
}
```


不合理的释放过程

上图中的释放过程可能存在一个问题，开发人员可能不是很清楚释放的过程，因为就执行了一个

```
IFORM_Release(pMe->inputForm);  
pMe->inputForm = NULL;
```

就把窗口设置为 NULL 了；其实在上面 2 行代码中，并没释放窗口指针分配的内存。关键在上面代码中的（IROOTFORM_Release）函数，该函数将产生雪崩效应，把在根窗口（pMe->rootForm）上的所有窗口，按后进先出的顺序释放所有已经分配的内存，这个时候才彻底删除窗口。

我们推荐如下图一样

```
IFORM *pTopForm = NULL;  
pTopForm = IROOTFORM_GetTopForm(pMe->pRoot);  
while (pTopForm)  
{  
    DBGPRINTF("删除窗口!");  
    IROOTFORM_PopForm(pMe->pRoot);  
    IFORM_Release(pTopForm);  
    pTopForm = IROOTFORM_GetTopForm(pMe->pRoot);  
}
```

释放窗口过程

在把所有窗口都释放，包括协同应用中其它对象的窗口，然后在释放根窗口。

甚至有些应用，在释放的时候，开始就把根窗口（ROOTFORM）释放，这是不合理的，非常可能导致了协同应用处理窗口释放产生问题。

➤ 主程序数据结构

每个应用必须有一个大的全局数据结构，保存每个窗口数据结构的指针，还应该保存 AEEApplet 对象以及根窗口（ROOTFORM）的指针、全局数据对象以及应用名称等数据，如图示（数据结构关系图示）。

应用内部全局数据应该放在该数据结构中。

➤ 窗口独立数据结构

每个窗口必须使用一个独立数据结构，把该窗口使用的所有控件和数据结构都放在其中。建议该窗口的 FORM 放第一项，每控件和相关内容都放一起，比如 LIST 控件，和它一起的是 LIST 的 ITEM 项数据指针，LIST 显示的 STATIC 控件等，如图示（数据结构关系图示）。

➤ 窗口间参数传递

窗口之间的数据传递，必须全部使用大的全局数据结构来传递。从该数据结构，可以拿到所有窗口下的任何数据，如图示。

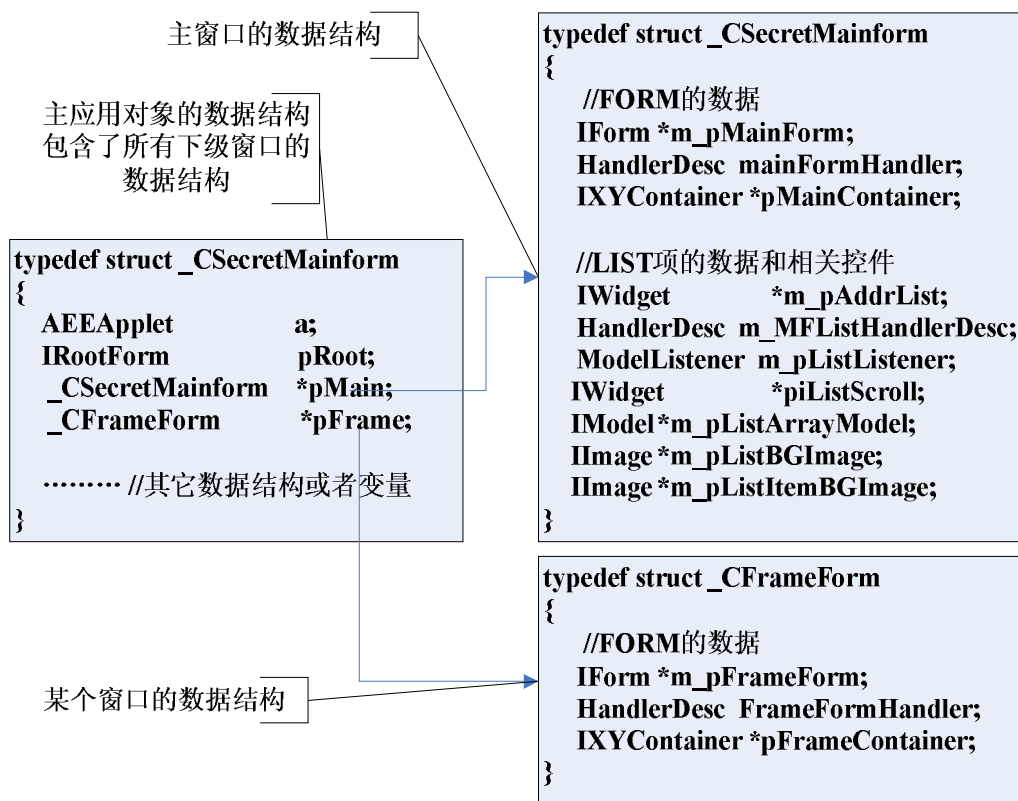
```

int CDeleteForms_RunDeleteForm(CAddress *pMe);
int CDeleteForms_RunSelectForm(CAddress *pMe);
int CDeleteForms_RunSelfFormExt(CAddress *pMe);

```

➤ 程序结构图示

下 2 图是程序结构的关系图和实例图示



数据结构关系图示

```

//主程序的结构
typedef struct _CSecretData
{
    AEEApplet          a;
    IRootForm          *rootForm;

    CSecretMainForm    *pMain;           //主窗口指针
    CSecretFrameForm    *pFrame;         //第2层窗口指针
    ICPrivateMode       *pICPrivateMode; //私密模式对象指针

    AECHAR              m_szHintTitle[32]; //提示标题,中文英文
    AECHAR              m_szHintText[80];  //提示内容
    char                m_szDebugInfo[160]; //调试信息

    char                m_szSecretBar[60];  //BAR文件
    AECHAR              m_szMainCaption[32]; //窗口标题, 中文英文
    AECHAR              m_szFrameCaption[32]; //

    uint8               dwAppIndexID;      //应用管理的ID号
}CSecretData;

```

数据结构实例图示

数据结构规范

➤ 数据结构名称定义

◆ 结构的名称

对内部使用的数据结构，建议结构名称前，全部加应用名称的前4个字符或者全名称，保证该结构不会和其它应用定义的结构重复定义。

很多应用都定义了 ListItem,如联系人为了自己定义的结构前加了 CADDRListItem,就是按上面的规则定义的结构。

```

//大数据量的时候，为了优化内存，使用下面结构
typedef struct _CADDRListItem
{
    AECHAR          *m_pMainTel;           //电话号码
    AECHAR          *m_pMainName;          //名称
    AECHAR          m_szLocaler[5];        //快速定位
    AECHAR          m_szLocate9Key[5];     //9键定位，1..9字符
    boolean         m_bHidden;             //私密状态
    uint16          m_dwRecOID;             //记录保存的表ID
    uint16          m_dwSavePos;            //保存在哪个表中位置
    uint16          m_dwGroupID;            //组ID保存下来
    boolean         m_bSelected;           //该项是否被选择中了
    IImage          *m_pImage;             //分组，SIM图标显示
    AECHAR          *m_szTitle;            //详细信息中用来保存标题的
    uint16          m_dwIndex;             //该项在队列中的位置
}CADDRListItem;

```

CADDRListItem 数据结构

我们不止 10 次，看到有些应用定义的数据结构名称和其它应用定义的数据结构名称一样，甚至和系统定义的数据结构名称一样，存在这些不可思议的现象。

◆ 公共的数据结构

公共使用的数据结构，建议定义在单独的头文件中；在开发环境中，可以直接使用该文件，但建议还是不要放到当前应用的目录下，因为如果这样，可能未来更新该文件不及时而且又再次包含新的文件，非常可能产生未知问题。

◆ 曾经的问题

1：重复定义：我们曾经在编译一个应用的时候，发现在模拟器下，在应用中该结构多次重复定义，很多个头文件中都定义了相同的数据结构。编译的时候，真的不知道哪个数据结构被引用了，虽然编译过了，数据结构也都一样的，程序不存在问题，可需要把它们编译成功非常麻烦。

2：定义了和系统一样的数据结构，这是最不应该的。

3：名称重复：在应用 A 中定义了 LISTITEM 数据结构，在应用 B 中也定义了 LISTITEM 数据结构，ARM 编译的时候，会有编译不了的编译错误。

➤ 数据结构中内存注意事项

◆ 中英文版本内存不一样。

考虑到英文版本的时候，宽字节字符个数比中文多，如果数据项不是很多，比如小于 50 项，可以直接在结构中分配英文版本需要的字符个数，如果数据项很多，比如可能达到 3000 项，那建议动态分配内存，并且注意优化问题。

◆ 大数据量时内存重复使用问题

如果数据量很大，并且反复使用，如果动态内存分配，显然频繁的分配不是个好的主意，因此建议在程序开始时就分配，然后指针重复指向该已经分配的内存区。

比如在上图（**CADDRListItem 数据结构**）中，结构中变量 `m_pMainTel` 会被重复使用，开始就分配了一个这样的内存：

```
AECHAR          m_szMainAddrTel[12][ABC_Tel_MAXLEN+2];    //电话号码
AECHAR          m_szMainAddrName[12][ABC_Name_MAXLEN+2];    //姓名
```

预先分配内存，避免重复分配

不管程序中有 1 项还是 3000 项，显示出来最多的就这 12 项，程序中就是重复使用这些内存，内存分配次数明显就少了很多。

代码编码规范（简要）

➤ 编码基本事项

每个程序员的编码方式都不一样，我们不对编码做特别的规定，但总的一个规则就是，希望能都按标准的方式来命名参数和变量，让代码更容易阅读和理解。

不规范的命名方式，代码不容易读，也可能几个月后，自己读起来也不理解。在代码中，建议多写说明和注释或者调试信息，保证读者能明白每行代码的意义。

我们特别强调代码的可读性，不可读的代码读起来令人难以接受，甚至产生厌恶的心情，对参数命名总是建议名称能表达参数代表的意义。

代码规范，函数清晰整洁，注释说明清晰明了，调试信息准确无误，将让后来者更容易接受维护代码，这应该是一个程序员应该做到的基本编码规则。

➤ 示例

这里给出一段代码，应该能说明代码规范的含义，这段代码滥用（`if、else`）导致代码无法阅读，该代码来自其他开发人员移交的一个应用中的一段代码，没做任何修改。

这是个读取文件的过程，修改前和修改后的，不规范的编码过程（程序中原始代码）如下：

```

static int yl_readfile(IShell* pshell, char* filename, char** pBuffer, unsigned int* ilen)
{
    IFileMgr*      pFileMgr      = 0;
    IFile*          hFile         = NULL;
    FileInfo*       lpInfo        = (FileInfo*)yl_malloc(sizeof(FileInfo));
    unsigned int     ireresult     = 0;

    if(SUCCESS == ISHELL_CreateInstance(pshell, AEECLSID_FILEMGR, (void**)&pFileMgr))
    {
        if(SUCCESS == IFILEMGR_GetInfo(pFileMgr, filename, lpInfo))
        {
            hFile = IFILEMGR_OpenFile(pFileMgr, filename, _OFM_READ);
            if(hFile)
            {
                if(ilen)
                    *ilen = lpInfo->dwSize;

                *pBuffer = yl_malloc(lpInfo->dwSize + 12);
                if(!*pBuffer){
                    ireresult = 4;
                }else{
                    IFILE_Read(hFile, *pBuffer, lpInfo->dwSize);
                    IFILE_Release(hFile);
                }
            }else{
                ireresult = 1;
            }
        }else{
            ireresult = 2;
        }
        IFILEMGR_Release(pFileMgr);
    }else{
        ireresult = 3;
    }
    FREE(lpInfo);
    return ireresult;
}

```

编码不规范

这段代码很乱，名称乱、返回值不规范等一大堆问题；最大问题在于滥用 else 导致代码不可阅读。

修改后代码（函数名称、参数、变量和返回值都没修改，这些值这里不讨论，由于函数图片一页放不下，这里入参和内存分配后 MEMSET 没写出来，这里只考虑代码可读性，下图代码中每花括号后面，建议空一行，代码行间不要太拥挤）：

```

static int y1_readfile(IShell* pshell, char* filename, char** pBuffer, unsigned int* ilen)
{
    int32          dwResult      = SUCCESS;
    IFileMgr*      pFileMgr      = 0;
    IFile*         hFile         = NULL;
    FileInfo*      lpInfo        = NULL;
    unsigned int    iresult       = 0;

    //入参检查.....
    DBGPRINTF("开始创建对象.");
    dwResult = ISHELL_CreateInstance(pshell, AEECLSID_FILEMGR, (void**)&pFileMgr);
    if (SUCCESS != dwResult)
    {
        DBGPRINTF("对象创建失败.");
        iresult = 3;
        goto _ReadFileError;
    }
    lpInfo = MALLOC(sizeof(FileInfo));
    if (NULL == lpInfo)
    {
        DBGPRINTF("内存分配失败.");
        iresult = 3;
        goto _ReadFileError;
    }
    dwResult = IFILEMGR_GetInfo(pFileMgr, filename, lpInfo);
    if (SUCCESS != dwResult)
    {
        DBGPRINTF("获取文件信息失败.");
        iresult = 2;
        goto _ReadFileError;
    }
    hFile = IFILEMGR_OpenFile(pFileMgr, filename, _OFM_READ);
    if (NULL == hFile)
    {
        DBGPRINTF("打开文件失败.");
        iresult = 1;
        goto _ReadFileError;
    }
    *ilen    = lpInfo->dwSize;
    *pBuffer = MALLOC(lpInfo->dwSize + 12);
    if(NULL == *pBuffer)
    {
        DBGPRINTF("内存分配失败.");
        iresult = 4;
        goto _ReadFileError;
    }
    dwResult = IFILE_Read(hFile, *pBuffer, lpInfo->dwSize);
    if (dwResult != lpInfo->dwSize)
    {
        DBGPRINTF("读文件失败(%d,%d).", lpInfo->dwSize, dwResult);
        iresult = 5;
        goto _ReadFileError;
    }
    iresult = 0;
    DBGPRINTF("读文件完成.");
_ReadFileError:
    if (hFile) IFILE_Release(hFile);
    if (pFileMgr) IFILEMGR_Release(pFileMgr);
    FREEIF(lpInfo);
    return iresult;
}

```

一般的编码规范

➤ 调试信息问题

在上图的代码中，打出了好几行调试，这是必须的；在 BREW 应用开发调试阶段，经常有可能出现手机系统崩溃，唯一能给我们知道产生系统崩溃的原因，就是调试信息。这些调试信息，如果不是每秒几十行，一般不会对系统速度产生影响。

想在 BREW 终端上写好稳定健壮的应用程序，调试信息是必不可少的，因为唯有它，才能知道调试中系统崩溃前一刻，程序可能跑到哪一行。否则，只能使用硬件调试器跟踪，那将是更为麻烦。一般 UI 应用崩溃，都可以使用调试信息找到应用崩溃问题。

◆ DBGPRINTF 调试信息

建议调试信息尽可能做到 5-10 行代码，至少有一行调试信息打印出来，不要太多，但也不要太吝啬。

◆ 写文件调试信息

有些应用可能需要使用写文件的方式保存日志，建议在关键地方写日志，文件到一定大小再清掉日志重新写。太大的日志会影响系统执行速度。

◆ 调试信息不应该放的地方

1: 窗口的函数中

```
boolean CAddrForms_ClassFormEventHandler(void *po, AEEEvent evt,
                                          uint16 wParam, uint32 dwParam)
{
    CAddress* pMe = (CAddress*)po;

    DBGPRINTF("窗口事件处理!"); //这里不应该打印调试信息。

    switch(evt) //事件处理
    {
        case EUT_KEY:
            switch(wParam)
            {
                case AUK_LEFT:
                case AUK_UP:
                case AUK_RIGHT:
                case AUK_DOWN:
                    break;
            }
    }
}
```

不应该出现调试信息的地方

窗口的进入点，在程序执行过程中，会收到很多消息，特别是根窗口的入口函数，导致影响到其它调试信息的查看。

2: 列表的显示函数中(XXXX_Adapter)


```

void CAddForms_ListFormShowDataAdapter(void *pUnused, void **ppValueIn,
                                       int nLen, void **ppValueOut, int *pnLenOut)
{
    CDataItem      *pItem = NULL;
    CAddress        *pMe   = NULL;

    DBGPRINTF("进入显示函数"); //这里不应该出现调试信息
}

```

会频繁打印调试信息

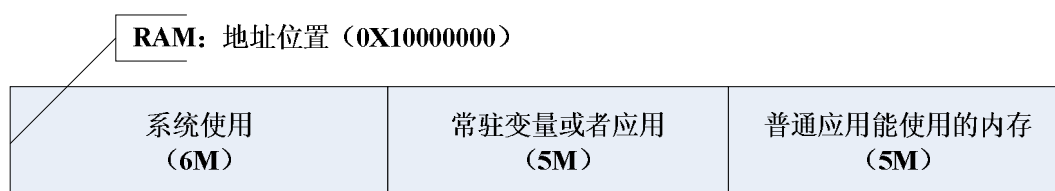
在显示函数中，记录每显示一条，就会打印出调试信息，也很影响其它调试信息的查看。

程序内存和堆栈

➤ 程序总的可用内存

◆ 总的内存

为了节约终端总体成本，终端的内存一般都非常有限，一般为 8M 或者 16M，很少有 32M 以上的内存，所以节约使用内存是开发应用非常重要的事情；就象现在 2 系列手机本身 RAM 就 16M 内存，在系统启动后，大概非常驻应用可以使用的内存为 5M 左右，所以非常驻应用的内存将是非常紧张，大部分应用在启动开始就需要做内存检查。如果内存不足，应用应该自动退出。



◆ 误区

我们曾经看到有些应用，操作中一下就分配 1M 甚至更多的内存，这是导致系统崩溃的前兆。

➤ 内存需求空间检查

◆ 应用需求内存检查

应用启动函数的第一件处事情，就是应该判断内存是否足够；一般情况下，判断有 (1024*1024*2) 字节内存就可以了；但有些大应用，如联系人，日程可以按最多可能启动的窗口数，每个窗口 300K 的总数来判断（今后窗口如果使用共享内存，这个值可能比较小些）。如果不足内存，调用桌面提示函数后，返回 EFAILED 结果提示用户启动失败。

◆ 接口需求内存检查

以对象的形式或者接口的形式提供给其它应用的情况，尽可能在入口前也进行检查内存，如果内存不足，这种情况需要直接提示后退出对象，而不是调用桌面的提示窗口，如拨号盘调用保存联系人接口，如果内存不足，联系人接口会提示后返回拨号盘窗口。

提示

如果控件和窗体都做成共享内存方式，那可能可以缩小些需要的内存。

➤ 函数内栈空间问题和错误 rex.c 841

终端开发特别需要注意栈空间问题，这里对它可能产生的情况做个说明，了解栈空间大部分情况下和 WINDOWS 开发不一样的地方，和操作可能产生的失败情况。

◆ 著名的 841 错误

系统总的栈空间为 64K，如应用使用栈空间超过 64K，将产生著名的 841 错误（在 rex.c 文件 841 行），这个错误直接是系统崩溃；一般情况下，该错误都可以通过硬件调试器找到；

堆栈溢出的错误可能，是有千百种，只能说，在写代码的时候，必须时刻注意这个问题，尽量避免。为避免产生这个错误，一般建议应用需要做些如下处理。

◆ 使用数组的情况

函数中的变量，大多数情况下应该预先声明，尽可能使用指针来分配内存，有时为了方便，使用数组，是可以方便些，但由于手机栈空间的限制（将在下面讲述堆栈溢出问题），对小数组（20 个字节以内），可以直接声明使用的。大的数组（如 100 个字节以上），在压栈过程需要更多注意。

◆ 数组改用指针

建议对 20 个字节以上的数组，都尽可能使用指针分配内存，并且分配后需要检查内存是否分配到。对分配 10K 以上的内存，这方面特别需要检查。

函数中尽量少使用大空间数组，而是使用指针分配内存空间；对大于 20 个字节的结构或者数组，都该使用 MALLOC 来分配空间。如把原来声明

```
AECHAR szTemp[10];
```

的过程改为

```
AECHAR *szTemp = NULL;
```

```
szTemp = MALLOC(10*sizeof(AECHAR));//检查内存是否为空后再使用
```

```
FREEIF(szTemp);
```

◆ 使用异步消息

函数处理中尽量使用异步消息机制（ISHELL_PostEvent）来处理。如果函数调用的其它函数非常多次，也是有可能导致堆栈溢出。这种处理的主要目的，就是把原来同步处理的过程，分成几个异步消息处理，避免频繁压栈导致栈空间溢出。

提示：

不要使用 ISHELL_SendEvent 来发送消息，该函数是同步函数。

◆ 入参使用指针

函数不能使用大数组做为参数传入。如果参数数组很大，比如 8K，可能产生的情况是直接崩溃或者提示堆栈溢出错误，并且使用硬件调试器也很困难找到问题所在。

◆ 参数错误例子

下面是写 VCARD 文件的函数，VCARD 结构声明：

```
//联系人编码，解码数据结构 8k+3k栈空间，该结构建议最好HALLOC空间，否则可能栈空间不够
typedef struct _address_LKM_UCARD
{
    void * pContactInfo; //联系人信息基本结构(话机或者sim卡)
    unsigned char Image[ ABI_IMAGE_MAXLEN + 1 ]; //图像流 8k栈空间
    int nImageSize; //图像流大小
    int groupNum; //联系人归属分组个数
    TCHAR groupNameArray[LKM_GRP_MAXRECORD_NUM][ABC_Name_MAXLEN]; //联系人归属分组数组
}LKM_UCARD;
```

写文件函数声明：

```
int32 CAddr_WriteVCardFile(CAddress* pMe,LKM_UCARD pVCard,char
*szFileName,int32 *dwLen);
```

这个函数的参数是个数组，是应用崩溃的主要原因；这个函数在模拟器上跑很多次都不会有问题，因为 WINDOWS 的堆栈不存在限制问题，但在手机上执行该函数，有时成功有时失败，就是因为太大的参数结构导致堆栈出了问题。

中英文版本资源规范

➤ 版本目录和资源 ID

我们在目录规范中例举了目录文件和资源文件目录，这里我们将对资源文件的数据进行些规范。

◆ 版本资源文件

中英文版本的资源文件，名称必须相同，资源 ID 也必须相同，唯有资源文件放置的目录不同。中文版本，资源文件放在\zhcn 目录下，英文版本资源文件放在\en 目录下，如果还有其它版本，需要建立相关的目录放置该版本资源文件。

◆ 资源 ID

中英文资源文件中绝不允许 2 个版本的资源 ID 不一样，因为引用的 BRH 头文件中定义了 ID 值，如果不一样在版本切换时，可能载入不了该资源。

◆ 载入过程

程序中也不允许使用 FOR 循环中通过 (IDS_STRING_ID+i) 的方式载入资源。如

```
for(i=0;i<10;i++)
```

```

{
    dwLoadResID = IDS_STRING_ID+i;
    //根据 dwLoadResID 连续载入资源。
}

```

本身过程没错误，问题再于 BREW 载入资源函数 (ISHELL_LoadResString) 可能存在 BUG，有时需要打乱资源 ID，再重新载入。

◆ 加速载入过程

很多应用在载入资源文件的时候，都会去检查下资源文件 ID 对应内容的大小（使用 ISHELL_GetResSize 函数获取大小），再根据这个大小分配内存，这是可靠的过程；

但一般情况下，程序启动就必须载入的资源，却也是这样处理，这就不是很合适；一般启动就载入的资源，会一直占用直到程序关闭，也一般都知道资源固定大小，所以这样的资源，应该启动就直接按大小分配，加速载入过程，优化启动速度。

➤ 调试环境和手机环境的资源

如果在调试环境下，可以直接通过宏定义，把资源文件直接载入调试，如下图（中英文版本资源）。如果是手机环境下，按如下处理都可以：

◆ 应用直接替换

可以判断当前手机的中英文版本，应用程序代码中根据版本转换文件路径，然后载入资源。

◆ OEM 层替换

可以直接使用版本规定的目录，如下图的 (langdir) 让 OEM 层在函数 (ISHELL_LoadResString) 中去替换到相应目录下找资源文件。如下图：

```

#ifdef _WIN32
#define ADDRFORMS_RES_FILE "fs:/mod/app_addrforms/en/app_addrforms.bar"
#define ADDRFORMS_RES_CONST "fs:/mod/app_addrforms/en/app_addrconst.bar"
#else
#define ADDRFORMS_RES_FILE "fs:/mod/app_addrforms/{langdir}app_addrforms.bar"
#define ADDRFORMS_RES_CONST "fs:/mod/app_addrforms/{langdir}app_addrconst.bar"
#endif

```

中英文版本资源

模拟器上已经测试好资源文件，就该把这些文件放到手机上，可以使用以下 2 种方法。

◆ 编译到 BIN 文件

把资源文件产生的 BAR 文件，编译到 BIN 文件中，在手机中产生虚拟的资源文件；这就需要在相应 min 文件中，把中文和英文的版本 BAR 文件都烧到手机中。根据每个应用的 min 文件，增加 min 中项的内容，如联系人增加的 min 项为：

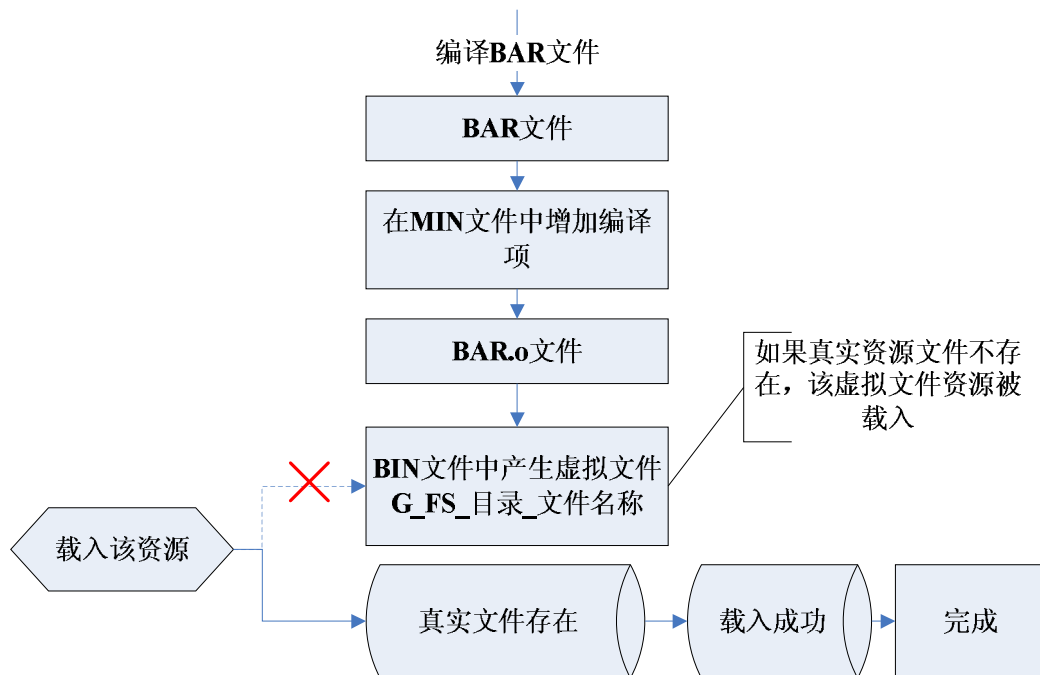
```
CONSTRESFILE_LANGFILES+=\
```

LANG/app_addrforms.bar|fs:/mod/app_addrforms/LANG/app_addrforms.bar

ARM 编译的时候，会把 LANG 换成相应版本的目录（如果是可以中英文切换的，那中文和英文的 BAR 文件都将编译到 BIN 文件中），这样 BAR 文件都烧入到手机中。

编译完成后，系统中将产生一个文件，格式为 G_FS_目录_文件名称；这是个虚拟文件，如果该真实存在，那这个虚拟文件将不会被使用。

整个过程如下图：



虚拟文件和资源文件的载入过程

◆ 下载到手机目录

也可以使用下载程序，在其中增加下载项，把资源文件下载到手机上。

◆ 优缺点

通过编译成 BIN 文件烧到手机上，有优点就是 BIN 的载入速度快（因为在 NOR 上，启动时被映射到虚拟文件中），但缺点是增加了 BIN 文件的大小，虽然 BIN 文件一般是有限制的，但一般都采用该方式。

把 BAR 直接下载到手机，优点是文件系统的空间都是足够的，并且启动速度也很快，但不如在 BIN 文件中安全（如可见的 BAR 或许有可能被用户删除的情况）。该采用哪种方式，看具体情况而定。

➤ 资源不可采用的方式

◆ 错误的资源处理方式

见过一些开发人员写的代码，只能说是忍无可忍的，代码来自真实的程序。

```

{
    AECHAR wcYourPhone[7] = {0x60A8, 0x7684, 0x624B, 0x673A, 0x4E0A, 0x6709, 0x0};
    AECHAR wcServer[6] = {0x670D, 0x52A1, 0x5668, 0x4E0A, 0x6709, 0x0};
    AECHAR wcContanct[4] = {0x8054, 0x7CFB, 0x4EBA, 0x0};
    AECHAR wcCalendar[3] = {0x65E5, 0x7A0B, 0x0};
    AECHAR wcItem[2] = {0x6761, 0x0};
    AECHAR wcDouHao[2] = {0xFF0C, 0x0};
    AECHAR wcJuHao[2] = {0x3002, 0x0};
    AECHAR wcBackup[3] = {0x5907, 0x4EFD, 0x0};
    AECHAR wcRestor[3] = {0x6062, 0x590D, 0x0};
    AECHAR wcFinish[5] = {0x5B8C, 0x6210, 0x3002, 0x0};

    AECHAR* wcDispStr = (AECHAR*)MALLOC(30 * 2);
    char cCount[6] = {0};
    AECHAR wcCount[6] = {0}; //客户端数据
    AECHAR wcCountServer[6] = {0}; //服务器端数据

    MEMSET(wcDispStr, 0, 30 * 2);
    if(E_BACKUP_CONTACT == pMe->b->eBackupContent)
    {
        WSTRCPY(wcDispStr, wcContanct);
    }
    else
    {
        WSTRCPY(wcDispStr, wcCalendar);
    }
}

```

错误的资源处理方式

想来也就开发人员自己知道这段代码的意义了。先不说堆栈问题，也不说代码十分不规范问题，就说资源，没中文说明，就说那些 UNICODE 代码，谁有办法知道什么意义，英文版本重新再写这段代码？

这样的编码方式非常不可取，希望今后我们的代码中不再出现这样的编码格式。

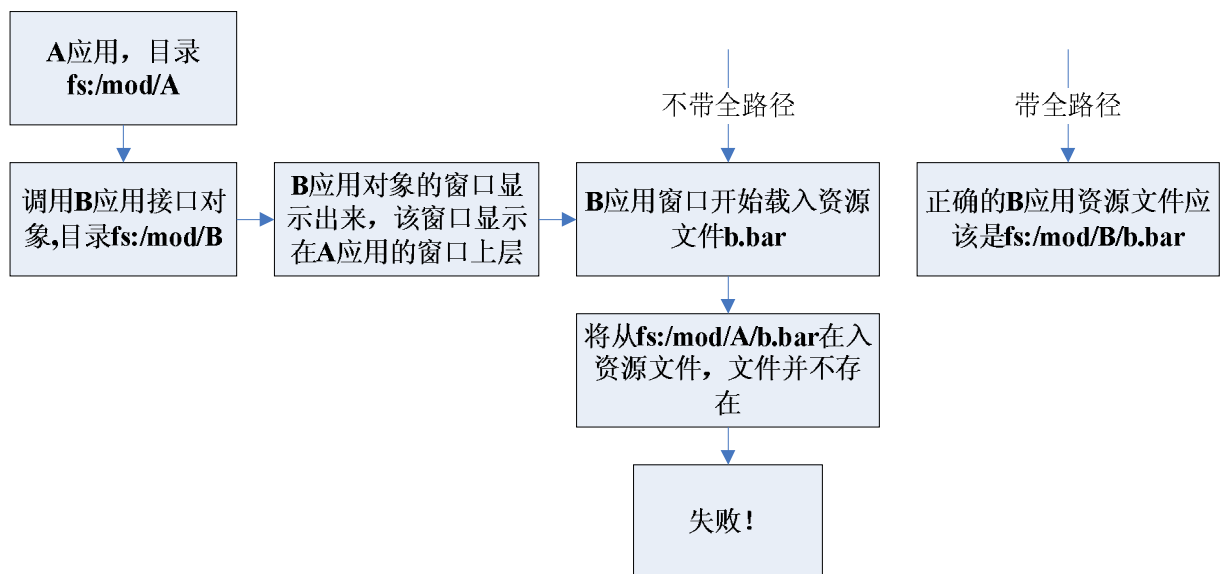
➤ 资源载入失败（ISHELL_LoadResString）现象。

资源载入，一般情况下都不存在问题，但曾经发生一些想不到的问题，这里也把它都列出来。

- ◆ 文件路径错误。
- ◆ 数据缓冲区内内存分配太小。

一般情况下使用（ISHELL_LoadResString）载入资源，目录错误和内存分配太小产生的载入失败，在开发过程中都很容易找出来；有时为了加快载入速度，不使用 ISHELL_GetResSize 来得到资源大小，而是分配足够的空间，偶尔分配不足也可能产生这样的问题。

应用找不到资源文件的问题，也可能在协同应用中产生出来，这就是为什么如果是协同处理事务的应用，目录和资源忘记都需要按全路径来定义的原因，如下图。



载入资源应该使用全路径

◆ 系统内部解析错误。

这个错误产生的过程非常奇怪，并且机会非常小，但确实发生过；

有个资源文件 300K，有文字和图片，把程序下载到几十台手机后，发现偶尔有一台手机的菜单资源文件不能载入（其它手机都正常），并且是连续的几个资源 ID 无法载入，通过反复测试，发现全部的资源文件中，载入失败的资源 ID 都是一段一段连续的，出现后产生的几率也是小于千分之一。

比如：

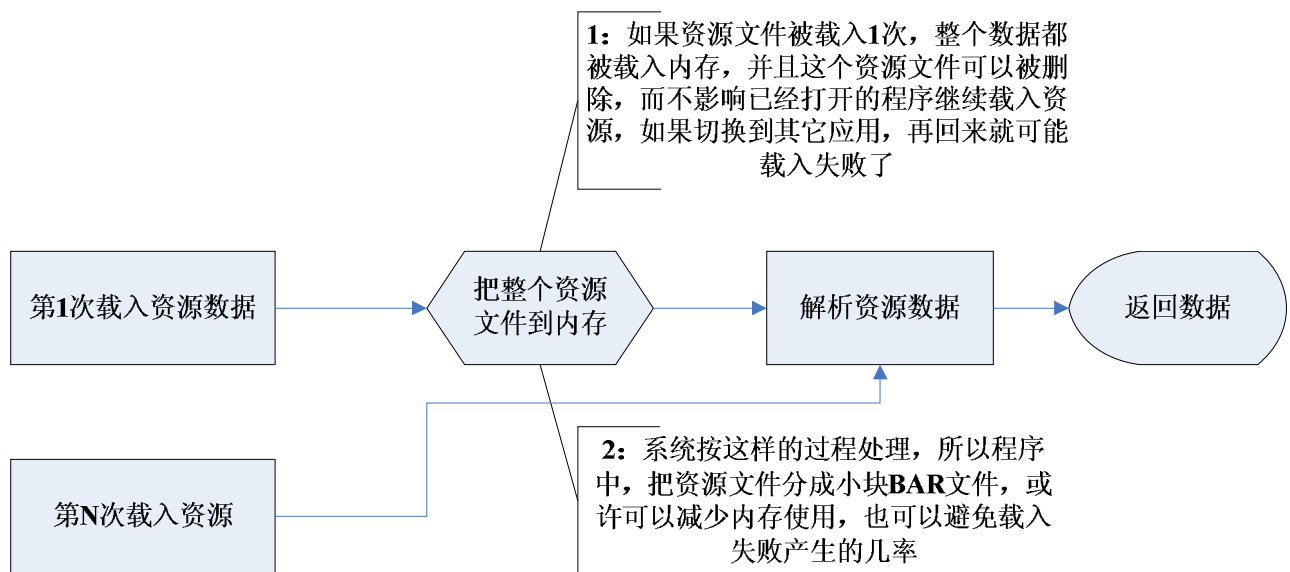
从 2001 到 2005 载入成功，从 2006 到 2010 失败，2011 到 2015 成功，2016 到 2018 失败。

当然，数百个文字资源 ID 可能一共就 10 来个载入失败。

产生这样的错误后

- 1：重新启动机器，可能就正常了。
- 2：切换到其他应用，再打开这个应用，也可能正常的。

分析了 BREW 资源载入的过程，大概入下，如图：



所以，大概是 BREW 对文件解码或者资源解析产生错误，因为失败后，函数返回结果是 0.

由于 BREW 对这方面失败产生的原因没有任何文档，产生的几率也非常小，所以为什么失败，也不是很清楚的，但只能通过某些方式避免这样的问题产生。

- 1: 把程序中一个应用的所有图片资源文件做成 1 个 BAR 文件，文字资源做成另外的 BAR 文件。
- 2: 产生了这样的问题，把资源顺序打乱，重新产生 BAR 文件，也许也能解决。
- 3: 把每个窗口的资源文件做成 1 个 BAR 文件，频繁载入次数，减少产生错误几率。

程序 CLSID 规范

➤ CLSID 是什么？

CLSID 是 BREW 系统对每个类对象维护的一个 ID 值，该值不可重复，系统中必须唯一。

在 BREW 应用中，每个控件、窗口或者应用，都必须指定一个 CLSID，该 CLSID 被编译到 MIF 文件中，操作系统将在启动的时候，将从所有的 MIF 文件中把所有的 CLSID 都读出；应用创建对象的时候，系统将通过类对象的 ID 找到相应的对象。

➤ CLSID 的定义

对于一般测试应用，可以指定任何 CLSID；但在商业版本中，每个应用的 CLSID 都必须集中管理，不能自定义 CLSID；动态应用的 CLSID 需要向 QUALCOMM 申请。

CLSID 一般以 16 进制形式存在，如 `#define AEECLSID_ADDRFORM 0x01500013`

➤ CLSID 和 BID 文件的位置

所有的 CLSID 都放在对应的 BID 文件中，一般情况 CLSID 和 BID 文件的名称，都使用大写，一个 BID 文件中可以定义多个 CLSID。

现在我们的应用，每个 CLSID 对应的 BID 文件都放在当前应用目录下，这给协同应用的引用过程造成困难，所以今后将考虑把所有的 CLSID 放到 BREW 系统 AEE 目录下，统一定义一个 CLSID 的头文件。

➤ 应用引用 CLSID

如果我们已经定义了一个包含所有 BID 的头文件，那直接引用该头文件即可；否则需要把 BID 文件加入头文件。

➤ CLSID 错误的做法

现在很多应用中，都发现为了方便快捷，直接定义了一个该 BID 对应的值，而不引用对应 BID 的文件（因为麻烦），这种做法是非常错误的。

比如，联系人的 BID 文件是 AEEIID_ADDRFORM.bid，该文件包含的联系人 BID 值为 0x01500013，某应用为了方便，直接定义了 `#define ADDRFORM 0x01500013`；然后使用该值创建联系人对象。

虽然这样的做法能创建成功，但这样的做法不能接受；如果今后我们还发现有这样的做法，希望能给予重罚。

经典的例子，如：

```
#define AEECLSID_UDISK          0x01100027
#define AEECLSID_APP_BREWMEDIAPLAY 0x0110002b
#define AEECLSID_APP_XREADER    0x01100049
#define AEECLSID_ZIPDEMO        0x0110005c
#define AEECLSID_USERBACKUP     0x01100051
#define AEECLSID_AUTOBACKUP     0x01100059
```

错误的 CLSID 引用

窗口和事件处理

➤ 应用程序组成和事件处理

◆ 基本组成

基于 BUIW 的应用，都采用图形用户界面，应用开发中主要使用下列内容：

- 1: 根窗口 (ROOTFORM)。
- 2: 窗口 (FORMS)。
- 3: 控件 (WIDGETS)。

根窗口 (ROOTFORM) 是一个抽象窗口，不做为窗口直接显示给用户，但根窗口 (ROOTFORM) 维护着窗口队列列表，是 BUIW 应用必须拥有的；每个应用只需要一个根窗口。

◆ 窗口消息和事件

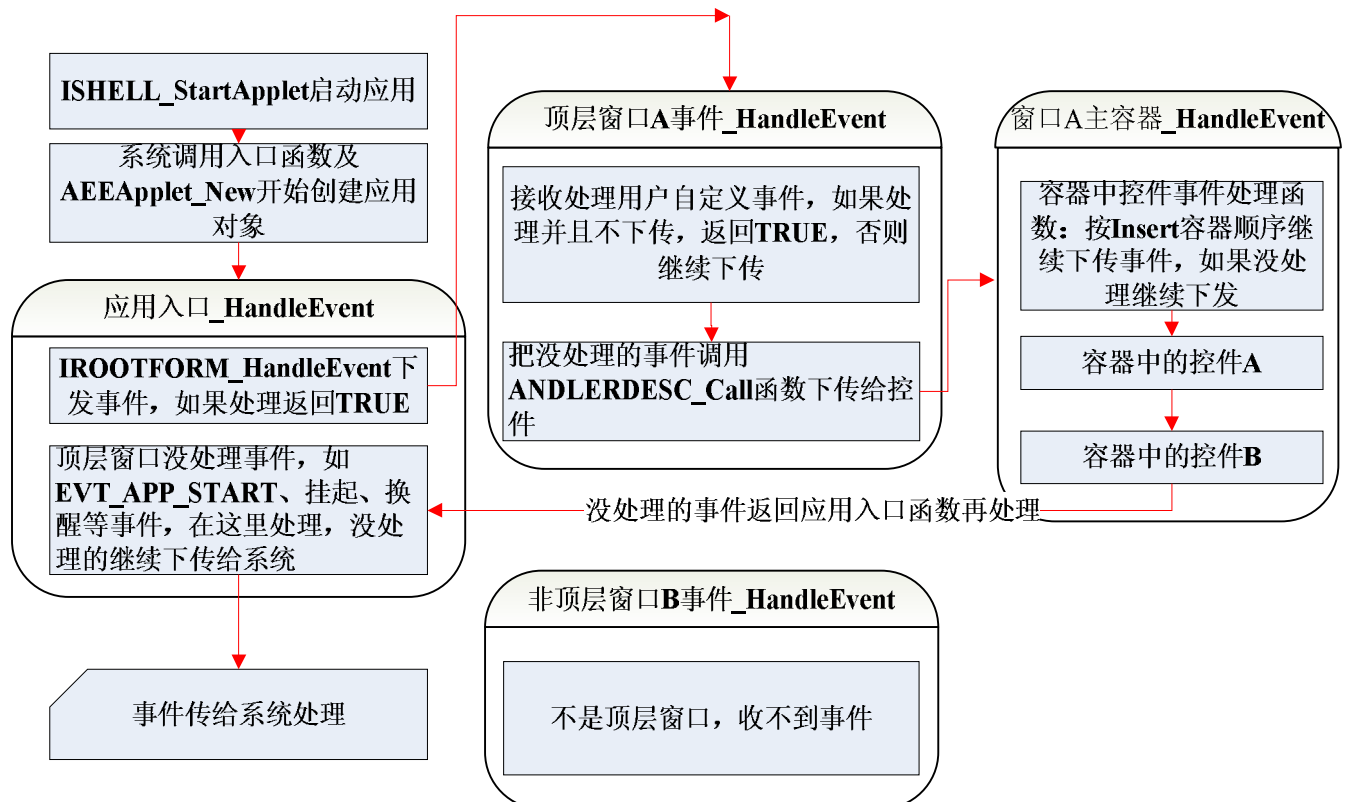
窗口 (FORMS) 和 WINDOWS 的窗口基本一样，窗口机制也和 WINDOWS 基本一样；每个窗口都需要有一个事件传递处理函数（一般按 XXX_EventHandler 命名，该函数主要是为取代窗口默认处理函数，让用户能在事件传递过程中，主动获取消息；

消息先从根窗口传递到顶层窗口的处理函数 (XXX_EventHandler)，再继续下传到每个控件 (WIDGET)，如果不是顶层窗口，将收不到应用发来的消息，该消息有可能被抛弃掉。

窗口采用消息处理过程，如果消息已经被处理，返回 TRUE，表示系统不需要再继续下发该消息，如返回 FALSE 表示需要系统进行下发消息，直到消息被处理；应用也可以处理事件后返回 FALSE，让系统继续把消息下发给下层控件或者系统默认处理函数。

◆ 事件传递过程

一般情况下，事件的传递过程如下图：



BUIW 事件传递 (图中事件没被处理的传递过程)

➤ 创建根窗口（ROOTFORM）

◆ 创建根窗口

- 《1》 创建 ROOTFORM 对象 的 CLSID 为（AEECLSID_ROOTFORM）。
- 《2》 一个 BUIW 应用只能有一个根窗口，根窗口本身不能放置控件，只能把窗口放在它的上面。
- 《3》 所有应用的窗口都必须放在（ROOTFORM）上才能显示出来。
- 《4》 根窗口被创建后，引用计数为 1。在根窗口（ROOTFORM）上 PUSH 窗口（如窗口 A），不会家根窗口的引用计数，但会增加（如窗口 A）的引用计数。
- 《5》 如果没特殊需求，不要主动增加根窗口的引用计数。

➤ 释放根窗口

◆ 释放根窗口和注意事项

根窗口的释放函数为（IROOTFORM_Release），当根窗口应用计数为 1 时（除非主动增加，否则都是为 1），根窗口对象释放时彻底被删除。

现在应用一般存在两种释放根窗口的方式，第一种在所有窗口释放前释放根窗口（这种处理方式不推荐），第二种方式是在所有窗口被释放后，再释放根窗口。第一种释放方式将使所有窗口的应用计数减 1，而一般情况第二种方式相对安全些，当然如果处理得好，都是没问题的。

◆ 建议

推荐还是建议统一在所有窗口被释放后，最后才释放根窗口，再协同处理方面少产生些麻烦。如果有产生白屏问题，可以再做特殊处理（保留最后一个窗口释放后，立刻根窗口一起释放）。

➤ 白屏问题

◆ 白屏闪现问题

在所有窗口都被从根窗口（ROOTFORM）移开后，有时可能产生白屏（闪一下白色屏），产生这样的情况有 2 种可能：

1：根窗口最后没被释放，一般是主动增加根窗口引用计数产生的结果，经常伴随着一大堆内存和窗口没释放。应该很容易就找到这样的问题所在，并且轻松解决。

2：经常发生在最后一个窗口被移开后到根窗口被 RELEASE 之间有很多的处理过程。如果出现这种情况，一般以以下 2 种方式解决

◆ 解决白屏问题

把关闭前需要处理的操作（如内存释放，文件关闭等）放到最后一个窗口关闭前处理，尽可能缩短最后一个窗口从根窗口移出前的操作。

◆ 错误的解决方式

在应用程序释放函数（一般是 XXX_FreeAppData）首先释放根窗口（ROOTFORM），但一般不推荐这样的处理过程。因为根窗口（ROOTFORM）在释放过程中，会把在它之上的所有窗口引用计数减 1，但一般窗口释放都采用（如下图）的处理方式，这是标准的处理方式，却可能因为根窗口（ROOTFORM）已经先被释放而非常有可能产生问题。

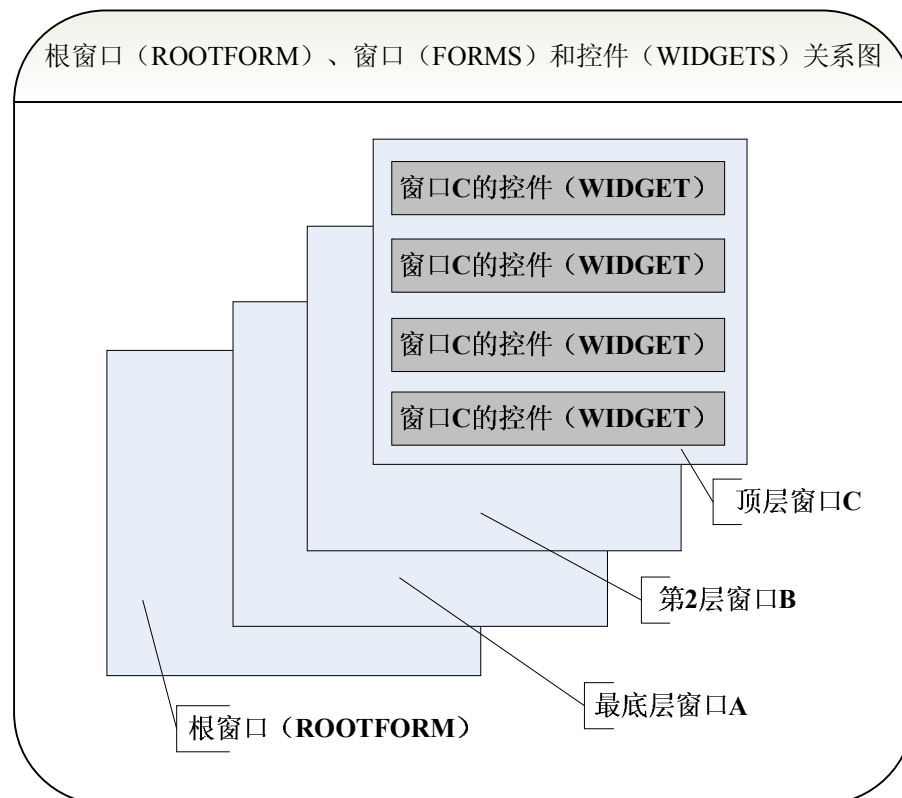
```
//-----  
//NAME:  
//DESC: 关闭窗口  
//-----  
void CAddrForms_ClassFormClose(CAddress *pMe)  
{  
    if (pMe->pClass == NULL) return;  
  
    DBGPRINTF("关闭窗口.");  
    IROOTFORM_PopForm(pMe->pRoot);  
    IFORM_Release(pMe->pClass->m_pClassForm);  
  
    return;  
}
```

一般应用关闭窗口的处理

➤ 创建窗口

◆ 窗口和根窗口的关系

窗口、根窗口和控件的关系，如下图的关系，显然根窗口维护着窗口队列。



◆ 创建窗口对象

窗口（FORMS）也可以看作是一个对象，使用标准创建接口函数（ISHELL_CreateInstance）来创建，如

```
dwResult = ISHELL_CreateInstance(pMe->piShell,AEECLSID_BLUEFORM,
                                (void**)&pMe->pClass->m_pClassForm);
if (dwResult != SUCCESS)
{
    DBGPRINTF("窗口(CClassForms)创建失败!");
    return EFAILED;
}
```

创建窗口

失败的错误代码可以参考（AEEError.h）头文件，一般情况是因为指定类没找到产生（ECLASSNOTSUPPORT 值为 3）错误。

➤ 释放窗口

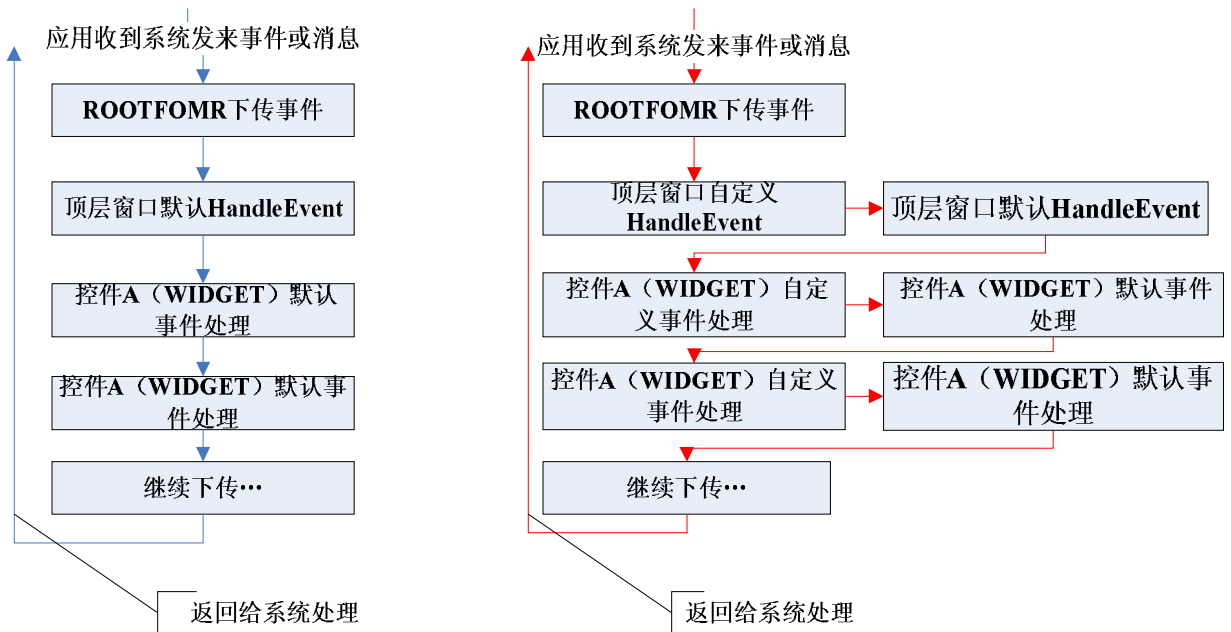
在窗口函数中说明。

➤ 窗口处理

◆ 设置窗口处理函数（XXX_HandleEvent）和关闭窗口处理函数（XXX_FormDelete）

窗口创建完成后，窗口默认有事件处理过程和关闭窗口后处理函数，但一般应用需要在默认函数被处理前获取消息或者事件，因此需要使用自定义事件处理函数取代窗口默认函数，在自定义函数被执行后，如果没返回将继续下传到默认函数处理。

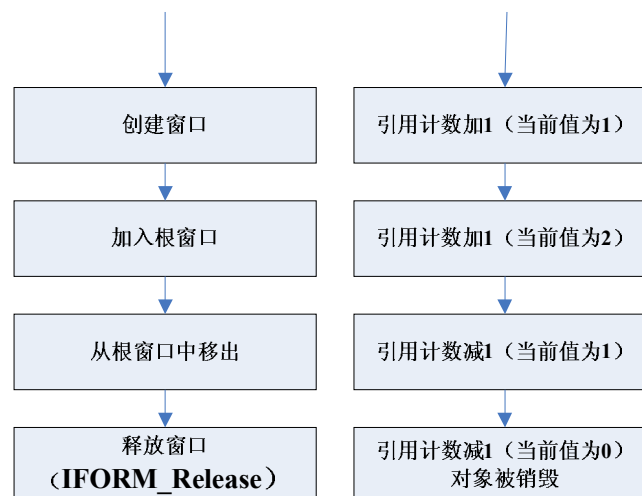
系统默认状态的事件处理过程和自定义 HandleEvent 事件处理过程。



左边是默认和右边是自定义事件处理（HandleEvent）过程

◆ 把窗口（FORM）加入根窗口（ROOTFORM）

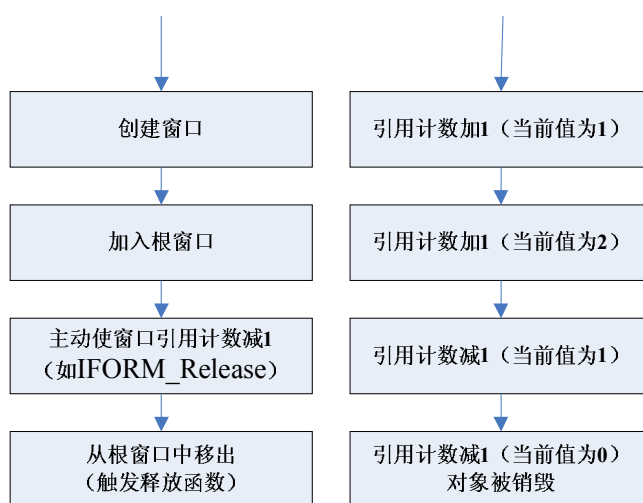
应用程序的窗口加入根窗口（ROOTFORM）后，才能显示出来，加入根窗口的函数为（IROOTFORM_PushForm）。加入根窗口后，窗口的引用计数为加 2，（创建后该值为 1）。



普通窗口

在根窗口上可以增加多个窗口（FORM），在加入后，顶层窗口将首先收到窗口激活属性，即（AEEEvent == EVT_WDG_SETPROPERTY 和 wParam == FID_ACTIVE 和 lParam == TRUE）。之后窗口进入正常执行状态。

部分窗口可能只保留引用计数为 1（如对话框），这样窗口在被从根窗口（ROOTFORM）中移出来时，就立刻触发窗口 Release 处理函数，使窗口运行中保存的所有数据都被清理干净。要主动把窗口的引用计数减 1，可以使用（IFORM_Release）来处理，但开发人员一定要清楚这个操作所表达的意义。



对话框窗口

◆ 把窗口从根窗口（ROOTFORM）移出来

把窗口从根窗口（ROOTFORM）队列中移出来，一般使用函数（IROOTFORM_RemoveForm）来执行，该函数是从根窗口队列列表中把某个窗口移出，不一定是顶层窗口，也可能是最底层窗口。

该函数直接把窗口的引用计数减 1，一般情况下并没触发窗口的删除函数，当然象对话框这样（本身引用计数为 1）的对象除外。

函数（IROOTFORM_PopForm）也是把窗口从根窗口中移出来，但该函数是把顶层窗口移出来，触发引用计数减 1 的同时，还触发下层窗口为顶层窗口的事件，即（AEEEvent == EVT_WDG_SETPROPERTY 和 wParam == FID_ACTIVE 和 lParam == TRUE）事件被出发。该函数一般和函数（IFORM_Release）同时出现，用来窗口关闭时释放。一般实现按如下图方式：

```

//启动时设置事件HandleEvent处理函数和窗口关闭时处理函数
HANDLERDESC_Init(&pMe->pClass->ClassFormHandler,
                  CAddrForms_ClassFormEventHandler, pMe, CAddrForms_ClassFormRelease);
IFORM_SetHandler(pMe->pClass->m_pClassForm, &pMe->pClass->ClassFormHandler);

void CAddrForms_ClassFormRelease(CAddress* pMe)
{
    //把窗口对象删除,包括控件和分配的内存等
    FREEIF(pMe->pClass);
    return;
}

void CAddrForms_ClassFormClose(CAddress *pMe)
{
    if (pMe->pClass == NULL) return;

    IROOTFORM_PopForm(pMe->pRoot);           //第1步
    IFORM_Release(pMe->pClass->m_pClassForm); //第2步

    return;
}

```

窗口关闭标准处理方式

按上图关闭窗口时,一般情况下,函数(IFORM_Release)执行时将同时(窗口引用计数为0)触发窗口删除函数,即第2步后将触发图中的(CAddrForms_ClassFormRelease)函数。在该过程中回删除整个窗口的对象,开发人员经常会犯在函数(IFORM_Release)被执行后、继续使用这里的pMe->pClass对象的错误,会产生程序崩溃问题。

◆ 窗口函数处理规范

现在开发的应用,很多开发人员都是监听函数中直接处理事件,即按下按钮后,在按钮事件中处理,这是WINDOWS一般使用的方式,即同步处理;但在BREW上,对事件的处理,希望以更稳健的异步方式执行,即在收到监听事件后,发送消息给当前窗口,在窗口事件处理函数的(EVT_COMMAND)中处理。

下图是按钮事件时,发送消息给窗口:

```

//-----
//NAME:
//DESC:确定按钮事件处理
//-----
static void CAddrForms_ClassFormBtnEventHandler(CAddress *pMe, ModelEvent *pev)
{
    if(pev->evCode == EVT_MDL_FOCUS_SELECT)
    {
        //这里pev->dwParam值为CLASS_FORM_BTN_OK
        ISHELL_PostEvent(pMe->piShell,pMe->ClsId,EVT_COMMAND,(int16)pev->dwParam,0);
        return;
    }
    return;
}

```


按钮事件处理

◆ 窗口事件处理示例

下图是联系人应用中的一个窗口基本事件处理过程，图中把窗口关闭过程、键盘处理过程和一般应用事件处理过程都分开处理，总的窗口事件处理过程非常明了。

```
//-----  
//NAME:  
//DESC:窗口事件处理函数，所有该窗口事件都通过消息发到该过程处理  
//-----  
boolean CAddrForms_ClassFormEventHandler(void *po, AEEEvent evt, uint16 wParam, uint32 dwParam)  
{  
    CAddress* pMe = (CAddress*)po;  
  
    switch(evt)  
    {  
        case EVT_KEY:  
            switch(wParam)  
            {  
                case AVK_CLR:  
                    CAddrForms_ClassFormClose(pMe,AVK_CLR);  
                    return TRUE;  
            }  
            break;  
  
        //关闭窗口  
        case EVT_BF_BUTTON:  
            CAddrForms_ClassFormClose(pMe,AVK_CLR);  
            return TRUE;  
  
        //窗口的事件处理部分内容  
        case EVT_COMMAND:  
            switch(wParam)  
            {  
                case CLASS_FORM_BTN_OK:  
                    DBGPRINTF("收到(CLASS_FORM_BTN_OK)事件!");  
                    return TRUE;  
  
                case CLASS_FORM_BTN_CANCEL:  
                    CAddrForms_ClassFormClose(pMe,wParam);  
                    return TRUE;  
            }  
            break;  
    }  
    return HANDLERDESC_Call(&pMe->pClass->ClassFormHandler, evt, wParam, dwParam);  
}
```

窗口事件处理过程

◆ 窗口的其它事件

窗口开始激活状态和非激活状态的事件，这个事件和 WINDOWS 的 ACTIVE 和 DEACTIVE 一样，一般把处理激活状态需要处理的事件放这里处理，比如私密应用状态该变了，在窗口激活后立刻去刷新数据。

```
case EUT_WDG_SETPROPERTY:
    switch(wParam)
    {
        //ACTIVE状态处理
        case FID_ACTIVE:
            if (dwParam == TRUE)
            {
                //窗口激活状态
            }

            if (dwParam == FALSE)
            {
                //窗口不再处于激活状态
            }
        }
    }
```

窗口激活和非激活状态收到的消息

提示：
一般不要在这个过程中处理太多太复杂的任务。

控件和事件处理

➤ 控件列表

一般应用所使用的控件，都在下列列表中，可能有些应用需要自定义部分特殊控件，需要控件组另外处理，但很大部分使用控件组合，一般都能满足应用的要求，控件列表如下。

控件 CLSID	控件类型	名称	简要说明
1 AEECLSID_XYCONTAINER	IXYContainer	容器控件	所有的控件都可以放在容器中, 也可以把容器的控件查询出来, 放在其它容器中, 即容器也是个控件.
2 AEECLSID_PROPCONTAINER	IPROPContainer	比例容器	容器, 但放在该容器的控件, 将按比例显示大小
3 AEECLSID_VIEWPORTWIDGET		VIEWPORT 控件	多页显示
4 AEECLSID_GRIDWIDGET		网格控件	
5 AEECLSID_BUTTONWIDGET	IWidget	按钮 (非标准)	按钮控件
6 AEECLSID_LISTWIDGET	IWidget	列表控件	
7			

8	AECLSID_IMAGEWIDGET	IWidget	显示图片控件	
9	AECLSID_STATICWIDGET	IWidget	静态文本控件	
10	AECLSID_SCROLLBARWIDGET	IWidget	滚动条控件	
11	AECLSID_PICKWIDGET	IWidget		比较少用
12	AECLSID_MENUWIDGET	IWidget	菜单控件	
13	AECLSID_CHECKWIDGET	IWidget	CheckBox 控件	
14				
15	AECLSID_TEXTWIDGET	IWidget	TEXT 控件	文本输入控件
16	AECLSID_RADIOWIDGET	IWidget	Radio 控件	

控件列表

这里需要特殊说明的是容器这个控件（IContainer、IXYContainer 和 IPropContainer），其它控件一般都是放入该容器控件，再把容器放入窗口显示出来。在后面将对直接放入窗口的控件操作进行说明。

容器控件本身也是个控件，可以通过查询接口，如（IXYContainer 的 IXYCONTAINER_QueryInterface）把容器控件查询出来，再放入其它容器中。

➤ 控件、容器和窗口关系

一般每个窗口都需要一个主容器，把容器和窗口绑定一起，其它控件或者容器可以直接放入容器显示出来，它们之间关系如下图。由于主容器是和窗口绑定，所以操作一般都是针对主容器进行的。

有时结构可能不这样，而是直接把控件放入窗口，但一般不建议这样处理，除非有特别需求。



窗口、容器和控件关系图

➤ 与 WINDOWS 同类控件的区别

➤ 控件的基本属性

开发人员必须先读头文件 (AEEWProperties.h)，把这个文件中的函数多读几遍，应该就能基本明白这些基本属性的设置过程。这里不对这部分内容做详细说明了。

这个文件包含了所有 BUIW 控件属性设置，如果平常开发中有些属性需要设置，直接去该文件查找。如果对部分属性需要修改或者增加功能，也可以深入 BUIW 控件去实现或者让控件来完成。

开发人员如果需要自己增强控件属性，也可以直接在应用中，把控件默认事件替换，在新函数中增加这些控件的属性。

➤ 创建控件和使用控件

在这部分将对平常使用的几个控件进行说明。

◆ 列表控件 (LIST) 使用和示例

LIST 控件是应用频繁使用的，关于它的使用方式，基本的过程这里就不列出来了，关键步骤会在这里结合例子和图示给予详细说明。比如应用中有个非常大的列表显示项，如联系人的名称和电话号码显示列表，下列图示处理整个过程（从功能需求到实现完成）。

1: 控件排列的需求

应用 LIST 需要显示的每项上的控件和大概位置（预计 LIST 的显示项内容），它们的排列大体如下：

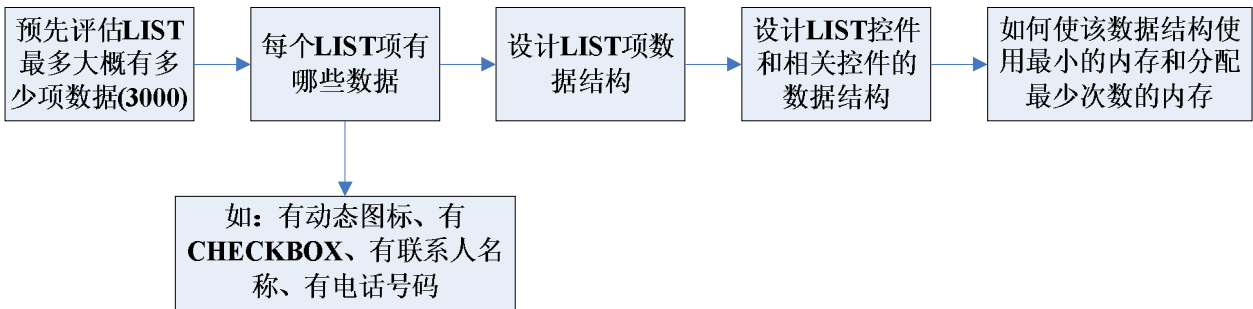


控件排列和显示

2: 使用 LIST 前思考的问题

如预计有最大可能超过 3000 联系人，需要思考内存以及数据优化问题（如下图）；而一般应用的 LIST 中的数据都比较少，如果比较少就不需要考虑这么复杂的处理。

这里考虑复杂的情况：



多记录数思考的问题

3: 设计 LIST 项数据结构

如考虑到有 3000 项的可能，但手机屏幕只能显示 6 到 8 项出来，不显示的项不载入显示，所以这里名称和电话号码用指针指向预先分配的 12 个字符数组（即缓冲区）：

如果记录少，显示内人也少，就直接分配这些内存更为合理。

```

//大数据量的时候，为了优化内存，使用下面结构
typedef struct _CADDRListItem
{
    AECHAR          *m_pMainTel;           //电话号码
    AECHAR          *m_pMainName;         //名称
    AECHAR          m_szLocaler[5];       //快速定位
    AECHAR          m_szLocate9Key[5];    //9键定位，1..9字符
    boolean         m_bHidden;           //私密状态
    uint16          m_dwRecOID;           //记录保存的表ID
    uint16          m_dwSavePos;         //保存在哪个表中位置
    uint16          m_dwGroupID;         //组ID保存下来
    boolean         m_bSelected;        //该项是否被选择中了
    IImage          *m_pImage;           //分组，SIM图标显示
    AECHAR          *m_szTitle;          //详细信息中用来保存标题的
    uint16          m_dwIndex;           //该项在队列中的位置
}CADDRListItem;

```

LIST 数据结构

4: 设计 LIST 相关数据结构

LIST 控件和相关控件的数据结构，把需要的控件和数据都列出来

```

//LIST项的数据和控件
IWidget          *m_pDeleteList;        //LIST控件
HandlerDesc      m_ListWidgetHandler;   //
IWidget          *m_piListScroll;       //滚动条
IArrayModel      *m_pListArrayModel;    //IMODEL
ModelListener    m_pDeleteListListener; //监听器

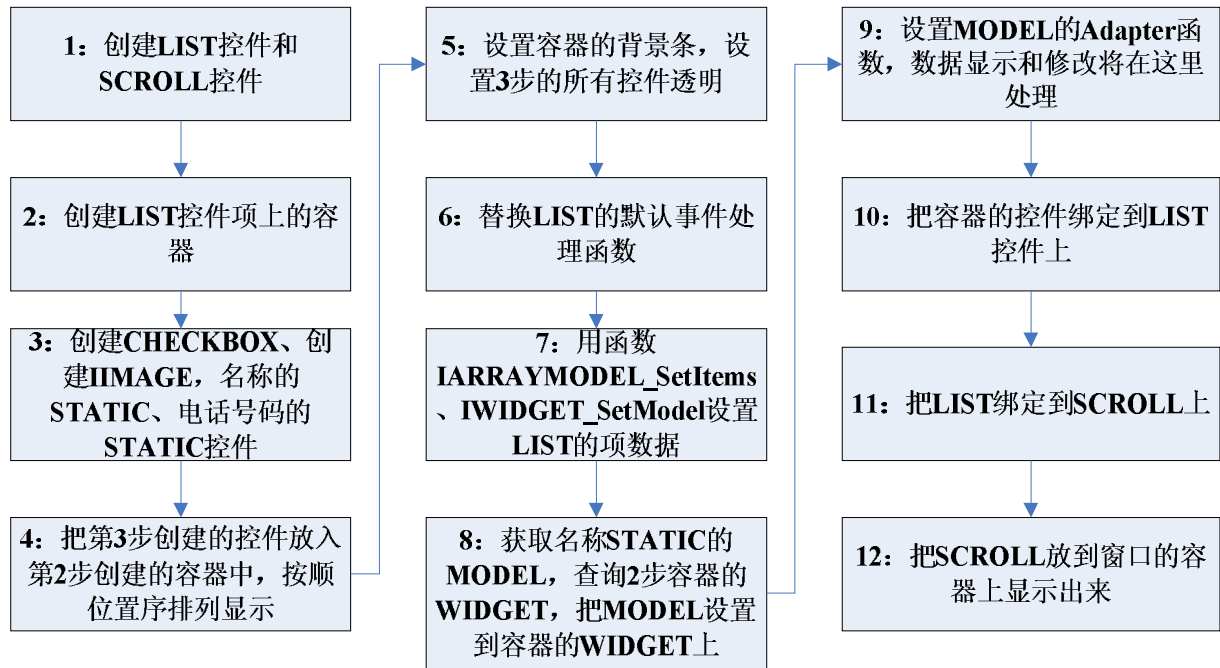
AECHAR          m_szDelAddrTel[12][ABC_Tel_MAXLEN+2]; //电话号码缓冲区
AECHAR          m_szDelAddrName[12][ABC_Name_MAXLEN+2]; //姓名缓冲区
CADDRListItem    *m_pDfListItemData;    //总数据项
int32            m_dwItemCount;          //LIST的项数
int32            m_dwCurRecCount;       //现在显示的项

//LIST控件容器控制
IXYContainer     *pDFXYListContainer;    //LIST项的容器
IWidget          *pDFContainerWidget;    //容器的子控件
IWidget          *m_pDFCheckBoxImgWidget; //CHECKBOX的图片
IWidget          *m_pDFListItemCheckBox; //CHECKBOX
IWidget          *m_pDFListItemIcon;     //
IWidget          *m_pDFListItemName;     //名称STATIC控件
IWidget          *m_pDFListItemTel;      //电话号码STATIC控件

```

LIST 相关控件项

5: 设计创建控件处理过程:



控件的创建显示过程

6: 实例代码

```

//替换LIST默认函数
HANDLERDESC_Init(&pMe->pDelete->m_ListWidgetHandler, CAddrForm_DFListWidget_HandleEvent, pMe, 0);
IWIDGET_SetHandler(pMe->pDelete->m_pDeleteList, &pMe->pDelete->m_ListWidgetHandler);

//设置LIST项数据内容
IARRAYMODEL_SetItems(pMe->pDelete->m_pListArrayModel, (void*)pMe->pDelete->m_pDFListItemData,
                    pMe->pDelete->m_dwItemCount, sizeof(CADDRListItem));
IWIDGET_SetModel(pMe->pDelete->m_pDeleteList, (IModel*)pMe->pDelete->m_pListArrayModel);

//获取名称的STATIC控件, 设置到容器的MODEL中, 设置显示函数, 绑定容器控件到LIST上
IWIDGET_GetModel(pMe->pDelete->m_pDFListItemName, AEEIID_VALUEMODEL, (IModel**) &pListtemValueModel);
IWIDGET_SetModel(pMe->pDelete->pDFContainerWidget, (IModel*)pListtemValueModel);
IVALUEMODEL_AdaptGet(pListtemValueModel, (PFNADAPTGET)CAddrForms_DeleteFormShowDataAdapter, (void *)pMe);
IVALUEMODEL_Release(pListtemValueModel);
IDECORATOR_SetWidget((IDecorator*)pMe->pDelete->m_pDeleteList, pMe->pDelete->pDFContainerWidget);

//双击的事件处理 linshuchun 2007.08.06
IWIDGET_GetViewModel(pMe->pDelete->m_pDeleteList, &pDeleteFormListViewModel);
IMODEL_AddListenerEx(pDeleteFormListViewModel, &pMe->pDelete->m_pDeleteListListener,
    (PFNLISTENER)CAddrForms_DeleteListFormListEventHandler, pMe);
IMODEL_Release(pDeleteFormListViewModel);

//绑定LIST到滚动条上
IDECORATOR_SetWidget((IDecorator*)pMe->pDelete->pListScroll, pMe->pDelete->m_pDeleteList);
  
```

实例代码

7: 设计显示函数。

```

//DESC: 这个过程显示LIST控件中每项的值
//-----
void CAddForms_DeleteFormShowDataAdapter(void *pUnused, void **ppValueIn,
                                           int nLen, void **ppValueOut, int *pnLenOut)
{
    CADDRListItem *pItem = NULL;
    CAddress *pMe = NULL;

    //参数检查...
    if (ppValueIn == NULL) return;
    if (pUnused == NULL) return;

    pMe = (CAddress *)pUnused;
    pItem = (CADDRListItem *)ppValueIn;

    IWIDGET_SetImage(pMe->pDelete->m_pDfListItemIcon, pMe->m_pListGCCardImage[0]);

    if (pItem->m_bHidden) //设置字体颜色
        IWIDGET_SetFGColor(pMe->pDelete->m_pDfListItemName, MAKE_RGB(255,0,0));
    else
        IWIDGET_SetFGColor(pMe->pDelete->m_pDfListItemName, RGB_BLACK);

    CCheckBox_SetValue(pMe->pDelete->m_pDfListItemCheckBox, pItem->m_bSelected);

    //这里显示名称
    if (pItem && pItem->m_pMainName)
    {
        *ppValueOut = pItem->m_pMainName; //显示当前项的名称,可以输出任何
        *pnLenOut = WSTRLEN(pItem->m_pMainName); //名称的长度,可以按最长(-1)显示
    }

    return;
}

```

显示函数

注意:

《1》效率问题

这个显示函数（如上图）会在每条记录被显示出来的是，执行一遍，即控件显示有刷新界面的时候，都调用该函数，所以非常频繁的执行，这就需考虑到代码的执行效率问题。最主要的一点，应该不要在该函数中处理需要时间比较长的任务，如读数据、频繁写日志、打印调试信息（频繁 DBGPRINTF 都不应该）等问题。

《2》创建多少控件问题

从该过程来看，并不需要每条记录都创建一组控件，而是共用一组控件，修改显示指针来更换显示数据。

《3》背景图片资源文件

所有的 LIST 背景图片，都必须使用 BUIW 提供的默认资源图片。

《4》是否显示滚动条

如果 LIST 项是不变的，可以根据实际情况显示滚动条，但如果时有时无，那都必须按有滚动条显示。

曾经的问题：

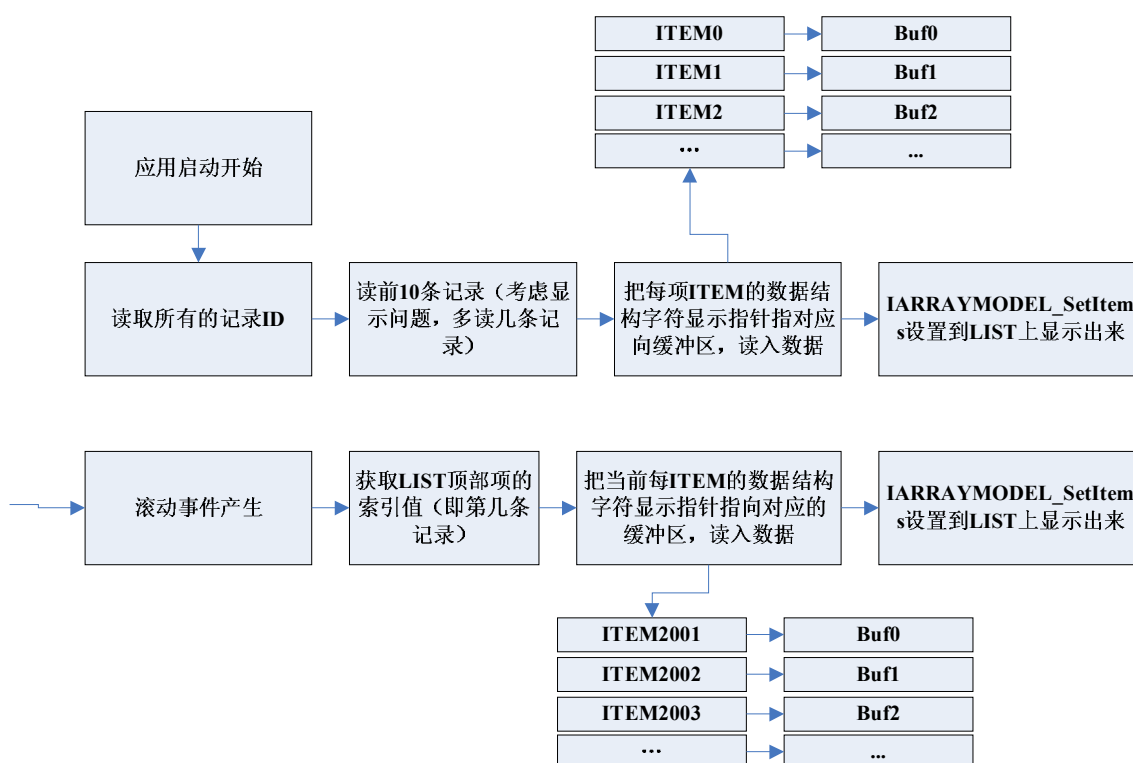
曾经看到有的应用，每条记录创建一组控件，记录不多（少于 10 条）经过处理，也能实现这些功能，但这个处理过程显然不合适，内存使用多，并且如果记录很多，无法处理。

8：滚动条事件的处理流程（缓冲显示优化内存）

手机 LIST 显示的内容，一般都显示比较少的字段，比如联系人列表显示，该表有 30 多个字段，但在列表中显示出来的就有 1 个名称或者加 1 个电话号码。

即使就 1 个名称，如果记录条数多，那如果都分配内存，也是个不小的数；因此内存也是按需要使用时才分配或者把指针指到已经分配的缓冲区。

下图中可以看出，在 LIST 中设置了可能 3000 条记录（在真实应用中滚动条拖动块能反映出该记录数），但用户能看到的就一个屏幕，最多 10 个联系人名称和电话，因此读的也就这 10 条。注意缓冲区（BUF0.....BUF12）重复被使用。

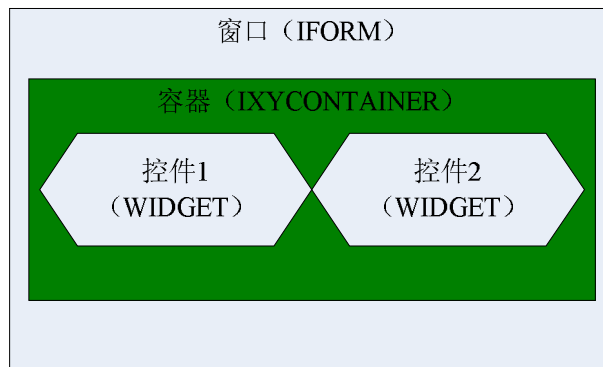


缓冲显示数据

◆ 容器控件 (IXYCONTAINER)

1: 与窗口的关系

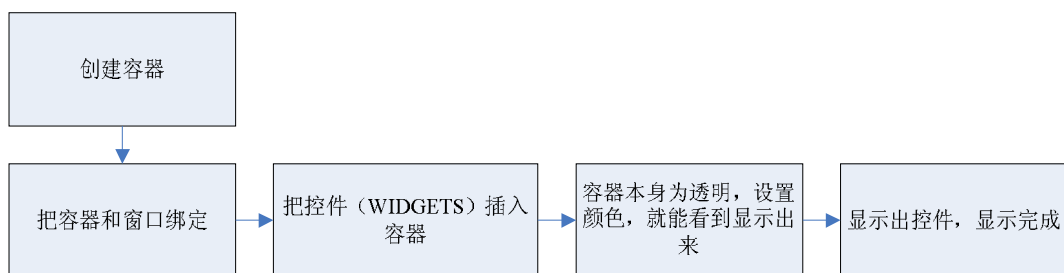
顾名思义，容器是用来存在控件，是连接控件和窗口的纽带。现在一般的做法是把控件放在容器中，容器放在窗口中显示出来的，关系入下图。



窗口、控件和容器

2: 创建和显示过程

这里结合图例把创建过程列出来，和一般控件的创建过程是一样的。



创建和显示容器过程

3: 代码实例

先查询容器控件并且绑定到窗口

```

void CAddrForms_ClassFormBindCtrl(CAddress* pMe)
{
    int32 dwResult = SUCCESS;
    IWidget* containerWidget = NULL;

    dwResult = IXYCONTAINER_QueryInterface(pMe->pClass->pClassMainXYContainer,
                                           AEEIID_WIDGET, (void**)&containerWidget);
    if(dwResult != SUCCESS)
    {
        DBGPRINTF("创建失败!");
        return;
    }

    IFORM_SetWidget(pMe->pClass->m_pClassForm, WID_BLUEFORM, containerWidget);
    IWIDGET_Release(containerWidget);
    return;
}

```

容器和窗口的绑定过程

注意:

上图查询过程增加了控件（containerWidget）的引用计数，所以控件（containerWidget）使用完就释放了。

把控件插入容器中，设置大小、位置和是否可见。

```

we.height      = ADDR_BUTTON_HEIGHT;
we.width       = ADDR_268_SCREEN_WIDTH;
wpos.x         = 0;
wpos.y         = 0;
wpos.bVisible  = TRUE;
IWIDGET_SetExtent(pWidget, &we);
IXYCONTAINER_Insert(pMe->pClass->pClassMainXYContainer, pWidget, WIDGET_ZNORMAL, &wpos);
IWIDGET_Release(pWidget);

```

控件插入容器的过程

4: 在 IXYCONTAINER 中的位置

控件 XY 容器中的位置，根据参数（WidgetPos wpos）来确定，这里的 XY 是指在容器中的坐标，即从（0，0）点开始，是否可见根据该结构的 bVisible 来确定。

控件的大小根据（WExtent we）中设置的值来确定，超出容器部分将不可见。

注意:

如果容器不可见，容器中的所有控件都将不可见。

◆ 比例容器

比例容器（IPROPCONTAINER）和 XY 容器基本是一样的，插入容器或者控件放到该比例容器的过程参考上面的 IXCONTAINER 过程；但主要区别在于比例容器中的控件是按比例显示

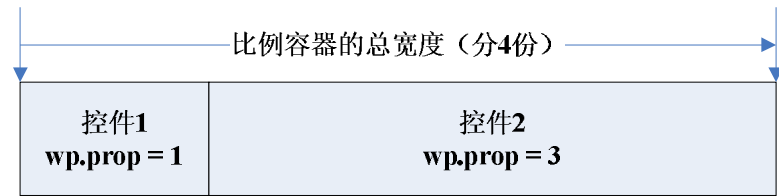
的。

在比例容器中插入 2 个控件（下图的 numberStatic、opStatic 控件），它们占的比例分别是 3 和 1，即分别占比例容器控件宽度的 3/4 和 1/4，设置过程如下图：

```
wp.bVisible = TRUE;  
wp.prop = 3;  
IPROPCONTAINER_Insert(numOpContainer, pMe->numberStatic, WIDGET_ZNORMAL, &wp);  
  
wp.prop = 1;  
IPROPCONTAINER_Insert(numOpContainer, pMe->opStatic, WIDGET_ZNORMAL, &wp);
```

插入比例容器的过程

图示：



提示：

该控件在实际中非常少被使用，进度条控件可能会使用该控件来设置颜色网格。

◆ **VIEWPORT 控件**
VIEWPORT 的使用

◆ **网格控件（GRID）**

◆ **按钮（非标准）**
BREW 提供的标准控件没有该控件，该控件有由控件组开发完成，具有 WINDOWS 上一样的事件响应功能；该控件的使用过程如下图：

- 1：创建对象。
- 2：设置高度宽度和可见属性。
- 3：插入容器，在窗口中能显示出来。
- 4：设置控件 ID 值。
- 5：设置是否有边线（FALSE 为没有）。
- 6：设置按钮标题。
- 7：设置背景图为 2 个字大小，并且是默认背景。
- 8：设置按钮事件响应函数。

```

//创建确定按钮
dwResult = ISHELL_CreateInstance(pMe->a.m_pIShell,AEECLSID_BUTTONWIDGET, (void**)&pMe->pMain->m_pMainBtnOK);
if (dwResult != SUCCESS)
{
    DBGPRINTF("按钮对象创建失败.");
    return;
}

we.height      = dwBtnHeight;
we.width       = TOMINFO_BUTTON_WIDGH;
wpos.x         = TOMINFO_BUTTON_LEFT;
wpos.y         = 0;
wpos.bVisible  = TRUE;
IWIDGET_SetExtent(pMe->pMain->m_pMainBtnOK, &we);
IXYCONTAINER_Insert(pButtonContainer, pMe->pMain->m_pMainBtnOK,WIDGET_ZNORMAL , &wpos);

MEMSET(pMe->pMain->m_pMainBtnOKCap,0,sizeof(pMe->pMain->m_pMainBtnOKCap));
WSTRCPY(pMe->pMain->m_pMainBtnOKCap,pMe->m_szTomBtnOK);

IWIDGET_SetProperty(pMe->pMain->m_pMainBtnOK,PROP_BTNID,TOMINFO_FORM_BTN_LEVEL01_04);
IWIDGET_SetProperty(pMe->pMain->m_pMainBtnOK,PROP_BTNNOSTYLE, (uint32)FALSE);
IWIDGET_SetProperty(pMe->pMain->m_pMainBtnOK,PROP_BTNCAPTION,(uint32)pMe->szBtnOKCap);
IWIDGET_SetProperty(pMe->pMain->m_pMainBtnOK, PROP_BTNBGDEFAULT, BTN_BGIMAGE_2);

IWIDGET_GetViewModel(pMe->pMain->m_pMainBtnOK, &pBtnModel);
IMODEL_AddListenerEx(pBtnModel, &pMe->pMain->m_pMainBtnOKListener,
                    (PFNLISTENER)CTomInfo_MainFormBtnEventHandler, pMe);

IMODEL_Release(pBtnModel);
DBGPRINTF("按钮(m_pBtnOK)创建完成!");

```

创建按钮过程

9: 在响应函数中发送消息给主窗口，异步处理事件。

```

//-----
//NAME:
//DESC:Button按钮事件处理
//-----
static void CTomInfo_MainFormBtnEventHandler(CTomInfoData *pMe, ModelEvent *pev)
{
    if(pev->evCode == EVT_MDL_FOCUS_SELECT)
    {
        ISHELL_PostEvent(pMe->a.m_pIShell,pMe->a.clsID,
                        EVT_COMMAND,TOMINFO_MAINFORM_BUTTON_EVENT,pev->dwParam);
        return;
    }
    return;
}

```

按钮事件

按钮注意事项

1: 点击完成后，应该先设置按钮为不可使用，处理后任务完成再设置为可用，主要考虑是如果任务处理比较长时间，频繁的点击后，可能产生系统非常繁忙的状态，甚至暴力测试的情况，有可能产生系统崩溃问题。

2: 事件建议使用异步处理，异步处理能减少压栈空间使用和系统有空闲时间处理其它任务

的机会。

◆ 显示图片控件

◆ 静态文本控件

静态文本控件（STATIC）控件是最基本的控件，非常容易使用，大概过程如下：

1：创建静态文本控件（创建、设置属性、设置大小可见、插入容器）。

```
//标题控件
dwResult = ISHELL_CreateInstance(pMe->piShell,AEECLSID_STATICWIDGET , (void**)&pMe->pClass->m_pClassFormStatic);
if (dwResult != SUCCESS)
{
    DBGPRINTF("创建对象失败.");
    return;
}

IWIDGET_SetFlags(pMe->pClass->m_pClassFormStatic, SWF_NOSHORTENTEXT | IDF_ALIGN_LEFT | IDF_ALIGN_BOTTOM);
IWIDGET_SetText(pMe->pClass->m_pClassFormStatic,pMe->pClass->m_szClassFormStaticCap,FALSE);
we.height      = CAPTION_WIDGET_HEIGHT;
we.width       = CAPTION_WIDGET_WIDTH;
wpos.x         = CAPTION_WIDGET_LEFT;
wpos.y         = CAPTION_WIDGET_TOP;
wpos.bVisible  = TRUE;
IWIDGET_SetExtent(pMe->pClass->m_pClassFormStatic, &we);
IXYCONTAINER_Insert(pMe->pClass->pClassMainXYContainer, pMe->pClass->m_pClassFormStatic,WIDGET_ZNORMAL , &wpos);
DBGPRINTF("创建(m_pClassFormStatic)完成!");
```

2：设置文本控件文本内容

一般有两种方式设置文本如下图（来自 Demo（计数器应用）代码），需要说明的是要特别注意函数（IWIDGET_SetText）的第三个参数。

《1》 让 Static 控件维护参数分配的内存。

函数（IWIDGET_SetText）第三个参数为 TRUE 表示当前控件在销毁的时候，会把第二个参数（buf）分配的内存释放。

```
AECHAR* buff = NULL;
buff = (AECHAR*) MALLOC(4);
buff[0] = pMe->operatorString[pev->dwParam];
IWIDGET_SetText(pMe->opStatic, buff, TRUE);
```

让静态控件维护内存

《2》 应用中开发人员维护参数分配的内存。

函数 (IWIDGET_SetText) 第三个参数为 FALSE, 说明该参数的内存需要开发人员自己维护; 下图是静态标题控件, 在结构中直接定义了 16*sizeof (AECHAR) 个字节。

```
//STATIC控件和标题
IWidget                                *m_pSelectFormStatic;
AECHAR                                m_szSFStaticCap[16];

IWIDGET_SetText(pMe->pSelect->m_pSelectFormStatic,
                pMe->pSelect->m_szSFStaticCap,FALSE);
```

应用维护内存

提示:

对于哪种方式, 开发人员自己一般都有想法, 但建议使用第二中方式, 是更为可靠的。以前有碰到过一些应用, 使用第一种方式处理, 但该控件的标题是可变的, 并且经常更换, 产生多次分配该内存的操作, 导致在释放的时候, 产生内存没 FREEIF 的情况。

◆ TEXT 控件

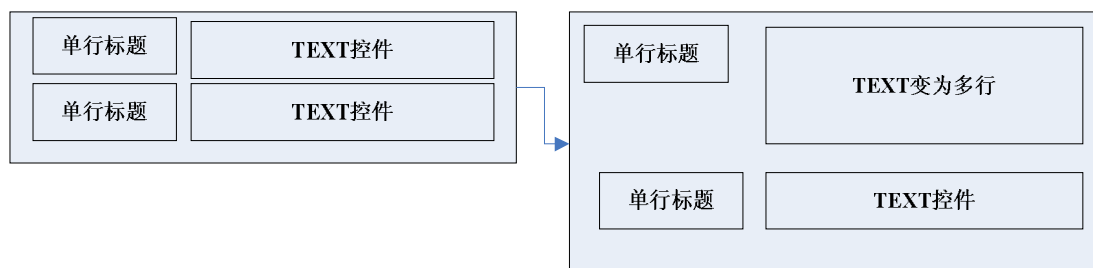
TEXT 控件的基本过程 (放入容器、设置高度宽度、是否可见等基本属性) 和其它标准控件的过程是一样的, 这里将不再描述, 主要把重点过程放在 TEXT 控件多行处理上。

《1》TEXT 多行的处理

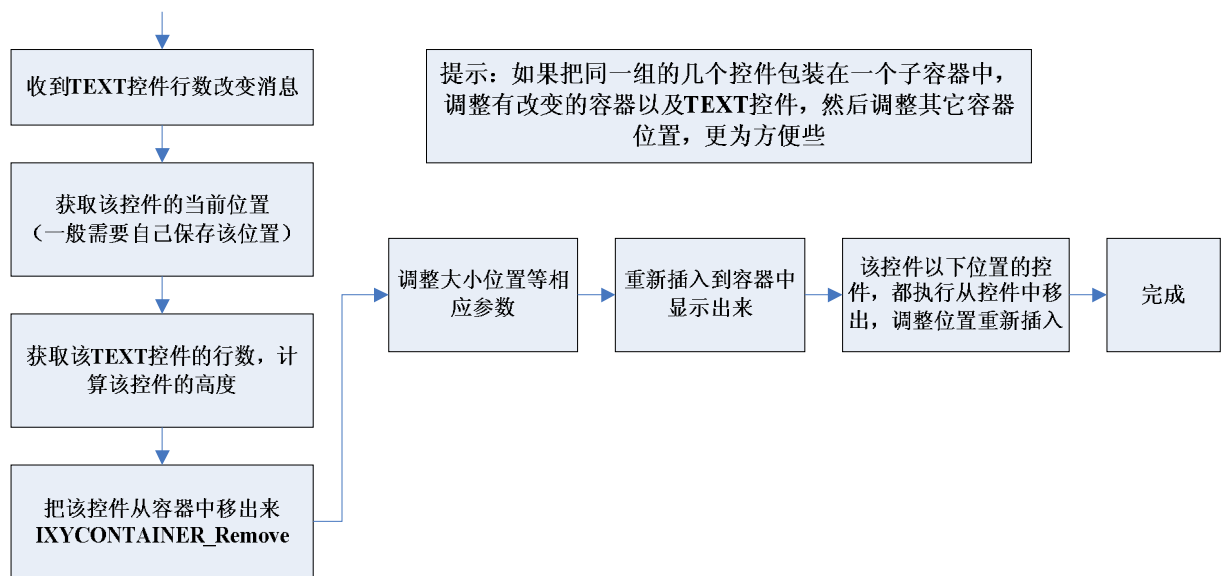
由于 BREW 没有提供对该控件自动按内容多少显示行数并且设置高度的过程, 所以在 BREW 每次行数改变的时候, 都必须重新设置新的高度和行数。

在 BREW 提供的标准控件基本功能中, 有函数 (IXYCONTAINER_Remove) 可以被用来实现把控件从容器中移出这个功能, 函数 (IXYCONTAINER_Insert) 可以把控件再插入容器中, 在这 2 个函数执行中间修改控件高度, 就可以实现多行处理的功能。

下面主要的思考方式就是如果控件高度发生改变, 在该控件位置下的所有控件都必须跟着改变。



行数改变的图示



多行处理流程

2: TEXT 控件的文本

获取 TEXT 控件的文本和设置 TEXT 控件的文本。

```

static __inline int IWIDGET_GetTextWidgetText(IWidget *piWidget, AECHAR **ppo)
{
    ITextModel *piModel;

    if (piWidget && IWIDGET_GetModel(piWidget, AEEIID_TEXTMODEL, (IModel **)&piModel) == SUCCESS)
    {
        TextInfo textInfo;

        ITEXTMODEL_GetTextInfo(piModel, &textInfo);
        *ppo = (AECHAR*)textInfo.pwText;
        ITEXTMODEL_Release(piModel);

        return SUCCESS;
    }
    return EFAILED;
}
  
```

获取文本的函数


```

static __inline int IWIDGET_SetTextWidgetText(IWidget *piWidget, const AECHAR *txt)
{
    ITextModel *piModel;

    if (piWidget && IWIDGET_GetModel(piWidget, AEEIID_TEXTMODEL, (IModel **)&piModel) == SUCCESS)
    {
        ITEXTMODEL_SetSel(piModel, 0, -1);
        ITEXTMODEL_ReplaceSel(piModel, txt, (int)(txt ? -1 : 0));
        ITEXTMODEL_Release(piModel);
    }

    return SUCCESS;
}

```

设置文本的函数

注意:

由于在 TAB 控件中使用 TEXT 控件，输入法的控制可能存在问题，所以有可产生问题，因此明确规定，不允许使用 TAB 控件。

◆ 滚动条控件

- 1: 和 LIST 控件绑定使用
- 2: 和 TEXT 控件绑定使用

◆ 菜单控件

普通的菜单

◆ CheckBox 控件

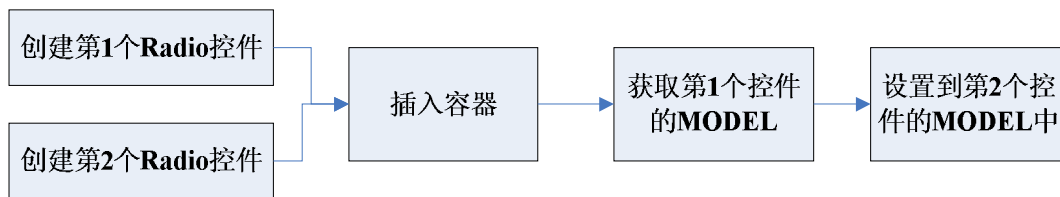
◆ TAB 控件

明确规定，不允许使用该控件。

◆ Radio 控件

控件 Radio 和 Windows 上的控件应该说是功能上基本一样的。这里不详细说明如何把控件放到窗口和容器中，主要描述把几个控件组放一起的过程。大概过程如下。

《1》 设置 Radio 控件组



设置 Radio 控件组的过程

《2》代码实例过程：

```

IWIDGET_GetViewModel(pMe->pMain->m_pWidget01, &pMe->pMain->m_pCheckValueModel);
IWIDGET_SetViewModel(pMe->pMain->m_pWidget02, (IModel*)pMe->pMain->m_pCheckValueModel);
IWIDGET_SetViewModel(pMe->pMain->m_pWidget03, (IModel*)pMe->pMain->m_pCheckValueModel);

```

设置 Radio 控件组

注意：

该过程使用的函数（IWIDGET_GetViewModel），会把第 2 个参数的引用计数加 1，最后需要减 1 才能释放该控件。

《3》设置和获取 Radio 控件的值。

```

//-----
//NAME:
//DESC: 获取Radio控件的值
//-----
boolean CSecret_RadioGetBool(IWidget *pWidget)
{
    IValueModel *pivm = NULL;
    boolean bResult = FALSE;

    if (IWIDGET_GetModel(pWidget, AEEIID_VALUEMODEL, (IModel**)&pivm) == SUCCESS)
    {
        bResult = IVALUEMODEL_GetBool(pivm);
        IVALUEMODEL_Release(pivm);
    }
    return bResult;
}

```

获取 Radio 值

```

//-----
//NAME:
//DESC: 设置Radio控件的值
//-----
void CSecret_RadioSetBool(IWidget *pWidget,boolean bValue)
{
    IValueModel *pvm = NULL;
    if (IWIDGET_GetModel(pWidget, AEEIID_VALUEMODEL,(IModel**)&pvm) == SUCCESS)
    {
        IVALUEMODEL_SetBool(pvm,bValue);
        IVALUEMODEL_Release(pvm);
    }

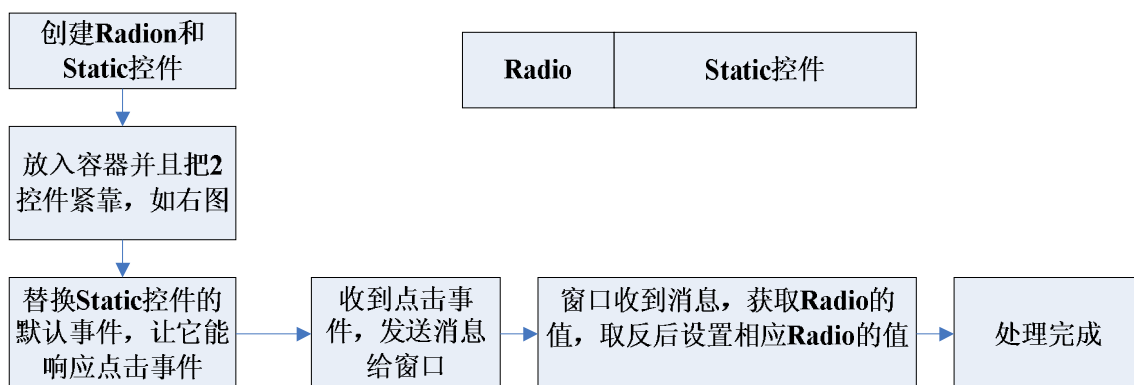
    return;
}

```

设置 Radio 值

◆ 如何把 CheckBox 和 Static 控件捆绑一起

捆绑 Radio 和 Static 控件为一组，让点击 Static 控件能 Radio 响应，该过程如下：



设置 Radio 组过程

提示：

这里也可以不用发送消息，而是直接设置值，因为处理过程很简单，不存在堆栈或者导致系统繁忙的状态。

➤ 引用计数问题。

◆ 认识引用计数

BUIW 的每个控件和窗口，都使用引用计数来保存该控件或窗口的状态，如果控件或窗口的引用计数为 0，表明该控件或窗口应该被销毁状态。它和 WINDOWS 的 DCOM 组件一样，也是采用这样的方式保存组件状态。

◆ 为什么这么强调引用计数

由于引用计数确定了控件或者窗口当前状态、是否被销毁、什么时间被销毁、销毁顺序等问题，所以对开发人员非常重要，需要非常明确这个计数。

引用计数在根窗口、窗口释放中非常重要，决定了释放的顺序问题，所以这个计数是开发人员必须明确清楚的问题。

不可动态增加根窗口（ROOTFORM）的引用计数，也不要动态增加窗口的引用计数；我们到目前为止，还没有过这样的需求，今后如果有，那一定是编码上的不合理。

◆ 哪些操作增加了引用计数

1: 创建对象（如 ISHELL_CreateInstance）

窗口和控件，在创建对象开始时，该对象计数被设置为 1，如下图的函数中的（me->nRefs）

```
void WidgetBase_Ctor(WidgetBase *me, AEEUTBL(IWidget) *pvt,
                    IModule *piModule, PFNHANDLER pfnDefHandler)
{
    me->nRefs = 1;           //引用计数被置为1
    me->piModule = piModule;
    ADDREFIF(piModule);
}
```

创建对象的引用计数为 1

2: 查询对象函数

该类型函数一般是容器类才有查询函数（如 IXYCONTAINER_QueryInterface），有（IXX_QueryInterface）类型接口的函数，应该都可以认为具有相同的特征，部分自定义控件也有该类型查询函数。

该函数最后将调用基类的查询函数，如下图中的过程。

```

uint32 WidgetBase_AddRef(IWidget *po)
{
    DECLARE_ME;

    return ++me->nRefs;
}

int WidgetBase_QueryInterface(IWidget *po, AEECLSID clsid, void **ppNew)
{
    if (clsid == AEECLSID_QUERYINTERFACE ||
        clsid == AEEIID_HANDLER ||
        clsid == AEEIID_WIDGET) {

        *ppNew = (void*)po;
        WidgetBase_AddRef(po);    //引用计数加1
        return SUCCESS;
    }

    *ppNew = 0;
    return ECLASSNOTSUPPORT;
}

```

查询接口增加对象引用计数

3: 获取子对象函数（如 IWIDGET_GetViewModel）

```

int WidgetBase_GetViewModel(WidgetBase *me, IModel **ppiModel)
{
    int nErr = 0;

    // if we've been requested to supply a view model,
    // and we don't have one, create it
    if (!me->piViewModel) {
        nErr = ModelBase_New(&me->piViewModel, me->piModule);
    }

    *ppiModel = me->piViewModel;

    // AddRef the returned view model
    if (me->piViewModel) {
        IMODEL_AddRef(me->piViewModel); //增加引用计数
    }

    return nErr;
}

```

查询子对象增加该对象引用计数

提示:

一般使用（IWIDGET_GetViewModel）获取的对象，都是局部变量，在使用完后，应该立刻释放掉，不需要保留该控件指针。如：

```

//-----
//NAME:
//DESC: 获取Radio控件的值
//-----
boolean CSecret_RadioGetBool(IWidget *pWidget)
{
    IValueModel *pivm = NULL;
    boolean bResult = 0;

    if (IWIDGET_GetModel(pWidget, AEEIID_VALUEMODEL, (IModel**)&pivm) == SUCCESS)
    {
        bResult = IVALUEMODEL_GetBool(pivm);
        IVALUEMODEL_Release(pivm);
    }

    return bResult;
}

```

pivm 局部变量，用完立刻释放

4: 对象插入容器 (IXYCONTAINER_Insert) 函数

包括比例容器和基本容器的插入过程,处理都是一样的。该过程增加了当前控件的引用计数,这个引用计数将在被从容器中移开时减掉 1,或者在最后释放时,从容器节点中去掉时减 1,具体的过程,可以参考如下代码内容。

```

void WidgetNode_Ctor(WidgetNode *me, IWidget *piw, AEERect *rc, boolean bVisible)
{
    WNODE_CTOR(me);

    RELEASEIF(me->piWidget);
    me->piWidget = piw;
    ADDREFIF(piw);

    me->rc = *rc;
    me->fVisible = bVisible;
    me->pixyRaise = NULL;
}

```

```

➡ WidgetNode_Ctor(WidgetNode * 0x03eee7e8, IWidget * 0x05c9d910, AEERect * 0x0013f924,
WidgetNode_NewForPos(const WidgetPos * 0x0013f924, IWidget * 0x05c9d910, WidgetNode * 0x03eee7e8,
ContainerBase_mkWidgetNode(ContainerBase * 0x05c9d508, const void * 0x0013f924,
ContainerBase_Insert(IContainer * 0x05c9d508, IWidget * 0x05c9d910, IWidget * 0x03eee7e8),
IXYCONTAINER_Insert(IXYContainer * 0x05c9d508, IWidget * 0x05c9d910, IWidget * 0x03eee7e8),
CSecret_MainFormCreateButton(_CSecretData * 0x03eeac78) line 356 + 31 bytes
CSecret_CreateMainForm(_CSecretData * 0x03eeac78) line 70 + 9 bytes
CSecret_HandleEvent(_IApplet * 0x03eeac78, unsigned short 0, unsigned short 0, unsigned short 0),
AEEApplet_HandleEvent(_IApplet * 0x03eeac78, unsigned short 0, unsigned short 0, unsigned short 0)

```

ready

IXYCONTAINER_Insert 增加了控件的引用计数

图中 IXYCONTAINER_Insert 子控件（如图地址为 0X05c9d910）在 WidgetNode_Ctor 函数中引用计数被增加了，这是在模拟器环境的调试信息中可以跟踪到。

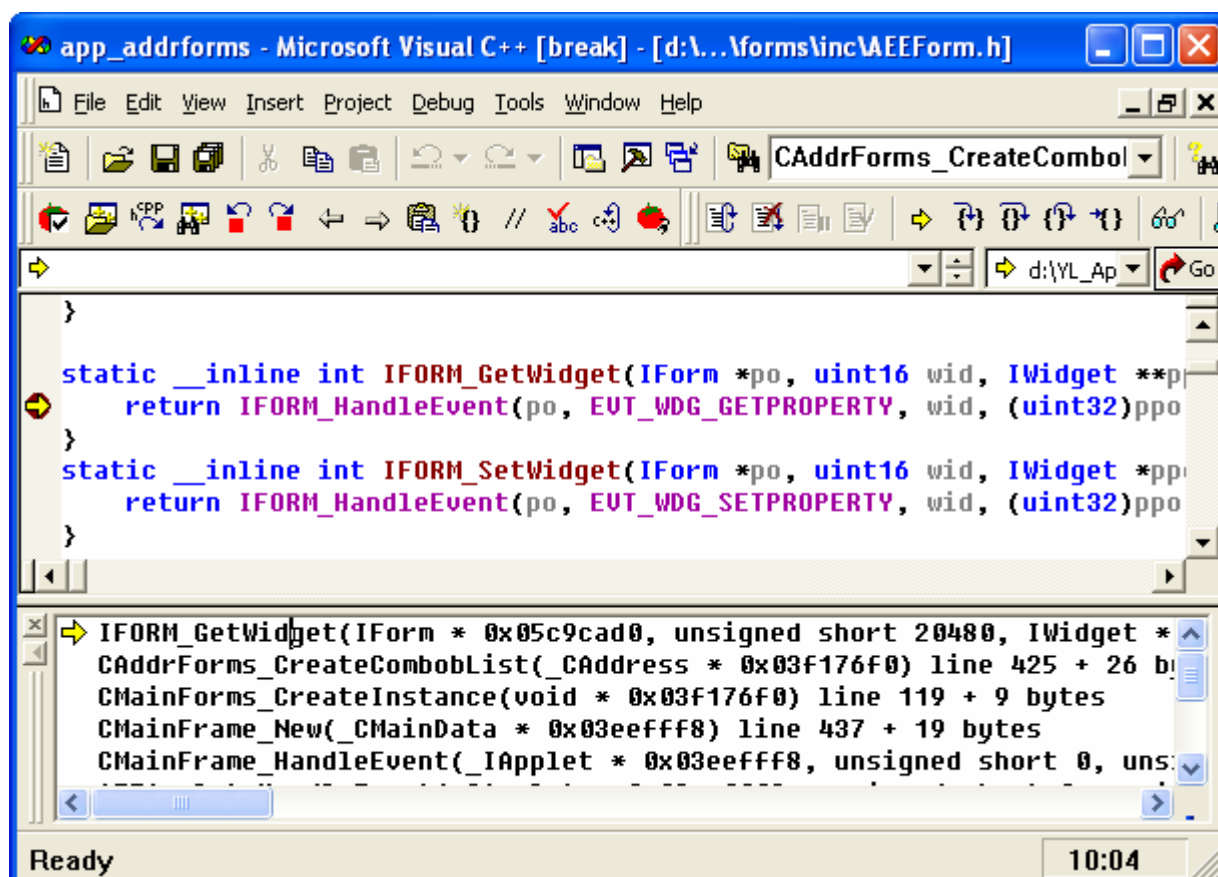
注意：

WNODE_CTOR(me)和后 2 行是把控件放入容器链表，并且是如下概念执行的。

```
#define WNODE_CTOR(p)      ((p)->pNext = (p)->pPrev = (p))
```

容器中控件链表

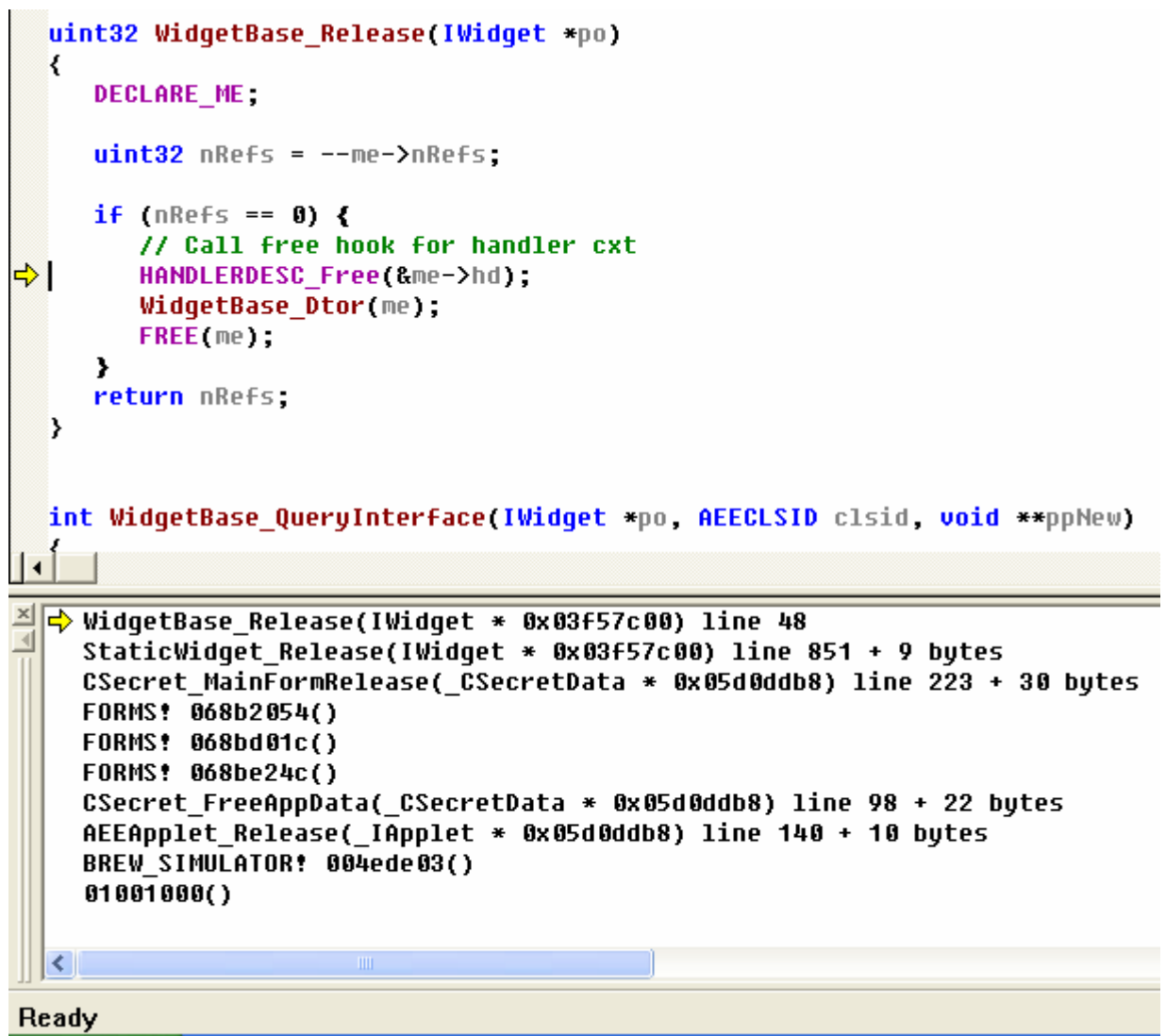
5:函数 (XXX_GetWidget)



```
switch(wParam) {  
    case WID_FORM:  
        *(IWidget **)dwParam = me->piWidget;  
        ADDREF IF(me->piWidget);  
        return TRUE;  
}
```

◆ 哪些窗口减少了引用计数

1: 释放对象函数（如 IWIDGET_Release、IFILEMGR_Release、IXYCONTAINER_Release 等）。下图是调用函数（IWIDGET_Release）释放一个 STATIC 控件的调试信息，从图中可以看到该过程最后也是调用了基类的释放过程。先对控件的引用计数减 1，如果判断是否真的 Release 控件分配的内存。



```
uint32 WidgetBase_Release(IWidget *po)
{
    DECLARE_ME;

    uint32 nRefs = --me->nRefs;

    if (nRefs == 0) {
        // Call free hook for handler cxt
        HANDLERDESC_Free(&me->hd);
        WidgetBase_Dtor(me);
        FREE(me);
    }
    return nRefs;
}

int WidgetBase_QueryInterface(IWidget *po, AEECLSID clsid, void **ppNew)
```

WidgetBase_Release(IWidget * 0x03f57c00) line 48
StaticWidget_Release(IWidget * 0x03f57c00) line 851 + 9 bytes
CSecret_MainFormRelease(_CSecretData * 0x05d0ddb8) line 223 + 30 bytes
FORMS! 068b2054()
FORMS! 068bd01c()
FORMS! 068be24c()
CSecret_FreeAppData(_CSecretData * 0x05d0ddb8) line 98 + 22 bytes
AEEApplet_Release(_IApplet * 0x05d0ddb8) line 140 + 10 bytes
BREW_SIMULATOR! 004ede03()
01001000()

Ready

释放过程跟踪

2: 释放对象函数（RELEASEIF），只需要看看该函数的定义，就能明白。


```

#define RELEASEIF(p)      RELEASEPPIF((IBase**) (void *) &p)

static __inline void RELEASEPPIF(IBase **p)
{
    if (*p) {
        (void)IBASE_Release(*p);
        *p = 0;
    }
}

```

RELEASEIF 函数

这个函数，调用了函数（IBASE_Release），看函数（IBASE_Release）定义如下：

```

#define IBASE_AddRef(p)    GET_PUTBL(p,IBase)->AddRef(p)
#define IBASE_Release(p)  GET_PUTBL(p,IBase)->Release(p)

```

说明这个函数，最后也是和函数（IWIDGET_Release）一样，对控件的引用计数减 1。到引用计数为 0 时，才真的开始释放。

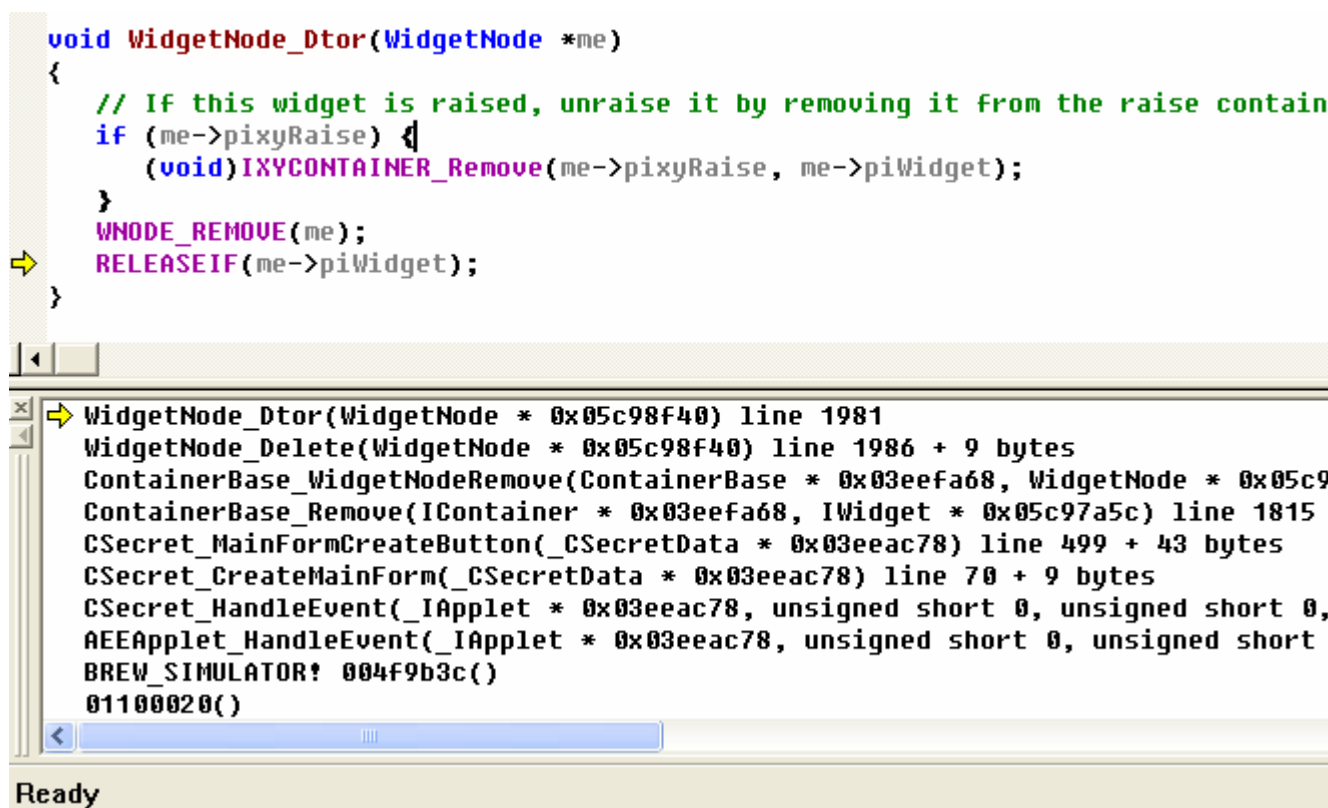
特别注意：

函数（RELEASEIF）和函数（IWIDGET_Release、IXYCONTAINER_Release）最大的区别是，函数（IWIDGET_Release）是有返回值的，可以看到当前控件的引用计数，而（RELEASEIF）不能。

3：从容器中移开控件函数（IXYCONTAINER_Remove）

下图中的调试，显示的是从一个容器的移开一个控件的全过程（插入容器，立刻移出）。最后是从容器节点中把该控件移出并使引用计数减 1。一般情况下当前该控件的引用计数为 2（创建开始为 1，插入容器加 1，因此移出后还为 1，下图的 RELEASEIF 并没有触发该控件的释构函数）。

如下图：



从容器中移开控件

➤ 替换控件默认函数

所有的控件都有默认的事件处理函数，开发人员可以根据需求，在开发过程中把这些默认函数替换为自定义函数，执行自定义函数后再执行默认函数。

这个过程和窗口的替换默认函数过程是一样的，具体也可以参考《如何让静态控件响应焦点事件》。

➤ 如何让静态控件响应焦点事件

◆ 控件响应点击事件的前提

控件需要响应点击事件的前提是，该控件必须有焦点，但 `STATIC` 控件并没有焦点，因此需要替换该控件的默认函数，在该函数中修改静态控件的参数，让控件能响应焦点事件。

静态控件在响应焦点事件经常在 `STATI` 和 `Radio` 以及 `CHECKBOX` 捆绑时使用。

◆ 如何修改默认函数

1: 声明 `HandlerDesc` 数据结构。

```

IWidget                                *m_pMainCommonModelWidget;           //STATIC控件
HandlerDesc                            m_pMainCommonDesc;                   //

```

2: 替换默认函数

```

HANDLERDESC_Init(&pMe->pMain->m_pMainCommonDesc, CSecret_SetSecretCommon_HandleEvent, pMe, 0);
IWIDGET_SetHandler(pMe->pMain->m_pMainCommonModelClsLabel, &pMe->pMain->m_pMainCommonDesc);

```

◆ 处理事件

定义新的函数处理。

```

//-----
//NAME:
//DESC:
//-----
boolean CSecret_SetSecretCommon_HandleEvent(void *po, AEEEvent evt, uint16 wParam, uint32 dwParam)
{
    CSecretData* pMe = (CSecretData*)po;
    boolean dwHandleRes = FALSE;

    switch(evt)
    {
    case EVT_WDG_CANTAKEFOCUS:
        *((boolean *)dwParam) = TRUE;
        return TRUE;

    case EVT_PEN_DOWN:
        return TRUE;

    case EVT_PEN_HITTEST:
        return TRUE;

    case EVT_PEN_UP:
        ISHELL_PostEvent(pMe->a.m_pIShell, pMe->a.clsID, EVT_COMMAND, IDS_COMMON_MODE_ID, dwParam);
        return TRUE;
    }

    dwHandleRes = HANDLERDESC_Call(&pMe->pMain->m_pMainCommonDesc, evt, wParam, dwParam);
    return dwHandleRes;
}

```

函数处理

事件中主要是修改了 EVT_WDG_CANTAKEFOCUS 事件的 dwParam 参数值，设置为可以

有焦点。在点击事件（EVT_PEN_UP）后给主窗口发送事件，通知控件被点击，主窗口根据这个事件响应相关事件。

➤ 焦点和 5 向键顺序

◆ 控件的焦点

有焦点的控件，分为有背景图（如按钮控件）和没背景图（如 TEXT 控件），如果有背景图的控件，需要改变时就设置相应图片，如果按钮图片，一般需要设置普通状态的图片、焦点状态图片、笔按下图片以及不可使用状态 4 种，如果是按钮文字图片，控件内部会对应处理。

◆ 键盘操作规则

标准应用都必须能使用键盘操作，如果只有 5 向键，那使用 5 向键必须能移动到所有有焦点的控件，其它操作可以使用触摸屏来完成。

◆ 5 向键顺序

5 向键的规则是先从左到右，从上到下，选择键响应事件。

➤ 如何创建一个自定义控件

应用窗口规范

➤ 正常窗口

一般设置正常窗口都有以下几项：

◆ 大小

◆ 按钮位置

◆ 应用菜单

◆ 编辑菜单

➤ 进度条窗口

◆ 进度条窗口的关闭和任务取消

出现进度条窗口，一般来说是为需要长时间处理任务才显示出来的，所以中间有可能产生很多不可预知的情况（如硬件异常中断、用户强制关闭、数据产生未知错误、文件系统异常等），应用开发人员首先需要考虑的第一点，应该是任何时候，都能让用户关闭该窗口。

如果该进度条中处理很多任务，某个任务在中间阶段是不能被立刻取消的，但也要能在下一个循环中取消下一个任务。如联系人应用批删除卡中联系人一样，在提交给底层应用删除一条卡联系人，这个时间用户点按钮取消操作，该条记录删除过程已经提交，不能被取消，但下一个循环将取消删除任务。

进度窗口一般都有个定时器，保证在处理时间没有返回，能自动关闭窗口。如联系人写卡过程，如果 15 秒后，底层没有返回，开始显示时间提示，30 秒后自动关闭该窗口。

◆ 进度条标题

标题应该能明确说明该操作的任务，如联系人批删除进度条，应该提示“批删除”或者“批删除联系人”。考虑屏幕宽度有限和英文版本，一般能明确说明操作就可以。

一般不建议进度条显示应用名称，或者“提示”。

◆ 内容或者进度显示

内容的显示，一般可以按 2 种方式显示：

1：显示文字

说明操作状态。如操作过程存在多步进行，比如先删除已经存在的很多文件，再写入新的很多文件，再导入某些数据库，这样的多步处理使用文字说明更为合理。

2：显示数字

操作进度，如果操作就一个步骤，但时间较长的过程，如粘贴一个大的文件，直接显示数字进度较为合理。

存在的误区

进度条的操作进度，和实际的操作进度应该分开，比如文件的 COPY 过程，用不着非常精确。

◆ 窗口大小

进度条窗口可按屏幕高度的 1/3 显示，并且显示在中间和合适位置。

提示：

不建议放在底部，影响视觉。

◆ 进度条按钮大小

一般建议该窗口的取消按钮，设置为标准的 2 个字按钮大小，部分开发人员可能觉的按 3 个字大小来设置按钮大小，可能更为美观，也是可以的。但不推荐比 2 个字标准按钮小，也不推荐比 3 个字按钮大小大。

◆ 进度条窗口的错误现象

《1》放一个 STATIC 控件显示进度。

有些开发人员为了方便，可能就在窗口上放一个 STATIC 控件，当提示进度的时候，显示该控件，并且处理任务，我们不推荐这样的做法。主要原因：

- 1：不美观，今天的智能手机应用不是 10 年前的黑白手机那样，我们特别强调任何影响视觉的边边角角，都应该修改。
- 2：没有标题，不能显示正在操作的具体任务。
- 3：可能需要处理更多的事情，比如把其它按钮、菜单都设置为不可使用等，更为麻烦。

《2》同步刷新数据。

大多数情况，不推荐进度条窗口在执行的时候，操作中的过程同时刷新背后窗口的数据，进度条操作一般都是时间比较长的任务，希望操作完成是越快越好。

《3》显示窗口在中间并且很小。

一般 BREW 开发人员，不是基于 BUIW 的应用，可能习惯放个小窗口在中间，建议窗口宽度应该和屏幕宽一样，两边对齐。

《4》放图片显示进度

放 2 张图片，动画显示，并且没取消按钮，如果任务比较长，不能正确提示用户进度进行位置。

➤ 半屏幕窗口

一般不推荐使用半屏窗口。

◆ 位置

◆ 按钮位置

➤ 全屏窗口

◆ 哪些应用使用了全屏窗口

部分应用可能需要设置全屏，如媒体播放器，拍照和拨号盘应用等。设置全屏或者正常状态的过程，该过程其实就是把托盘显示或者隐藏起来的过程。

◆ 设置全屏窗口

现在开发包提供函数 `SetFullScrn(TRUE)` 就可以设置全屏。`SetFullScrn(FALSE)` 就是正常窗口。

➤ 全屏窗口规范

考虑到协同调用应用的可能，所以建议全屏需要按如下规范。

◆ 应用内部

如果是应用内部，主窗口全屏，全部正常子窗口必须都是全屏；但对话框和本身就半屏窗口不考虑这种情况。

◆ 协同应用

如果是协同调用其它应用，主窗口是全屏，调用协同应用前必须设置为正常窗口状态，因为没法保证所有协同应用都一定能处理全屏，可能导致显示上的不整齐。

如现在出现的情况就是，如果拨号盘按全屏显示后，调用保存联系人接口，结果联系人窗口显示出来出现不整齐的现象，影响视觉美观。

程序<关于>版本号管理规范

程序版权规范

➤ 版权

加强版权规范，每个程序的所有代码文件，都需要在页头加入版权信息，强调哪怕就一行代码，也必须有版权信息。版权信息的具体内容，可参考相关部门提供的相关文档。

➤ 作者和修改内容

在版权信息的后面，必须加上程序名称（如宇龙 268 联系人）、作者（如西门吹雪）、日期（如 2006-05-05）以及修改的内容。

我们强调版权信息，如果发现应用的代码文件，没有版权信息，将给予重 X。

程序划屏处理规范

➤ 135 度斜线划屏

每个应用，在任何窗口，按 135 度角划屏幕，都应该显示相关应用的关于窗口。由于这样的处理方式很特殊，所以建议在显示该窗口的时候，把如果不是当前应用的窗口或者是其它对象的窗口都关闭。

如果顶层窗口是当前应用的对话框，建议把对话框也关闭。

➤ 90 度斜线划屏（改变私密状态）

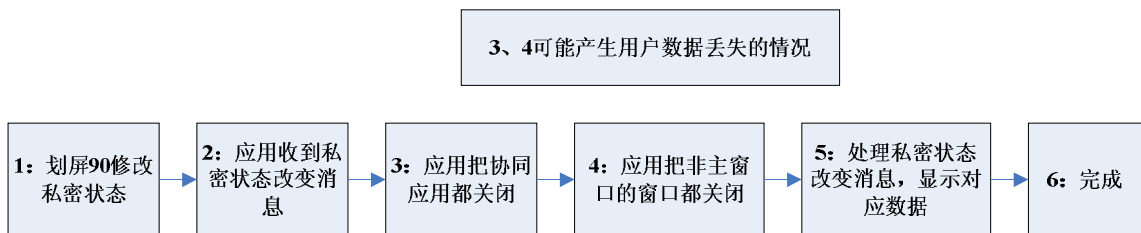
从桌面进行 90 度划屏幕，显示私密用窗口。如果私密状态改变，和私密有关系的应用，如联系人，短信、日程等，都需要处理私密改变消息（EVT_PRIVATE）。

◆ 应用该如何处理收到的私密消息

窗口在收到私密消息后，和私密有关的应用（如联系人、短信、日程、通话记录等）都需要处理该消息，看了很多应用的代码，各开发人员都有自己的处理方式。

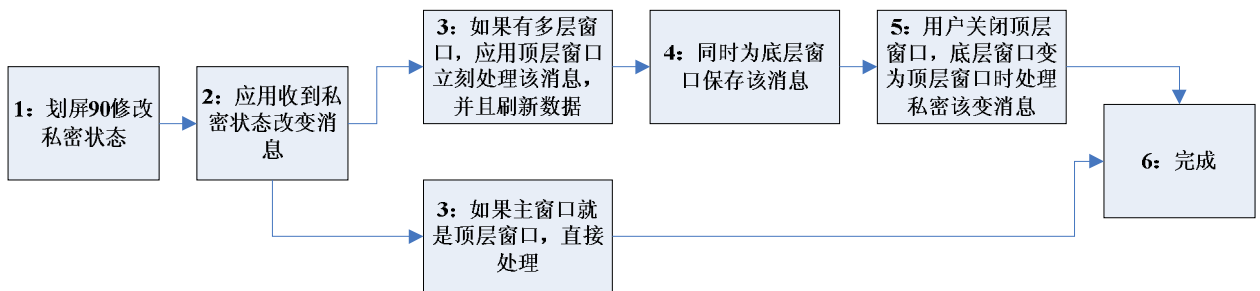
这里列出 2 种私密状态改变的处理方式。

第一种方式，收到消息，立刻把所有不是主窗口的窗口都关闭，在主窗口中立刻处理，虽然简单了很多，但存在用户输入的数据丢失的风险。



收到消息，关闭非主窗口的所有窗口

第二种方式，收到消息，顶层窗口立刻处理该消息，用户立刻能看到私密状态改变的效果，并且保存该消息，如果用户关闭顶层窗口，下层窗口将收到变为顶层出口的消息，立刻处理私密状态改变消息，直到所有窗口都被处理。显然处理非常复杂了很多，但中间没有导致用户输入的数据丢失的可能。



保存消息每窗口处理的过程

用户数据是不可丢失的关键数据，所以开发人员在开发过程中，必须尽最大可能保存用户输入的数据，任何情况都必须保证用户数据的安全性，想象如果一个用户输入了很多数据，却在最后丢失了，那对用户来说将是非常郁闷的事情，也应该是开发人员非常遗憾的事情。

程序异常处理

➤ 程序异常意识

开发人员应该时刻有异常意识，由于终端一般是用户界面操作，界面的操作无法预料是否会按设计者所想的一步步处理，所以任何地方，都需要处理异常情况。

BREW 使用的是真实的内存，而不是 WINDOWS 那样虚拟的内存，所以一旦出现内存异常，一般就是系统崩溃或者内存泄露，也就是重新启动或者出现 BPOINT 1，这两者都不能接受。

异常真是无处不在，所以不能都罗列出来，我们大概列举了几种经常产生的异常。

◆ 内存没释放

从以往的经验来看，95%以上的异常都是由于内存没释放或者重复释放产生的。我们将在注意事项中专门讲述 BPOINT1 和 BPOINT3 的问题。

由于内存没释放，经常就是用着用着就没内存了；内存重复释放，可能情况就是系统突然崩溃。

◆ 内存不足的异常

这种情况很容易产生，终端内存非常紧张，所以内存不足是必须时时刻刻首先考虑的问题；分配内存后，一定要检查内存是否已经分配到。如果没有，应该进行异常处理，或者提示失败（一般情况是连提示需要分配的内存也分配不到，只能等着系统崩溃）。

如果内存不足，创建对象或者窗口，都可能产生失败，但根据以往的经验，如果到由于内存不足而创建对象失败的时候，一般也就是只能等着重新启动机器了。

◆ 用户强制关闭应用的异常（AVK_END）

时刻提醒用户可能强制关闭应用，所以在任何时候，都需要处理这个可能。在任何窗口用户按这个键，都要能保证程序被关闭或者退到后台执行而没内存问题。

◆ 资源数据错误的异常

程序在载入资源时，需要检查资源是否被真的载入了，如果载入失败，需要目录或者资源文件是否存在，这样的情况一般发生在调试阶段。

我们有发现资源文件存在但资源部分项载入失败的问题，将在注意事项中说明。

◆ 用户数据错误的异常

1：如果是用户输入数据，需要检查数据是否合法，是否超长等问题。数据合法性的检查是非常重要的。

2：中间断电或者读取数据失败，是否可能产生问题，都需要检查。

3：系统文件损坏，或者配置文件损坏或者配置文件被用户删除等情况都有可能产生。

➤ 使用 goto 处理异常

◆ 正确使用 goto 语句

有时一个函数需要处理很多步骤，这些步骤可能需要分配多次内存、读取多个配置文件、进

行多步操作等情况，如果中间产生错误，将立刻返回，但这个时候如果是使用 `return` 直接返回，每个 `return` 前将需要处理释放分配的内存、设置内存指针为 `NULL` 等情况，如果步骤很多，将象滚雪球一样越来越大，显然处理就有点罗嗦。

使用 `goto` 语句，当如果产生异常，就直接 `goto` 到对应的标签位置，善后处理 1 次就可以。

◆ 不要滥用 `goto` 语句

不要滥用 `goto` 语句，如果一个函数中有多个 `goto` 处理标签，将有可能产生混乱，比如从一个标签在 `goto` 到另外一个标签，再跳到另外一个标签，将非常容易产生问题混乱。

一个函数中使用一个标签，并且就处理异常情况，这样的 `goto` 语句能使应用异常处理更为清晰，但如果滥用 `goto` 将使代码向面条一样混乱，代码无法理清逻辑。

➤ 异常的提示信息

◆ 准确标题信息

提示信息的标题，应该根据当前的情况来判定，是该使用“提示”还是该使用当前应用的名称，需要自己斟酌；比如当前在联系人应用中，提示用户需要输入数据才能保存，那标题为“提示”即可；但如果是拨号盘调用联系人，联系人需要提示内存不足，那标题应该为“联系人”，表明这是联系人应用提示的内容。

对提示的标题，一般是“提示”还是应用名称，或者“错误”，不做特别限制，但标题不应该出现“警告”这样的提示信息。

◆ 准确的内容提示

提示的内容，应该做到以下 3 点：

- 1：字字斟酌，语气和谐。
- 2：内容准确，表达明了。
- 3：不应该出现“你”、“您”这样的称呼。

◆ 准确的图标

提示的图标有 4 种，分别是提示信息、警告信息、确认信息和一般信息，这 4 种图标表达的意思是不一样的，需要分清楚。

程序互斥规范

程序自动化编译规范

程序宏定义规范

➤ 应用内部的宏定义

对使用宏定义来判断哪些开关需要打开和关闭的情况，更加希望这个宏的名称应该非常容易读懂，不只是自己读懂，更希望其他读者能读懂。

《1》 经常使用的一些数字，或者可能修改的一些值，并且在很多地方使用，设置为宏可以更为方便，如果联系人话机的记录总数、判断最少需要内存总数,如

```
#define LKM_MAXRECORD_NUM      1100           //话机最大记录个数
#define ADDR_MIN_MEM_2M        (1024*1024*2)   //联系人最少需要 2M
内存
```

《2》 宏的定义应该能直接表达用途，下面的宏定，是存在缺陷的。

```
#define comic                  //数码图册接口开关

#define DOUBLE_USB_DISK       //双U盘开关

#ifndef DOUBLE_USB_DISK
    #ifdef AEE_STATIC
        #define EXPLORE_ROOT_DIR    AEEFS_NAND_ROOT
    #else
        #define EXPLORE_ROOT_DIR    AEEFS_ROOT_DIR
    #endif
#endif

#ifdef DOUBLE_USB_DISK
char* EXPLORE_ROOT_DIR = NULL;
#endif
```

不规范的宏定义

上图中的宏（comic）这个宏定义不规范，如果不看注释，不明白什么意思。宏（EXPLORE_ROOT_DIR）的定义更是不可接受，这样的定义将极大可能产生问题。

➤ 应用间的宏定义

如果几个应用共用一个宏定义，建议该宏放到一个公共头文件中。使用 `extern` 引用的做法不是很合理。

➤ 宏定义的名称

名称应该能表示该宏的意义，并且能保证该宏不会被其它应用重复定义，如果重复定义，可能编译不了，也可能产生不可预知的错误。

如果有计算的

调试信息规范

➤ 日志文件

◆ 日志文件的目录和大小。

如果应用能通过 QDXM 的调试信息，把一般出现的问题都解决，那不推荐使用写文件的形式来保存日志，但是如通讯应用、闹钟应用等这样的应用，不可预测什么时间会产生问题或者很难重现，那还是需要以写文件的形式保存日志。

日志文件一般应该放到 USB 盘目录下，方便读取查看，当然也可以放当前应用的固定目录下。

日志文件应该不要超过 100K，如果文件太大，频繁读写速度慢，并且会影响系统运行的速度，我们曾经看到一个日志超过 400K，频繁的读写，导致任何操作都非常缓慢。

◆ 否写日志

通过宏来判断是否写日志，一般不要使用时间或者日期来判断是否写日志。

◆ 正式版本

发布正式版本应该非常容易就能去掉**写文件形式**的调试信息。

➤ QDXM 调试信息

◆ 不要频繁打印调试信息

不要频繁打印调试信息，如定时器每秒打 50 条调试信息和内存查看每秒打印几百条信息；不可以在 xxx_HandleEvent 事件开始加任何调试信息，在该处加调试信息可能导致每秒几百条调试信息；

一般的调试信息不影响速度，但非常多的调试信息，影响速度并且找不到有价值的调试信息。

◆ 使用中文

调试信息尽可能是用中文。

◆ 内容准确不罗嗦

调试信息应该简单明了，不要使用个人名称或者一大串 (!@#%^&*=_+) 号什么的。虽然这样的调试信息个人很容易找到，但看起来非常不雅观，显得罗嗦。

全局变量和__inline 函数

➤ 全局变量

应用中间需要协同处理，有些对象可能同时被其它应用使用，所有使用全局变量的时候，需要仔细考虑，应该做到以下 3 点。

◆ 慎用全局变量

全局变量应该做到少用慎用，如果能不使用全局变量，尽量不要使用。

◆ 命名全局变量

如果定义了全局变量，建议使用_gXxxxx 这样的格式来定义这个变量，并且一定要保证这个变量必须是唯一的。

◆ 修改全局变量

修改全局变量的值，应该通过全局变量维护者提供函数来修改，并且需要打出调用应用的调试信息，保证值被修改后能知道谁修改了。

➤ __inline 函数

◆ 优缺点

手机空间不是很多，建议函数少用__inline 关键字，虽然会提高运行该函数的运行速度，但该__inline 函数会增加 bin 文件的空间。

◆ 哪些函数建议使用__inline

如果应用中有些函数，需要要求效率特别高，需要特别注意速度，那该函数应该使用__inline 关键字（比如需要排序的应用）；否则一般不建议使用，大部分的应用是不需要这么高性能的。

大数据量处理 CPU 时间限制

由于受限于应用独占 CPU 时间限制问题，一般不使用 FOR、WHILE 处理大数据的连续任务，一般来说，我们可以使用以下 3 种方式，把任务划分为小块处理。

如果能估计判断 FOR、WHILE 执行的时间少于 5 秒，那放心使用。

➤ 为什么不能使用 FOR、WHILE 连续处理大数据量

◆ CPU 时间限制

理论上现在 UI 可以一直占用 CPU 时间为 30 秒左右，但实际上 UI 大概最多只能占用 16 秒左右，如果一直占用 CPU 时间超过 16 秒非常有可能将直接导致系统崩溃。所以在程序中如果需要处理某些复杂的过程（如压缩文件），需要把任务过程划分成小块处理，每小任务占用的 CPU 时间不应该超过 5 秒，目的就是让系统在处理该任务的时候，还有机会处理其它应用（如同时拨打电话，播放媒体文件等）。

这个限制特别体现在系统繁忙的情况，非常突出。

◆ 提示

不要因为不能处理大数据量而都不使用 FOR、WHILE 处理其它任务。

➤ 使用 ISHELL_POSTEVENT 消息处理。

◆ 消息机制

通过发送消息给系统，系统在下一消息处理循环后发回来继续进行，但这样的处理机制需要应用维护中间状态的变量；一般来说，这种机制非常稳定，如果中间有其它应用需要运行，如电话拨入拨出、多媒体同时播放等，当前应用转到后台后也能稳定执行。

◆ 处理消息位置

消息机制和 Windows 的机制是一样的，但推荐把消息都发送到窗口处理函数下（**EVT_COMMAND**）事件中，应用从这里开始消息事件，消息参数 wParam、dwParam 可以根据需要自定义。

◆ 消息丢失问题

消息函数（ISHELL_PostEvent）理论上是有非常小的可能消息丢失，但一般情况下不会的，该几率应该小于万分之一（系统极度繁忙的情况）；以前曾经测试过这个函数，一百万次的执行，没发现有丢失问题。

既然有丢失可能，那应用就需要多考虑丢失的情况产生的异常问题。

◆ 性能问题

和连续处理 FOR、WHILE 循环相比，效率稍微低些，但一般可以接受，并且暴力测试也是非常稳定。

◆ 休眠状态

需要注意，在进入大批量处理数据时，不要让系统进入休眠状态，应该在处理完成后，才进入休眠状态。如何让程序不进入睡眠状态，参考（**不让手机进入休眠状态**）

➤ 使用 ISHELL_SETTIMER

定时器函数（ISHELL_SETTIMER）经常被使用，回调函数是指在从当前时间开始到指定豪

秒数后被执行，具体该函数处理过程，可以参考帮助文件。

在使用该函数过程中，需要特别注意和考虑下列情况：

◆ 休眠挂起状态

当系统进入休眠后，为了节能，定时器可能会被挂起，这时回调函数可能不会被执行，所以对和时间有关系的应用，需要在唤醒时立刻去读系统时间，否则可能导致时间不正确。（如时间助手中的计时器，当系统休眠时，屏幕关闭，定时器将不被执行，中间唤醒后，立刻读系统时间才能显示正确）。

◆ 解决系统休眠

可以通过获取背光状态，比如设置背光状态为 100（一直亮着），在应用运行完成后设置回原来的背光状态，可以解决休眠挂起的问题。

◆ 取消定时器

定时器使用完成后，一定要使用（ISHELL_CancelTimer）取消该定时器，如果不取消，不能保证是否会产生未知问题。

◆ 定时器间隔周期

函数参数（dwMSecs 计时器有效期）一般不能小于 50，特别是不能为 0ms、1ms 和负值，太小的值会导致系统极度繁忙而解决不了暴力测试的问题。

有时为了系统稳定，可能通过牺牲效率换取系统安全运行的可能，间隔周期太大和太小都是合理。

◆ 周期和暴力测试问题

如果回调函数处理的过程比较复杂，处理周期比较长，建议 dwMSecs 设置比较大些；让系统空闲些能处理其它应用的任务；在以往的测试过程中，发现周期处理长的情况（如批删除文件、联系人日常的备份恢复等），如果参数（dwMSecs）设置比较小如(50ms),暴力测试也是有可能导致系统繁忙而崩溃。

➤ ISHELL_Resume 函数处理重复执行的过程

函数（ISHELL_Resume）是向 AEE 外壳注册回调，AEE 外壳将在下一次调用事件循环时调用此回调函数，一般情况下，系统一有空闲时间，该函数就会被调用。该函数在处理大量数据的时候，频繁被使用，如联系人备份和恢复。

◆ 效率问题

该函数在效率上要比（ISHELL_SETTIMER）函数高，因为该函数在系统有空闲就被执行，显然系统将一直处于繁忙状态。

◆ 休眠问题

不会因为系统休眠就停止。

◆ 取消回调

使用完成后，必须使用（CALLBACK_Cancel）取消。在使用过程中，发现异常情况下处理的可能结果是导致内存泄漏（point1）。所有有使用到（AEECallback 结构的对象）都必须使用（CALLBACK_Cancel）取消。特别是用户按（AVK_END）的情况，需要安全处理。

◆ 暴力测试问题

系统极度繁忙的时候，暴力测试更容易崩溃。

◆ 如何使用

下图中显示如何使用该函数。

```
//在pMe声明 AEECallback 结构数据
AEECallback      pRecAllDelGroupCB;

pMe->dwCurDelPos = 0; //回调函数开始，维护中间状态值
pMe->pRecAllDelGroupCB.pNotifyData = (void *)pMe;
pMe->pRecAllDelGroupCB.pfnNotify = xLinkMan_RecovAllDelGroupCallBack;
pMe->pRecAllDelGroupCB.pfnCancel = NULL;
ISHELL_Resume(pMe->piShell,&pMe->pRecAllDelGroupCB);
return SUCCESS;
```

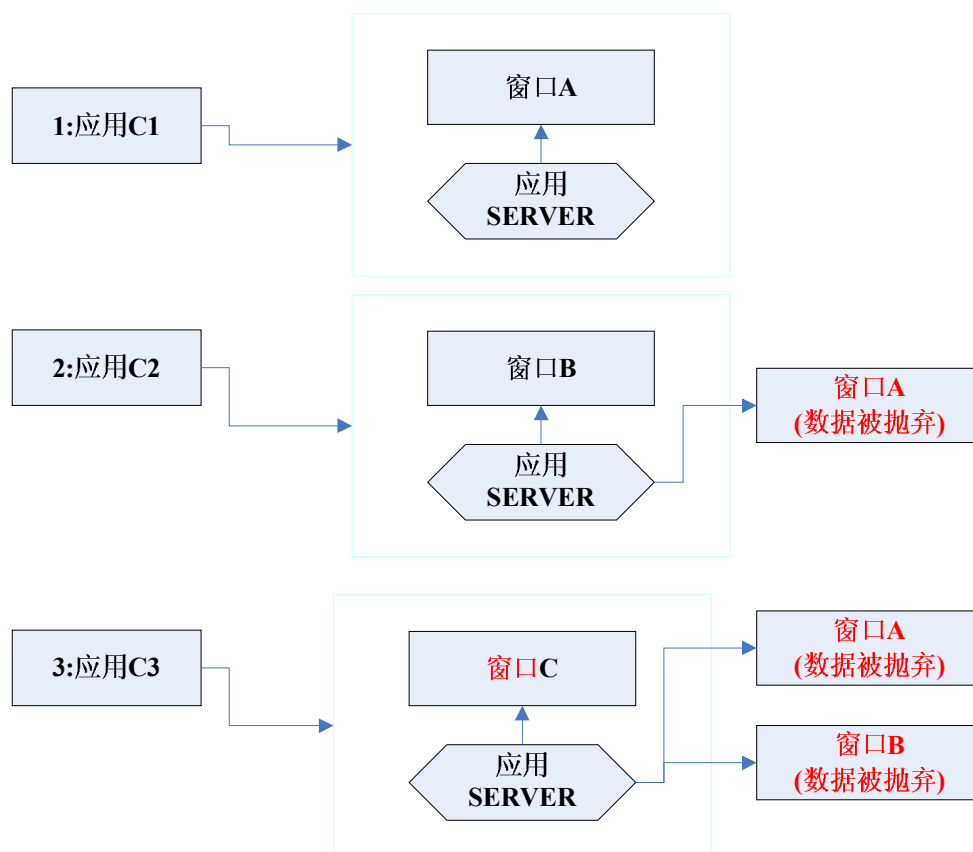
使用 ISHELL_REMUSE

带窗口的接口或对象规范

一般来说，BREW 的应用大部分是独立处理业务的，即应用就一个对象实例存在，这个类型的应用不管是自己内部处理业务，还是外部应用协同处理，都使用相同的对象实例。

➤ 单实例对象

单个实例对象的应用，其数据处理过程如下图：



单实例对象图

当应用 C1 启动应用（如上图 SERVER）时，提交了数据，这时候窗口 A 保存了数据，但还没处理完成的时候，应用 C2 也启动了应用（SERVER），窗口 A 的数据将被抛弃，如果应用 C3 再启动应用（SERVER）的时候，窗口 B 的数据也被抛弃，即同一时间只能保存一份数据（如操作按 1、2、3 进行时，只有窗口 C 的数据被保留下来）。

➤ 单实例对象优缺点

这样的处理模式，有优点也存在缺点：

◆ 优点

- 1：应用（SERVER）将比较容易处理，开发过程相对简单些，处理边角问题相对比较少些，只要保证应用内部数据安全，就没其它问题了；其它应用传递参数可以通过带参数启动，处理不麻烦。
- 2：系统内存需求相对较少。
- 3：和其它应用关联少。

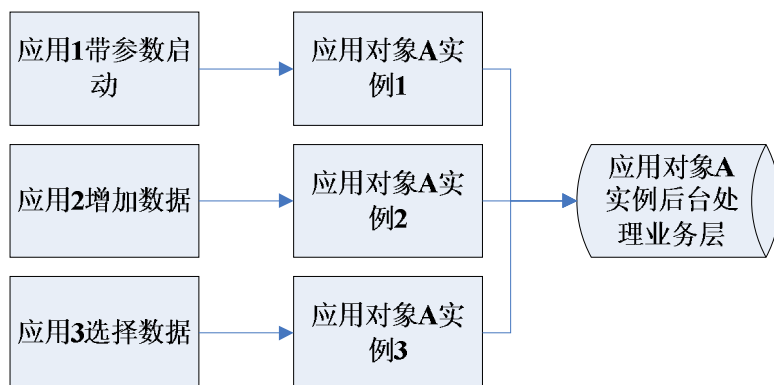
◆ 缺点

由于同一时间只能保存一份数据，部分用户数据在多应用同时处理业务过程中，可能导致用户数据丢失，这在一般用户操作中是不可忍受。

➤ 多实例对象

但也有些任务,需要几个应用之间需要协同处理任务(如短信、邮件需要选择联系人等)。为处理协同过程,应用可能需要包装成一个对象,在某些条件下,该对象可能被同时创建几个实例(如在邮件选择联系窗口打开,创建了联系人对象实例,同时打开短信选择联系人,再创建一个联系人对象实例,2个对象都是独立存在的)。

下图是多实例处理一个业务的过程:



多实例对象图

由于采用多实例处理模式,应用对象(如图中应用对象A)将以控件模式,提供给应用(如图的应用1),这和WINDOWS的带窗口的DLL是一样的。每个对象实例在表面上是独立存在的,业务处理在写入后台处理层前都是独立的。

➤ 多实例对象优缺点

多实例处理比单实例麻烦很多,但为保证用户数据安全以及多应用协同处理业务,一般建议还是采用这种方式实现。

◆ 数据安全

不存在用户从当前应用输入了内容,从其它应用进入后数据把已经输入的数据丢失的可能,保证了数据的安全性。

◆ 内存需求较大

一般终端设备内存都比较紧张,因此在创建多实例的时候,经常都需要判断现有内存是否满足创建多对象需要的内存空间。当内存非常少并且跑几个实例对象,更大可能导致内存紧张。

◆ 应用程序更为复杂

当多个对象同时存在的时候,需要处理临界区和安全锁等问题,同时需要处理好各个对象的数据同步问题,来保证每实例数据的更新同步。

如果没处理好这个问题,可能最糟糕的情况就是同时写入同一条记录,记录结果却被另外一个实例对象写入的另一条记录修改了,导致数据产生错误或者记录不对而用户却不知道。

◆ 更多的异常处理

需要更多的异常处理来保证用户进行的操作都能被安全执行。

◆ 释放更为麻烦

多实例对象可能存在 2 个或者更多个，并且可能每对象可能存在很多个窗口，在最后释放或者在处理 BREW 标准按键处理(AVK_END)时，非常的困难。

多实例对象的标准规范：

➤ 标准创建接口

必须提供标准创建接口，使用 CLSID 就可以创建该对象，现在很多应用在这方面都不标准，有些是提供了 CLSID 和创建接口，但其它部分（如参数设置）不标准。

我们特别强调，不能直接使用定义一个数字宏来确定一个 CLSID, CLSID 必须是包含在 BID 文件中。

➤ 标准 Release 接口

必须实现标准（XXX_Release）接口，调用该接口，当引用计数为 0 的时候，能全部清空对象内部数据。

➤ 能被动移出（_REMOVE）

◆ 能动态移出所有窗口

如果对象内部可能存在多个窗口，该对象必须能保证所有窗口被 Remove 出来而不产生错误。这个过程应该伴随着该对象被释放的过程。

◆ 不能只移出一个顶部窗口

如果该对象存在多个活动窗口，不能只移出一个窗口。因为这样不能保证对象数据完整性。

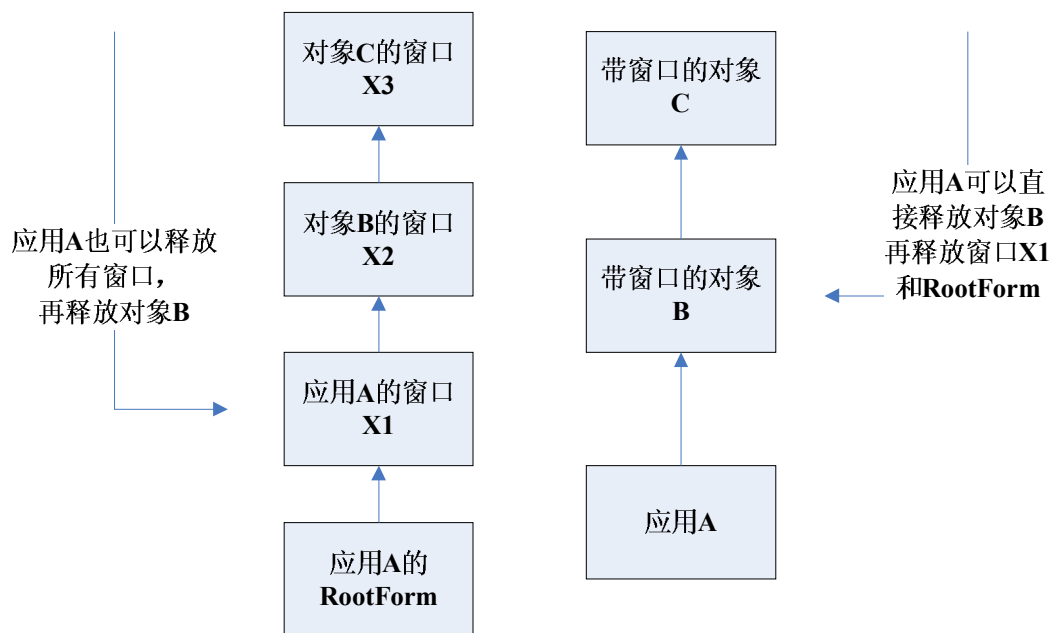
➤ 多窗口 Release

◆ 对象内部有多个窗口

对象内部有多个窗口，其它调用该对象的应用，不管是使用直接释放该对象还是使用还是先释放所有窗口，再释放对象，都应该支持。

◆ 对象中还创建其它对象

如果对象中还存在创建的其它应用的对象接口或者窗口，（XXX_Release）接口也能保证所有数据被释放干净，将在后面更为详细说明这个释放顺序问题。



多对象的释放

这个释放过程存在 2 种情况：

1：应用 A 直接释放对象 B，对象 B 将释放对象 C 和它的窗口以及对象 B 的所有窗口；最后再释放应用 A 的所有窗口。

2：应用 A 直接释放所有窗口，当释放了对象 C 的窗口 X3 后，开始释放窗口 X2，这时将在释放 X2 过程中把对象 C 释放干净。同时在释放 X1 的过程中把对象 B 释放干净。然后再释放应用 A 的其它数据。

提示：

可能存在 ABC 这 3 个对象，但不许可再创建第 4 个对象，太多层次容易产生问题，并且更容易产生内存不足等问题。

➤ 内存

◆ 应用检查内存

应用启动需要检查内存，这在应用基本规范部分已经说明。

◆ 对象需要检查内存

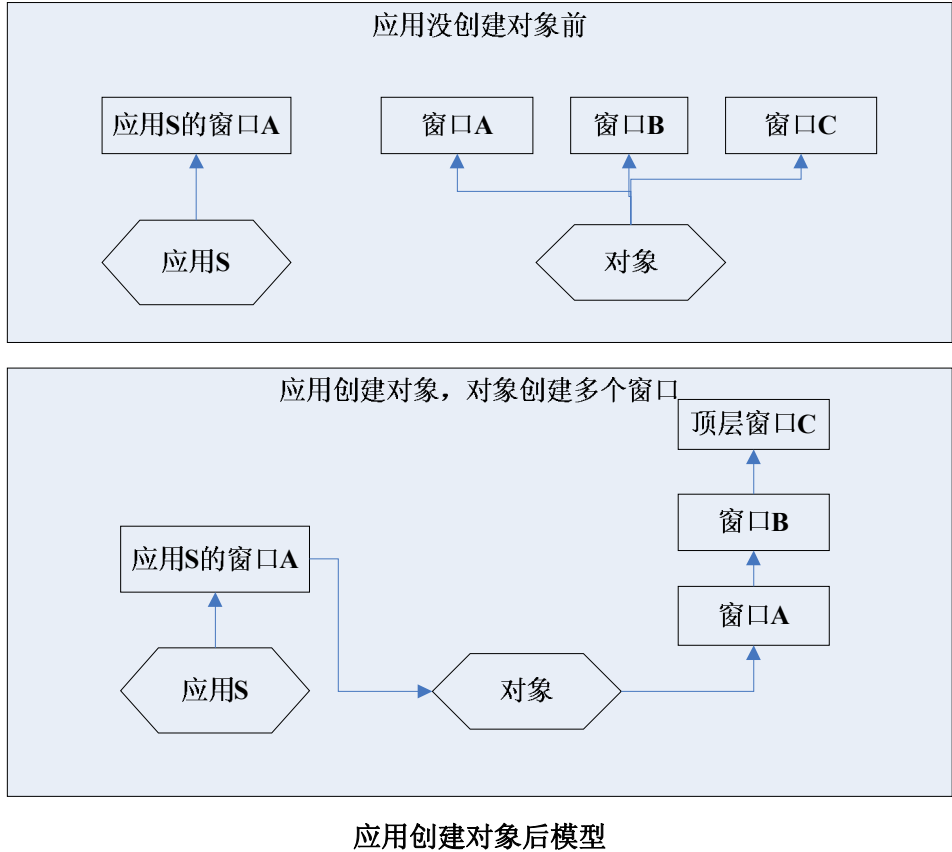
对象被创建前，需要检查系统内存是否足够，这个过程应该在对象内部进行检查，即原则就是谁需要内存谁自己检查内存是否足够。

➤ 对象示例

我们给出一个多实例的基本模型，分别是对象创建前后的关系，先释放所有窗口的过程和主

动释放对象的过程，实际程序开发，应该比这个模型复杂很多。整个过程大概如下图

- ◆ 应用程序和对象创建前
 - ◆ 应用创建了对象后
- 这 2 个过程如下图。

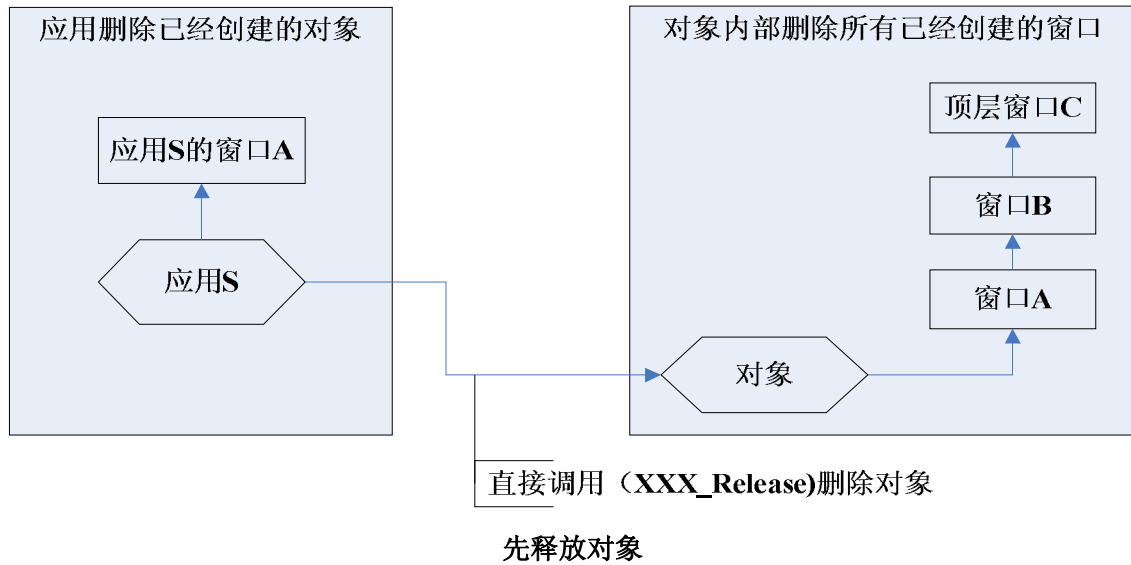


- ◆ 把所有窗口都释放

```
IForm *pTopForm = NULL;
pTopForm = IROOTFORM_GetTopForm(pMe->pRoot);
while (pTopForm)
{
    DBGPRINTF("删除窗口!");
    IROOTFORM_PopForm(pMe->pRoot);
    IFORM_Release(pTopForm);
    pTopForm = IROOTFORM_GetTopForm(pMe->pRoot);
}
```

释放所有窗口

- ◆ 先释放对象

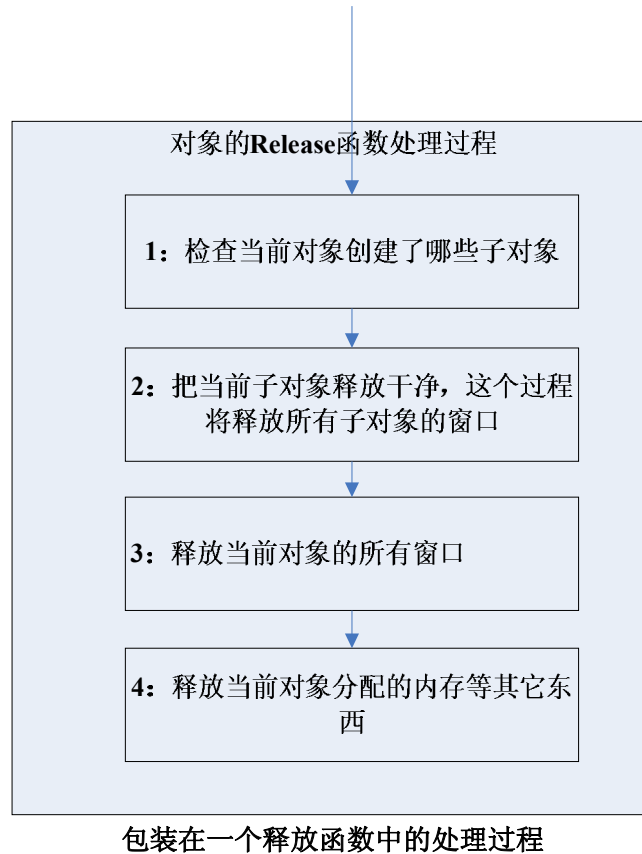


➤ 释放函数示例

我们知道可能存在一个问题，该在什么地方释放这个过程呢？该函数怎么处理才比较安全？

◆ 释放函数该处理过程

释放函数必须包装在一个过程中，并且必须先检查该对象可能创建了几个子对象，因为子对象的存在，可能就是顶层窗口是属于该子对象的。所以就要先释放掉子对象，然后再释放自己内部的窗口。



◆ 对象的数据

估计做为对象提供，该对象应该都有很对数据需要维护，因此这些数据也窗口无关，可以在最后释放，才能保证所有数据都被清除干净。

◆ 误区

- 1: 窗口的数据和对象的数据结构混在一起，没法在释放窗口后，单独释放对象。
- 2: 释放函数不合理，不能取到当前对象下的所有可能创建的子对象。
- 3: 没法知道当前对象下，已经打开了几个子窗口。
- 4: 判断每个窗口是否存在，再一个一个释放，可能产生顺序不对。

ARM 编译项

➤ 如何在把应用编译入手机 BIN 文件

我们以一个应用（app_dict）编译到手机需要的步骤和过程进行说明：

◆ 把应用放到编译目录下

把 app_dict 文件复制到 main/目录下，该应用程序必须在模拟器上基本编译完成，把警告信息大部分都去掉。

◆ 在 OEMModTableExt.c 文件中增加

该文件在目录 (F:\Newprj3.35.L_2618_20080409\brew_3.1\porting\oem\src) 下可以找到。

```
extern int dict_Load(IShell *ps, void * pHelpers, IModule ** pMod);
```

```
和 {AEEFS_MIF_DIR"app_dict.mif", dict_Load},
```

◆ 在 incpath.min 文件中增加

```
DICTAPP= $(SRCROOT)/ui/main/app_dict 和 $(DICTAPP)
```

这里相当于把应用的目录加到编译环境中，这样 ARM 编译，就能在相应的 (SXNAX.dep) 中找到。

◆ 在 dmss_qsc60x0.mak 文件中增加：

```
$(DICT_APP_OBJS)
```

需要注意的是，这个 ID 应该是在编译应用的 min 文件中定义的。

◆ 在 dmss_objects.min 文件中增加：

```
include $(DICTAPP)/app_dict.min
```

这个过程就是把应用的编译项加入编译环境。ARM 将找到该文件并且按文件定义的规则进行编译。

◆ 在 dmss_rules.min 中增加，

前提是把 dictlib.a 库文件放到..\build\ms\SXNRX\libs\底下

```
QCTLIBS := $(QCTLIBS) $(LIBDIR)/dictlib.a
```

➤ 如何修改 min 文件

◆ min 文件的意义

该文件的意义在于定义了编译的内容，ARM 将按该编译规则进行编译。

◆ 增加 C 文件

◆ 注意事项

1: 空格的问题。

➤ nand 和 nor 的区别

➤ 设置文件系统区

➤ 性能优化

- ◆ 性能优化的需求
- ◆ 显示过程的优化
- ◆ 资源载入的优化

开发注意事项

➤ 如何在模拟上调试唤醒挂起

应用程序经常需要模拟器上模拟唤醒和挂起的行为，在手机上就是挂到后台去执行的过程，或者挂起就自动关闭应用的过程。

模拟器模拟挂起的行为非常简单，就是菜单里点一个带窗口的菜单，如载入设备，然后就应用就会收到挂起的消息，点取消载入设备，应用收到唤醒的消息。

➤ 如何让系统不进入休眠状态

有些应用可能需要比较长时间处理任务，可能超过几分钟，如联系人的备份恢复，批删除几百条可联系人等操作，时间都有可能超长，一般情况的手机，为了节能，在 15 秒或者 30 秒屏幕没响应，将自动进入休眠状态（省电模式），在进入这种模式后，定时器就有可能不被执行，因此，长时间处理任务的时候，需要让系统不进入休眠状态，直到任务处理完成。

一般整个操作可以分 2 步。

◆ 获取当前系统的背光值

当长时间任务处理开始时，显示进度条窗口后，先获取当前窗口的背光值，保存下来（该过程一般在窗口创建后就获取）如：

```
#ifdef AEE_STATIC
    pMe->pProcDlg->dwSaveBackLight = GetBacklightEx(); // 获取原来背光值
    SetBacklightLocal(TRUE, 100); // 设置背光常亮
#endif
```

获取背光值

◆ 取消背光

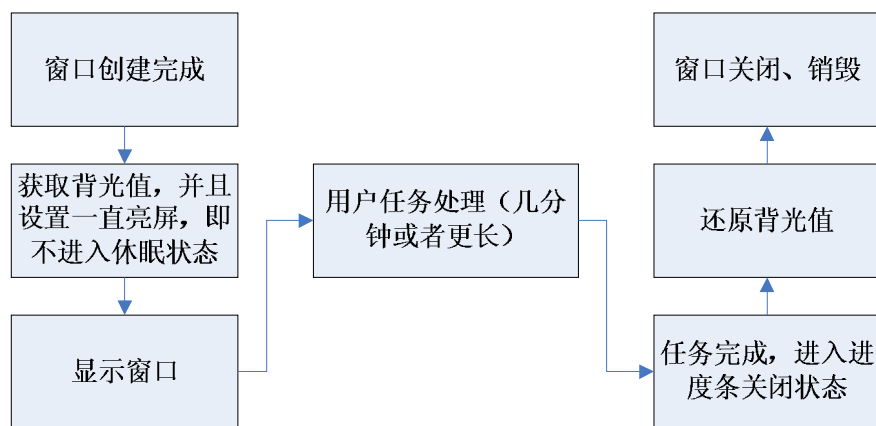
当任务处理完成后，需要把背光设置为用户设置的背光值（该过程一般在进度条窗口被关闭时处理）。

```
#ifdef AEE_STATIC
    SetBacklightLocal(TRUE, pMe->pProcDlg->dwSaveBackLight); // 还原背光值
#endif
```

还原背光值

◆ 图示

一般全操作过程如下图：



背光的过程

➤ UI 界面应用和底层应用交互的过程

很多应用在处理任务时，都有可能需要和底层有关系，如闹钟和提醒的过程，联系人写卡和删除卡联系人的过程，短信的发送过程等。

我们这里以联系人写入卡的过程为例子，详细说明该操作过程的步骤。

◆ 向底层注册回调函数

联系人应用在准备向卡写入数据的过程前，需要判断卡是否能写入（是否有空间，是否处于可写状态）。然后向底层应用注册回调函数。

◆ 开始向底层写入数据

联系人应用在检查完成数据参数正确后，显示进度条窗口，然后向底层写入数据，并且保存该数据（名称和电话号码，数据 ID 等）。

◆ 底层调用回调函数

底层在写入完成后，回调函数，联系人获得已经写入成功或者失败的结果。联系人获得

结果后，给自己发送一条消息，让回调函数立刻返回。因为回调函数不应该阻塞。

◆ 更新数据和相关模块数据

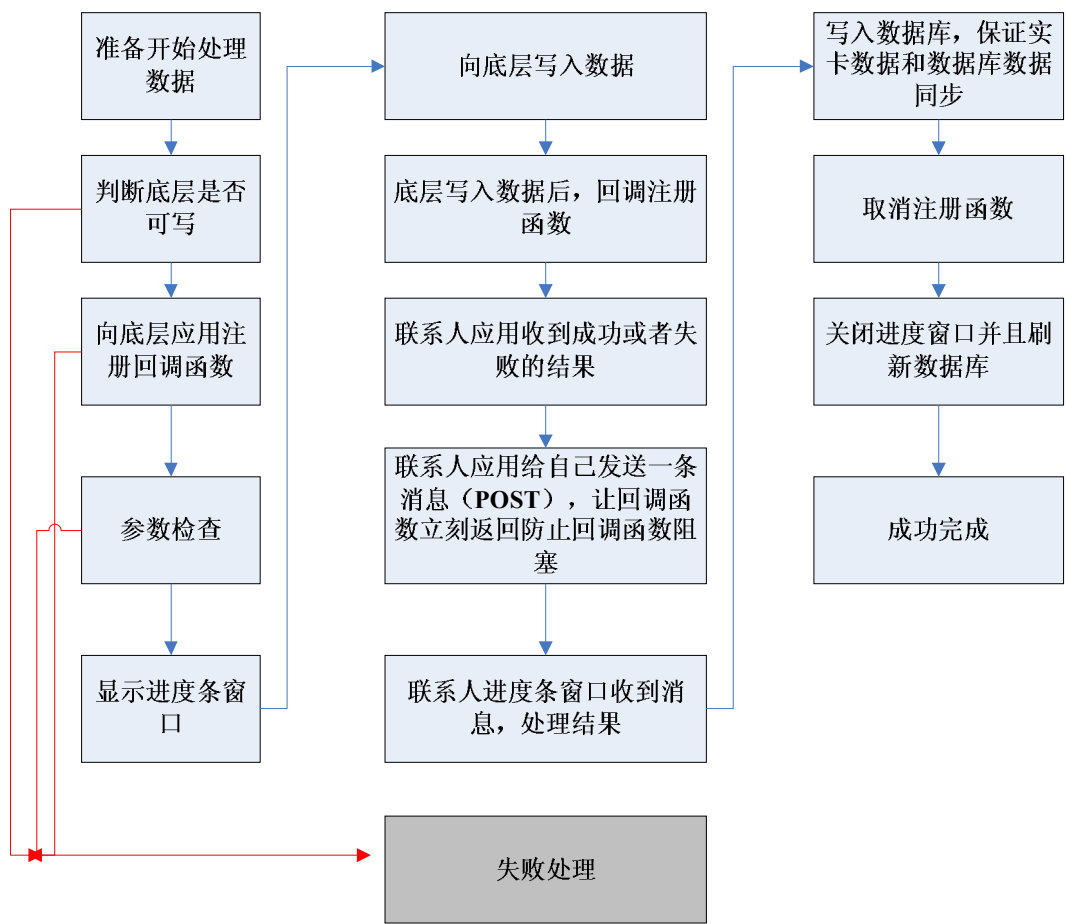
联系人进度条窗口收到消息后，保存数据到数据库中。

◆ 从底层取消

关闭进度条窗口，取消已经注册的函数，同时刷新界面数据。
处理完成。

◆ 图示

下图写卡数据的过程，大部分应用和底层交互的过程一般相同。



联系人应用写卡的过程

在真实的写卡过程中，中间阶段需要判断的状态是非常多的（很多的异常处理过程都需要考虑的），这里不仔细列出。

注意：

2 个进程之间通讯的时候，不要阻塞处理，而是一方收到消息后后，读取数据完成时，立刻让函数返回，这中间不要有其它过程在处理。

➤ ClearCase 上应该保存哪些文件

所有应用最后都将使用自动化编译，也为了代码完整和维护容易，所以开发人员在完成应用后，需要把所有开发过程中用到的程序源代码，测试代码，所以这里列出必须要上传的代码和文件。

◆ 应用的源代码

使用的 UI 界面应用，都必须使用 VC6++ 开发，源代码包含 .c 和 .h 文件，这些文件必须都要上传到 ClearCase 上。每次上传后必须打好 Baseline，并且这些代码，应该按规范的目录保存。

◆ 应用完整的资源文件

应用开发人员必须是使用的资源文件、图片、必须的媒体文件等等，都需要上传，即所有应用使用到的文件都需要保存下来。

在这里完整的资源文件，不是只指该文件是否存在，还应该是指这些 .mif、.bar 文件是否能在相应的应用开发工具中编译。

曾经看到一个应用，在 ARM 下编译到手机，没发现问题，因为 .brh 文件和 .bar 文件都存在并且 ID 值相等能对应，但发现资源文件中的 .brx 文件和 .brh 文件没法对应（值相等、名称不一样），这样就产生不能对这个资源文件做修改，如果修改就得在资源编辑器中重新编译这个文件，几十项的修改也是个不小的工作，特别是这样低级的错误让人心烦。

◆ 应用的批处理文件

应用的批处理文件，如删除所有 o 文件的批处理文件、自动化编译需要的批处理文件等。

◆ 应用配置文件

应用中使用的配置文件，如果程序启动后可以自动产生，那可以不上传，否则都必须保留。

◆ 完整的测试代码

所有的测试代码，测试应用例子，都必须上传，并且应该做到下载后立刻就能编译测试通过。

◆ 误区

有些应用把项目文件（.dsp、.dsw）放在一个目录，而把其它 .c、.h 文件代码放在另外的一个目录，在上传的时候，就上传 .c、.h 代码，这样就可能产生其他维护代码的人员需要重新搭建开发环境，并且重新折腾资源文件等。

➤ RELEASEIF 和 IWIDGET_Release 的异同

◆ 共同点

◆ 区别

➤ **ModelListener 的取消问题。**

◆ 使用监听对象 (ModelListener)

数据库结构 (ModelListener) 在每次被 (IMODEL_AddListenerEx) 函数调用后, 该函数中的参数函数和数据结构都会被系统维护, 直到用户取消, 因此都必须取消函数 (LISTENER_Cancel) 把系统维护的数据取消。

◆ 不取消监听对象可能产生的结果

如果很多监听函数没有被取消, 可能会产生的问题是:

- 1: 一般可能会有内存问题, 也有可能不存在 BPOINT1 和 BPOINT3。
- 2: 不可预知的系统崩溃, 就看运气了。
- 3: 不可预知的系统崩溃一般可以使用硬件调试器找到该问题。

◆ 注意

我们曾经在这个函数中, 碰到过巨大的问题, 花了很长的时间才找到这个监听器是必须取消的。

➤ **BPOINT1 和 BPOINT 3 的错误。**

应用是基于 C 开发的, 所以内存分配使用 MALLOC 函数, 分配的是实地址, 但 BREW 系统不会在应用关闭时, 主动释放该应用分配的内存, 因此每个 MALLOC 分配的内存都必须调用 FREEIF 函数来收回这些内存块; 在开发过程中, 可能由于开发人员没考虑周到, 程序出现内存问题是经常出现的。一般都是以下情况。

◆ 内存泄露 (BPOINT1)

1: 内存分配后没释放。

没释放的情况很多, 但大部分在调试阶段都能找到并且解决, 主要考虑的是异常退出时, 是否能都处理好内存问题。

一般没释放内存的过程, 在 VC++ 调试中都能找到。

2: 控件创建后没释放。

有时开发人有对引用计数认识不够明确, 可能产生最后没释放的控件, 但这样的问题, 一般伴随着一大片的 BPOINT1, 在模拟器上就能非常容易找出问题所在。

3: 窗口创建后没释放。

一般和控件没释放的过程是一样的。

4: 对象使用 AE CALLBACK 后没取消。

我们曾经的 ISOCKET 接口中使用了 AE CALLBACK 数据结构, 发现在 ISOCKET 强制关闭后, 必须取消这个结构中的过程, 否则有内存问题。虽然发现了这样的情况, 但我们

没仔细研究这个问题的根本原因。

◆ 内存重复释放 (BPOINT3)

最有名的 BPOINT3 内存重复释放问题，在开发阶段就必须解决的，任何时间出现这个问题，最后都是导致系统崩溃。因此任何情况下都不许可这个问题产生。

与内存没释放相比，这个问题严重百倍，因此更需要慎重对待。

一般内存释放函数都是使用 BREW 系统提供的 2 个函数，

FREE () 和 FREEIF ();

FREE () 函数释放了内存，但并没有把该内存指针设置为 NULL，存在安全隐患。

我们推荐都必须使用 FREEIF () 函数释放内存。这 2 个函数区别如下：

```
#define FREEIF(p)    if (p) { FREE((void*)p); (p) = 0; }
```

FREEIF 判断内存是否为空并且释放后置 NULL

所以内存释放后，都必须设置为 NULL，防止重复释放问题。

提示：

在千百个错误中，这个错误是最容易导致系统死机的。

◆ 内存越界

有一千种的可能使内存越界，但最大的可能是使用 (MEMCPY) 内存产生越界，在开发中产生这样的问题，一般都是笔误。

现在检查的好几个程序，产生内存越界，都是这个函数产生的，所有开发人员应该多多慎重。多加检查，多测试，一般都能找到这个问题。

◆ 内存问题的建议

我们对内存问题的最大的忧虑在于内存越界和 BPOINT3 产生的问题，前者经常导致不可预测的系统崩溃，而后者经常是调试信息打出提示后系统崩溃。

- 1: 建议使用 PC_LINT 对程序代码进行检查。
- 2: 使用 (MEMCPY) 函数的时候，尽可能多检查是否产生越界问题。
- 3: 加强普通测试和压力测试，没有频繁的测试，程序不可能非常稳定。

◆ 采取的措施

现在在开发阶段，我们就在手机上增加了如果内存有问题则响一下告警音的提示，并且把该有问题的应用和产生内存问题的地址等都保存到文件中。这种方式应该能解决很多平常产生的问题。

➤ **OEM 层不应该处理 UI 的事情**

➤ **文件操作注意**

- ◆ 不能同时对一个文件进行操作

➤ **树型文件夹问题**

在手机文件系统中，树型目录肯定是存在的，这个 PC 的文件目录一样，但手机毕竟没 PC 那么强大的管理功能，并且受限于手机 CPU 的处理时间限制（每次最多极限只能连续占用 30 多秒），因此对文件管理这部分，需要把处理过程分成小块处理，每次处理一部分。

这里对打开文件目录查看该目录下有多少文件和文件子目录的过程不做描述，因为一般每次打开目录，最多只会列出 256 个文件和目录，这个过程一般能在几秒内完成的，毕竟在手机上要一次显示 10000 个文件和目录不大可能并且也没有任何意义。

一般手机会有 2 个文件目录，一个是手机系统带的 USB 盘，该 U 盘目录一般最大就 96M（自带 2G 的可能也是存在的，可手机成本显然提高）。另外一个 T 卡的文件目录，用户购买的，现在的 T 卡一般都是 2G 容量或者更大。

下面将对这 2 个目录进行说明。

◆ **系统 USB 文件目录**

是手机结构中，曾经图示了 USB 在存储介质中的位置。如下图：



◆ **T 卡文件目录**

➤ 编译环境下不应该有垃圾文件

关于 BAR 文件编译

- 1: 目标机上真实的文件
- 2: 目标机上虚拟的文件

➤ mif 文件中的项意义

提交版本前测试项

今后版本提交前，都应该给出一份下列项的测试数据，这些数据很多是十分非常重要的和必须测试通过的，特别是直接影响用户视觉的项，更加需要特别注意；这里出了大部分内容。

➤ 应用启动测试

◆ 干净环境的启动测试

这里干净的环境是指，刚做完成 FILEDOWN 后，除了系统需要的文件外和应用正常的配置文件外，没有用户数据（如数据库中没记录、文件目录中都没用户文件等情况），所有的启动是否正常。

◆ 丢失配置文件的启动测试

配置文件丢失的时候，启动过程是否能正确提示或者提示错误后退出；但必须保证不能出现以下情况：

- 1: 系统崩溃。
- 2: 内存泄露。
- 3: 程序继续执行，但产生文件或者数据损坏。
- 4: 用户数据丢失。

文件丢失的情况非常少，但是有可能出现，所以这些测试在应用中应该做到的，即需要处理这样的异常情况，来保证程序的安全稳定。

◆ 安全模式下的启动

安全模式下最大的特点是网络没启动和部分应用需要的相关应用没启动（如联系人的共享内存没启动），但应该保证应用能跑起来，不产生错误或者能在产生错误后提示错误原因。

用户进入安全模式的情况非常少，但还是有可能进入这种模式，所以**推荐**这里也是需要做些

安全措施，大部分的应用不必关系这种情况，只有和网络有关的应用（如联系人、短信、拨号盘等应用），才需要考虑保证启动后不产生系统崩溃、文件损坏和用户数据丢失。

提示：

安全模式下如果是由于系统没启动相关应用，而产生的非重大的问题（重大问题是指系统崩溃、文件损坏和丢失、用户数据被破坏等），一般可以不追究。

➤ 编译应用和功能测试

◆ 提交版本前

在打好 BASELINE 后，需要自己编译一次并烧到手机上，测试没问题后，才能把 BASELINE 发给版本发布编译人员。

在过去的开发过程中，有些开发人员，每次发布测试新版本前，频繁提交应用版本，并且自己也没仔细测试，出现这样情况有可能导致版本编译人员在编译中也频繁产生错误。

特别提示：

最近有过一次案例，开发人员一个晚上一个应用就提交了 3 个版本，全编译人员检查了前 2 个版本应用功能正确，第 3 个版本没检查，导致发布的颁布存在一个明显的 BUG，这就是开发人员频繁提交版本并且没有仔细测试的结果。

◆ 自动化编译问题

需要注意的是，今后采用的是自动化编译，如果任何应用自动化编译中有问题，将导致出开发环境的速度延后，提交版本需要特别慎重。

所以提交版本前，一定要多多编译和测试，需要的是非常慎重对待。

◆ 修改注意的问题

后期应用的修改问题，任何修改都不是小事，哪怕就一行代码的修改，甚至是一个字符的修改，都需要重新编译后，自己**全部功能界面测试**通过才能提交 BASELINE。

特别提示：

对于后期修改，这里需要特别提醒，以我自己为例，有个应用（别人移交的应用）的资源文件，在资源文件的内容中，修改了一个标点，这个修改该非常小了，编译过了，也烧到手机上进行了测试，发现这个问题已经修改正确。但做梦也没想到的这个应用，在这个资源文件下的另外一个资源图片，并不是需要的图片，却名称一样，导致产生了另外一个 BUG（不正确的图片被编译到资源文件中了）。事情麻烦的是，到版本发布后才发现这个 BUG，这是无法让人能接受的事实。

所以在开发中，特别是到后期出全面测试版本的时候，更是需要十分慎重。

➤ 启动速度测试

◆ 空记录启动的时间

理论上，从进入应用到应用主界面被完全显示出来的时间，关键应用不应该超过 600 毫秒，非关键应用可以在 750 毫秒或者再多点；现在的大部分关键应用，优化后都基本能实现到这个启动速度。如优化后还是确实无法实现到这个启动速度，需要说明理由。

优化应用启动速度，是每个应用必须尽最大努力做到的，这个过程直接对用户产生视觉影响，太慢的启动速度，是十分无法接受的；对启动速度的口号必须是“优化优化再优化！”。

这里的启动时间，是指进入应用到应用显示完成的时间，一般从 QDXM 的调试信息可以看出，或者从应用中打印时间来判断，也是可以的。

提示：

有些应用可以先让窗口显示出来，再加载数据，可以提高显示的视觉效果，但这个过程不应该产生视觉上的闪屏、数据重复刷屏现象以及其它不良效果。

◆ 满记录启动的时间

满数据的情况，理论上启动时间应该比空记录的启动时间基本多一点点，比如多 100 毫秒。如果多出很多，就需要考虑优化问题了。

一般情况下，应用启动后都是先读取记录总数、数据库表的所有 ID、数据库表中的**前几条记录**，然后显示出来，所以空记录和满记录的启动时间差距，理论上，时间不受太大的影响，因为这 3 个过程理论上是不需要多少时间的，出发应用多次打开关闭数据库，那另当别论。

提示：

如果数据库频繁打开关闭，满记录的载入数据的时间将产生非常大的影响，所以应用开发人员尽可能少进行频繁打开关闭数据库的操作。

➤ 大数据量操作测试

每个应用如果存在数据库，都必须进行最大数据量测试，大数据量测试方面很多，这里列出关键部分内容。

◆ 载入数据需要的时间

每个应用一般都会对操作中数据载入进行优化，不会把所有记录一次全部载入，而是每次只载入一页的数据，等用户有其它操作后，再重新载入相应的数据。

所以满数据操作中载入时间主要就是考虑程序是否对这方面进行了优化。

◆ 删除所有数据需要的时间

大部分应用都要测试全部删除数据的时间，这方面的时间一般不做明确的限制，能越快越好，速度很可能受文件系统的限制，因为文件系统的速度将对批删除产生非常的影响，如果文件系统非常慢，删除的过程将非常长，严重影响用户操作；在这种情况下，对批删除的优化就更为重要。

◆ 满数据量下所有可能进行的操作

主要测试是在 LIST 控件中，拖拉是否产生延迟、卡住等效果，一般不允许这方面的效果。现在 2938 中的通话记录这方面（延迟、卡住）的效果特别明显，使用起来特别不舒服。

如果程序内存许可的情况，需要特别提高速度，可以把数据库表的记录 ID 和标题数据读到共享内存中，然后每次拖拉都根据内存中的 ID 去读记录，速度将能提高很多，但缺点就是占用内存比较大。

➤ 系统极度繁忙测试（暴力测试）

理论上，所有应用都需要进行压力测试，即当系统极度繁忙的时候，再操作其它应用是否还能稳定安全执行。

◆ 应用的暴力测试（单个应用）

- 1: 如应用被频繁（几百次的操作）增加修改、批增加、批删除等操作，是否安全稳定执行。
- 2: 部分按钮或者其它控件频繁被点击是否产生崩溃（每秒 1 次或者更快的点击）。
- 3: 应用内部几百次反复操作、屏幕几千次的点击（随机点击任何窗口下任何东西都可能被点到）、媒体连续几十个小时的播放、程序数百次的打开关闭、文件传输几百 M 甚至更大更多次数的等操作。
- 4: 频繁测试、极限状态的测试、各种不可思议的测试，都是必须的。

曾经的历史：

曾经日程应用，在测试人员 30 分钟暴力的连续点击下产生崩溃，当然经过优化后解决了这个问题。

◆ 系统繁忙的暴力测试（多个应用）

其它应用在处理多任务，如备份恢复正在进行，媒体播放也在进行，电话是否能拨入，拨入后是否产生白屏或者不显示来电，或者停顿效果等不良影响。

◆ 暴力测试的提示（更高的品质）

暴力测试是应用开发后、版本发布前必须测试通过的，该测试的结果往往能测试出一些没有优化或者代码不良的问题以及某些不可思议的操作未处理等情况，这也说明需要开发人员写出更高质量的代码、需要在开发中经常思考更多的情况以及如何更加优化代码的执行效率等等。

“优化无止境”，这应该是终端开发人员需要放在心中的一面镜子。

➤ 应用互斥测试

◆ 同时对 T 卡的写文件

◆ 同时对数据库的操作

➤ 占用内存测试（启动内存和最大内存）

内存非常紧张，所有应用在提交版本前，都应该对占用内存做一个总结，给出大概占用多少内存的数据，一般可以把内存信息打印出来，每次进出应用，内存是否稳定。

我们曾经出现过每次进出应用，没有出现 BPOINT1 但内存减少的现象，所以只看 BPOINT 是否有内存泄露，也不非常可靠的，跟踪内存状态数据才能准确判断是否有内存问题。

以下项目是每个应用都应该测试的：

◆ 内存稳定情况

进出应用几十次，看看占用的内存是否有增加或者减少的情况，理论上该值在进出几次后，应该是稳定的。如果有不稳定情况，应该查查问题的原因。

◆ 空记录启动后占用的内存

需要几次进出后，查看占用的内存总数，应该每次看看峰值大小，是否超出许可。

提示：

曾经的记录是某个应用，启动后就分配所有记录的内存，产生单个应用需要内存峰值达到 4M 的情况。

◆ 满记录后启动占用的内存

满记录特别需要注意内存的峰值情况。

◆ 所有窗口打开后占用的内存

即用户把所有窗口打开，包括协同应用的窗口；应用可能达到 4、5 层窗口，甚至更多的情况，都必须检查这样的情况是否产生内存不足。

◆ 使用过程是否有内存泄露

有些应用在使用过程中内存减少，但没 BPOINT1 和 BPOINT3 产生的情况，很多应用会检查启动时是否有内存问题，但没检查使用过程是否产生问题，所以这是需要检查的。

◆ 是否有 BPOINT1 和 BPOINT3 产生内存问题

这在文档中已经提到多次，这里就不多说了。

➤ 操作响应速度

◆ 操作响应的速度

窗口操作的响应速度，包括打开、关闭以及内部按钮和其它控件的操作速度，这些速度直接影响用户的视觉体验，是开发过程需要测试优化的。

◆ 对数据库的操作响应速度（批删除、批增加）

一般不对操作单条记录响应时间做限制，但需要对批处理的速度进行检查，这些操作如果快不了，需要给出问题所在，如是系统的原因、文件操作速度慢的原因或者其它内在的情况。

◆ 对底层任务处理的响应速度

◆ 批删除文件，COPY 文件的响应速度

能快尽量快，但需要保证系统的稳定，兼容稳定和速度，在两者间进行选择。

➤ 系统时间测试

和时间有关的应用，需要测试时间改变项。

◆ 当前时间下正常情况

所有操作的正常处理情况，正常的时间切换不产生问题，这是应用基本的功能点。

◆ 网络更换（如启动 C 网同步时间）

同步到当前时间的情况，是否相应数据修改正确（如 C 网自动同步时间）。

◆ 修改为 1980 年前情况

一般现在只考虑 1980 年后的时间，如果改到前面，是否有问题。

◆ 修改为 2050 年后的情况

同上。

◆ 时区改边的情况

修改时区是否正确，不能到了巴黎就使用不了。

➤ 待机测试

◆ 正常待机

各个应用是否正常，正常待机后，该在中间自动启动的应用是否能启动（如闹钟，提醒等）。

◆ 强制待机

唤醒后屏幕是否能正常点亮。

◆ 待机后来电和短信

如果来短信或者来电，屏幕是否能正常点亮并且是否能正确启动应用。

➤ 挂起和唤醒测试

◆ 正常挂起和唤醒后

界面是否正常（如不产生花屏），应用在正常窗口、对话框窗口、全屏窗口等情况。

是否有内存问题，很多应用在挂起和唤醒中做了很多操作，需要测试这方面的内容。

如果主题已经切换，是否能正确显示。

挂起的应用，如果程序正在进行，是否还能正常进行。这方面的例子，如联系人批删除情况，如果挂起了，将继续进行处理，不受影响。

➤ T 卡插拔测试

和 T 卡有关的应用，需要加强插拔卡这方面的功能测试，这方面涉及的应用主要有文件系统、文件管理器、电子书、媒体播放器、T 卡备份恢复等等应用。频繁插拔卡主要测试以下内容：

◆ 是否产生系统崩溃

中间插拔卡时，由于程序需要的文件已经不存在，是否产生系统问题，或者正在处理，收到卡插入消息，是否能正常处理。

◆ 是否产生内存泄露等

应用是否产生内存问题，如 BPOINT1 和 BPOINT3；是否产生控件没释放或者中间临时变量没处理好的情况，比如应用中没取消定时器等情况。

◆ 是否文件丢失

如果从 PC 上向手机 COPY 一个几百 M 的文件，正在 COPY 的时候，拨卡是否导致崩溃、是否文件可以删除或者其它问题、整个文件系统是否丢失或者损坏。

◆ 是否应用执行失败

1：当前应用是否能在失败提示后关闭。

2：是否影响到其它应用的执行。

◆ 系统极度繁忙的时候是否更大几率产生问题

如后台数据备份正在执行，前台媒体播放器在放 T 卡中的 MP3，而这时如果插拔卡是否有问题，几十次甚至上百次的测试，是否安全稳定。

➤ 断网测试

与网络有关的应用，需要测试网络断开和启动的情况。

◆ 强制关闭网络的测试

1：有网络有关的应用，如联系人和短信应用，在关闭网络的的时候，和该网络有关的数据

是否隐藏。

2: 在对卡操作的过程, 是否能正确处理这样的异常情况。

3: 是否存在数据库被破坏和数据丢失的情况产生。

4: 如产生错误, 是否正确提示用户操作失败。

◆ 反复打开关闭应用的情况

1: 在 2 个网络间重复切换, 打开关闭等起, 这样的操作是否产生不良影响, 如联系应用和短信应用的批处理操作, 同时打开关闭, 频繁操作, 是否产生内存、系统崩溃或者无法操作等一切异常情况。。

2: 网络应用是否正确处理, 如和网络有关的应用 (WAP、邮件等是否能拨号上网等情况) 是否都正常。

➤ 新建默认项测试

应用新建一条记录的时候, 该有默认名称的记录或者可以默认名称的记录, 是否给了默认的数据。这样的过程例如:

◆ 默认标题

1: 新建任务应用的默认标题。

2: 新建 T 卡备份的默认标题。

◆ 默认日期时间

1: 设置提醒时间

2: 闹钟默认时间

3: 平常日期时间选择的默认时间