MMI 实例培训教程

MTK 无线通信平台

傅贵

MAIL: mail@fugui.name

MSN: fu_gui@msn.com

第一部份：基础

第一章：开始

准备

在开始之前，假定你已经能够在手机中看到如下一些画面：

Idle

MainMenu

图 1.1

Hello, World

学程序都由"Hello, World"开始，我们也这样，下面是我们在手机中"Hello, World"的显示效果：

图 1.2

DOS 中的"Hello, World"实现如下：

```
main ()
{
    printf ("Hello, World ") ;
}
```

代码 1.1

当然，我们平台所用的操作系统不是 DOS，所以 DOS 下的函数我们基本不能用，包括程序入口方式也得改。

程序入口

嵌入式系统与通用系统不同，我们的应用程序通常与整个系统是固定在一起的。通俗一点说，MMI 就是一个大的程序，我们写小程序就是大程序的一个个固定分支。要写自己的程序，首先就得在大程序中添加一个新入口。当然，添加新入口有点麻烦，所以我们就暂时借用现有程序的入口 goto_main_menu，这是主菜单的入口函数（在 MainMenu.c 中），在 Idle 时按左软键就会进入主菜单。

我们要借用的代码如下（注：红色部份为修改后的代码，后同）：

```
void mmi_myapp_entry(void)
{
    //我们的程序由此开始!
}

void goto_main_menu(void)
{
    //将主菜单切换成我们的程序：
    mmi_myapp_entry();
    return;
    ……
}
```

代码 1.2

下几章会详细介绍怎样添加一个新入口，到时候我们再将此处改回来。

打印文本

我们用来显示文本串的函数原型如下：

```
void (*gui_print_text) (UI_string_type _text);
```

这个函数跟 DOS 中的 printf 使用方式差不多：

```
void mmi_myapp_entry(void)
{
    //我们的程序由此开始!
    gui_print_text(L"Hello, World");
}
```

代码 1.2

因为我们平台默认是基于多国语言的，所以基本上与文本相关的函数都只接受 Uinicode 编码，在这里我们以 L"Hello, World" 方式强制将字串转换为 Uinicode 编码输入。最终运行结果如下(开机以后按左软键)：

图 1.3

可是如图中所示我们看不到任何效果！

为什么会这样？这是因为我们平台里面，当所有绘画动作的代码结束时，如果不强制刷新屏幕是见不到效果的。

刷新屏幕

函数 gui_BLT_double_buffer 用来刷新屏幕：

```
void mmi_myapp_entry(void)
{
    //我们的程序由此开始!
    gui_print_text(L"Hello, World");
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

代码 1.3

四个参数指明了我们要刷新的范围，一般我们都默认刷新整个屏幕，结果如下：

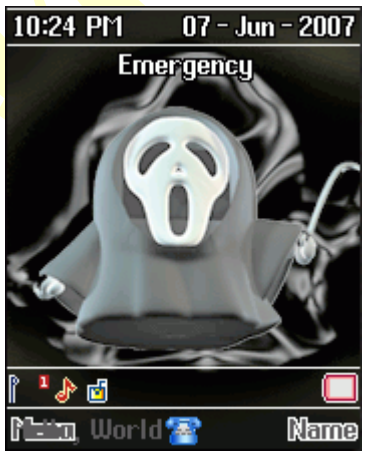


图 1.4

"Hello, World"在屏幕的左下角，当然，这样的效果还不能让人满意。

清屏

首先，我们得将背景去掉，因为我系统默认不会做任何事情，所以什么事都得自己来！

使用函数 clear_screen 就可以将整个屏幕刷成白色：

```
void mmi_myapp_entry(void)
{
    //我们的程序由此开始!
    clear_screen();
    gui_print_text(L"Hello, World");
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

代码 1.4

结果如下：

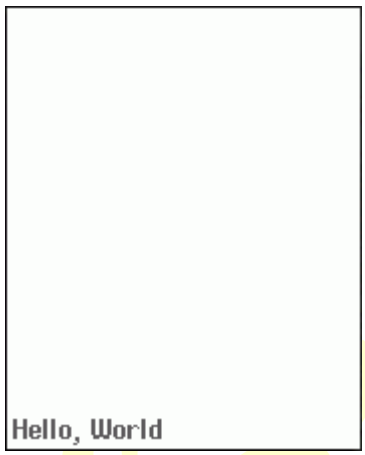


图 1.5

这样好看多了，下一步就是修改文本显示位置以及文本颜色：

文本属性

函数 gui_move_text_cursor 用来设置文本输出的起始位置，函数 gui_set_text_color 用来修改文本颜色：

```
void mmi_myapp_entry(void)
{
    //我们的程序由此开始!
    clear_screen();
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
    gui_print_text(L"Hello, World");
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

代码 1.4

结果如下：

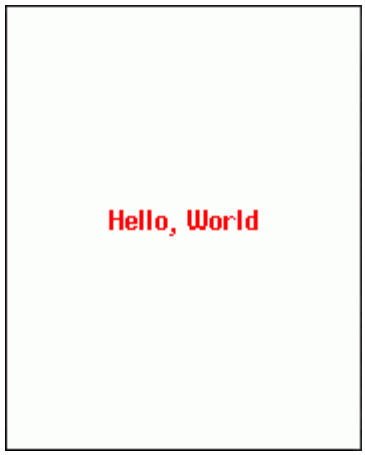


图 1.5

因为文本属性设置是针对整个系统的，并且不会有默认值，所以在每次输出文本前我们都要记得重设文本属性。

总结

至此，一个最简单的 MMI 程序已经出来，当然，这只是能够看一下而已，要想让程序正常运行，我们还得进一步完善。

第二章：屏幕

新的屏幕

上一章的屏幕输出后，我们只要稍微等上一会，就会发现屏幕上多出一些不想要的东西，见下图：

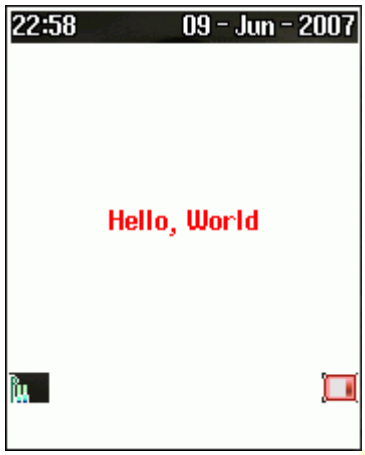


图 2.1

仔细看一下就会发现这是 Idle 中的东西，会什么会出现在这里？因为当我们进入一个新的程序时，如果不手动退出上一个程序，那么上一个程序一直都是活着的，这些多出来的东西就是因为我们没有强制退出 Idle。

退出上一个程序的方法是在进入新程序时调用 EntryNewScreen：

```
void mmi_myapp_entry(void)
{
    EntryNewScreen(MAIN_MENU_SCREENID, NULL, NULL, NULL);
    clear_screen();
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
    gui_print_text(L"Hello, World");
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

代码 2.1

全屏显示

但是调用 EntryNewScreen 后，我们还是会在屏幕中见到一些别的东西，见下图：

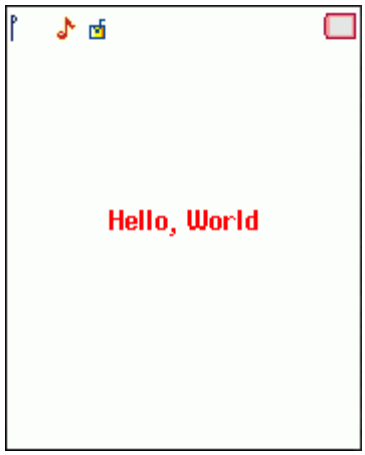


图 2.2

这一次的东西跟上个程序无关，这是系统默认显示的状态信息条，在大的屏幕中，如果你不强制关掉状态条，系统会一直将其显示出来，关掉状态条有几种办法，最简单的方式是在进入新屏幕时调用 entry_full_screen：

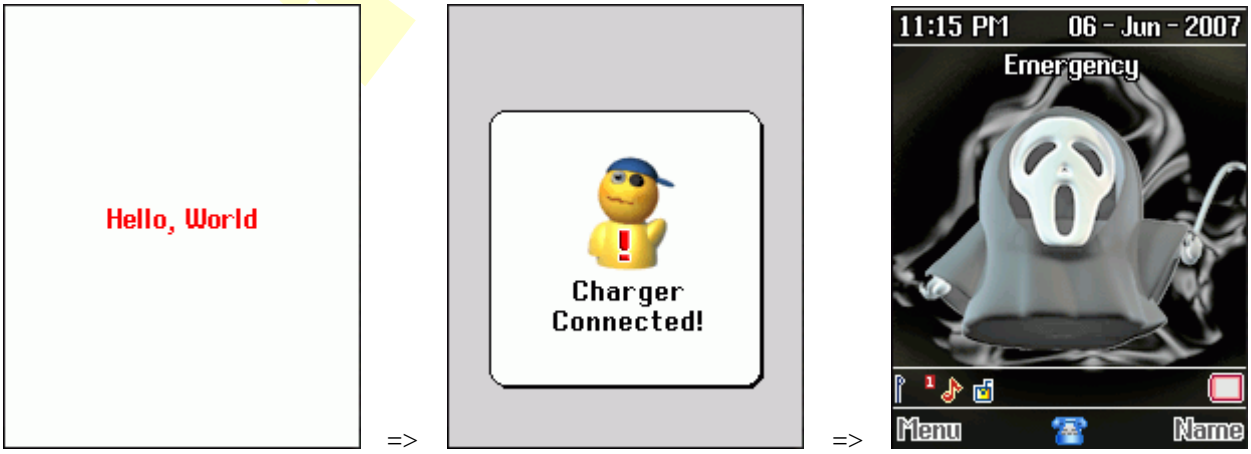
```
void mmi_myapp_entry(void)
{
    EntryNewScreen(MAIN_MENU_SCREENID, NULL, NULL, NULL);
    entry_full_screen();
    clear_screen();
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
    gui_print_text(L”Hello, World”);
    gui_BLT_double_buffer(0, 0, UI_device_width – 1, UI_device_height – 1);
}
```

代码 2.2

这样，我们才算是完完全全进入新的程序。

屏幕历史

在我们的程序运行的时候，如果有新的程序出来，同样会强制退出我们自己的程序，比如插上充电器时弹出提示框等，但新程序结束后，就会发现系统会直接返回 Idle，见下图：



我们的程序 插入充电器弹出对话框 对话框结束后直接返回 Idle

图 2.3

显然这样用户操作起来会十分的不便，所以在我们系统中为了解决此问题而建立了一套屏幕历史管理机制，只要在调用 EntryNewScreen 时传入我们新进入屏幕的 ID 及屏幕的入口函数，系统就会在下次调用 EntryNewScreen(也就是进入更新的屏幕)时，自动将我们的屏幕加入到历史记录中，当新的屏幕退出后，系统也会将我们的屏幕从历史中弹出并显示出来。

EntryNewScreen 的函数原型如下：

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

其中第一个参数表明新显示屏幕的序号，每个屏幕都有一个全局唯一的序号，这样方便系统管理。

第二个参数指明屏幕的退出函数，系统在强制退出我们的屏幕时会自动调用此函数，我们可以在其中作一些资源释放等方面的工作，如果实在没事可做就可以将此参数设为空。

第三个参数指明屏幕的入口函数，只有传入此参数，系统才会将屏幕自动加入历史。

第四个参数暂时不使用。

```
void mmi_myapp_entry(void)
{
    EntryNewScreen(MAIN_MENU_SCREENID, NULL, mmi_myapp_entry, NULL);
    entry_full_screen();
    clear_screen();
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
    gui_print_text(L"Hello, World");
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

代码 2.3

这里有一点要解释一下，我们的屏幕有点类似于 Windows 系统中的窗口概念，即一个应用程序在某个状态时的显示模式及交互方式，但有一点不同，我们的屏幕是独占整个显示系统及交互系统的。也就是说我们的系统在任何时候只能由一个屏幕来控制。

还要说明一点，一个程序可能会由多个屏幕组成，这跟 Windows 中一个程序可能有多个窗口类似，因为我们这里的范例程序一直只有一个屏幕显示，所以我们就暂时将“屏幕”与“程序”等同起来，这样方便叙述。

手动加入历史

如果不想让系统自动加入屏幕历史，可以尝试用下面的方式手工加入：

```
void mmi_myapp_entry(void);
void mmi_myapp_exit(void)
{
    history currHistory;
    S16 nHistory = 0;

    currHistory.scrnID = MAIN_MENU_SCREENID;
    currHistory.entryFuncPtr = mmi_myapp_entry;
    pfnUnicodeStrcpy((S8*) currHistory.inputBuffer, (S8*) & nHistory);
    AddHistory(currHistory);
}

void mmi_myapp_entry(void)
{
```

```
    EntryNewScreen(MAIN_MENU_SCREENID, mmi_myapp_exit, NULL, NULL);
    entry_full_screen();
    clear_screen();
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
    gui_print_text(L"Hello, World");
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

代码 2.4

当 EntryNewScreen 的第三个参数为空时，系统就不会自动加入历史，所以我们就可以在 mmi_myapp_exit 中手动添加进历史。

返回最近的屏幕

当然，有进入就有退出，退出屏幕也需要手动执行。我们通常用 GoBackHistory 通知系统将历史中最后一次显示的屏幕弹出来，如下：

```
void mmi_myapp_entry(void)
{
    EntryNewScreen(MAIN_MENU_SCREENID, NULL, mmi_myapp_entry, NULL);
    entry_full_screen();
    clear_screen();
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
    gui_print_text(L"Hello, World");
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    SetKeyHandler(GoBackHistory, KEY_RSK, KEY_EVENT_UP);
}
```

代码 2.5

我们通常将右软键设为返回最近显示的屏幕。

总结

至此，一个可观看可操作的简单 MMI 程序已经出来了，下面要做的就是将我们的程序标准化，规范化。

第三章：程序

新程序

规范化就是要将我们的程序独立出来。
独立分为两大步：代码独立与数据独立：
　　代码独立：就是将程序代码放到单独的文件中，这是本章要讲的主要内容。
　　数据独立：也就资源独立，第四章会详细讲述怎样将资源独立出来。
在修改之前，我们先将我们的程序命名为“MyApp”。

添加程序文件

一般新加的 MMI 程序都放到 plutommi\MMI 下面：
先创建三个目录：

plutommi\MMI\MyApp	程序总目录
plutommi\MMI\MyApp\MyAppInc	头文件目录
plutommi\MMI\MyApp\MyAppSrc	源文件目录

在 plutommi\MMI\MyApp\MyAppSrc\下创建程序源文件：
　　MyAppSrc.c

在 plutommi\MMI\MyApp\MyAppInc\下创建程序头文件：
　　MyAppProt.h
　　MyAppDefs.h
　　MyAppTypes.h
　　MyAppGprot.h

下面将分别介绍每个文件的作用：

MyAppProt.h

　　MyAppProt.h 用来放本程序的所有函数申明，但此头文件只被本程序的源文件所加载。

```
#ifndef _MYAPPPROT_H
#define _MYAPPPROT_H

#include "MyAppGprot.h"
extern void mmi_myapp_exit(void);
extern void mmi_myapp_entry(void);

#endif /* _MYAPPPROT_H */
```

代码 3.1

MyAppTypes.h

MyAppTypes.h 用来放本程序所需的类型、结构、常量定义。

MyAppGprot.h

MyAppGprot.h 也是用来放函数申明，但此头文件是被别的程序加载的，也就是说此文件所申明的都是对外接口。

```
#ifndef _MYAPPGPROT_H
#define _MYAPPGPROT_H
#include "PixtelDataTypes.h"
#include "MyAppTypes.h"
extern void mmi_myapp_entry(void);
#endif /* _MYAPPGPROT_H */
```

代码 3.2

MyAppDefs.h

MyAppDefs.h 用来放本程序的资源 ID 定义，下一章会详细讲述。

MyAppSrc.c

MyAppSrc.c 是本程序的主源文件，在这里我们会将程序主函数 mmi_myapp_entry 与 mmi_myapp_exit 从 MainMenu.c 中转移到此处（因转移简单，就不详细列出了）。另外，我们可以将常用的一些头文件包含进来，以方便后面程序设计，比如下例一些头文件：

```
#include "MMI_features.h"
#include "PixtelDataTypes.h"
#include "gdi_include.h"
#include "wgui.h"
#include "gui.h"
#include "Globaldefs.h"
#include "CustDataRes.h"
#include "gui_themes.h"
#include "wgui_categories.h"
#include "wgui_softkeys.h"
#include "HistoryGprot.h"

#include "MyAppDefs.h"
#include "MyAppTypes.h"
#include "MyAppProt.h"
……
```

代码 3.3

将文件加入项目

因我们使用的是 ARM 编译器，要想将文件加入项目，必须手动将新文件路径添加到以下几个数据表文件中：
修改 make\plutommi\下的三个文件：
1、plutommi.lis： 此文件用来指明 MMI 所要编译的所有源文件。在文件中添加如下一行：

```
plutommi\MMI\MyApp\MyAppSrc\MyAppSrc.c
```

2、plutommi.inc：此文件用来指明 MMI 所有头文件所在目录（因我们的源文件中加载头文件都不会指明路径，所以一定要在此申明）。在文件中添加如下一行：

```
plutommi\MMI\MyApp\MyAppInc
```

3、plutommi.pth：此文件用来指明 MMI 所有源文件所在目录。在文件中添加如下一行：

```
plutommi\MMI\MyApp\MyAppSrc
```

程序开关

为了尽量精简最终生成的烧录程序，我们一般都会给每个小程序加上自己的编译开关，并将自己程序所有代码都包含进编译开关。
MMI 的编译开关一般都放到文件 plutommi\Customer\CustResource\PLUTO_MMI\MMI_featuresPLUTO.h 中,按照如下方式添加：

```
……
/*****************************************************************************
 [Application]: MyApp
 *****************************************************************************/
#define __MMI_MYAPP__
……
```

代码 3.4

如下所示，我们一般也会将入口加进编译开关：

```
#include "MyAppGprot.h"
void goto_main_menu(void)
{
#ifdef __MMI_MYAPP__
      //将主菜单切换成我们的程序：
      mmi_myapp_entry(void)();
      return;
#endif /* __MMI_MYAPP__ */
}
```

代码 3.5

最后一步，必须在 DOS 中重新 build new 一次，否则添加不会生效(build new 同时会生成新的模拟器)。

总结

本章讲了怎样在我们系统中添加一个 MMI 新程序，当然了，这只是推荐做法，并不需强制执行。下一章我们会继续讲怎样将资源独立。

第四章：资源

资源介绍

通常将程序使用的数据分为动态数据与静态数据两种，动态数据即程序运行时才能知道的数据，一般是由程序动态生成。而静态数据是固定的，在编译时即可将其转换成二进制数据存进最终的烧录文件中，我们将这种静态数据称之为资源。常见资源有以下几种类型：字串、图像、菜单、字库、主题、声音以及某些程序单独使用的资源。
在常见资源中，一般新加的程序只会修改其中的三种，即字串、图像与菜单。所以我们下面讲的资源也只与这三种相关。
添加一项资源通常分三步：原料，ID，装载。

原料： 即原材料，如图像就是准备一副新图，字串就是准备各种语言的 Unicode 编码。
ID： 即资源项的别名，程序只能通过 ID 来获取资源项（ID 一般定义在 XXDefs.h 中）。
装载： 装载在编译目标烧录文件之前就会被执行，其目的有两个：一是将原材料转换成二进制数据，二是生成将 ID 与二进制数据联系起来的映射表。

资源装载预编译程序是 plutommi\Customer\ResGenerator\mtk_resgenerator.exe，这个程序在每次编译目标烧录文件之前临时编译生成的。下面的修改基本上与这个程序有关。

添加文件

在 plutommi\Customer\CustResource\PLUTO_MMI\Res_MMI 下面创建一个新文件：
plutommi\Customer\CustResource\PLUTO_MMI\Res_MMI\Res_MyApp.c
并在文件中添加一个函数 PopulateMyAppRes：

```
#include "StdC.h"
#ifdef DEVELOPER_BUILD_FIRST_PASS
#include "PopulateRes.h"
#include "MMI_features.h"
#include "GlobalMenuItems.h"
#include "MyAppDefs.h"

void PopulateMyAppRes(void)
{

}
#endif/* DEVELOPER_BUILD_FIRST_PASS */
```

代码 4.1

此文件用在预编译时装载资源。每个程序都有自己的资源装载文件，这些文件与 plutommi\Customer\ResGenerator\mtk_resgenerator.exe 一起生成 mtk_resgenerator.exe 并在 Windows 下被执行。

修改 Makefile

在文件 plutommi\Customer\ResGenerator\Makefile 中添加如下两行：

```
    -I "../../MMI/MainMenu/MainMenuInc"   \
    -I "../../MMI/MyApp/MyAppInc"   \
```

代码 4.2

此文件是资源装载预编译程序的 Makefile。

修改 PopulateRes.c

修改 plutommi\MMI\Resource\ PopulateRes.c：

```
……
extern void PopulateMainDemoRes(void);
extern void PopulateMyAppRes(void);
……

void PopulateResData(void)
{
    ……
    PRINT_INFORMATION(("Populating Main Menu Resources\n"));
    PopulateMainMenuRes();

    PRINT_INFORMATION(("Populating MyApp Resources\n"));
    PopulateMyAppRes();
    ……
}
```

代码 4.3

mtk_resgenerator.exe 在执行时会呼叫到这里面的 PopulateResData。

修改 readexcel.c

修改 plutommi\Customer\ResGenerator\readexcel.c（如果找不到此文件，此步骤可省略）。

```
……
#include "SettingDefs.h"
#ifdef __MMI_MYAPP__
#include "MyAppDefs.h"
#endif /* __MMI_MYAPP__ */
……
```

代码 4.4

字串资源有自己单独的装载预编译程序 readexcel.exe，此程序在 mtk_resgenerator.exe 呼叫完后会被接着生成并执行。

资源 ID

在加 ID 之前先得为本程序添加一个基础 ID，因所有程序的资源 ID 都是各自为政各定义各的，但是这些 ID 又不能冲突（每种类型的资源 ID 都是在同一个取值空间），所以我们就用这些基础 ID 将每个程序的 ID 取值隔离开来。

基础 ID 统一定义在 plutommi\MMI\Inc\MMIDataType.h：

```
……
typedef enum
{
    ……
    RESOURCE_BASE_RANGE(MAIN_MENU,          600),
    RESOURCE_BASE_RANGE(MYAPP,              100),
    ……
} RESOURCE_BASE_ENUM;
……

/*****************************************************************************
* Main Menu
*****************************************************************************/
#define   MAIN_MENU_BASE            ((U16) RESOURCE_BASE_MAIN_MENU)
#define   MAIN_MENU_BASE_MAX        ((U16) RESOURCE_BASE_MAIN_MENU_END)
RESOURCE_BASE_TABLE_ITEM(MAIN_MENU)

/*****************************************************************************
* MyApp
*****************************************************************************/
#define   MYAPP_BASE                ((U16) RESOURCE_BASE_MYAPP)
#define   MYAPP_BASE_MAX            ((U16) RESOURCE_BASE_MYAPP_END)
RESOURCE_BASE_TABLE_ITEM(MYAPP)
……
```

代码 4.5

重点是在 RESOURCE_BASE_RANGE(MYAPP, 100) 这里的 100 表示我们的程序 ID 定义不会超过 100 个（是任何一种类型的资源 ID 数量都不会超过 100，不是所有加起来）。

还有一种资源是跟屏幕历史控制有关，即我们第二章所讲的屏幕的序号，前面没有定义只好用 Main Menu 的屏幕序号充数，下面我们就给自己的程序加上屏幕序号（也定义在 MyAppDefs.h 中）：

```
typedef enum
{
    SCR_MYAPP_MAIN = MYAPP_BASE + 1,
} SCREENID_LIST_MYAPP;
```

代码 4.6

下面就将我们的主程序改过来：

```
void mmi_myapp_entry(void)
{
    EntryNewScreen(SCR_MYAPP_MAIN, NULL, mmi_myapp_entry, NULL);
    ……
}
```

代码 4.7

总结

本章重点讲了资源相关的基础知识，切记以上的步骤最好严格遵守，以免添加资源时出现不必要的麻烦！接下来几章将会详细介绍这三种资源的使用方法。

第五章：字串资源

介绍

这一章我们会将字串"Hello, World"转移到资源中去，并为其添加多国语言版本。

字串 ID

先在 MyAppDefs.h 中添加字串 ID：

```
typedef enum
{
    STR_MYAPP_HELLO = MYAPP_BASE + 1,
} STRINGID_LIST_MYAPP;
```

代码 5.1

字串资源

在 plutommi\Customer\CustResource\PLUTO_MMI\ref_list.txt 中添加一行（注：最好不要在 Excel 中编辑此文件，因为 Excel 默认会为每个字串加上双引号）：

Enum Value	Module Name	Max	Description	English	Tr_Chinese	Si_Chinese	Thai
.....	.....	.....	.....	.....	.....	.....	....
STR_DEF_ENCODING_	Undefined	4	Default Encodir	UCS2	UCS2	UCS2	ยูซีเอส
STR_MYAPP_HELLO	Undefined	11	Hello, World	Hello, World	你好,世界	你好,世界	Hello,
STR_MAINMENU_STYL	Undefined	14	MainMenu Style	MainMenu Styl	主選單風格	主选单风格	รูปแบบ
.....	.....	.....	.....	.....	.....	.....	....

图 5.1

下面是对表格每一列的描述：
第一列是字串 ID，与 MyAppDefs.h 中定义的要完全一致。
第二列是字串所属的程序名（可任意写，仅供自己参考）。
第三列指本字串的最大长度(取所有语言中文本最长的一个)。
第四列是字串描述，也可任意写。
从第五列起就是各种语言所对应的字串。

字串装载

在函数 PopulateMyAppRes 中添加一行：

```
void PopulateMyAppRes(void)
{
    ADD_APPLICATION_STRING2(STR_MYAPP_HELLO,"Hello, World","MyApp.");
}
```

代码 5.2

宏 ADD_APPLICATION_STRING2 用来装载字串，第一个参数用来放字串的 ID，第二个参数是字串的默认显示，当 ref_list.txt 里面找不到相应内容时就拿这个字串来充数，第三个参数是字串的描述，可忽略。

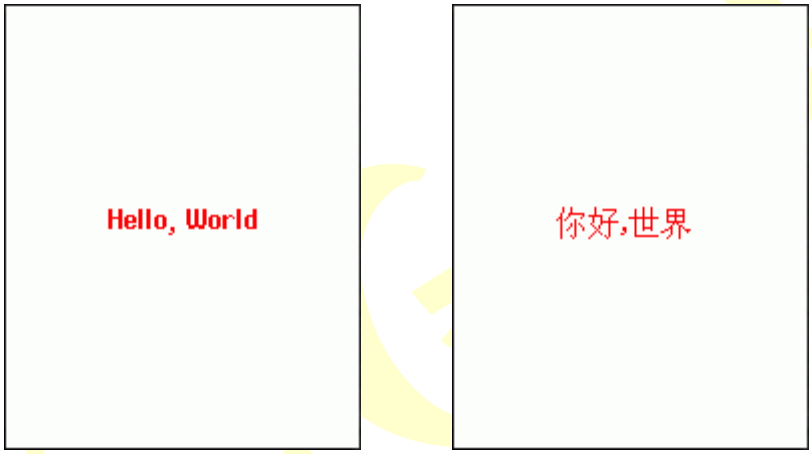
字串读取

使用函数 GetString 可将字串资源读取出来：

```
void mmi_myapp_entry(void)
{
    ……
    gui_print_text((UI_string_type)GetString(STR_MYAPP_HELLO));
    ……
}
```

代码 5.3

依据系统当前所使用的语言(由用户在“话机设置”中选取)，GetString 会返回相应的字串，如下：



英文 中文

图 5.2

总结：

本章主要讲述如何将字串资源化，建议将所有输出字串强制以资源方式存储，以方便多国语言版本管理。

第六章：菜单资源

介绍

在第一章中，为了简单起见我们将 MainMenu 作为程序入口，这一章我们会将入口改成自己新添加的菜单项，新菜单项放在[MainMenu]->[Organizer]->[Hello, World]。

菜单项 ID

所有菜单的 ID 都放在头文件 plutommi\MMI\Inc\GlobalMenuItems.h 中：

```
enum GLOBALMENUITEMSID
{
    IDLE_SCREEN_MENU_ID = 1,
    ……
    ……
    MENU_ID_MYAPP_HELLO,
    MENU_ID_DEVAPP_START,
    MENU_ID_DEVAPP_END = MENU_ID_DEVAPP_START + 100,
  MAX_MENU_ITEMS_VALUE,
  MENU_ITEM_END
};
```

代码 6.1

注：新菜单 ID 一定要放在 MAX_MENU_ITEMS_VALUE 之前，否则会出问题。

菜单加载

首先，我们需要将 MENU_ID_MYAPP_HELLO 加入到 Organizer 的下级列表中，按如下方式修改 Res_MainMenu.c(main menu 及 main menu 下一级子菜单都在此文件中加载)：

```
……
typedef enum
{
#if defined(__MMI_CALENDAR__)
    ORG_ENUM_CALRNDAR,
#endif
#if defined(__MMI_TODOLIST__)
    ORG_ENUM_TODOLIST,
#endif
    ORG_ENUM_ALARM,
#if defined (__MMI_WORLD_CLOCK__)
    ORG_ENUM_WORLDCLOCK,
#endif
#ifdef __MMI_MESSAGES_CLUB__
```

```
    ORG_ENUM_SERVICE,
#endif
#ifdef __MMI_MYAPP__
    MENU_ENUM_MYAPP_HELLO,
#endif/* __MMI_MYAPP__ */
    ORG_ENUM_TOTAL
} OrganizerMenu;
……
……
#if defined(__MMI_VERSION_2__)
void PopulateMainMenuRes(void)
{
    ……
    /* oganizer */
    ADD_APPLICATION_MENUITEM((
        MAIN_MENU_ORGANIZER_MENUID,
        IDLE_SCREEN_MENU_ID,
        ORG_ENUM_TOTAL,
        #if defined(__MMI_CALENDAR__)
            ORGANIZER_CALENDER_MENU,
        #endif
        #if defined(__MMI_TODOLIST__)
            ORGANIZER_TODOLIST_MENU,
        #endif
            ORGANIZER_ALARM_MENU,
        #if defined (__MMI_WORLD_CLOCK__)
            ORGANIZER_WORLDCLOCK_MENU,
        #endif
        #ifdef __MMI_MESSAGES_CLUB__
            EXTRA_SHORTCUTS_MENUID,
        #endif
        #ifdef __MMI_MYAPP__
            MENU_ID_MYAPP_HELLO,
        #endif
        SHOW,
        MOVEABLEWITHINPARENT|INSERTABLE,
        DISP_LIST,
        MAIN_MENU_ORGANIZER_TEXT,
        MAIN_MENU_ORGANIZER_ICON
    ));
    ……
}
```

代码 6.2

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

第 1 个参数：新加菜单项的 ID。

第 2 个参数：新菜单项的上一级菜单 ID，即通常所说的 Parent ID。

第 3 个参数：此菜单的下一级菜单项总个数。此处假设总个数为 **N**。
第 4 个参数----第 4+N 个参数：分别为每一个子菜单项的 ID。
第 4+N+1 个参数：隐藏属性，一般设为 SHOW，此参数暂时不起作用。
第 4+N+2 个参数：菜单项转移属性，有以下一些可选属性 NONMOVEABLE，MOVEABLEWITHINPARENT，MOVEABLEACROSSPARENT， INSERTABLE，SHORTCUTABLE。可将这些属性任意组合起来，NONMOVEABLE 表示关闭转移属性，目前大部份菜单项都是用的这个属性。SHORTCUTABLE 表示此菜单项将添加到快捷菜单列表中。除了这两个外其它的属性目前暂时都不会用到。
第 4+N+3 个参数：下级菜单的显示风格，有以下一些风格，可任选其一：

DISP_LIST：	列表显示，绝大部份普通菜单都是用此风格。
DISP_MATRIX：	矩阵显示，如九宫格，十二宫格等等，一般主菜单都是用此风格。
DISP_CIRCULAR_3D：	循环 3D 显示，只有主菜单才会用到。
DISP_PAGE：	翻页风格，每个菜单项一页，一般只会在主菜单中用到。
DISP_FIXED_GRID：	此风格暂时可忽略。
第 4+N+4 个参数：	此菜单项的显示文本串 ID。
第 4+N+5 个参数：	此菜单项的小图标 ID。

然后加载 MENU_ID_MYAPP_HELLO 本身：

```
void PopulateMyAppRes(void)
{
    ADD_APPLICATION_STRING2(STR_MYAPP_HELLO,"Hello, World","MyApp.");
    ADD_APPLICATION_MENUITEM((MENU_ID_MYAPP_HELLO,
        MAIN_MENU_ORGANIZER_MENUID, 0, SHOW,SHORTCUTABLE, DISP_LIST,
    STR_MYAPP_HELLO, 0));
}
```

代码 6.3

因 MENU_ID_MYAPP_HELLO 没有下级菜单，所以第三个参数设为 0。

在我们平台中，每个菜单项的行为都由菜单项自己控制，系统所能做的只是在高亮此项（即按上下方向键选中此项）时发个通知过来。当然，每个菜单项都要在开机时告知系统将由哪个函数来接受通知，SetHiliteHandler 就是用来做此事的，其第一个参数是菜单项 ID，第二个参数是用来接受通知的函数指针。我们通常会为每个程序建一个初始化函数，此函数只在开机时运行一次，如下面代码所示的 mmi_myapp_init，在此函数中我们会将本程序所有菜单项都注册一遍。
在菜单项接受通知的函数中，我们通常所做的只有一件事，即更改左右软键的响应函数，其中最重要的就是左软键，正如前面所讲，我们要将 MENU_ID_MYAPP_HELLO 做为程序的入口菜单项，那么我们所要做的就是将左软键的响应函数设为 mmi_myapp_entry。
注：左软键的响应函数必须得改，否则此菜单项将无法进入下一级菜单。右软键则可有可无，因为绝大部份菜单项的右软键响应函数都是相同的 GoBackHistory。

修改见 MyAppSrc.c：

```
……
void mmi_myapp_hilite_hello(void)
{
    SetLeftSoftkeyFunction(mmi_myapp_entry, KEY_EVENT_UP);
}

void mmi_myapp_init(void)
```

```
{
    SetHiliteHandler(MENU_ID_MYAPP_HELLO, mmi_myapp_hilite_hello);
}
……
```

代码 6.4

同时这几个函数都要在头文件中申明一下，见代码 6.5 及代码 6.6：

MyAppProt.h

```
……
extern void mmi_myapp_hilite_hello(void);
……
```

代码 6.5

MyAppGprot.h

```
……
extern void mmi_myapp_init(void);
……
```

代码 6.6

MMITask.c

前面所讲的开机初始化函数 mmi_myapp_init 我们通常在 InitAllApplications 中呼叫，如下代码所示：

```
……
#ifdef __MMI_MYAPP__
#include "MyAppGprot.h"
#endif
……

void InitAllApplications(void)
{
    ……
#ifdef __MMI_MYAPP__
    mmi_myapp_init();
#endif
    ……
}
```

代码 6.7

最终结果如下，菜单列表 Organizer 中的最后一项是我们的”Hello,World”，只要高亮此项并单击左软键就会进入我们的程序：

图 6.1

总结：

本章讲了如何添加菜单资源，以及在程序中菜单的使用方式，最后以八个字总结一下我们平台的菜单特性：统一装载，分散使用。

第七章：图像资源

介绍

上一章添加的菜单项我们并没有为之加上图标，图标与字串一样，要先将其加入资源，然后通过 ID 将其读取出来。

图像 ID

先在 MyAppDefs.h 中添加字串 ID：

```
typedef enum
{
    IMG_MYAPP_HELLO = MYAPP_BASE + 1,
} IMAGEID_LIST_MYAPP;
```

代码 7.1

新加目录

通常在 plutommi\Customer\Images\ 下面有如下一些目录：

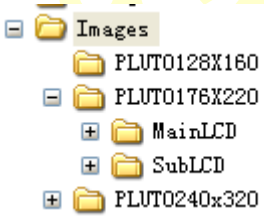


图 7.1

因我们主屏用的尺寸是 176X220，并且此图像是用于主屏(用于副屏的情况不多)，所以在 PLUTO176X220\MainLCD 下我们给程序新建一个叫“MyApp”的目录，如：
plutommi\Customer\Images\PLUTO176X220\MainLCD\MyApp

加入图片

在 MyApp 目录下新加一副图片 SB_MYAPP. Bmp：

装载图片

```
void PopulateMyAppRes(void)
{
```

```
    ……
    ADD_APPLICATION_IMAGE2( IMG_MYAPP_HELLO,
    CUST_IMG_PATH"\\\\MainLCD\\\\MyApp\\\\SB_MyApp.bmp" , "Hello World!" );
    ……
}
```

代码 7.2

宏 ADD_APPLICATION_IMAGE2 用来加载图像资源，其第一个参数是图像 ID，第二个参数图像存放的路径，路径前面加的宏 CUST_IMG_PATH 在运行的时候会自动转化为相应的图像根目录，这里是 plutommi\Customer\Images\PLUTO176X220\，第三个参数是此图像的描述，只起参考作用。

将图片作为菜单 ICON

将前一章所加的菜单项 MENU_ID_MYAPP_HELLO 的最后一个参数改为 IMG_MYAPP_HELLO：

```
void PopulateMyAppRes(void)
{
    ……
    ADD_APPLICATION_MENUITEM((MENU_ID_MYAPP_HELLO,
        MAIN_MENU_ORGANIZER_MENUID, 0, SHOW,SHORTCUTABLE, DISP_LIST,
    STR_MYAPP_HELLO, IMG_MYAPP_HELLO));
}
```

代码 7.3

最终结果如下：

图 7.2

总结：

本章讲了如何添加图像资源，当然图像不止 BMP 一种格式，但其添加方式都一样，第十一章会详细讲述各种图像资源的使用方式。

第二部份：绘画

第八章：开始

MMI 架构

MMI 的全称是 Man Machine Interface，即人机接口或人机介面。人机介面分为文字介面(如 DOS)和图形介面(如 Windows)两种类型，我们平台分属于简单的图形介面。下面是我们平台的 MMI 简洁架构图：

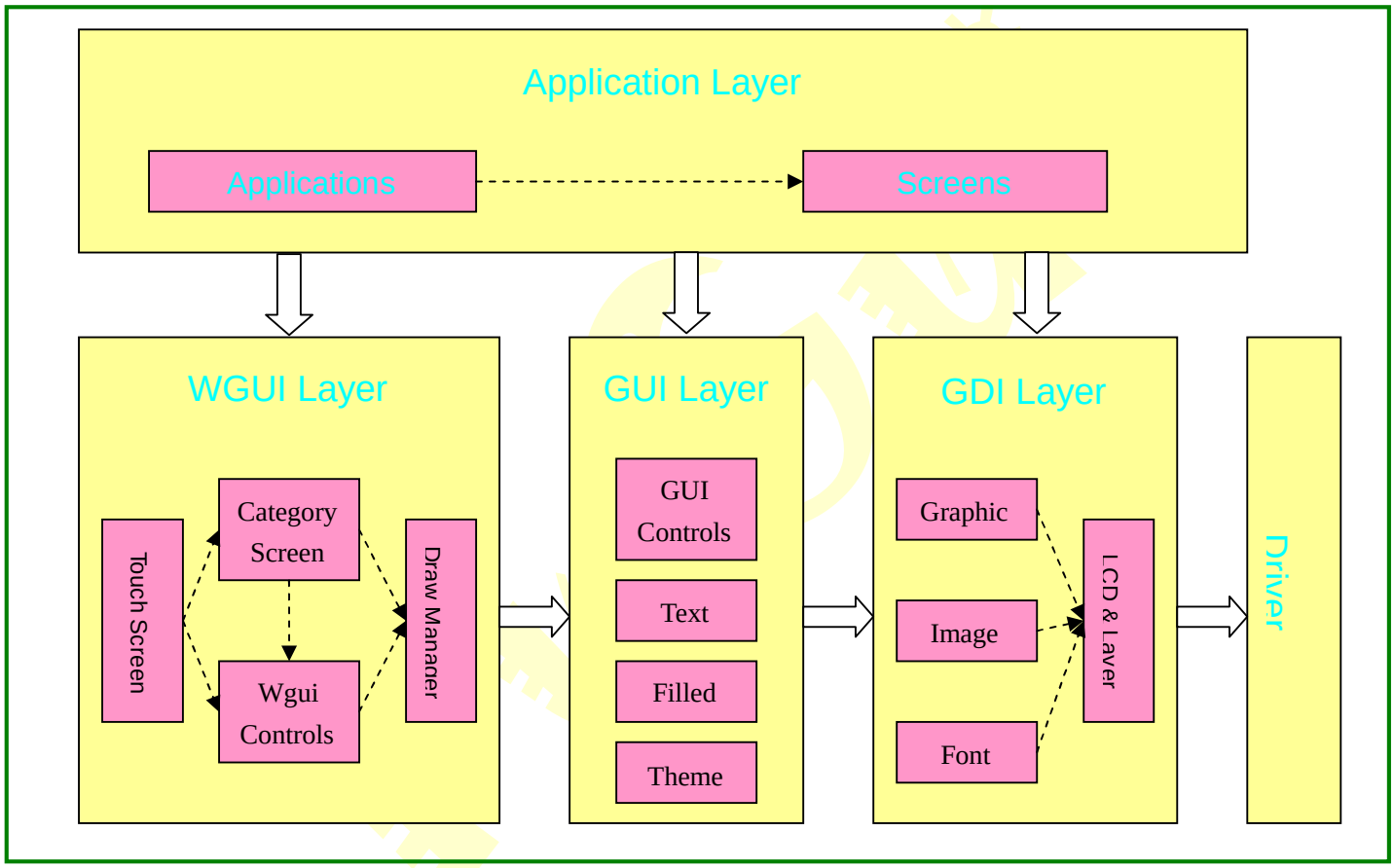


图 8.1

最上层的 Application Layer 在第一部份已详细介绍，通常我们要做的也就是这一层的开发。Application Layer 往下就是平台的图形子系统，图形子系统一般由系统提供商提供，本教程第二部份要讲的都是图形子系统的内容。图形子系统再往下是硬件驱动，这已经不在我们的讨论范畴。

图形系统

在讲图形系统之前我们先讲一下 GUI，GUI 的全称是 Graphical User Interface，即图形用户界面或图形用户接口。原始意义的 GUI 是针对最终用户来说的，也就是一种供用户使用的图形操作介面。而从开发的意义来说，这里的用户应该是开发 Application Layer 的程序员(也即二次开发程序员)，GUI 也变成了操作系统提供给应用层的一整套绘画函数接口，也就是说，从开发意义上来说 GUI 与图形系统是同一个概念。

当图形系统变复杂时，我们会进一步将图形系统细分为三个子系统：

模板子系统：将与用户交流的功能介面分类整理，取其原型归类后形成模板。如 Windows 系统性中的窗口模板(对话框，单文档或多文档等)，控件模板(按钮，文本框等)等。还有我们平台中的 Category Screen、WGUI Controls 等都属于模板类接口。简而言之，能自行绘制并且能自行与用户交流的元件其原型我们都可以称之为模板。

绘画子系统：此系统之接口只负责绘画，不负责与用户交流，通常模板类元件都有其相应的绘画类接口。

设备子系统：也即 GDI(Graphical Device Interface)，图形设备接口。此类接口负责与设备协调，并对上层形成一套简洁统一的接口。一般情况下此类接口只能被图形系统内部使用。

WGUI Layer

WGUI，其字面意义是 Wrap GUI，也即包装后的 GUI，我们可以认为 WGUI 就是图形系统中的模板子系统。WGUI 分为四个部份：

Category Screen：可以认为是屏幕模板集，与 Windows 系统中的窗口模板类似。

WGUI Controls：控件模板，我们系统中控件的概念与 Windows 系统中控件的概念一致。

Touch Screen：Touch Screen 可以认为是图形系统中最核心的控制模块。在我们平台早期版本时，WGUI 中的各种元件在系统运行时都是各自为政，这样虽然可以保证元件的灵活性，但是造成了公共资源协调困难。以前的公共资源都是焦点元件(系统任何时候只有一个元件属于激活状态)独占，当触摸屏出现后，因为触摸屏事件的随意性极大，一般的元件很难全盘掌控，所以后来在 WGUI 中加了一个 Touch Screen 模块，用以全盘接手触摸屏事件的管理。

Draw Manager：Draw Manager 是伴随 Touch Screen 一起出现的，因 Touch Screen 模块为方便控制会记录 Screen 中所有元件的属性及状态，为减轻图形系统代码冗余，将 Category Screen 中的元件也统一起交由 Draw Manager 绘制，Draw Manager 会依据每个元件的属性及状态依次将其绘制出来。

GUI Layer

这里所说的 GUI Layer 应该是前面讲的图形系统中的绘画子系统，一般绘画子系统的接口会分成以下几种类型：

图形：即通过程序计算，以画点为基础绘制出各种几何图形。

图像：以外部的图像资源为基础，通过各种图像解码器绘制出相应的点阵图。

文本：即文字或文本串的输出。

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

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

我们平台的绘画子系统与设备子系统分的不是很清楚，以上所说的几种类型中控件、填充及文本是专属 GUI 的，图形与图像虽然在 GUI 中也有接口，但我们在应用层中更多的是直接使用 GDI 中的接口，虽然这样不太合理，但目前状态下也只能这样(因 GUI 中的功能都不太完善)。

有一点要注意，在很早期的版本中，GUI 接口名称都以 pixtel_UI_ 为前缀(主要是 06 年以前的代码)，后来都改成以 gui_ 为前缀了，所以如果您的代码很老，要记得将实例中的代码做相应的改动。

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

GDI Layer

GDI Layer 分为以下几个部份：

Graphic：图形类接口，GDI 中的图形绘制一般会有相应的硬件加速。

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

Font：即字库与字体管理，用以绘制单个文字。

LCD & Layer：控制每个硬件屏幕(目前主要是主屏及副屏)，以及与屏幕相辅的缓冲区 Layer 管理。

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

排版常量

再列出一些绘画中常用的尺度常量：

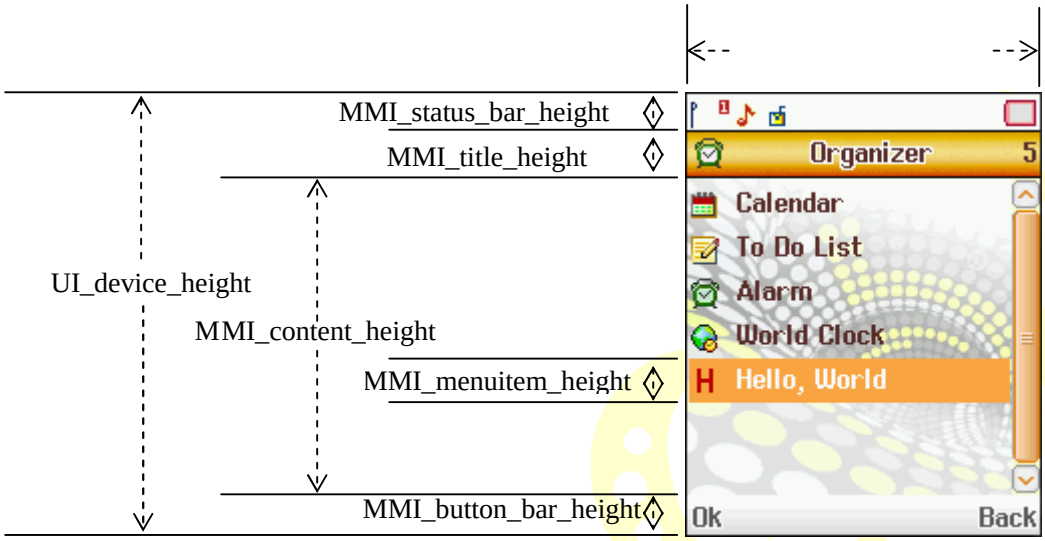


图 8.2

以上常量可能会有一些宏与之作用相同，如 MMI_MENUITEM_HEIGHT 与 MMI_menuitem_height 都表示菜单项的高度，但建议使用 MMI_menuitem_height，因为 MMI_MENUITEM_HEIGHT 是菜单项高度的系统初始值，而这个高度在运行时是有可能变化的，MMI_menuitem_height 会依情况自行变化，所以在使用排版常量时建议使用上面列出的这些值。

常用颜色

这是一些系统中常用的颜色：

GUI 颜色	GDI 颜色	RGB 值	颜色示例
UI_COLOR_RED	GDI_COLOR_RED	255, 0, 0	
UI_COLOR_GREEN	GDI_COLOR_GREEN	0, 255, 0	
	GDI_COLOR_BLUE	0, 0, 255	
UI_COLOR_BLACK	GDI_COLOR_BLACK	0, 0, 0	
UI_COLOR_WHITE	GDI_COLOR_WHITE	255, 255, 255	
	GDI_COLOR_GRAY	127, 127, 127	
UI_COLOR_GREY		192, 192, 192	
UI_COLOR_LIGHT_GREY		120, 120, 120	
UI_COLOR_DARK_GREY		64, 64, 64	
UI_COLOR_3D_FILLER		180, 180, 200	

图 8.3

不同的颜色可以直接通过 RGB 值组合而来，例如 RED=255, BLUE = 103, GREEN = 102，组合起来的颜色:  ,我们可以用以下方式直接使用：

GUI: color my_color = {255, 103, 102, 100};

GDI: gdi_color my_color = gdi_act_color_from_rgb(255, 255, 103, 102);

注：GUI 一行中的最后一个参数“100”与 GDI 一行中的第一个参数“255”是通常所说的 Alpha 值(透明度)，此参数目前用处不大，可忽略。

总结：

本章简单的讲了一下我们平台 MMI 架构以及图形系统总体概念。下面几章我们基本上会以 GUI 及 GDI 为主，控件及屏幕模板在第三部份会详细讲述。

第二部份我们每章都会尽量将所有绘画接口演示一遍，可自行选其一动手实践一遍，也可选每章结束时列出来的习题实践。

第九章：文本

第一部份我们已经用过一些基本的文本函数：

```
void mmi_myapp_entry(void)
{
    ……
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
    gui_print_text((UI_string_type)GetString(STR_MYAPP_HELLO));
    ……
}
```

代码 9.1

图 9.1

字体

常用的字体属性有如下几个：

属性	描述
bold	粗体，如“**Hello,World**”。
italic	斜体，如“*Hello,World*”。
underline	下划线，如“Hello,World”。
size	字号大小，因象形文字每种字号都要占用很大资源空间，故目前字号只对英文及数字有效（常用字号有 SMALL_FONT：小号，MEDIUM_FONT：中号，LARGE_FONT：大号，SUBLCD_FONT：副屏字体，DIALER_FONT：拨号字体，VIRTUAL_KEYBOARD_FONT：虚拟键盘字体）。

图 9.2

以上几种属性都可以在结构体 stFontAttribute 中设置，使用方式如下例：

```
void mmi_myapp_entry(void)
{
```

```
    stFontAttribute     f = {0};
    f.size = LARGE_FONT;
    ……
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
    gui_set_font(&f);
    gui_print_text((UI_string_type)GetString(STR_MYAPP_HELLO));
    ……
}
```

代码 9.2

代码 9.2 演示如何设置字号大小，其它属性的使用方法类似，设置完后由函数 gui_set_font 负责将属性值传给系统。上例代码的演示结果如下：

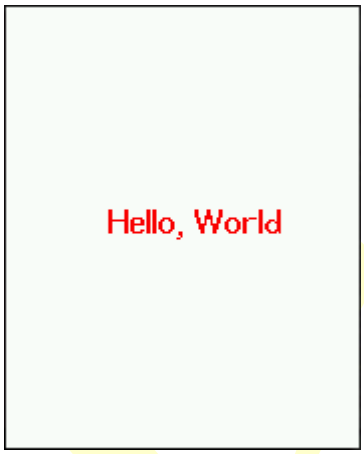


图 9.3

stFontAttribute 中还有以下几个不常用的属性：

属性	描述
color	字体颜色，已被 gui_set_text_color 取代，此属性目前无效。
type	字体类型，暂时无效。
oblique	倾斜，斜率是 italic 一半，很少用。
smallCaps	小写锁定，暂时无效。

图 9.4

带边框的文本

```
void mmi_myapp_entry(void)
{
    stFontAttribute     f = {0};
    f.size = LARGE_FONT;
    ……
    gui_move_text_cursor(50, 100);
    gui_set_text_color(UI_COLOR_RED);
```

```
    gui_set_font(&f);
    gui_set_text_border_color(UI_COLOR_GREEN);
    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO));
    ……
}
```

代码 9.3

带边框的文本由函数 gui_print_bordered_text 输出，gui_print_bordered_text 与 gui_print_text 使用方式类似，但记得在使用之前要先用函数 gui_set_text_border_color 设好边框颜色。

结果如下：

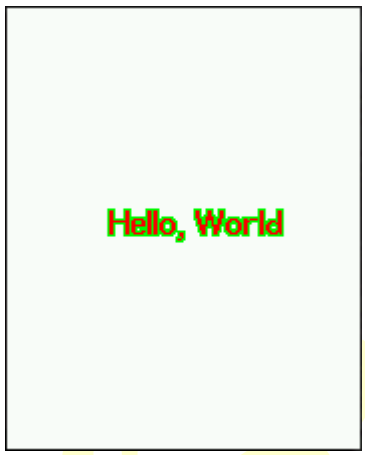


图 9.5

精确排版

要精确排版得先测量好字串的宽与高，如下代码演示了如何将字串左右居中：

```
void mmi_myapp_entry(void)
{
    S32 x, y, w, h;
    stFontAttribute     f = {0};
    f.size = LARGE_FONT;
    ……
    gui_set_text_color(UI_COLOR_RED);
    gui_set_font(&f);
    gui_set_text_border_color(UI_COLOR_GREEN);
    gui_measure_string((UI_string_type)GetString(STR_MYAPP_HELLO), &w, &h);
    x = (UI_device_width - w) / 2;
    y = (UI_device_height - h) / 4;
    gui_move_text_cursor(x, y);
    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO));
    ……
}
```

代码 9.4

函数 gui_measure_string 用来测量字串宽高，其第一个参数是要测量的字串，第二三个参数是宽高变量的地址。当然在测量之前要先设好字体，因为字串的宽高会依字体不同而不同。

结果如下：

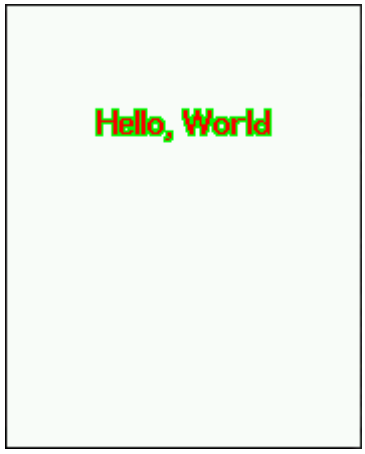


图 9.6

练习

请尝试完成如图 9.7 所演示的效果：

图 9.7

第十章：图形

点

图形都是以点为基础画出来的，下面代码将在屏幕的正中间画一个点：

```
void mmi_myapp_entry(void)
{
    ……
    gui_putpixel(UI_device_width / 2, UI_device_height / 2, UI_COLOR_BLACK);
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    ……
}
```

代码 10.1

如图 10.1,在屏幕的正中间有一个象素的黑点：

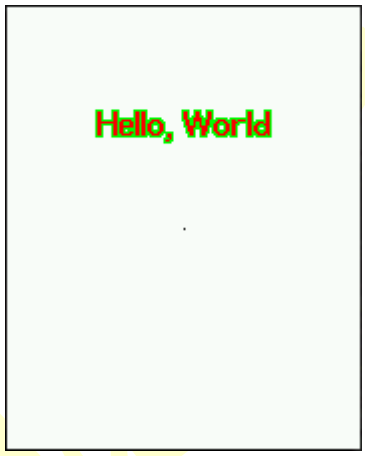


图 10.1

另外，函数 gdi_draw_point 也可以达到同样的效果。

线

函数 gui_line 可以画条一个象素宽的直线：

```
void mmi_myapp_entry(void)
{
    ……
    gui_line(30, 100, 150, 140, UI_COLOR_BLACK);
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    ……
}
```

代码 10.2

结果如下图：

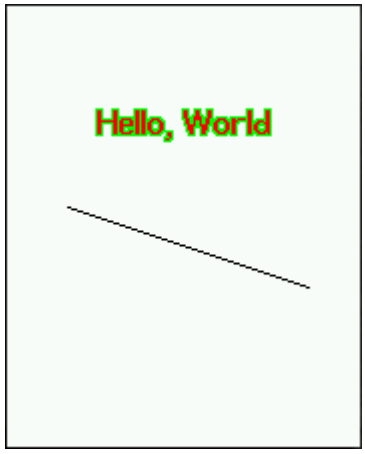


图 10.2

画线的还有如下一些函数：

名称	描述
gdi_draw_line	与 gui_line 作用完全相同，只是当线是水平或垂直的时候，此函数会做一些硬件加速，因此建议多用此函数。
gdi_draw_line_style	画带风格的线，如：- ，— — — — — — — — 等。
gui_draw_vertical_line	画垂直线，可用 gdi_draw_line 取代。
gui_draw_horizontal_line	画水平线，可用 gdi_draw_line 取代。
gui_wline	与 gui_line 类似，只是可以画不同宽度的线（其余所有画线的函数均只能画一个象素宽的线）。

图 10.3

框

```
void mmi_myapp_entry(void)
{
    ……
    gui_draw_rectangle(x - 4, y - 4, x + w + 4, y + h + 4, UI_COLOR_RED);
    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO));
    ……
}
```

代码 10.3

gui_draw_rectangle 用来画边框，函数 gdi_draw_rect 与 gui_draw_rectangle 效果完全一样。

结果如下图：

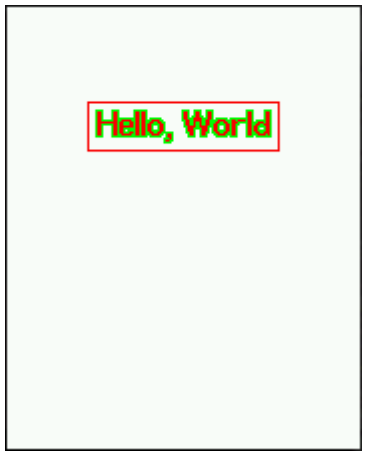


图 10.4

填充矩形

填充矩形是一个实心的框，使用方式如下例：

```
void mmi_myapp_entry(void)
{
    ……
    gui_draw_rectangle(x - 4, y - 4, x + w + 4, y + h + 4, UI_COLOR_RED);
    gui_fill_rectangle(x, y, x + w, y + h, UI_COLOR_GREY);
    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO));
    ……
}
```

代码 10.4

结果如下，一个灰色的填充矩形：

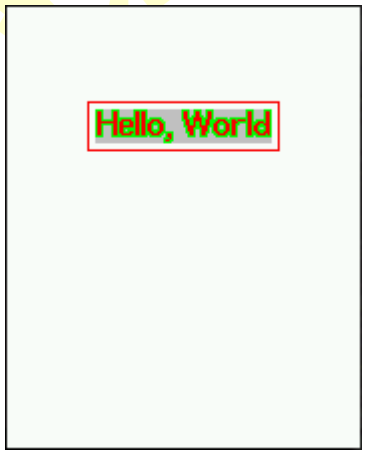


图 10.5

其它填充矩形：

名称	描述	范例
gui_cross_hatch_fill_rectangle	十字纹(单色与背景色交替)，函数 gdi_fill_dot_rect 效果完全一样。	
gui_hatch_fill_rectangle	百叶窗(单色与背景色交替)。	
gui_alternate_cross_hatch_fill_rectangle	十字纹(两种颜色交替)。	
gui_alternate_hatch_fill_rectangle	百叶窗(两种颜色交替)。	

图 10.6

带框填充矩形

```
void mmi_myapp_entry(void)
{
    ……
    gdi_draw_frame_rect(x - 4, y - 4, x + w + 4, y + h + 4,
        gdi_act_color_from_rgb(255, 204, 255, 102), GDI_COLOR_RED, 3);
    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO));
    ……
}
```

代码 10.5

如图 10.7，这是一个边框色为 GDI_COLOR_RED，填充色为 gdi_act_color_from_rgb(255, 204, 255, 102)，并且边框宽度为 3 的带边框填充矩形：

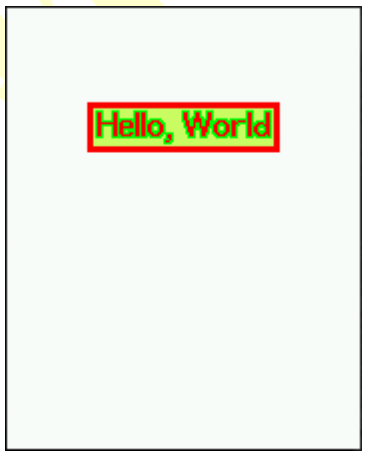
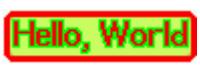


图 10.7

其它带框填充矩形为：

名称	描述	范例
gdi_draw_round_rect	圆角矩形	

gdi_draw_button_rect	按钮风格的矩形	
gdi_draw_shadow_rect	带阴影的矩形	
gdi_draw_gradient_rect	递进色填充矩形	

图 10.8

练习

请输出如图 10.9 类似的画面：

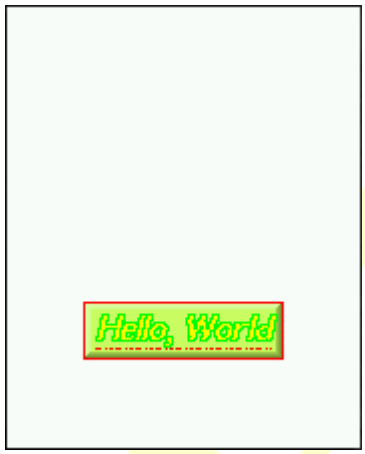


图 10.9

第十一章：图像

图像分类

图像按显示方式可分为静态图像与动画两种。
按图像格式可分为 BMP、JPG、GIF、PNG 等等。
按存储方式可分为以下三种：

存储方式	描述
资源	如第七章所示，图像资源是我们平台内部用的最多的一种存储方式。资源又分两种使用方式，一是资源 ID，一是资源 Buffer，资源 Buffer 即以 GetImage(IMAGE_ID)方式由资源 ID 转换过来的。
文件	即在系统运行时从文件系统中动态获取的图像。
Buffer	与资源存储方式不同，资源存储的图像内容中加入了我们平台自定义的格式数据，而这所说的 Buffer 只有纯粹的图像数据(如网络在线下载的临时图像数据等)。

图 11.1

静态图像

暂时借用主菜单中 ORGANIZER 的图标：

```
……
#include "MainMenuDef.h"
……
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw_id(30, 110, MAIN_MENU_MATRIX_ORGANIZER_ICON);
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    ……
}
```

代码 11.1

显示结果如下：

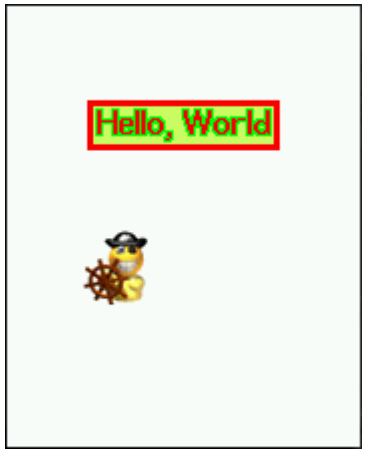


图 11.2

同样的，下面这样写可以达到完全一样的效果：

```
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw(30, 110,   (U8*)GetImage(MAIN_MENU_MATRIX_ORGANIZER_ICON));
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    ……
}
```

代码 11.2

gdi_image_draw_id 是以资源 ID 方式显示图像，gdi_image_draw 则是以资源 Buffer 方式显示图像。

当图像是放在文件系统中的时候(假设存储路径为：”D:\\MM_OR.gif”)，如：

图 11.3

此时的图像显示方式为：

```
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw_file(100, 100, (S8*)L"D:\\MM_OR.gif");
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    ……
}
```

代码 11.3

gdi_image_draw_file 只用将文件的存储路径传入即可(注意要用 Unicode 编码)。

显示结果如下图：

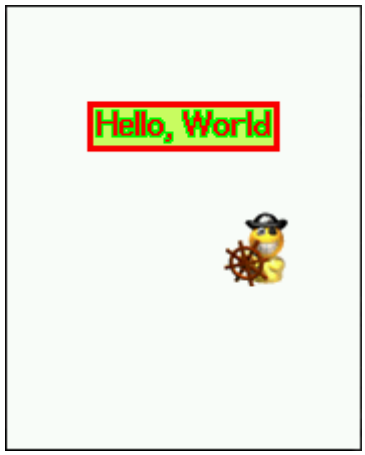


图 11.4

我们还可以对图像缩放：

```
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw_resized_id(30, 100, 20, 30, AIN_MENU_MATRIX_ORGANIZER_ICON);
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    ……
}
```

代码 11.4

函数 gdi_image_draw_resized_id 除了要传入 x, y 座标外，还要传入想要输出的宽与高。以上代码显示的结果如下：

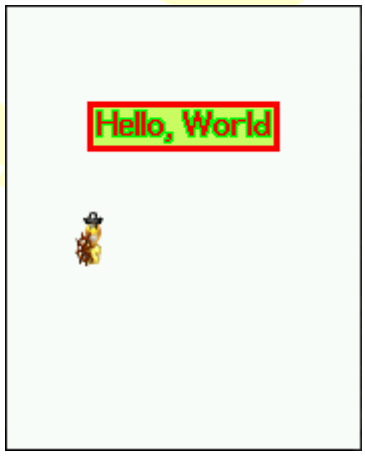


图 11.5

常用的静态图像显示函数有如下一些：

函数	说明
gdi_image_draw_id	资源 ID，不缩放。
gdi_image_draw	资源 Buffer，不缩放。
gdi_image_draw_file	文件，不缩放。
gdi_image_draw_ext	Buffer，不缩放。

gdi_image_draw_resized_id	资源 ID，可缩放。
gdi_image_draw_resized	资源 Buffer，可缩放。
gdi_image_draw_resized_file	文件，可缩放。
gdi_image_draw_resized_ext	Buffer，可缩放。

图 11.6

注：除 Buffer 图像外，其余储存方式的图像函数均不用传入格式参数，系统会自动识别图像格式。

动画

动画的显示方式与静态图像类似，我们也借用 ORGANIZER 中的动画图标，如下例代码：

```
……
gdi_handle my_anim;
void mmi_myapp_entry(void)
{
    ……
    gdi_anim_draw_id(50, 100, MAIN_MENU_MATRIX_ORGANIZER_ANIMATION, &my_anim);
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    ……
}
```

代码 11.5

因此处无法演示出动画，故只能各位自行体验效果，如下图：

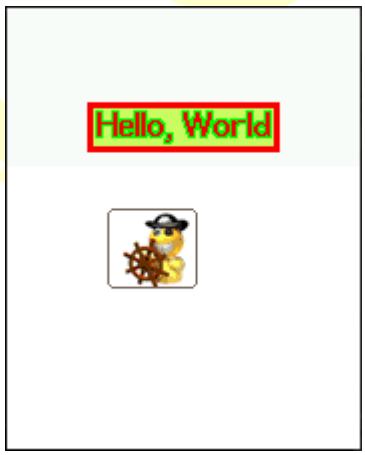


图 11.7

参数 my_anim 主要用来停止动画，如下代码所示：

```
gdi_handle my_anim;
void stop_my_anim(void)
{
    gdi_anim_stop(my_anim);
}
void mmi_myapp_entry(void)
{
```

```
    ……
    gdi_anim_draw_id(50, 100, MAIN_MENU_MATRIX_ORGANIZER_ANIMATION, &my_anim);
    SetKeyHandler(stop_my_anim, KEY_LSK, KEY_EVENT_UP);
    ……
}
```

代码 11.6

上例代码运行后，只要按左软键，动画就可以停止了。

与静态图像类似，每种储存类型的动画都分可缩放与不可缩放两种显示方式，另外，动画还有自己的两种显示方式：
只画一次：　　即动画只从第一帧显示到最后一帧，然后不再继续下一轮循环。
指定开始帧：　　指定动画由哪一帧开始画。
常用的静态图像函数有如下一些：

函数	说明
gdi_anim_draw_id	资源 ID，不缩放。
gdi_anim_draw_id_once	资源 ID，不缩放，只画一次。
gdi_anim_draw	资源 Buffer，不缩放。
gdi_anim_draw_frames	资源 Buffer，不缩放，指定开始帧。
gdi_anim_draw_resized	资源 Buffer，可缩放。
gdi_anim_draw_once	资源 Buffer，不缩放，只画一次。
gdi_anim_draw_file	文件，不缩放。
gdi_anim_draw_file_resized	文件，可缩放。
gdi_anim_draw_file_frames	文件，不缩放，指定开始帧。
gdi_anim_draw_mem	Buffer，不缩放
gdi_anim_draw_mem_frames	Buffer，不缩放，指定开始帧。
gdi_anim_draw_mem_resized	Buffer，可缩放
gdi_anim_draw_mem_once	Buffer，不缩放，只画一次。

图 11.8

图像都可以用 gdi_image_get_dimension_id 测量出宽高(动画与静态图像都用此函数)，使用方式如下：

```
void mmi_myapp_entry(void)
{
    ……
    gdi_image_get_dimension_id(MAIN_MENU_MATRIX_ORGANIZER_ICON, &w, &h);
    x =    (UI_device_width - w) / 2;
    y = (UI_device_height - h) / 2;
    gdi_image_draw_id(x, y, MAIN_MENU_MATRIX_ORGANIZER_ICON);
    ……
}
```

代码 11.7

上面代码演示如何上下左右居中显示图像，结果如下：

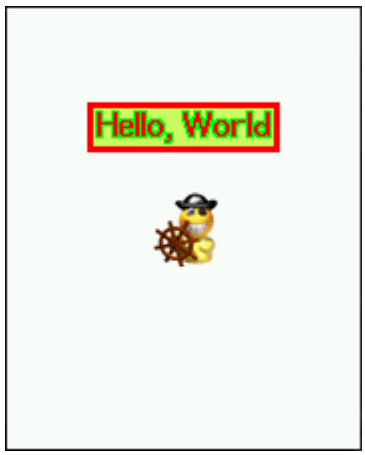


图 11.9

练习

请尝试输出以下画面：

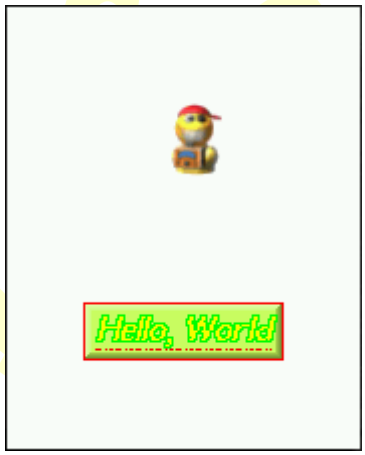


图 11.10

第十二章：背景

背景介绍

图形系统中每种可视的元件都可以为其加上背景，背景是元件显示风格的最大衬托，如果将所有元件的背景统一变化，则很容易使整个系统的显示风格产生变化，所以通常的图形系统会将所有元件的背景集中起来，再加一些别的容易控制又显眼的显示元素，就形成了整个系统的主题(Theme)。

我们系统中的主题大部分是以背景为基础，所以掌握了背景的使用方式，也就差不多掌握了主题的修改方法。

每个背景就是一个图形或图像填充起来的矩形，我们只要初始化结构体 UI_filled_area，然后交由函数 gui_draw_filled_area 画出来即可。

UI_filled_area 的定义如下：

```
typedef struct _UI_filled_area
{
    dword                   flags;          //总控制标志
    UI_image_type           b;              //背景图像
    gradient_color*         gc;             //递进颜色
    color                   c;              //背景色
    color                   ac;             //替换色
    color                   border_color;   //边框颜色
    color                   shadow_color;   //阴影颜色
    UI_transparent_color_type transparent_color; //透明色
} UI_filled_area;
```

代码 12.1

参数 flags 是结构体的总控制中心，通常按以下方式组合成：
flags = 类型标志 | 边框标志 | 阴影标志;

类型标志有如下一些：

类型标志	说明
UI_FILLED_AREA_TYPE_COLOR	颜色
UI_FILLED_AREA_TYPE_GRADIENT_COLOR	递进颜色
UI_FILLED_AREA_TYPE_TEXTURE	纹理
UI_FILLED_AREA_TYPE_BITMAP	图像
UI_FILLED_AREA_TYPE_HATCH_COLOR	百页窗(背景色 c 与原始底色交替)
UI_FILLED_AREA_TYPE_ALTERNATE_HATCH_COLOR	交替百页窗(背景色 c 与替换色 ac 交替)
UI_FILLED_AREA_TYPE_CROSS_HATCH_COLOR	十字纹(背景色 c 与原始底色交替)
UI_FILLED_AREA_TYPE_ALTERNATE_CROSS_HATCH_COLOR	交替十字纹(背景色 c 与替换色 ac 交替)
UI_FILLED_AREA_TYPE_NO_BACKGROUND	无背景
UI_FILLED_AREA_TYPE_3D_BORDER	3D 背景
UI_FILLED_AREA_TYPE_CUSTOM_FILL_TYPE1	自定义背景 1

UI_FILLED_AREA_TYPE_CUSTOM_FILL_TYPE2	自定义背景 2

图 12.1

后面会详细演示每种类型的用法。

边框标志有如下一些：

flags	说明	范例
UI_FILLED_AREA_BORDER UI_FILLED_AREA_SINGLE_BORDER	单边框(一个象素宽)	
UI_FILLED_AREA_DOUBLE_BORDER	双边框(两个象素宽)	
UI_FILLED_AREA_ROUNDED_BORDER | UI_FILLED_AREA_DOUBLE_BORDER	圆角边框	
UI_FILLED_AREA_3D_DEPRESSED_BORDER | UI_FILLED_AREA_DOUBLE_BORDER	3D 下陷边框	
UI_FILLED_AREA_DEPRESSED_BORDER | UI_FILLED_AREA_DOUBLE_BORDER	3D 突起边框	
UI_FILLED_AREA_LEFT_ROUNDED_BORDER	左圆角边框	
UI_FILLED_AREA_RIGHT_ROUNDED_BORDER	右圆角边框	

图 12.2

阴影标志有如下一些：

flags	说明	范例
UI_FILLED_AREA_SHADOW | UI_FILLED_AREA_DOUBLE_BORDER	单阴影(一个象素宽)	
UI_FILLED_AREA_SHADOW | UI_FILLED_AREA_DOUBLE_BORDER| UI_FILLED_AREA_SHADOW_DOUBLE_LINE	双阴影(两个象素宽)	

图 12.3

颜色

下面演示以颜色为背景的使用方法：

```
void mmi_myapp_entry(void)
{
    ……
    UI_filled_area filler = {0};

    EntryNewScreen(SCR_MYAPP_MAIN, NULL, mmi_myapp_entry, NULL);
    entry_full_screen();
    clear_screen();

    filler.flags = UI_FILLED_AREA_TYPE_COLOR | UI_FILLED_AREA_BORDER | UI_FILLED_AREA_SHADOW;
    filler.c = UI_COLOR_GREY;
```

```
    filler.border_color = UI_COLOR_DARK_GREY;
    filler.shadow_color = UI_COLOR_3D_FILLER;
    gui_draw_filled_area(20, 20, 156, 150, &filler);
    ……
}
```

代码 12.2

这是一个灰色填充，暗灰色单边框，亮色单阴影的背景，效果如下图：

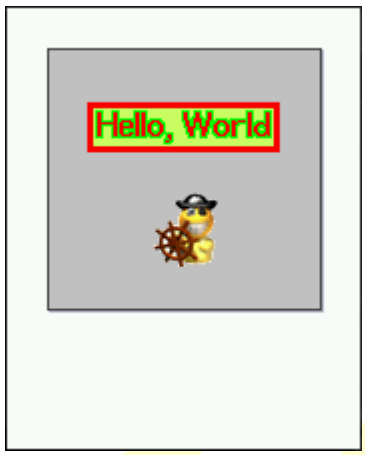


图 12.4

递进颜色

下面演示递进色的使用方法：

```
void mmi_myapp_entry(void)
{
    ……
    UI_filled_area filler = {0};
    static color g_colors[3] = {{255,0,0},{0,255,0},{0,0,255}};
    static U8 perc[2] = {30,70};
    gradient_color gc = { g_colors, perc, 3 };

    EntryNewScreen(SCR_MYAPP_MAIN, NULL, mmi_myapp_entry, NULL);
    entry_full_screen();
    clear_screen();

    filler.flags = UI_FILLED_AREA_TYPE_GRADIENT_COLOR;
    filler.gc = &gc;
    gui_draw_filled_area(20, 20, 156, 150, &filler);
    ……
}
```

代码 12.3

递进色需要用到一个结构体 gradient_color，其定义如下：

```
typedef struct _gradient_color
{   color *c;                    //颜色列表，数量由最后一个参数 n 决定。
```

```
    byte   *p;                  //百分比列表，个数为 n – 1，依次表示两个相邻颜色递进宽度占整个宽度的百分比。
    byte   n;                   //颜色数量。
} gradient_color;
```

代码 12.4

如上例，总共有三个颜色红、绿、蓝，其中红绿递进所占的百分比为 30%，绿蓝递进所占的百分比为 70%，最终显示如下：

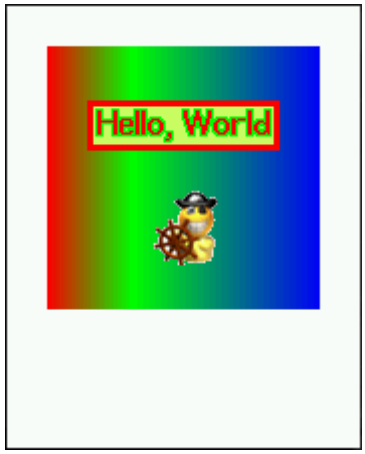


图 12.5

另外还有两个参数控制递进色的显示方式：

UI_FILLED_AREA_HORIZONTAL_FILL：　水平方式递进显示，此为默认方式，可以不用设。
UI_FILLED_AREA_VERTICAL_FILL：　垂直方式递进显示，从上到下递进显示。
UI_FILLED_AREA_FLIP_FILL：　反转显示，将递进色从右至左，或从下至上显示。

图像

我们以下面这副图(UI_FILLED_AREA_TYPE_BITMAP)作为背景：

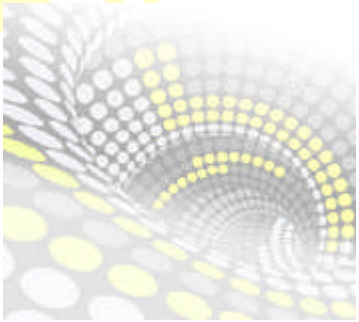


图 12.6

使用方式如下：

```
void mmi_myapp_entry(void)
{
    ……
    filler.flags = UI_FILLED_AREA_TYPE_BITMAP;
    filler.b = GetImage(IMG_GLOBAL_SUB_MENU_BG);
    gui_draw_filled_area(20, 20, 156, 150, &filler);
```

```
    ……
}
```

代码 12.5

显示结果为：

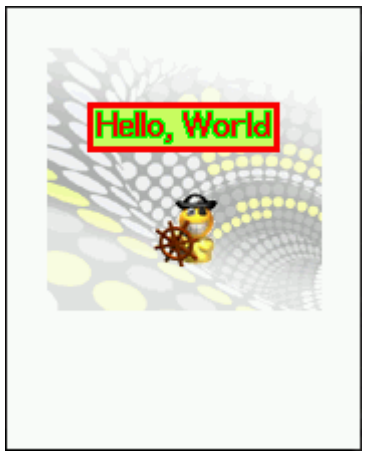


图 12.7

纹理

纹理就是不停的用一副图填充背景，直到填满为止，以下图为例：

图 12.8

使用方式为：

```
void mmi_myapp_entry(void)
{
    ……
    filler.flags = UI_FILLED_AREA_TYPE_TEXTURE;
    filler.b = GetImage(IMG_FLEXIBLE_TITLEBAR_BG);
    gui_draw_filled_area(20, 20, 156, 150, &filler);
    ……
}
```

代码 12.6

结果如下图：

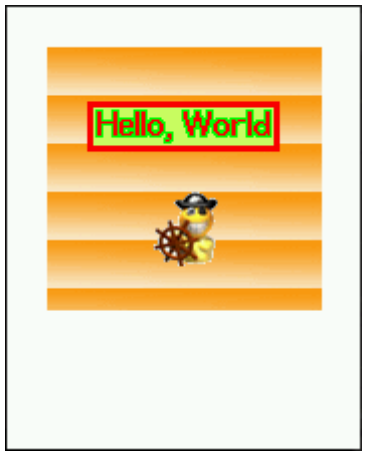


图 12.9

3D 效果

下面是 3D 效果的背景显示：

```
void mmi_myapp_entry(void)
{
    ……
    filler.flags = UI_FILLED_AREA_TYPE_3D_BORDER;
    filler.c = UI_COLOR_GREY;
    gui_draw_filled_area(20, 20, 156, 150, &filler);
    ……
}
```

代码 12.7

结果如下：

图 12.10

还有两种 3D 效果的背景：
UI_FILLED_AREA_TYPE_CUSTOM_FILL_TYPE1
UI_FILLED_AREA_TYPE_CUSTOM_FILL_TYPE2
效果如下：

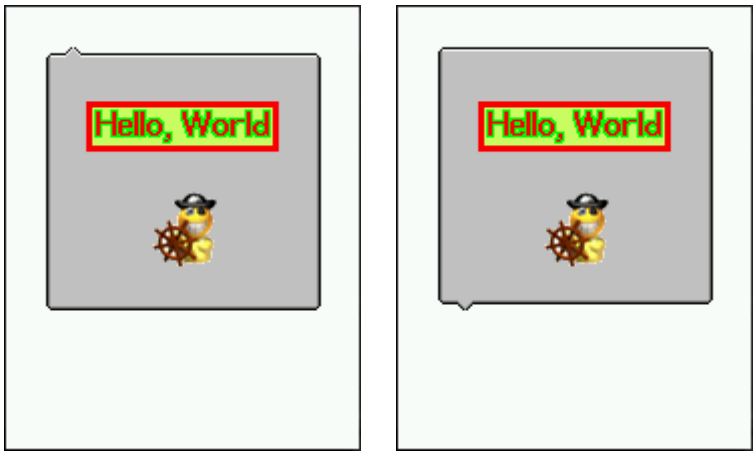


图 12.11

百页窗及十字纹

下面是百页窗及十字纹的显示效果：

flags	等效的图形函数	范例
UI_FILLED_AREA_TYPE_CROSS_HATCH_COLOR	gui_cross_hatch_fill_rectangle	Hello, World
UI_FILLED_AREA_TYPE_HATCH_COLOR	gui_hatch_fill_rectangle	Hello, World
UI_FILLED_AREA_TYPE_ALTERNATE_CROSS_HATCH_COLOR	gui_alternate_cross_hatch_fill_rectangle	Hello, World
UI_FILLED_AREA_TYPE_ALTERNATE_HATCH_COLOR	gui_alternate_hatch_fill_rectangle	Hello, World

图 12.12

动画背景

动画背景无法用填充区域来实现，参照第十一章的实现如下：

```
#include "Mmi_phnset_dispchar.h"
……
gdi_handle my_anim;
void mmi_myapp_entry(void)
{
    ……
    EntryNewScreen(SCR_MYAPP_MAIN, NULL, mmi_myapp_entry, NULL);
    entry_full_screen();
    clear_screen();
    gdi_anim_draw_id(0, 0, IMG_ID_PHNSET_ON_0, &my_anim);
    ……
}
```

代码 12.8

我们用开机动画作为整个画面的背景，如下图：

图 12.13

但是这样实现会有一个很大的缺陷：只要动画跳到第二帧以后，动画上面的文本及图像都会被盖住了，解决办法请看下章“层及特效”。

练习

请输出以下画面：

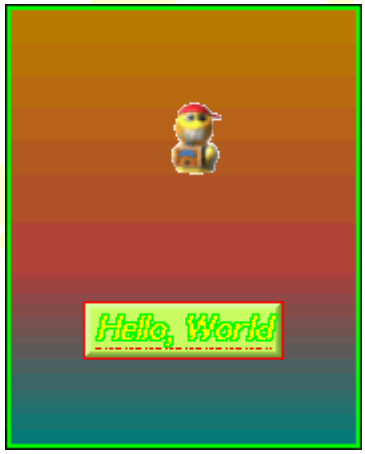


图 12.14

第十三章：层

介绍

层主要有两个作用：

缓冲：在某些频繁更新的界面中，如果某些显示元素一直不变化，我们就可以将这些元素提取出来画到一个模拟的屏幕中，当界面需要更新时，只需将要刷新的元素更新到另外一个模拟屏幕，而后将两个模拟屏幕合并到真正屏幕上，这样做就省掉了不变元素的重画时间(某些元素重画很耗时，如图像显示等)，从而减轻了系统负担及加速画面更新。我们把这样的模拟屏幕就叫做层，也可以说层就是屏幕的缓冲空间。

特效：因为层的格式简单且统一，并且一般的图形系统中都会用硬件来加速层合并，所以在层合并时加上些特效会很方便并且实现极快。我们系统中的特效有通透，半透明，剪切等等。

在上一章中，我们讲到了动画作背景的缺陷，有了层以后，我们就可以将不变的文本及图像放到新建的一个层中，将动画放到背景层中，每当有动画换帧时，只需将新的帧画到背景层中，然后合并两个层到屏幕(动画刷新时自动会合并)，这样做上层的文本等就不会被动画盖住了。

准备

为了方便演示层特效，我们将画一副大的浅色图作为背景，如下：

```
void mmi_myapp_entry(void)
{
    ……
    EntryNewScreen(SCR_MYAPP_MAIN, NULL, mmi_myapp_entry, NULL);
    entry_full_screen();
    clear_screen();
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    ……
}
```

代码 13.1

效果如下图：

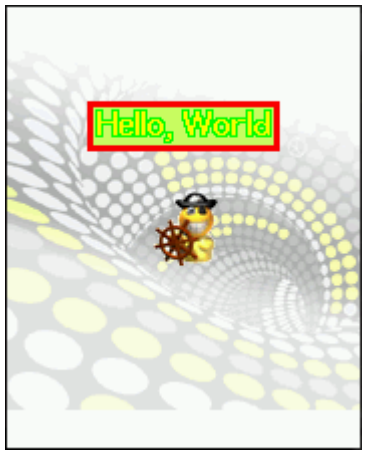


图 13.1

创建层

创建层需要先建一个层句柄(可以把层句柄当作一个指向层的索引)，我们都是通过层句柄来控制层的。函数 gdi_layer_create 用来创建层，其前四个参数指出层的位置及大小(位置是以实际屏幕左上角为原点的)，最后一个参数是刚创建的层句柄地址，用以返回所创建的层。下面代码中将创建一个宽 136，高 130 的层：

```
gdi_handle my_layer;
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(20, 20, 136, 130, &my_layer);
    ……
}
```

代码 13.2

要注意一点的是，因为创建层时系统要为其分配动态内存空间，而系统保留的内存一般只够创建一个 UI_device_width* UI_device_height 大小的层，所以如果调用 gdi_layer_create 时内存不足系统就会 ASSERT。解决的办法是使用函数 gdi_layer_create_using_outside_memory，从函数名就可以看出需要你自己申请内存，然后作为参数传进去创建层。

激活层

在我们的图形系统中，任何时刻有且只能有一个层处于激活状态，所有的绘画函数都是默认画到这个层中，所以想要在层上绘画必须先将其激活。

创建层并不会自动激活，我们还需要手动将其激活：

```
gdi_handle my_layer;
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(20, 20, 136, 130, &my_layer);
    gdi_layer_set_active(my_layer);
    ……
}
```

代码 13.3

基础层

系统开机的时候会为每个硬件屏幕创建一个基础层，基础层有以下几个特点：

1、 基础层由系统创建，无法删除。
2、 与硬件屏幕完全重合(位置与大小都一样)。
3、 系统默认的激活层，EntryNewScreen 时系统会自动将基础层激活。
4、 显示更加快速，基础层存储于芯片内的 flash 中，所以在其上面绘画极快，一般我们会将刷新频繁的内容放在基础层上。

由以上几个特点看出，在不使用多层的情况下，我们完全可以将基础层当成是硬件屏幕来看待，也就是说普通程序完全可以忽略层的概念。

尽管可以创建多少个层，但基础层肯定是不能浪费的，这里我们将在其上绘制背景图(在激活新层之前我们的东西都默认画到基础层中)，在新建层上绘制文字及图标。

因系统一般只在 EntryNewScreen 时才会自动将基础层激活，为避免特殊情况下使用层混乱，通常在新层上绘画完毕后，我们会主动将基础层还原为激活状态：

```
void mmi_myapp_entry(void)
{
    ……
    gdi_handle base_layer;
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(20, 20, 136, 130, &my_layer);
    gdi_layer_set_active(my_layer);
    ……
    gdi_layer_get_base_handle(&base_layer);
    gdi_layer_set_active(base_layer);
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

代码 13.4

函数 gdi_layer_get_base_handle 用来获取基础层的句柄。另外，用函数 gdi_layer_restore_base_active()也以起到相同的作用。

合并

函数 gui_BLT_double_buffer 用来合并层，但在使用之前先得用 gdi_layer_set_blt_layer 指明是哪几个层需要合并，函数 gdi_layer_set_blt_layer 能接受四个层句柄，也就是说我们系统同一时刻最多只能合并四个层(当然创建的层可以不止这个数)。另外要注意参数的顺序，第一个传入的层是放在最底下的，然后依次往上码。

要切记任何时候绘制完毕后一定要记得将层合并，否则你是看不到显示效果的。

```
gdi_handle my_layer;
void mmi_myapp_entry(void)
{
    ……
    gdi_handle base_layer;
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(20, 20, 136, 130, &my_layer);
    gdi_layer_set_active(my_layer);
    ……
    gdi_layer_get_base_handle(&base_layer);
    gdi_layer_set_active(base_layer);
    gdi_layer_set_blt_layer(base_layer, my_layer, NULL, NULL);
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}
```

代码 13.5

我们也可以直接用 gdi_layer_blt 来合并层，效果一样，如下所示：

```
gdi_handle my_layer;
void mmi_myapp_entry(void)
{
    ……
    gdi_handle base_layer;
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(20, 20, 136, 130, &my_layer);
    gdi_layer_set_active(my_layer);
    ……
    gdi_layer_get_base_handle(&base_layer);
    gdi_layer_set_active(base_layer);
    gdi_layer_blt(base_layer, my_layer, NULL, NULL, 0, 0, UI_device_width, UI_device_height);
}
```

代码 13.6

合并结果如下：

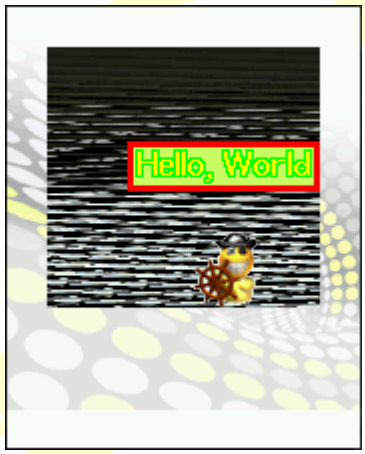


图 13.2

上面的显示结果我们发现两个问题，一是文本与图标向右下的偏移了一定距离，二是背景图被乱码遮盖住了。下面我们先来解决第一个问题：

层坐标系

首先我们要更正一个概念，在我们的图形系统中，所有绘画函数所使用的坐标参数原点并不是硬件屏幕的左上角，而是当前激活层的左上角，如下图所演示：

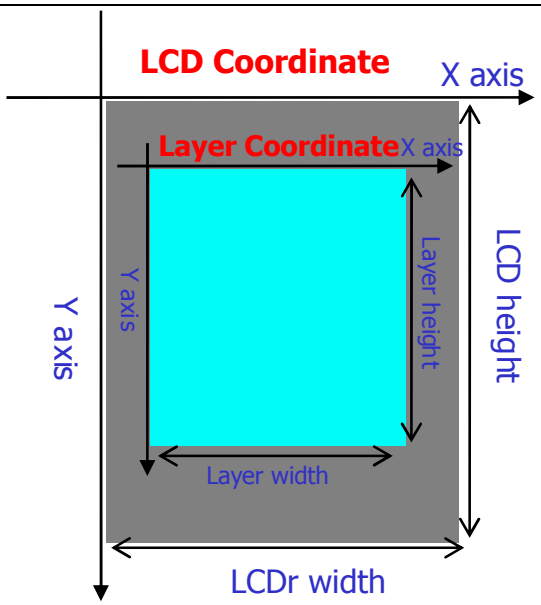


图 13.3

我们之前对文本图标的精确排版都是针对硬件屏幕来说，因以前都是在基础层上绘制，而基础层与屏幕重叠，所以不会出现问题。现在转移到新层之上后，我们就需要将这个偏移反向抵消掉：

```
gdi_handle my_layer;
void mmi_myapp_entry(void)
{
    ……
    gdi_handle base_layer;
    S32 layer_offset_x = 20, layer_offset_y = 20;
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(layer_offset_x, layer_offset_y, 136, 130, &my_layer);
    gdi_layer_set_active(my_layer);

    gui_set_text_color(text_color);
    gui_set_font(&f);
    gui_set_text_border_color(UI_COLOR_GREEN);
    gui_measure_string((UI_string_type)GetString(STR_MYAPP_HELLO), &w, &h);
    x = (UI_device_width - w) / 2 - layer_offset_x;
    y = (UI_device_height - h) / 4 -layer_offset_y;
    gui_move_text_cursor(x, y);

    gdi_draw_frame_rect(x - 4, y - 4, x + w + 4, y + h + 4,
        gdi_act_color_from_rgb(255, 204, 255, 102), GDI_COLOR_RED, 3);

    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO));

    gdi_image_get_dimension_id(MAIN_MENU_MATRIX_ORGANIZER_ICON, &w, &h);
    x =    (UI_device_width - w) / 2 - layer_offset_x;
    y = (UI_device_height - h) / 2 - layer_offset_y;
```

```
    gdi_image_draw_id(x, y, MAIN_MENU_MATRIX_ORGANIZER_ICON);
    ……
    gdi_layer_get_base_handle(&base_layer);
    gdi_layer_set_active(base_layer);
    gdi_layer_blt(base_layer, my_layer, NULL, NULL, 0, 0, UI_device_width, UI_device_height);
}
```

代码 13.7

结果如下，现在排版正确了：

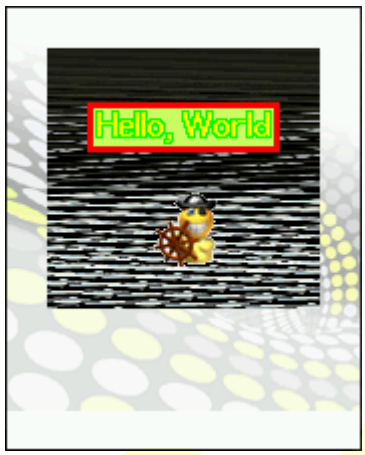


图 13.4

通透

现在再来解决背景图被遮住的问题。
首先我们要将新层清理一下，用函数 gdi_layer_clear 就可以将整个层刷成单一颜色(层激活后要立即执行)，这里我们将其刷成纯蓝色。
但是这样背景图还是会被一片蓝色盖住，要解决这个就要用到我们层特效“通透”了，只要用函数 gdi_layer_set_source_key 将某一颜色设为层的通透色，在层合并的时候，系统会自动将层中与通透色相同的颜色忽略掉(就是说这一点上看到的是底下层的颜色)。
代码如下：

```
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(layer_offset_x, layer_offset_y, 136, 130, &my_layer);
    gdi_layer_set_active(my_layer);
    gdi_layer_clear(GDI_COLOR_BLUE);
    gdi_layer_set_source_key(TRUE, GDI_COLOR_BLUE);
    ……
    gdi_layer_get_base_handle(&base_layer);
    gdi_layer_set_active(base_layer);
    gdi_layer_blt(base_layer, my_layer, NULL, NULL, 0, 0, UI_device_width, UI_device_height);
}
```

代码 13.8

显示结果为：

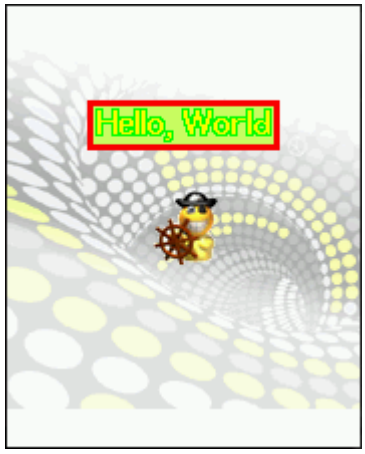


图 13.5

剪切

我们再来演示层的另一个特效“剪切”。
所谓剪切，就是在层中设一个限制区域，只有在这个区域中的绘画才是有效的，否则就会被自动忽略。
剪切特效有两个特点：

1、 每个层一定有而且只能有一个剪切区域。
2、 剪切区域一经设置，永久生效。所以剪切区域用完后最好用 gdi_layer_reset_clip 还原(如不还原则有可能什么东西都画不上来)。

剪切用函数 gdi_layer_set_clip 来实现，如下所示：

```
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(layer_offset_x, layer_offset_y, 136, 130, &my_layer);
    gdi_layer_set_active(my_layer);
    gdi_layer_clear(GDI_COLOR_BLUE);
    gdi_layer_set_source_key(TRUE, GDI_COLOR_BLUE);
    gdi_layer_set_clip(40, 25, 100, 100);
    ……
}
```

代码 13.9

效果见下图：

图 13.6

半透明

下面演示半透明特效的使用方法：

```
void mmi_myapp_entry(void)
{
    ……
    gdi_image_draw_id(0, 0, IMG_ID_PHNSET_WP_3);
    gdi_layer_create(layer_offset_x, layer_offset_y, 136, 130, &my_layer);
    gdi_layer_set_active(my_layer);
    gdi_layer_clear(GDI_COLOR_BLUE);
    gdi_layer_set_source_key(TRUE, GDI_COLOR_BLUE);
    gdi_layer_set_opacity(TRUE, 128);
    ……
}
```

代码 13.10

gdi_layer_set_opacity 的第一个参数指明要不要开启半透明效果，第二个参数是透明度的取值，范围从 0 至 255，值越小表示透明度越高，当取值为 0 时就会完全被透掉，255 即完全不透明。
效果如下图：

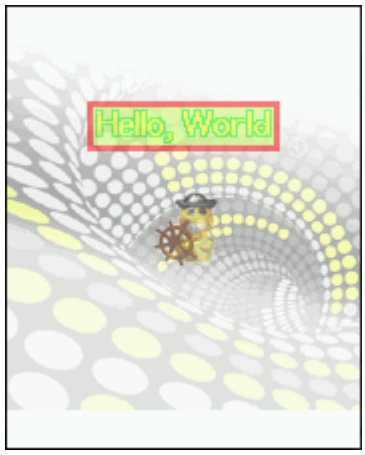


图 13.7

释放层

创建层需要为其分配内存空间，所以层用完后也要手动将其释放(切记一定要释放，否则别的程序就无法创建层了)，方法如下：

```
gdi_handle my_layer;
void mmi_myapp_exit(void)
{
    ……
    gdi_layer_free(my_layer);
}
void mmi_myapp_entry(void)
{
    ……
    EntryNewScreen(SCR_MYAPP_MAIN, mmi_myapp_exit, NULL, NULL);
    ……
}
```

代码 13.11

锁屏

假设我们屏幕中有一个 title 以及一个 menu，你可能会看到如下的绘制流程：

```
draw screen start

    draw title start
    ……
    gdi_layer_blt(……)
    draw title end

    draw menu start
    ……
    gdi_layer_blt(……)
    draw menu end

……
gdi_layer_blt(……)
draw screen end
```

代码 13.12

从上面可以看出 gdi_layer_blt 被调用了三次，虽然我们只需最后一次就足够了，但 title 及 menu 的中的 gdi_layer_blt 不能少，因为 title 及 menu 都有可能会独立于 screen 之外刷新，而在我们系统中频繁的调用 gdi_layer_blt 会极大影响系统性能。为解决这个我们引进了一套锁屏机制，其原理是在绘画时加入一个计数器，只在当计数器为零时 gdi_layer_blt 才会真正起作用，我们再看下一个流程图：

```
draw screen start----------------------------------------计数器 = 0
gdi_layer_lock_frame_buffer();---------------------计数器 = 1
    draw title start
```

```
    gdi_layer_lock_frame_buffer();----------计数器 = 2
    ……
    gdi_layer_unlock_frame_buffer();----------计数器 = 1
    gdi_layer_blt(……)
    draw title end
    draw menu start
    gdi_layer_lock_frame_buffer();----------计数器 = 2
    ……
    gdi_layer_unlock_frame_buffer();----------计数器 = 1
    gdi_layer_blt(……)
    draw menu end
……
gdi_layer_unlock_frame_buffer();----------计数器 = 0
gdi_layer_blt(……)
draw screen end
```

代码 13.13

gdi_layer_lock_frame_buffer()会将计数器加一，gdi_layer_unlock_frame_buffer()会将计数器减一。显而易见，上面流程图中只有最后一次 gdi_layer_blt 才会起实际作用。下面是代码中使用范例：

```
void mmi_myapp_entry(void)
{
    ……
    EntryNewScreen(SCR_MYAPP_MAIN, mmi_myapp_exit, NULL, NULL);
    gdi_layer_lock_frame_buffer();
    ……
    ……
    gdi_layer_unlock_frame_buffer();
    gdi_layer_blt(base_layer, my_layer, NULL, NULL, 0, 0, UI_device_width, UI_device_height);
}
```

代码 13.14

注：gui_lock_double_buffer 与 gui_unlock_double_buffer 跟上述两个函数功能一致，可参考函数 setup_UI_wrappers。

练习

请尝试输出以下画面：

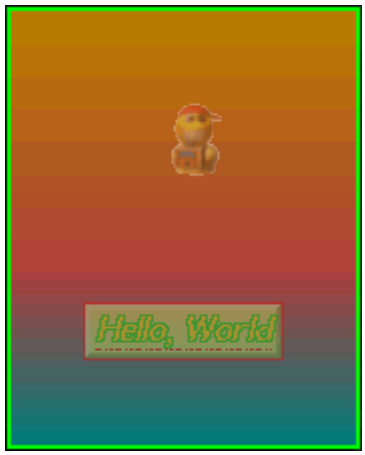


图 13.8

第三部份：互动

第十四章：开始

互动

MMI 之根本就是互动。互动分两边——人机互动与机人互动：

人机互动：即用户输入，通常有键盘输入与触摸屏输入两种。

机人互动：即机器反馈，最常见的是画面反馈与声音反馈，本教程第二部份的“绘画”就是最基础的画面反馈。

从写程序的概念来说，我们把互动分为三个等级：

原始级：程序接收用户输入后，以基础的绘画反馈用户。

控件级：程序创建控件，由控件接管部份用户输入，并由控件自行反馈用户。

屏幕级：由屏幕模板全盘控制用户输入与画面反馈。

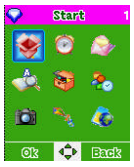
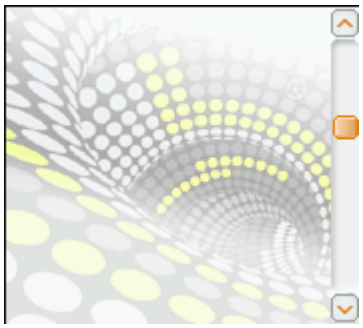
第三部份主要讲述控件级与屏幕级互动。

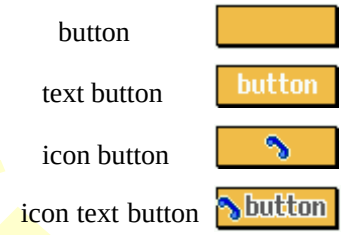
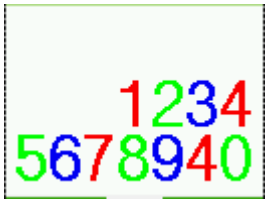
如第八章所讲，控件分为 GUI Control 与 WGUI Control 两种，GUI Control 加上输入控制就成了 WGUI Control，所以第三部份前几章我们将以“GUI Control+键盘输入+触摸屏输入=WGUI Control”的方式为大家讲述。

接下来几章我们会演示如何现将现有的程序整理成一个屏幕模板，并将屏幕模板嵌入 Draw Manager 及 Touch Screen 中。

在开始之前我们先简要的游览一下系统中的常见控件与常见屏幕：

控件

名称	描述	范例
Fixed matrix menu	**矩阵菜单框架** 右小图所示的即是矩阵菜单。 注：名称中的 Fixed 意即所有菜单项都是同一种类型（最原始的菜单中每项都可以是不同类型，因其极复杂所以从来不用）。 	
Fixed list menu	**列表菜单框架** 如右小图所示的即是列表菜单，菜单都是由菜单框架与菜单项组合而成。菜单框架主要用来控制菜单项的排版及状态。 菜单框架与后面所列的四种菜单项两两组合就成了各种不同的菜单。 	
Fixed text menuitem	**文本菜单项** 文本菜单项分列表型与矩阵型两种，如范例所示：	

Fixed icontext menuitem	**文本图标菜单项** 文本图标菜单项也分列表型与矩阵型两种，如范例所示：	
Fixed icontext list menuitem	**多列菜单项** 即多列图标与多列文本组合在一起的菜单项，如范例中上是两个图标加一个文本，范例中下是两列文本。	
Fixed twostate menuitem	**两状态菜单项** 菜单项在运行是有两种状态，典型的是复选框与单选框。	
Button	**按钮** 按钮有四种类型，如范例所示： 左右软键都是 icontext button，只是其中的 icon 一般都会省掉。	
Scrollbar	**滚动条** 如范例所示，这是三种风格的滚动条，分别用于不同尺寸的屏幕。 范例中的是竖向滚动条，另外还有横向滚动条。	
Progress	**进度条**	
Slide	**滑杆**	
Multitap input	**输入法候选字符列表** 输入法中按键后当有多个字符可供选择时，就显示此列表以用户选择。 如范例所示，在英文输入法时按数字键“5”时会出现此列表，用户可以在此列表中选择自己想要的字符(可连续按此键切换选择或用触摸屏点选)。	
Single line input box	**单行输入框** 一般在菜单式内嵌式编辑中多用单行输入框。	
Multi line input box	**多行输入框** 多行输入框多用于全屏状态下复杂文本输入，比如中文输入等。	
Dialer input box	**拨号输入框** 拨号时输入数字用。	

Dialer input box (touch screen)	**拨号输入框（触摸屏）** 用于触摸屏的拨号输入框。	
Status bar	**状态条** 状态条用于显示当前的系统信息。 状态条有三条： Status bar 0：主屏最上面的横状态条，在主屏的 Idle 或一般程序中均可显示。 Status bar 1：主屏右边的竖状态条，一般在 Status bar 0 中图标排不下的时候就会将图标放到 Status bar 1 中(只能在主屏的 Idle 中显示)。 Status bar 2：副屏上面的状态条。	
Title bar	**标题条** 标题条包括左边的图标，中间的标题，右边的菜单高亮项序号三部份。	
Information bar	**输入法信息条** 用于多行编辑框信息显示，包括左边的输入法指示，右边的输入字符数信息两部份。	
Scrolling text	**滚动文本** 当文本条过长无法完全显示时，就可以启用滚动文本条。	
Popup description	**弹出提示框** 当高亮某个菜单项时，我们可以用弹出提示框显示一些有关此菜单项的信息。	
Virtual keyboard	**虚拟键盘** 触摸屏中编辑时用的虚拟键盘。	

图 14.1

屏幕

<table>
<tr><td rowspan="2">菜单</td><td>Category6Screen
 

文本列表菜单</td><td>Category22Screen
 

文本矩阵菜单</td><td>Category15Screen
 

图标文本列表菜单</td><td>Category14Screen
 

图标文本矩阵菜单</td></tr>
<tr><td>Category11Screen
 

单选框菜单</td><td>Category13Screen
 

复选框菜单</td><td>Category73Screen
 

多列菜单</td><td>Category57Screen
 

内嵌编辑菜单</td></tr>
<tr><td>提示框</td><td>Category7Screen
 

文本提示框</td><td>Category8Screen
 

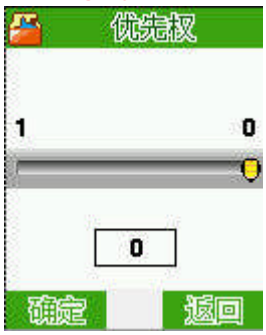
文本图标提示框</td><td>Category74Screen
 

长文本提示框</td><td>Category165Screen
 

全屏弹出提示框</td></tr>
<tr><td>其它</td><td>Category5Screen
 

全屏多行编辑框</td><td>Category111Screen
 

全屏单行输入框</td><td>Category142Screen
 

图片游览器</td><td>Category87Screen
 

数值选取</td></tr>
</table>

图 14.2

基础画面

为方便后面几章演示，我们先将输出画面改成以下方式：

图 14.2

代码修改如下：

```
void mmi_myapp_entry(void)
{
    S32 x, y, w, h;
    color text_color = {255, 255, 0, 100};

    EntryNewScreen(SCR_MYAPP_MAIN, mmi_myapp_exit, NULL, NULL);
    gui_lock_double_buffer();
    entry_full_screen();
    clear_screen();

    gui_set_text_color(text_color);
    gui_set_text_border_color(UI_COLOR_GREEN);
    gui_measure_string((UI_string_type)GetString(STR_MYAPP_HELLO), &w, &h);
    x = (UI_device_width - w) / 2;
    y = MMI_title_y;
    gui_move_text_cursor(x, y);
    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO));

    gui_unlock_double_buffer();
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
    SetKeyHandler(GoBackHistory, KEY_RSK, KEY_EVENT_UP);
}
```

代码 14.1

第十五章：控件

GUI Control

本章所讲的控件是 GUI Control，因控件种类多无法一一演示，我们将以“文本图标列表菜单”为大家示范。

控件使用一般都有三个步骤：创建、设置、显示：

创建：基本上就是创建一个此控件的结构体对象，一般都是申明一个全局对象，我们很少用动态对象，一是我们系统动态内存管理不是很成熟，二是因为我们的屏幕中一般控件数量比较少，用全局的比较方便。另外要注意一点，我们系统中控件一般都会有一个类如 gui_create_*control_name*()的函数，此函数不是用来创建控件，而是初始化控件对象的。

设置：形如 gui_create_XXX，gui_set_XXX，gui_resize_XXX 之类的都是控件设置类接口。

显示：显示接口一般都类似于 gui_show_*control_name*()。

菜单是由菜单框架及 n 个菜单项组成，所以“文本图标列表菜单”要创建两种控件：Fixed list menu 及 Fixed icontext menuitem。

菜单框架

我们先需要创建一个列表菜单框架：

```
fixed_list_menu My_fixed_list_menu;        //列表菜单框架
void mmi_myapp_entry(void)
{
    ……
    gui_move_text_cursor(x, y);
    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO));

    //开始显示菜单
    memset(&My_fixed_list_menu, 0, sizeof(fixed_list_menu));
    gui_create_fixed_list_menu(&My_fixed_list_menu, 20, MMI_content_y + 5, 136, MMI_content_height - 50);
    MMI_current_menu_type = LIST_MENU;
    //显示菜单结束

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

代码 15.1

gui_create_fixed_list_menu 用来初始菜单框架的一些基本属性。MMI_current_menu_type 是一个全局标志，用来标志当前菜单显示风格(虽然有点多此一举，但代码中一定要有，否则会显示不正常)。

菜单框架显示效果如下：

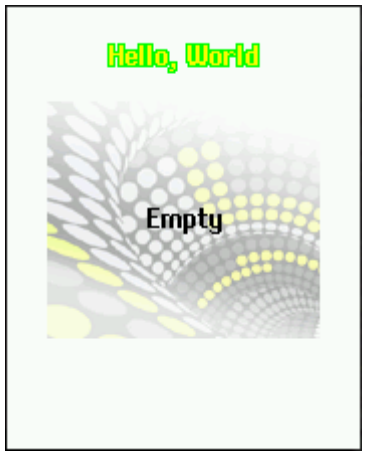


图 15.1

菜单项公共属性

第二步是创建菜单项，菜单项有两种属性，一是公共属性，我们把所有菜单项属性中值完全相同的属性都合在一起，如每项高宽等，这样即方便控制又节省空间。二是单项属性(每项都不同)，如菜单项文本图标等。

```
……
fixed_icontext_menuitem My_fixed_icontext_menuitem_common;          //菜单项公共属性
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    memset(&My_fixed_icontext_menuitem_common, 0, sizeof(fixed_icontext_menuitem));
    gui_create_fixed_icontext_menuitem(&My_fixed_icontext_menuitem_common, 136, 18);                    //宽高
    gui_fixed_icontext_menuitem_set_text_position( &My_fixed_icontext_menuitem_common, 24, 0);      //文本偏移距离
    My_fixed_icontext_menuitem_common.flags |=  UI_MENUITEM_DISABLE_BACKGROUND;       //统一标志符
    //显示菜单结束
    ……
}
```

代码 15.2

菜单项单项属性

下面初始化每个菜单项的单项属性。

在初始化之前我们先加几个文本串资源，添加方法可参考第五章“字串资源”：

```
typedef enum
{
    STR_MYAPP_HELLO = MYAPP_BASE + 1,
    STR_MYAPP_HELLO_MTK,
    STR_MYAPP_HELLO_TIBET,
    STR_MYAPP_HELLO_LHASA,
    STR_MYAPP_HELLO_SINKIANG,
```

```
    STR_MYAPP_HELLO_MONGOLIA,
    STR_MYAPP_HELLO_SIAN,
    STR_MYAPP_HELLO_CHENGTU,
} STRINGID_LIST_MYAPP;
```

代码 15.3

文本资源如下图：

STR_MYAPP_HELLO	Undefined	11	Hello, World	Hello, World	你好,世界
STR_MYAPP_HELLO_MTK	Undefined	11	Hello, MTK	Hello, MTK	你好,联发
STR_MYAPP_HELLO_TIBET	Undefined	11	Hello, Tibet	Hello, Tibet	你好,西藏
STR_MYAPP_HELLO_LHASA	Undefined	11	Hello, Lhasa	Hello, Lhasa	你好,拉萨
STR_MYAPP_HELLO_SINKIANG	Undefined	15	Hello, Sinkiang	Hello, Sinkiang	你好,新疆
STR_MYAPP_HELLO_MONGOLIA	Undefined	15	Hello, Mongolia	Hello, Mongolia	你好,内蒙古
STR_MYAPP_HELLO_SIAN	Undefined	11	Hello, Sian	Hello, Sian	你好,西安
STR_MYAPP_HELLO_CHENGTU	Undefined	15	Hello, Chengtu	Hello, Chengtu	你好,成都

图 15.2

菜单项创建方法如下：

```
……
#define My_fixed_list_n_items   (8)                                          //菜单项项数
fixed_icontext_menuitem_type My_fixed_list_menuitems[My_fixed_list_n_items];  //icontext 型菜单项列表
void *My_fixed_menuitem_pointers[My_fixed_list_n_items];                      //指向菜单项列表的索引列表
void mmi_myapp_entry(void)
{
    S32 i;
    ……
    //开始显示菜单
    ……
    memset(&My_fixed_list_menuitems, 0, sizeof(fixed_icontext_menuitem_type) * My_fixed_list_n_items);
    for (i = 0; i < My_fixed_list_n_items; i++)
    {
        My_fixed_list_menuitems[i].item_text = (UI_string_type)GetString(STR_MYAPP_HELLO + i);    //菜单项文本
        My_fixed_list_menuitems[i].item_icon = (PU8) GetImage(IMG_GLOBAL_L1 + i);                 //菜单项图标
        My_fixed_list_menuitems[i].flags = (UI_MENUITEM_CENTER_TEXT_Y | I_MENUITEM_CENTER_ICON_Y);
        My_fixed_list_menuitems[i].item_icon_handle = GDI_ERROR_HANDLE;
        My_fixed_menuitem_pointers[i] = (void*)&My_fixed_list_menuitems[i];                       //给索引列表赋值
    }
    //显示菜单结束
    ……
}
```

代码 15.4

My_fixed_menuitem_pointers 是指向菜单项中每一项的索引列表，主要用来通知菜单框架每一个菜单项的数据地址。

联合菜单框架与菜单项

联合就是将菜单项的相关属性传递给菜单框架，要传递的属性有：菜单项索引列表，菜单项公共属性，菜单项项数，菜单项功能接口等。

```
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    My_fixed_list_menu.items = My_fixed_menuitem_pointers;                          // 菜单项索引列表
    My_fixed_list_menu.common_item_data =  (void*)&My_fixed_icontext_menuitem_common;   // 菜单项公共属性
    My_fixed_list_menu.n_items = My_fixed_list_n_items;                             // 菜单项项数
    gui_set_fixed_list_menu_item_functions(&My_fixed_list_menu,                     // 菜单项功能接口
        gui_show_fixed_icontext_menuitem,                   //显示菜单项函数
        gui_measure_fixed_icontext_menuitem,                //测量菜单项函数
        gui_highlight_fixed_icontext_menuitem,              //高亮函数
        gui_remove_highlight_fixed_icontext_menuitem,       //失去高亮函数
        gui_hide_fixed_icontext_menuitem,                   //菜单项隐藏函数
        NULL                                                //菜单项重设大小函数
    );
    //显示菜单结束
    ……
}
```

代码 15.5

菜单框架是依靠传递过来的功能接口控制所有菜单项的，当要控制某一项的时候，就把菜单项索引值传入相应的功能接口，也就是说菜单框架只认接口不认菜单项。因每种类型的菜单项都有自己的一套接口，所以不管什么类型的菜单项，只要照上面这样联合起来，菜单框架都能控制。

显示菜单

最后是显示菜单：

```
S32 My_fixed_list_highlight_item = 0;           //菜单高亮项索引
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    My_fixed_list_menu.highlighted_item = -1; //菜单高亮项索引，此为菜单框架内部属性值，在跳转之前一定要设为-1
    gui_fixed_list_menu_goto_item(&My_fixed_list_menu, My_fixed_list_highlight_item);
    gui_show_fixed_list_menu(&My_fixed_list_menu);
    //显示菜单结束
    ……
}
```

代码 15.6

在显示之前要先用 gui_fixed_list_menu_goto_item 设置好菜单的高亮项，否则菜单会显示不正常。

当菜单项文本过长，高亮时其会自动滚动，所以我们退出菜单后还要手动通知所有菜单项停止滚动，实现如下：

```
void mmi_myapp_exit(void)
{
    gui_fixed_icontext_menuitem_stop_scroll();
}
```

代码 15.7

最终显示结果为：

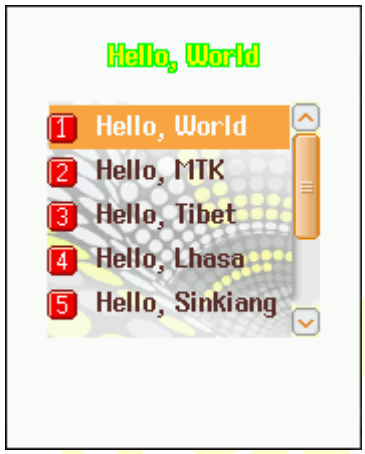


图 15.3

现在菜单还只能看不能操作，下一章我们将为其加上键盘操作。

第十六章：键盘

键盘介绍

手机键盘虽然模样多变，但其按键功能基本上固定，下面是稍微标准一点的手机键盘图：

图 16.1

系统内部会给每种功能的按键定一个代号，不管键盘外观及布局如何改，只要按键功能不变，其按键代码就不必改变。如果有新功能按键出现，我们就为其增加一个新的代码。

下面是针对图 16.1 的按键代码图：

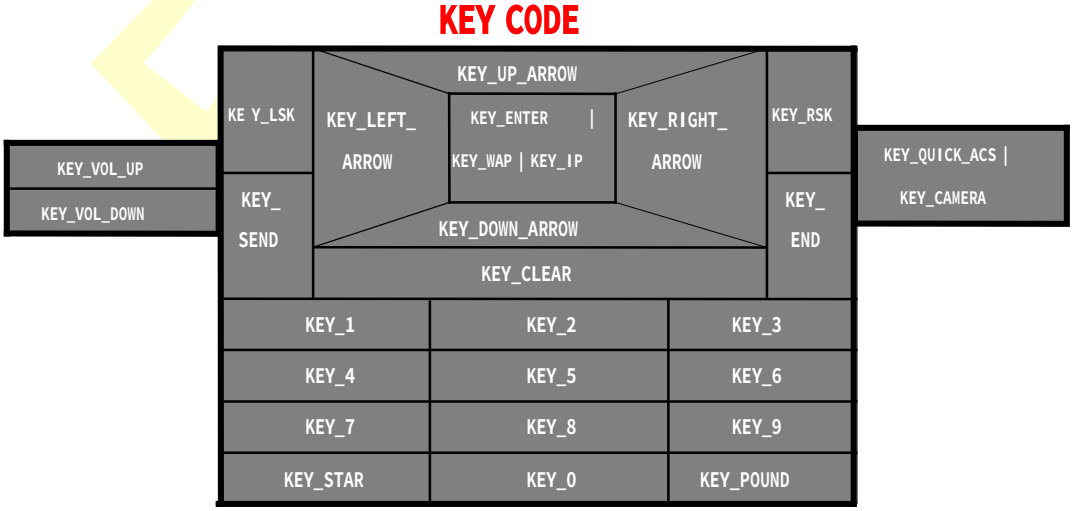


图 16.2

下面是我们对每个功能按键的描述：

按键名	描述	KEY CODE
数字键 0—9	最原始的数字键盘，但其功能还可由程序自行定义。	KEY_0 — KEY_9

拨号键	主要有两个用途：一是打电话时用来拨号，二是 Idle 时按此键进入通话记录列表。	KEY_SEND
终止键	终止键主要有三个作用： 一、打电话时用来挂断。 二、绝大部份程序用此键来返回 Idle. 三、长按此键开关机。	KEY_END
星号键	与固话的星号键作用类似。	KEY_STAR
井号键	与固话的井号键作用类似。	KEY_POUND
左右中软键	与画面联系最紧密的功能按键，一般左软键用来进入，右软键用来退出，中软键一般只在触摸屏中会用到，如下图在拨号界面中，中软键就是中间的小图标，只要点击就会拨号出去： 	KEY_LSK KEY_RSK KEY_CSK
上下左右方向键	方向按键	KEY_UP_ARROW KEY_DOWN_ARROW KEY_LEFT_ARROW KEY_RIGHT_ARROW
确认键	大部份情况下可当左软键用，在 Idle 时按此键可进入 WAP 游览器，在拨号介面按此键可拨 IP 电话。	KEY_ENTER KEY_WAP KEY_IP
清除键	编辑介面时此键用来删除字符(相当于 PC 键盘的退格键)。	KEY_CLEAR
下下音量键	用来调节音量，一般放在手机左侧面。	KEY_VOL_UP KEY_VOL_DOWN
拍照键	用来拍照，一般放在手机右侧面。	KEY_CAMERA KEY_QUICK_ACS

图 16.3

按键的操作方式有很多种，其分别描述如下：

操作方式	描述	KEY TYPE
按下	将按键按下。	KEY_EVENT_DOWN KEY_FULL_PRESS_DOWN
放开	放开按键。	KEY_EVENT_UP
长按	按下按键后不动，一定时间到后将会触发长按事件，一般情况下是 1.5 秒后，可参考 KPD_LONGPRESS_PERIOD。	KEY_LONG_PRESS
重复	长按事件触发以后，系统将会以一定频率不断触发“重复”事件，一般是间隔 0.5 秒，可参考 KPD_REPEAT_PERIOD。 注：长按事件只会触发一次，重复事件将会一直持续下去，直到放开按键。	KEY_REPEAT
半按	将按键按下一半，只有特殊的按键支持此事件(比如经过特殊处理的拍照键)，目前很少用到。	KEY_HALF_PRESS_DOWN
半按放开	半按后放开按键。	KEY_HALF_PRESS_UP

图 16.4

为某个按键设置响应函数我们一般用 SetKeyHandler，其第一个参数是响应函数的地址，第二个参数是按键代码，第三个参数是按键操作方式。

代码准备

为使演示更直观，我们为菜单加上一项功能：当高亮菜单某一项里，上方主文本区将显示此项的文本串。
首先得将主文本显示代码提取出来以方便文本串刷新：

```
void mmi_myapp_draw_text(S32 index)      //显示主文本区函数，index 为菜单高亮项的序号
{
    S32 x, y, w, h;
    color text_color = {255, 255, 0, 100};

    gui_lock_double_buffer();

    gui_reset_clip();
    gui_set_text_color(text_color);
    gui_set_text_border_color(UI_COLOR_GREEN);
    gui_measure_string((UI_string_type)GetString(STR_MYAPP_HELLO + index), &w, &h);
    x = (UI_device_width - w) / 2;
    y = MMI_title_y;
    gui_move_text_cursor(x, y);

    gui_fill_rectangle(0, y, UI_device_width - 1, y + h, UI_COLOR_WHITE);
    gui_print_bordered_text((UI_string_type)GetString(STR_MYAPP_HELLO + index));

    gui_unlock_double_buffer();
    gui_BLT_double_buffer(0, y, UI_device_width - 1, y + h);
}

void mmi_myapp_highlight_handler(S32 item_index)        //菜单项高亮时被系统调用的回调函数
{
    mmi_myapp_draw_text(item_index);
}

void mmi_myapp_entry(void)
{
    ……
    mmi_myapp_draw_text(0);
    //开始显示菜单
    ……
    My_fixed_list_menu.item_highlighted = mmi_myapp_highlight_handler;
    gui_show_fixed_list_menu(&My_fixed_list_menu);
    //显示菜单结束
    ……
}
```

代码 16.1

这里的 My_fixed_list_menu.item_highlighted 与上一章提到的菜单项功能接口中的高亮接口不是同一个概念，菜单项功能接口是由 GUI 内部提供的，不能更改。这里的 item_highlighted 是由应用层提供的。当菜单项高亮时菜单框架会依次呼叫这两个函数。

按键操作

下面我们为上下方向键加上按键响应：

```
void my_fixed_list_goto_previous_item(void)     //跳到菜单的前一项
{
    gui_lock_double_buffer();
    gui_fixed_list_menu_goto_previous_item(&My_fixed_list_menu);    //菜单框架将跳到前一项
    gui_show_fixed_list_menu(&My_fixed_list_menu);
    gui_unlock_double_buffer();
    gui_BLT_double_buffer( My_fixed_list_menu.x, My_fixed_list_menu.y,
        My_fixed_list_menu.x + My_fixed_list_menu.width, My_fixed_list_menu.y + My_fixed_list_menu.height);
}

void my_fixed_list_goto_next_item(void)          //跳到菜单的下一项
{
    gui_lock_double_buffer();
    gui_fixed_list_menu_goto_next_item(&My_fixed_list_menu);         //菜单框架将跳到下一项
    gui_show_fixed_list_menu(&My_fixed_list_menu);
    gui_unlock_double_buffer();
    gui_BLT_double_buffer( My_fixed_list_menu.x, My_fixed_list_menu.y,
        My_fixed_list_menu.x + My_fixed_list_menu.width, My_fixed_list_menu.y + My_fixed_list_menu.height);
}

void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    //显示菜单结束
    SetKeyHandler(my_fixed_list_goto_previous_item, KEY_UP_ARROW, KEY_EVENT_DOWN);
    SetKeyHandler(my_fixed_list_goto_next_item, KEY_DOWN_ARROW, KEY_EVENT_DOWN);
    ……
}
```

代码 16.2

运行后，当我们按“下方向键”时菜单将会跳到第二项：

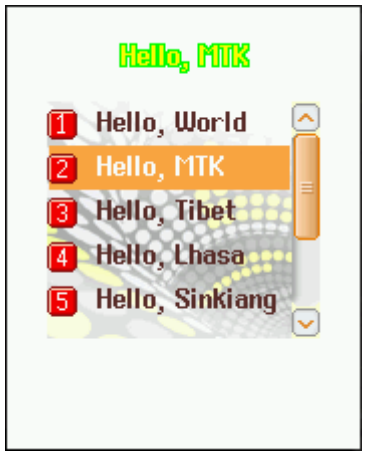


图 16.5

群组按键

接下来我们为每个数字键加上按键响应。

因数字键数量多，如果我们为每一个按键都注册一次，代码将会很冗余，所以我们将改用接口 SetGroupKeyHandler。其第一个参数是按键响应函数地址，第二个参数是按键代码列表，第三个参数是注册按键数量，第四个参数是按键操作方式。

```
void my_fixed_list_goto_item(void)
{
    U16 keycode, keytype;
    S32 index = 0;

    gui_lock_double_buffer();
    GetkeyInfo(&keycode, &keytype);      //获取当前用户操作的按键代码
    index = keycode - KEY_1;             //计算出当前按的是哪一个数字
    gui_fixed_list_menu_goto_item(&My_fixed_list_menu, index);
    gui_show_fixed_list_menu(&My_fixed_list_menu);
    gui_unlock_double_buffer();
    gui_BLT_double_buffer(My_fixed_list_menu.x, My_fixed_list_menu.y,
        My_fixed_list_menu.x + My_fixed_list_menu.width,
        My_fixed_list_menu.y + My_fixed_list_menu.height);
}

void mmi_myapp_entry(void)
{
    U16 shortcut_keys[My_fixed_list_n_items] =
    {
        KEY_1, KEY_2, KEY_3, KEY_4,
        KEY_5, KEY_6, KEY_7, KEY_8
    };
    ……
    //开始显示菜单
    ……
```

```
    //显示菜单结束
    SetGroupKeyHandler(my_fixed_list_goto_item, (PU16)shortcut_keys, My_fixed_list_n_items, KEY_EVENT_UP);
    ……
}
```

代码 16.3

运行后当按下数字键“8”时，菜单会立即跳第八项：

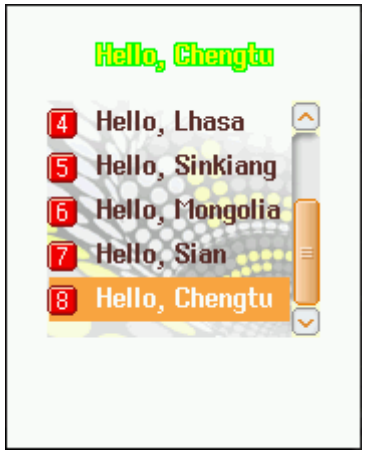


图 16.6

下一章我们将继续为菜单加上触摸屏响应。

第十七章：触摸屏

触摸屏介绍

触摸屏操作简单、自然、方便，更贴近画面显示，也更符合用户的使用习惯。

与电脑中的鼠标相比，触摸屏在定位方式与操作方式上均有很大的不同：

定位方式：鼠标是一种相对定位系统，每次操作的位置和前一次位置都有关联。而触摸屏是一种绝对定位系统，每一次操作的位置与上一次没有任何关系，想要选哪就直接点哪。

操作方式：鼠标的操作方式很不确定，不同的鼠标使用起来也不同(比如单键，双键，三键，滚轮等等)。而触摸屏的操作方式相对统一，基本操作不外乎“点下”、“移动”、“放开”三种，其它的操作方式都是此三种衍变而来。

简要的说，鼠标是定位辅以操作，触摸屏则是操作辅以定位。

从程序的角度来看，我们平台的触摸屏操作方式分为三级：硬件级、驱动级、MMI 级：

硬件级：由触摸屏硬件产生，大部分情况下硬件只会产生一种操作方式，即触摸笔点放，至于具体是点是放由驱动自己判断。

驱动级：驱动收到硬件通知后，先依据前一次操作以确定当前是点下还是放开(点下后一定是放开，放开后一定是点下)，若是点下则启动定时器，然后以一定频率产生另外的操作如“移动”、“长按”、“重复”等等。

每生成一个新的操作，驱动都会立即从硬件读取触摸笔的当前坐标，然后将操作方式与触摸笔坐标一起保存起来，并通知 MMI 尽快处理。

MMI 级：基本上与驱动级操作一一对应。

驱动级操作方式

接到硬件通知后，驱动会先判断是点下还是放开，下面是接下来的处理流程：

点下：如果是点下则先判断是否处在手写模式，如果不是则看当前点下的坐标是否在手写启动区域，一般在编辑介面中我们会将某一块区域设为手写启动区域。如下图，中间的红色框框住的区域就是手写启动区域：

图 17.1

只要点在启动区域，我们就会开启手写模式，接下来就按不同模式不同处理：

非手写模式：先生成一个触摸笔点下的操作(PEN_DOWN)，然后启动两个定时器：

触摸笔移动定时器：一般以 80 毫秒(参考 MMI_PEN_SAMPLING_PERIOD_1)为一周期不停的检测触摸笔的位置，如果位置发生变化则产生一个移动操作(PEN_MOVE)。此定时器直到触摸笔放开后自行停止。

触摸笔长按定时器：一般定时 600 毫秒后(参考 MMI_PEN_LONGTAP_TIME)触摸笔还按着不动则产生一个长按操作(PEN_LONGTAP)。注意在这期间不能有 PEN_MOVE 发生，否则定时器会自行中止。PEN_LONGTAP 产生后会接着启动另一个定时器：

触摸笔重复定时器：以 300 毫秒(参考 MMI_PEN_LONGTAP_TIME)为周期不停的产生重复操作(PEN_REPEAT)，跟长按一样如有 PEN_MOVE 发生则定时器自行中止。

手写模式：先判断当前是否点在手写扩展区域，一般情况下手写扩展区域都设为全屏，只有点在手写扩展区域才会以手写模式继续处理，否则将立即转为非手写模式走上面讲的流程。

当点击是在手写扩展区域时，则生成一个手写点下的操作(STROKE_DOWN)，然后启动两个定时器：

手写移动定时器：一般以 20 毫秒(参考 MMI_PEN_SAMPLING_PERIOD_2)为一周期不停的检测触摸笔的位置，如果位置发生变化则产生一个手写移动操作(STROKE_MOVE)。此定时器直到触摸笔放开后自行停止。

手写长按定时器：一般定时 800 毫秒后(参考 MMI_PEN_STROKE_LONGTAP_TIME)触摸笔还按着不动则产生一个手写长按操作(STROKE_LONGTAP)。在这期间不能有 STROKE_MOVE 发生，否则定时器会自行中止。

放开：先会中止所有定时器，然后按当前是否为手写模式产生不同的操作：

非手写模式：生成一个触摸笔放开的操作(PEN_UP)。

手写模式：生成一个手写放开的操作(STROKE_UP)。

从上面的流程可以看出，手写与非手写最大的区别就是生成移动操作的频率不同，这样做主要是为了减轻系统负担。

下面再游览一下我们上面讲的流程：

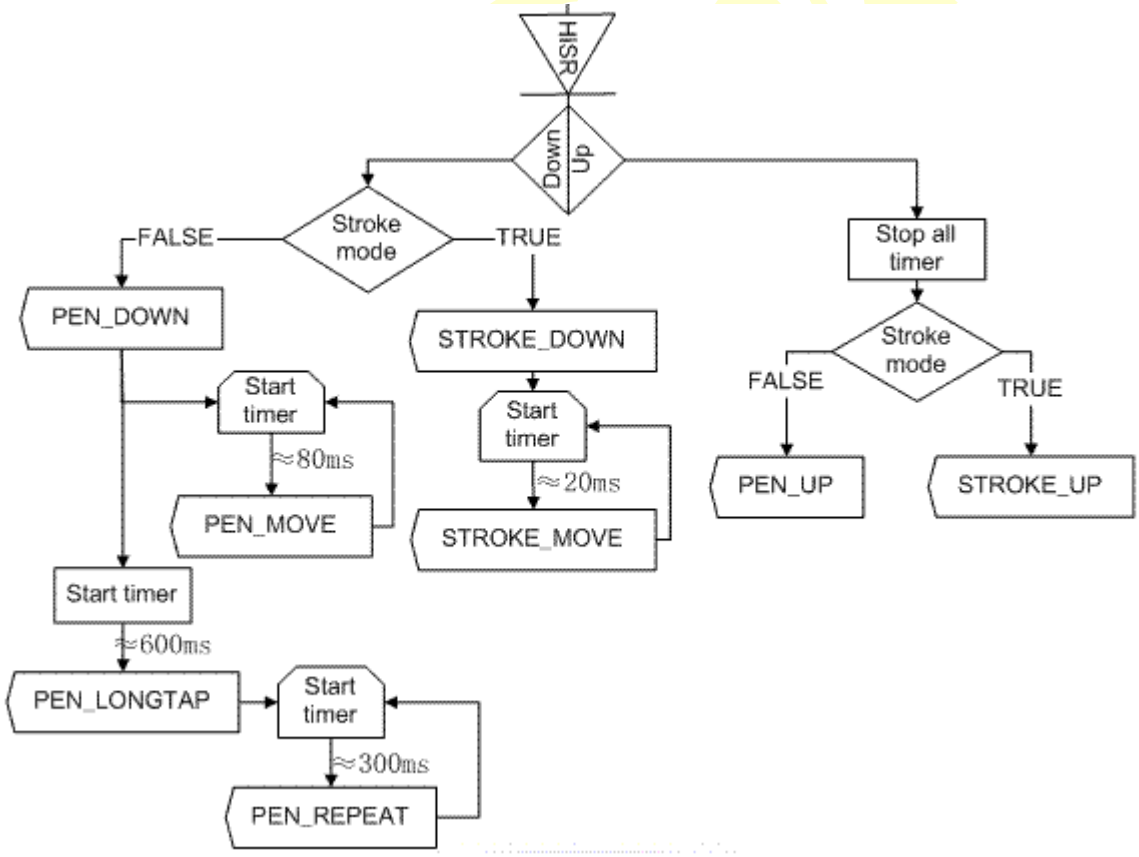


图 17.2

另外驱动还有两个不常用的操作（这两个操作 MMI 应用层可以不作考虑）：

PEN_ABORT：中止操作，驱动会先将用户操作缓存下来，留待 MMI 空闲的时候再处理，但如果 MMI 长期不处理则会导致缓存空间爆满，此时驱动就会产生一个 PEN_ABORT 通知 MMI。

TP_UNKNOWN_EVENT：未知操作，MMI 如收到此操作则表示系统出现异常了。

MMI 操作方式

下面是驱动操作方式与 MMI 操作方式的转换流程图：

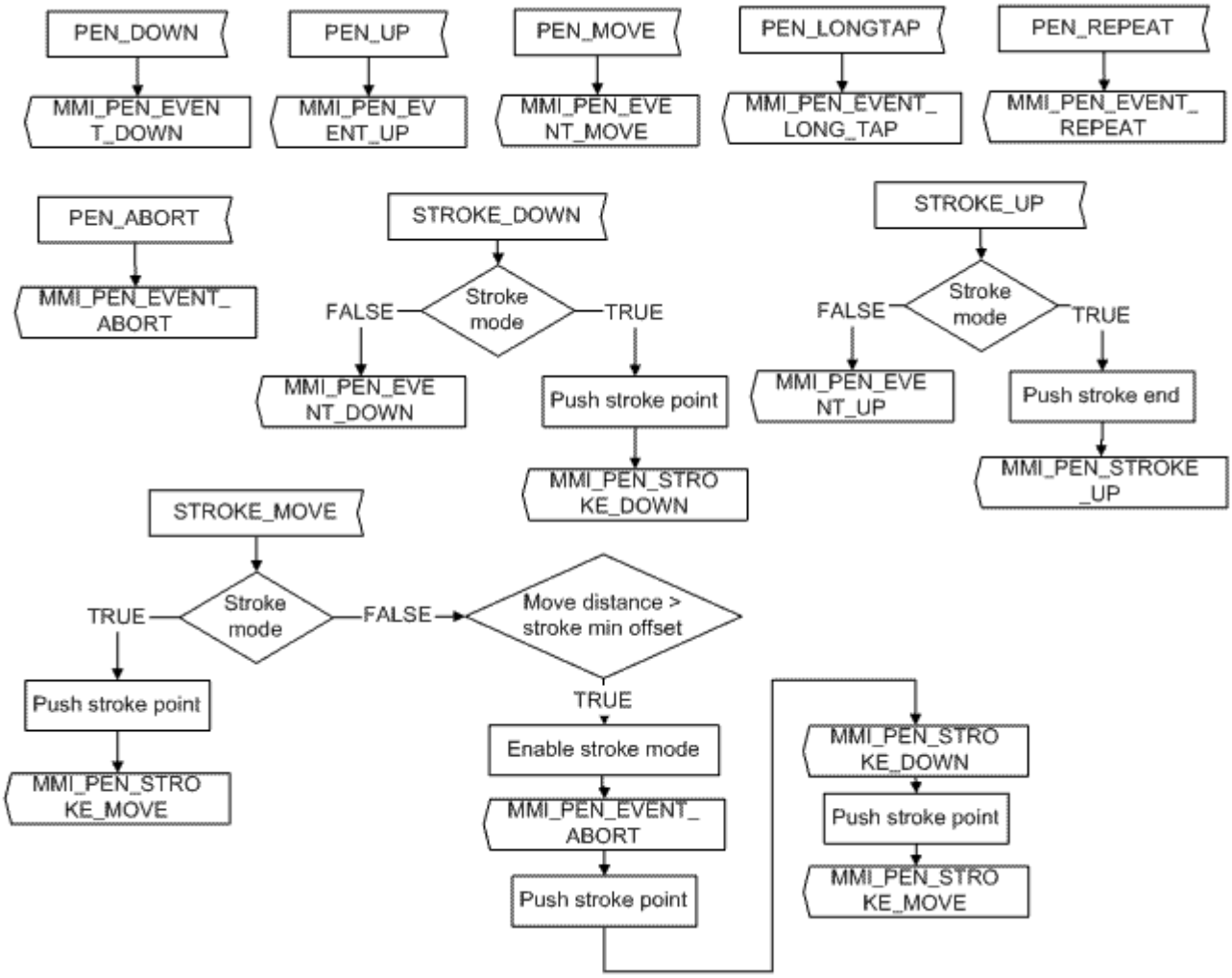


图 17.3

从上面流程图可以看出，非手写模式的转换是完全一一对应的，手写模式则要进一步处理，而这处理的关键是“MMI 手写模式”。“MMI 手写模式”与“驱动手写模式”是不同的概念。当第一次 STROKE_DOWN 产生后，只有 STROKE_MOVE 一定的距离，我们才说进入“MMI 手写模式”。如果没进入 MMI 手写模式，我们会将 STROKE_DOWN、STROKE_MOVE 及 STROKE_UP 一一转换常规操作 PEN_DOWN、PEN_MOVE 及 PEN_UP，这样设计主要是考虑到在写字的时候用户还有可能对文本区进行一些常规操作(如移动光标等)。

菜单的触摸屏操作

下面我们为菜单加上触摸屏操作。

菜单有自己的触摸屏处理功能，我们只需把用户操作传递给菜单即可：

```
//此函数将用户操作传递给菜单
void mmi_myap_pen_list_menu_hdlr(
    mmi_pen_point_struct point,              //触摸笔坐标
    mmi_pen_event_type_enum pen_event        //操作方式
```

```
)
{
    gui_list_pen_enum menu_event;

    gui_lock_double_buffer();
    if (!gui_fixed_list_menu_translate_pen_event(&My_fixed_list_menu, pen_event, point.x, point.y, &menu_event))
    {
        return;    //如果触摸笔不在菜单显示区域内则不做处理
    }

    if (menu_event == GUI_LIST_PEN_HIGHLIGHT_CHANGED ||
        menu_event == GUI_LIST_PEN_NEED_REDRAW || menu_event == GUI_LIST_PEN_ITEM_SELECTED)
    {
        //如果菜单处理后使菜单显示内容有了变化，我们需手动重画菜单
        gui_show_fixed_list_menu(&My_fixed_list_menu);
    }
    gui_unlock_double_buffer();
    gui_BLT_double_buffer( My_fixed_list_menu.x, My_fixed_list_menu.y,
        My_fixed_list_menu.x + My_fixed_list_menu.width, My_fixed_list_menu.y + My_fixed_list_menu.height);
}
```

代码 17.1

用户操作需要手动注册响应函数：

```
……
void mmi_myapp_pen_up_hdlr(mmi_pen_point_struct point)
{
    mmi_myap_pen_list_menu_hdlr(point, MMI_PEN_EVENT_UP);
}
void mmi_myapp_pen_down_hdlr(mmi_pen_point_struct point)
{
    mmi_myap_pen_list_menu_hdlr(point, MMI_PEN_EVENT_DOWN);
}
void mmi_myapp_pen_move_hdlr(mmi_pen_point_struct point)
{
    mmi_myap_pen_list_menu_hdlr(point, MMI_PEN_EVENT_MOVE);
}
void mmi_myapp_pen_repeat_hdlr(mmi_pen_point_struct point)
{
    mmi_myap_pen_list_menu_hdlr(point, MMI_PEN_EVENT_REPEAT);
}
void mmi_myapp_pen_long_tap_hdlr(mmi_pen_point_struct point)
{
    mmi_myap_pen_list_menu_hdlr(point, MMI_PEN_EVENT_LONG_TAP);
}

void mmi_myapp_entry(void)
```

```
{
    ……
    //开始显示菜单
    ……
    //显示菜单结束
    mmi_pen_register_down_handler(mmi_myapp_pen_down_hdlr);      //注册“点下”操作响应函数
    mmi_pen_register_up_handler(mmi_myapp_pen_up_hdlr);          //注册“放开”操作响应函数
    mmi_pen_register_move_handler(mmi_myapp_pen_move_hdlr);      //注册“移动”操作响应函数
    mmi_pen_register_repeat_handler(mmi_myapp_pen_repeat_hdlr);  //注册“重复”操作响应函数
    mmi_pen_register_long_tap_handler(mmi_myapp_pen_long_tap_hdlr);// 注册“长按”操作响应函数
    ……
}
```

代码 17.2

另外手写模式的注册函数如下：

mmi_pen_register_stroke_down_handler： 手写点下

mmi_pen_register_stroke_move_handler： 手写移动

mmi_pen_register_stroke_up_handler： 手写放开

mmi_pen_register_stroke_longtap_handler： 手写长按

运行后，用触摸笔将滚动条拖到中间，然后点击第三项，显示结果如下：

图 17.4

第十八章：高级控件

WGUI Control

高级控件，也即 WGUI Control。
与 GUI Control 不同，每种高级控件在系统启动的时候即已创建一份实体，我们在用的时候只需将这份实体重新初始化并显示即可。
下面我们将第十五章所讲的 GUI 菜单改成 WGUI 菜单。

初始化菜单框架

初始化方式与 GUI 菜单类似：

```
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    move_fixed_list(20, MMI_content_y + 5);          //设置位置
    resize_fixed_list(136, MMI_content_height - 40); //设置宽高
    MMI_current_menu_type = LIST_MENU;               //设置显示风格
    disable_menu_shortcut_box_display = 1;           //关掉标题条右边的快捷序号显示框
    //显示菜单结束
    ……
}
```

代码 18.1

要记得把快捷序号显示框关掉，WGUI 菜单框架默认联合了快捷序号显示，如不手动关掉每次高亮操作其会自动显示出来。

初始化菜单项公共属性

函数 create_fixed_icontext_menuitems 用来初始化菜单项公共属性：

```
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    create_fixed_icontext_menuitems();
    //显示菜单结束
    ……
}
```

代码 18.2

初始化菜单项单项数据

```
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    for (i = 0; i < My_fixed_list_n_items; i++)
    {
        add_fixed_icontext_item(get_string(STR_MYAPP_HELLO + i), (PU8)GetImage(IMG_GLOBAL_L1 + i));
    }
    //显示菜单结束
    ……
}
```

代码 18.3

联合菜单框架与菜单项

```
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    associate_fixed_icontext_list();
    //显示菜单结束
    ……
}
```

代码 18.4

注册按键

```
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    register_fixed_list_highlight_handler(mmi_myapp_highlight_handler);    //注册高亮接口
    register_fixed_list_keys();                                            //注册方向键
    register_fixed_list_shortcut_handler();                                //注册数字键
    //显示菜单结束
    ……
}
```

代码 18.5

显示菜单

```
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    fixed_list_goto_item_no_redraw(0);          //先高亮第一项
    show_fixed_list();                          //显示菜单
    //显示菜单结束
    ……
}
```

代码 18.6

显示效果跟第十五章一样：

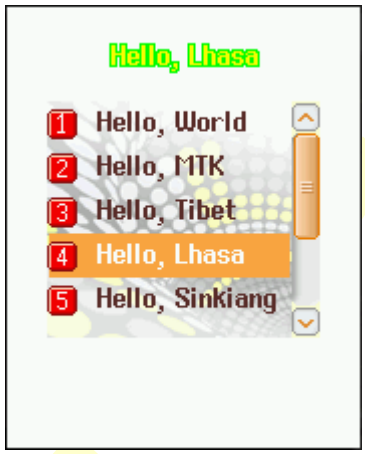


图 18.1

其它高级控件

其它高级控件的使用方式均类似，下面再举一个左右软键的例子：

```
void mmi_myapp_entry(void)
{
    ……
    //开始显示菜单
    ……
    //显示菜单结束

    //开始显示左右软键
    change_left_softkey(STR_GLOBAL_OK, IMG_GLOBAL_OK);          //设置左软键文本及图标
    change_right_softkey(STR_GLOBAL_BACK, IMG_GLOBAL_BACK);     //设置右软键文本及图标
    show_softkey(MMI_LEFT_SOFTKEY);                             //显示左软键
    show_softkey(MMI_RIGHT_SOFTKEY);                            //显示右软键
    SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);       //注册右软键响应函数
    //显示左右软键结束
```

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

代码 18.7

左右软键显示后就不能用 SetKeyHandler 来注册按键响应了，得改用 SetLeftSoftkeyFunction 与 SetRightSoftkeyFunction。显示结果如下：

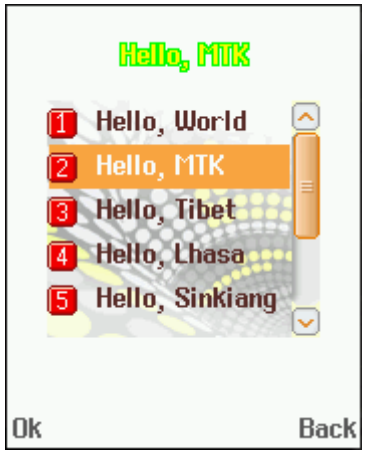


图 18.2

第十九章：屏幕模板

介绍

我们系统中的屏幕模板接口都类似于 ShowCategoryXXXScreen，这里的 XXX 是数字。数字没有任何含义，只是个序号而已，因为我们平台早期规划得不是很好，加上屏幕模板很难命名，所以才出现这样命名不太合理的情况。

屏幕模板可以简单的分为两大类，一类是通用性模板，这些模板都是可重复使用的。二是非通用性模板，都是某些特殊程序的专用模板，从某种意义上来说这些可以不叫做模板，只是为方便代码统一管理才放到模板中来。

新模板

我们将建一个新的模板，暂时命名为 ShowCategory888Screen，下面先将我们程序的所有功能都移到模板中来：

```
void ShowCategory888Screen()
{
    S32 i;

    //初始化屏幕
    EntryNewScreen(SCR_MYAPP_MAIN, mmi_myapp_exit, NULL, NULL);
    gui_lock_double_buffer();
    entry_full_screen();
    clear_screen();

    //初始化菜单框架
    move_fixed_list(20, MMI_content_y + 5);
    resize_fixed_list(136, MMI_content_height - 40);
    MMI_current_menu_type = LIST_MENU;
    disable_menu_shortcut_box_display = 1;

    //初始化菜单项公用数据
    create_fixed_icontext_menuitems();

    //联合菜单框架与菜单项
    associate_fixed_icontext_list();

    //初始化菜单项单项数据
    for (i = 0; i < My_fixed_list_n_items; i++)
    {
        add_fixed_icontext_item(get_string(STR_MYAPP_HELLO + i), (PU8)GetImage(IMG_GLOBAL_L1 + i));
    }

    //注册按键
    register_fixed_list_shortcut_handler();
```

```
    register_fixed_list_keys();
    register_fixed_list_highlight_handler(mmi_myapp_highlight_handler);
    //显示菜单
    fixed_list_goto_item_no_redraw(0);
    show_fixed_list();

    //显示左右软键
    change_left_softkey(STR_GLOBAL_OK, IMG_GLOBAL_OK);
    change_right_softkey(STR_GLOBAL_BACK, IMG_GLOBAL_BACK);
    show_softkey(MMI_LEFT_SOFTKEY);
    show_softkey(MMI_RIGHT_SOFTKEY);
    SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);

    gui_unlock_double_buffer();
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}

void mmi_myapp_entry(void)
{
    ShowCategory888Screen();
}
```

代码 19.1

当然，这样的模板别人是无法使用的，接下来我们将经常会被改动的元素提取出来：

可重用的模板

常见的变化元素有：左右软键的文本图标，菜单项的内容，菜单项的初始高亮项序号，以及按键的响应函数等。

```
//将 mmi_myapp_highlight_handler 改为 cat888_highlight_handler，并将 mmi_myapp_draw_text 中内容并入其中
void cat888_highlight_handler(S32 index)
{
    ……
    gui_measure_string(MMI_fixed_icontext_menuitems[index].item_text, &w, &h);
    ……
    gui_print_bordered_text(MMI_fixed_icontext_menuitems[index].item_text);
    gdi_layer_unlock_frame_buffer();
    gui_BLT_double_buffer(0, y, UI_device_width - 1, y + h);
}

void ShowCategory888Screen(
    U16 left_softkey, U16 left_softkey_icon,     //左软键文本及图标
    U16 right_softkey, U16 right_softkey_icon,   //右软键文本及图标
    S32 number_of_items,                         //菜单项项数
    U16 *list_of_items,                          //菜单项文本列表
    U16 *list_of_icons,                          //菜单项图标列表
```

```
    S32 highlighted_item)                    //初始菜单高亮项序号
{
    ……

    //初始化菜单项单项数据
    for (i = 0; i < number_of_items; i++)
    {
        add_fixed_icontext_item(get_string(list_of_items[i]), (PU8)GetImage(list_of_icons[i]));
    }

    //注册按键
    register_fixed_list_shortcut_handler();
    register_fixed_list_keys();
    register_fixed_list_highlight_handler(cat888_highlight_handler);

    //显示菜单
    fixed_list_goto_item_no_redraw(highlighted_item);
    show_fixed_list();

    //显示左右软键
    change_left_softkey(left_softkey, left_softkey_icon);
    change_right_softkey(right_softkey, right_softkey_icon);
    show_softkey(MMI_LEFT_SOFTKEY);
    show_softkey(MMI_RIGHT_SOFTKEY);

    ……
}

void mmi_myapp_entry(void)
{
    S32 i;
    U16 list_of_items[My_fixed_list_n_items];
    U16 list_of_icons[My_fixed_list_n_items];

    EntryNewScreen(SCR_MYAPP_MAIN, mmi_myapp_exit, NULL, NULL);

    for (i = 0; i < My_fixed_list_n_items; i++)
    {
        list_of_items[i] = STR_MYAPP_HELLO + i;
        list_of_icons[i] = IMG_GLOBAL_L1 + i;
    }
    ShowCategory888Screen(
        STR_GLOBAL_OK, IMG_GLOBAL_OK,
        STR_GLOBAL_BACK, IMG_GLOBAL_BACK,
        My_fixed_list_n_items,
        list_of_items,
```

```
            list_of_icons,
            0);
    SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
}
```

代码 19.2

提出绘画部份

我们通常会将模板中真正的绘画部份提取出来放到一个单独的接口中，这样方便在显示的时候临时刷新整个屏幕(ShowCategory888Screen 中就只放模板初始化部份)：

```
void RedrawCategory888Screen(void)
{
    gdi_layer_lock_frame_buffer();

    //显示菜单
    show_fixed_list();

    //显示左右软键
    show_softkey(MMI_LEFT_SOFTKEY);
    show_softkey(MMI_RIGHT_SOFTKEY);

    gdi_layer_unlock_frame_buffer();
    gui_BLT_double_buffer(0, 0, UI_device_width - 1, UI_device_height - 1);
}

void ShowCategory888Screen(
    U16 left_softkey, U16 left_softkey_icon, U16 right_softkey, U16 right_softkey_icon,
    S32 number_of_items, U16 *list_of_items, U16 *list_of_icons,
    S32 highlighted_item)
{
    ……
    //初始化菜单
    fixed_list_goto_item_no_redraw(highlighted_item);
    //show_fixed_list();

    //初始化左右软键
    change_left_softkey(left_softkey, left_softkey_icon);
    change_right_softkey(right_softkey, right_softkey_icon);
    //show_softkey(MMI_LEFT_SOFTKEY);
    //show_softkey(MMI_RIGHT_SOFTKEY);

    RedrawCategoryFunction = RedrawCategory888Screen;
    RedrawCategoryFunction();
}
```

代码 19.3

当需要刷新整个屏幕时，只需要调用 RedrawCategoryFunction 即可，这样就不必使每次刷新都将屏幕重新初始化一遍。

模板历史管理

在保存屏幕历史的时候，我们通常会将屏幕中相关的一些显示状态也保存下来，比如菜单中当前高亮第几项等等，这些状态我们一般都由屏幕模板自己来管理。为方便演示我们先给程序加上一项功能：当用户按左软键时，将弹出一个提示框以显示当前所选中项的文本：

```
void mmi_myapp_popup()
{
    EntryNewScreen(0, NULL, mmi_myapp_popup, NULL);
    ShowCategory165Screen(
        STR_GLOBAL_OK, IMG_GLOBAL_OK,
        STR_GLOBAL_CANCEL, IMG_GLOBAL_BACK,
        get_string(STR_MYAPP_HELLO + MMI_fixed_list_menu.highlighted_item),
        IMG_GLOBAL_ACTIVATED, NULL
    );

    SetLeftSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
    SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
}

void mmi_myapp_entry(void)
{
    ……
    SetLeftSoftkeyFunction(mmi_myapp_popup, KEY_EVENT_UP);
    SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
}
```

代码 19.4

上面代码也顺便让大家了解一下下系统中现有的模板是如何使用的。

运行后，我们选中菜单中某一项并按下左软键，会弹出如下面中间小图所显示的提示框，但是这时用户再按右软键返回，就会发现菜单高亮项又变成第一项了：

图 19.1

模板历史添加方法如下：

```
void mmi_myapp_exit(void)
{
    history currHistory;
    ……
    GetCategoryHistory(currHistory.guiBuffer);          //添加屏幕历史时顺便将模板历史加入其中
    AddHistory(currHistory);
}

U8 *cat888_get_category_history(U8 *history_buffer)
{
    get_list_menu_category_history(0, history_buffer);  //返回屏幕模板的历史记录
}

S32 cat888_get_category_history_size()
{
    return (sizeof(list_menu_category_history));        //返回屏幕模板的历史记录大小
}

void ShowCategory888Screen(
    U16 left_softkey, U16 left_softkey_icon,
    U16 right_softkey, U16 right_softkey_icon,
    S32 number_of_items, U16 *list_of_items, U16 *list_of_icons,
    S32 highlighted_item,
    U8 *history_buffer)
{
    ……

    //初始化菜单
    if (set_list_menu_category_history(0, history_buffer))
    {   //如果屏幕是 GoBack 进来的，则历史不为空
        fixed_list_goto_item_no_redraw(MMI_fixed_list_menu.highlighted_item);
    }
    else
    {   //如果历史为空则用 highlighted_item 初始化
        fixed_list_goto_item_no_redraw(highlighted_item);
    }

    ……
    RedrawCategoryFunction = RedrawCategory888Screen;
    GetCategoryHistory = cat888_get_category_history;
    GetCategoryHistorySize = cat888_get_category_history_size;
    RedrawCategoryFunction();
}

void mmi_myapp_entry(void)
{
```

```
    ……
    EntryNewScreen(SCR_MYAPP_MAIN, mmi_myapp_exit, NULL, NULL);
    guiBuffer = GetCurrGuiBuffer(SCR_MYAPP_MAIN);//获取当前屏幕模板的历史记录，如果第一次进屏幕则记录为空
    ShowCategory888Screen(
        STR_GLOBAL_OK, IMG_GLOBAL_OK,
        STR_GLOBAL_BACK, IMG_GLOBAL_BACK,
        My_fixed_list_n_items, list_of_items, list_of_icons,
        0, guiBuffer      //将模板历史记录传给模板
    );
    ……
}
```

代码 19.5

在将屏幕压入历史时，我们都会顺便调用 GetCategoryHistory 将模板状态一起压入历史(EntryNewScreen 时一般会自动压入)，在应用时我们再将历史取出来，并交由屏幕模板使用。

退出模板

如果在屏幕退出时模板需要做某些善后处理，我们可以将处理函数的地址交由 ExitCategoryFunction，屏幕退出时会自动呼叫到 ExitCategoryFunction。

```
void mmi_myapp_exit(void)
{
    ……
    //gui_fixed_icontext_menuitem_stop_scroll();     //在 reset_fixed_list 会自动停止滚动
}
void ExitCategory888Screen(void)
{
    ClearHighlightHandler();            //清掉所有高亮响应函数
    reset_softkeys();                   //重置左右软键
    reset_menu_shortcut_handler();      //重置快捷序号框
    reset_fixed_list();                 //重置列表菜单
}
void ShowCategory888Screen(
    U16 left_softkey, U16 left_softkey_icon, U16 right_softkey, U16 right_softkey_icon,
    S32 number_of_items, U16 *list_of_items, U16 *list_of_icons,
    S32 highlighted_item, U8 *history_buffer)
{
    ……
    ExitCategoryFunction = ExitCategory888Screen;
    RedrawCategoryFunction = RedrawCategory888Screen;
    GetCategoryHistory = cat888_get_category_history;
    GetCategoryHistorySize = cat888_get_category_history_size;
    RedrawCategoryFunction();
}
```

代码 19.6

使用新模板

现在可以重复使用我们的新模板了，下面我们来试用一下：

```
#include "Worldclock.h"
#define My_fixed_list_n_items   (30)
void mmi_myapp_entry(void)
{
    S32 i;
    U16 list_of_items[My_fixed_list_n_items];
    U16 list_of_icons[My_fixed_list_n_items];
    U8   *guiBuffer = NULL;

    EntryNewScreen(SCR_MYAPP_MAIN, mmi_myapp_exit, NULL, NULL);
    guiBuffer = GetCurrGuiBuffer(SCR_MYAPP_MAIN);
    for (i = 0; i < My_fixed_list_n_items; i++)
    {
        list_of_items[i] = STR_WCLOCK_CITY1 + i;
        list_of_icons[i] = IMG_GLOBAL_L1 + i;
    }
    ShowCategory888Screen1(
      STR_GLOBAL_OK, IMG_GLOBAL_OK,
      STR_GLOBAL_BACK, IMG_GLOBAL_BACK,
      My_fixed_list_n_items, list_of_items, list_of_icons,
      0, guiBuffer);
    SetLeftSoftkeyFunction(mmi_myapp_popup, KEY_EVENT_UP);
    SetRightSoftkeyFunction(GoBackHistory, KEY_EVENT_UP);
}
```

代码 19.7

显示效果如下：

图 19.2

这个屏幕模板目前还无法进行触摸屏操作，虽然也可以按照第十七章的方法为其加上，但如果每个屏幕模板都这样弄，无疑会使代码变得很冗余，下一章“高级模板”我们将会为大家介绍解决办法。

第二十章：高级模板

介绍

第八章我们讲过，在 WGUI 中除了屏幕模板与控件模板外还有两个模块“Touch Screen”与“Draw Manager”，现在再重温一下这两个模块的概念：

Touch Screen：Touch Screen 主要是为了协调屏幕中的触摸屏操作管理。要协调触摸屏操作，就要知道当前屏幕中有哪些控件，为此我们在 WGUI 建了一个“模板数据库”，在数据库中我们收录了每个模板的控件列表。当有触摸屏操作时，Touch Screen 就会从数据库中取出当前模板的控件集，并挨个询问这些控件要不要处理当前触摸屏操作，如果控件回应要处理，则将控制权完全交给此控件。

下面是 Touch Screen 的简要流程图：

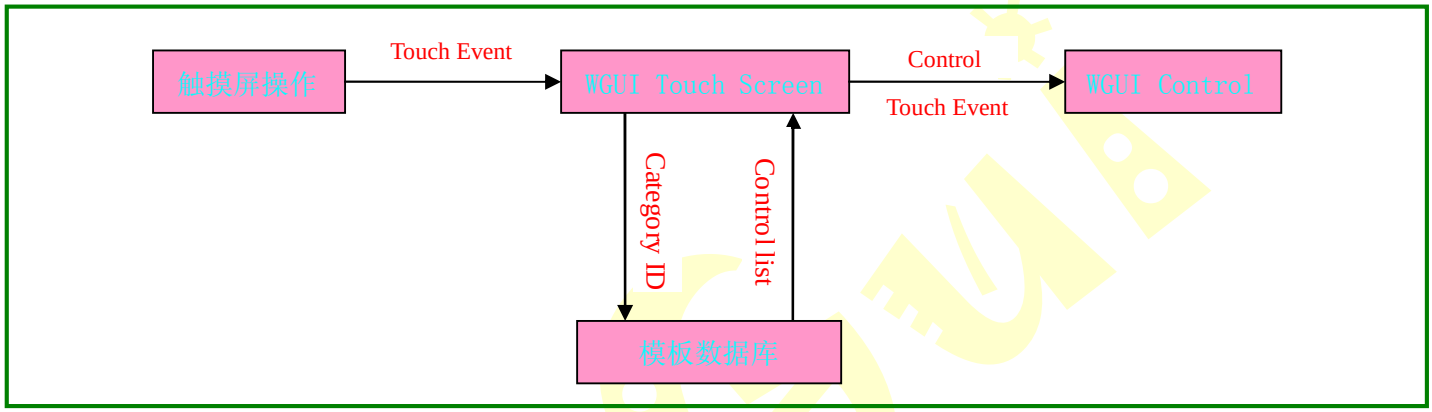


图 20.1

注：WGUI 中的 Touch Screen 将完全接管所有触摸屏操作，所以第十七章所讲的触摸屏响应注册接口在高级模板中都不可以使用。

Draw Manager：Draw Manager(后面将简称为 DM)主要是为了减轻代码冗余。Touch Screen 在模板数据库中保存了每个模板的控件列表，DM 一并将每个控件的属性集(如排版数据，控制标志等)也加入其中，当要绘制模板时 DM 将控件集与属性集一起取出来，然后依次通知每个 WGUI 控件，WGUI 控件收到 DM 的通知与相关的属性集后立即将自己绘制出来。

下面是 DM 的简要流程图：

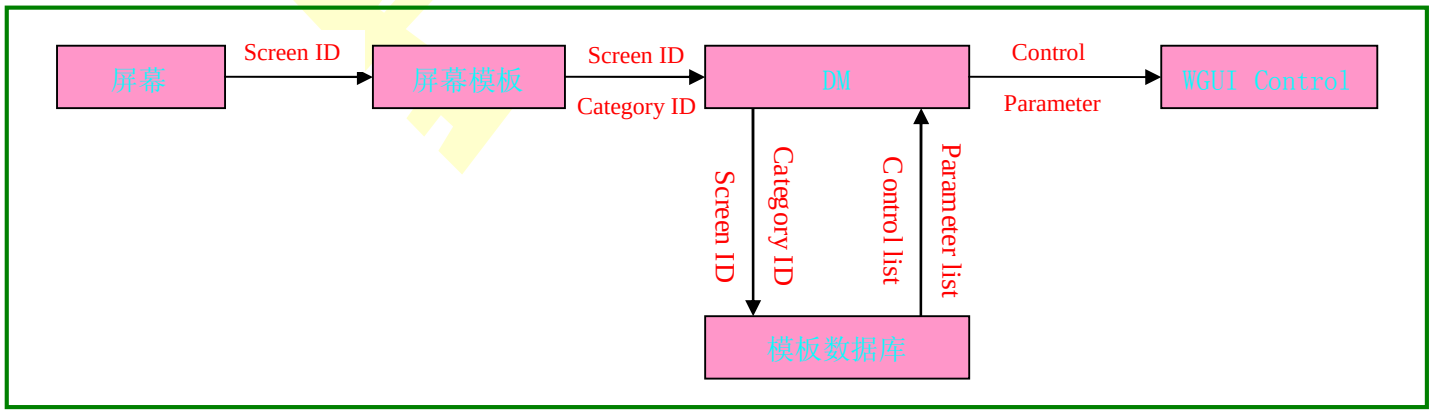


图 20.2

普通屏幕模板加入 TouchScreen 与 DM 后，就成了高级屏幕模板，本章将为大家介绍如何将普通模板转化为高级模板。

模板数据库

高级模板的重中之重是“模板数据库”，数据库存在 CustCoordinates.C 中，下面将详细讲述数据库的构成。
首先要讲的是映射表 g_categories_controls_map，映射表中每一项代表一个模板，其结构定义如下：

```
typedef struct
{
    U16 category_id;                //模板 ID，CategoryScreen、DM、TouchScreen 之间主要是通过模板 ID 相互交流
    U8 *control_set_p;              //控件列表
    S16 *default_coordinate_set_p;  //属性列表
    S16 *rotated_coordinate_set_p;  //旋转屏幕的属性列表，目前基本不用
} dm_category_id_control_set_map_struct;
```

代码 20.1

模板 ID 定义在 wgui_categories_defs.h 中，我们将新模板的 ID 命名为 MMI_CATEGORY888_ID：

```
enum MMI_CATEGORY_ID_LIST
{
    MMI_LIST_CATEGORY_ID = 1,
    MMI_CATEGORY5_ID,
    ……
    MMI_CATEGORY_NSM275,
    MMI_CATEGORY888_ID    //我们屏幕的模板 ID
};
```

代码 20.2

然后在映射表中加上一项：

```
const dm_category_id_control_set_map_struct g_categories_controls_map[] =
{
    {MMI_CATEGORY5_ID, (U8 *) category5, (S16 *) coordinate_set5, NULL},
    {MMI_CATEGORY6_ID, (U8 *) list_menu_category, (S16 *) common_coordinate_set, NULL},
    ……
    {MMI_CATEGORY888_ID, (U8 *) category888, (S16 *) coordinate_set888, NULL}
};
```

代码 20.3

我们的模板包含了两个控件“列表控件”与“系统按键条”：

```
const U8 category888[] =
{
    3,
    DM_BASE_LAYER_START,
    DM_LIST1,
    DM_BUTTON_BAR1,
};
```

代码 20.4

第一个参数表示我们模板包含的控件数，这里为数字“3”是因为我们还加上了一个控制类型的控件 DM_BASE_LAYER_START，后面会详细讲此控件的作用。
控件在列表中放置的顺序也有讲究，越往后的控件显示越靠上层，也越容易接收触摸屏操作。

接下来定义我们模板的属性集：

```
const S16 coordinate_set888[] =
{
    DM_FULL_SCREEN_COORDINATE_FLAG,
    20, MMI_CONTENT_Y + 5, 136, MMI_CONTENT_HEIGHT - 40, DM_NO_FLAGS,
    DM_DEFAULT_BUTTON_BAR_FLAG, MMI_SOFTKEY_WIDTH,
};
```

代码 20.5

每个控件需要多少个属性由控件自行决定，读取属性也要由控件自己来做，控件只需取完属性后通知 DM 取了几个即可。换句话说属性集的存取 DM 概不负责(这样也有风险，只要有一个控件弄错了就会导致连锁错误)。

大部分控件的属性集定义是有规律可讲的，一般是 5 个属性：X，Y，Width，Height，Flags。

如果当前控件读到的第一个属性是下面表格中列的一些值，则读取属性的方式另有变化（P(x)表示第一个属性值后面的第 x 个值，P(0)即当前属性值）：

第一个属性值	X	Y	Width	Height	Flags
DM_DUMMY_COORDINATE	0	0	0	0	无
DM_NULL_COORDINATE	-1	-1	-1	-1	无
DM_FULL_SCREEN_COORDINATE_FLAG	0	0	LCD Width	LCD Height	无
DM_FULL_SCREEN_COORDINATE	0	0	LCD Width	LCD Height	P(1)
DM_CONTENT_COORDINATE_FLAG	Content X	Content Y	Content Width	Content Height	无
DM_CONTENT_COORDINATE	Content X	Content Y	Content Width	Content Height	P(1)
DM_POPUP_SCREEN_COORDINATE_FLAG	Popup X	Popup Y	Popup Width	Popup Height	无
DM_POPUP_SCREEN_COORDINATE	Popup X	Popup Y	Popup Width	Popup Height	P(1)
DM_DEFAULT_TITLE_BAR_FLAG	Title X	Title Y	Title Width	Title Height	无
DM_DEFAULT_TITLE_BAR	Title X	Title Y	Title Width	Title Height	P(1)
DM_DEFAULT_BUTTON_BAR_FLAG	0	Button Bar Y	LCD Width	Button Bar Height	无
DM_DEFAULT_BUTTON_BAR	0	Button Bar Y	LCD Width	Button Bar Height	P(1)
非以上所列值	P(0)	P(1)	P(2)	P(3)	P(4)

图 20.3

另外，属性集中的第一个属性不是给控件的，而是模板的基础属性。

在模板数据库还有一张映射表 g_screenid_coordinate_sets_map，此映射表以 Screen ID 为索引，因为每个屏幕模板都会有多个屏幕使用到，而有些屏幕并不完全依照模板显示，这时就可以在此映射表为屏幕加上一些特殊的属性集，显示的时候 DM 会将此属性集取出来并优先使用。

模板数据库控件

模板数据库中的控件总体来说可以分为三种类型：

控制类型控件：严格来说这些并不能叫控件，应该叫“控制”更为合适。因为这些控件并不拿来显示，而是用来控制其它控件的显示。

注：下面表格第三列中的“布局”，如果值为“标准”表示按照一般的属性读取方式，否则为自定义读取方式。括号中 DM 表示控件以数据库中的值初始化，CategoryScreen 表示控件以 ShowCategoryXXXScreen 中的值初始化(还是按以前普通模板的方式)。

名称	描述	属性集
DM_BASE_LAYER_START	激活基础层。 从 DM_BASE_LAYER_START 到 DM_BASE_LAYER_END 之间的控件都画在基础层上	**布局：** 无 **Flags：** 无
DM_BASE_LAYER_END	结束基础层活动状态，此控件基本上不用。	**布局：** 无 **Flags：** 无
DM_NEW_LAYER_START	创建一个新层，并将其激活。 从 DM_NEW_LAYER_START 到 DM_NEW_LAYER_END 之间的控件都画在新层上	**布局：** 标准(DM) **Flags：** DM_NO_FLAGS
DM_NEW_LAYER_END	结束新层活动状态，此控件基本上不用，新层在屏幕退出时由 DM 自动释放。	**布局：** 无 **Flags：** 无
DM_ALIGNED_AREA_START	控件对齐开始。 此控件开始，至 DM_ALIGNED_AREA_END 结束，中间的所有控件将上下等距对齐。 注：如若中间有单行或多行编辑框，则自动压缩其大小不超过对齐区域的大小。	**布局：** 标准(DM) **Flags：** DM_ALLIGNED_AREA_EQUAL_SPACE_TOP_AND_BOTTOM：间距为零，上下居中。 DM_ALLIGNED_AREA_MULTILINE_SCROLL_IF_REQUIRED：如若是间距为零，上下居中，此 Flag 自动将多行编辑框压缩并使其自动滚动。 DM_ALLIGNED_AREA_NO_BACK_FILL：对齐区域填充背景。
DM_ALIGNED_AREA_END	控件对齐结束。	**布局：** 无 **Flags：** 无

图 20.4

标准显示类控件：这些控件都有 WGUI Control 原型：

名称	描述	属性集
DM_CIRCULAR_MENU1	循环 3D 菜单。 只有主菜单才会用到。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_LIST1	列表菜单。	**布局：**标准(CategoryScreen) **Flags：** DM_CATEGORY_CONTROL_COORDINATES：菜单布局强制使用 DM 数据。
DM_INLINE_FIXED_LIST1	内嵌编辑列表菜单。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_MATRIX_MENU1	矩阵菜单。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_DYNAMIC_LIST1	动态列表菜单。 菜单项在显示的时候动态加载(WGUI 菜单项数组容量有限，当实际项数超出容量时，就只能显示到哪一项再初始化哪一项)。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_ASYNCDYNAMIC_LIST1	异步动态列表菜单。 动态加载时采用异步机制(加载很耗时，为了不阻塞用户操作，只能异步加载完后再显示)。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_SINGLELINE_I	单行编辑框。	**布局：**标准(DM 位置，CategoryScreen 宽高)

NPUTBOX1		**Flags：** DM_SINGLE_LINE_INPUTBOX_SPECIFIC_HEIGHT：强制使用 DM 宽高。
DM_MULTILINE_INPUTBOX1	多行编辑框。	**布局：**标准(DM) **Flags：** DM_FIXED_MULTILINE_INPUTBOX_HEIGHT：强制使用 CategoryScreen 高度。 DM_FIXED_MULTILINE_INPUTBOX_WIDTH：强制使用 CategoryScreen 宽度。 DM_FIXED_MULTILINE_INPUTBOX_HEIGHT_NO_MULTITA：去掉编辑框的 Multitap 部份。
DM_HORIZONTAL_TAB_BAR	水平属性页。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_DATE_TIME_DISPLAY	日期时间	**布局：** 无 **Flags：** 无
DM_TITLE1	标题条。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_STATUS_BAR1	状态条	**布局：** DM_DEFAULT_STATUS_BAR_FLAG： x(0)，y(0)，x1(0)，y1(0)，x2(LCD Width)，y2(LCD Height)，flags(无)。 DM_DEFAULT_STATUS_BAR： x(0)，y(0)，x1(0)，y1(0)，x2(LCD Width)，y2(LCD Height)，flags(P0)。 非以上参数： x(P0)，y(P1)，x1(P2)，y1(P3)，x2(P4)，y2(P5)，flags(P6)。 **Flags：** DM_NO_FLAGS
DM_LSK	左软键。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_RSK	右软键。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_BUTTON	按钮。	**布局：**标准(CategoryScreen 优先，若没设则用 DM) **Flags：** DM_NO_FLAGS
DM_CALENDAR	日历。	**布局：**标准(DM) + 31 个自定义参数 **Flags：** DM_NO_FLAGS
DM_SLIDE_CONTROL	滑杆。	**布局：**标准(DM) **Flags：** DM_SLIDE_CONTROL_VALUE_POS_HEAD：字串放于滑杆前。 DM_SLIDE_CONTROL_VALUE_POS_TAIL：字串放于滑杆后。 DM_SLIDE_CONTROL_VALUE_POS_NEXT_LINE：字串换行。 DM_SLIDE_CONTROL_VALUE_POS_NONE：默认放置。
DM_SCROLL_TEXT	滚动文本条。	**布局：**标准(DM) **Flags：** DM_SCROLL_TEXT_LEFT_ALIGN_X：左对齐。 DM_SCROLL_TEXT_CENTRE_ALIGN_X：左右居中。 DM_SCROLL_TEXT_RIGHT_ALIGN_X：右对齐。 DM_SCROLL_TEXT_TOP_ALIGN_Y：顶对齐。

		DM_SCROLL_TEXT_CENTER_ALIGN_Y：上下居中。 DM_SCROLL_TEXT_BOTTOM_ALIGN_Y：底对齐。 DM_SCROLL_TEXT_USE_FONT_HEIGHT：高度设为文本原始高。
DM_PERCENTAGE_BAR	进度条。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_EMS_INPUTBOX1	EMS 编辑框	**布局：**标准(DM) **Flags：** 与 DM_MULTILINE_INPUTBOX1 一致。
DM_DIALER_INPUT_BOX1	拨号输入框。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS

图 20.5

注：名称中带的数字“1”表示这是模板中此种控件的第一个实体，因为 WGUI Control 一般只有一份实体，所以我们目前还只有 1 没 2。

扩展显示类控件：

名称	描述	属性集
DM_POPUP_BACKGROUND	弹出提示框背景。	**布局：**标准(DM) **Flags：** DM_DRAW_POPUP_BACKGROUND_3D：提示框用 3D 效果。 DM_POPUP_BACKGROUND_HATCH_FILL：用十字编纹填充整个屏幕。 DM_POPUP_BACKGROUND_GREYSCALE：将整个屏幕半灰色显示。
DM_WALL_PAPER	画桌面墙纸。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_SCR_BG	画屏幕背景。	**布局：** 无 **Flags：** 无
DM_CATEGORY_CONTROLLED_AREA	自画区域 1。	**布局：**标准(用户自定) **Flags：** DM_NO_FLAGS
DM_CATEGORY_CONTROLLED_AREA2	版画区域 2。	**布局：**标准(用户自定) **Flags：** DM_NO_FLAGS
DM_NWAY_STOPWATCH	多面向秒表。	**布局：**标准(DM) + 31 个自定义参数 **Flags：** DM_NO_FLAGS
DM_TYPICAL_STOPWATCH	一般秒表。	**布局：**标准(DM) + 31 个自定义参数 **Flags：** DM_NO_FLAGS
DM_BASE_CONTROL_SET1	基础组合控件 1，组合了以下控件： MAINLCD_176X220 || MAINLCD_240X320: DM_STATUS_BAR1, DM_TITLE1, DM_BUTTON_BAR1 MAINLCD_128X160 || MAINLCD_128X128: DM_TITLE1, DM_BUTTON_BAR1	**布局：**无 **Flags：** 无
DM_BASE_CONTRO	基础组合控件 2，组合了以下两个控件：	**布局：**无

L_SET2	DM_STATUS_BAR1, DM_BUTTON_BAR1	**Flags：** 无
DM_BUTTON_BAR1	系统按键条，如下图：  包括左软键，右软键，中软键(或方向箭头)，按键条背景四个部份。	**布局：**标准(DM) + ButtonWidth(P5) **Flags：** DM_BUTTON_DISABLE_BACKGROUND：关掉背景显示，一般有全屏背景都会关掉，如：  DM_BUTTON_BAR_UP_ARROW：画  中的上箭头。 DM_BUTTON_BAR_DOWN_ARROW：画下箭头。 DM_BUTTON_BAR_LEFT_ARROW：画左箭头。 DM_BUTTON_BAR_RIGHT_ARROW：画右箭头。
DM_STRING	文本串。	**布局：**标准(DM) **Flags：** DM_LEFT_ALIGN_X：左对齐。 DM_CENTRE_ALIGN_X：左右居中。 DM_RIGHT_ALIGN_X：右对齐。 DM_TOP_ALIGN_Y：顶对齐。 DM_CENTER_ALIGN_Y：上下居中。 DM_BOTTOM_ALIGN_Y：底对齐。 DM_STRING_BORDERED：带边框的字串。 DM_BACK_FILL：背景填充。
DM_IMAGE	图像。	**布局：**标准(DM) **Flags：** DM_LEFT_ALIGN_X：左对齐。 DM_CENTRE_ALIGN_X：左右居中。 DM_RIGHT_ALIGN_X：右对齐。 DM_TOP_ALIGN_Y：顶对齐。 DM_CENTER_ALIGN_Y：上下居中。 DM_BOTTOM_ALIGN_Y：底对齐。
DM_BACK_FILL_AREA	背景填充。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_RECTANGLE	矩形。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS
DM_LINE	线条。	**布局：**标准(DM) **Flags：** DM_NO_FLAGS

图 20.6

普通模板加入 DM

将上一章的 ShowCategory888Screen 修改如下：

```
void ShowCategory888Screen(
    U16 left_softkey, U16 left_softkey_icon, U16 right_softkey, U16 right_softkey_icon,
    S32 number_of_items, U16 *list_of_items, U16 *list_of_icons,
```

```
    S32 highlighted_item, U8 *history_buffer)
{
    dm_data_struct dm_data;      //模板参数
    ……
    //初始化菜单，因 DM 以模板 ID 为索引保存历史，此处也要同步以模板 ID 为索引使用历史
    if (set_list_menu_category_history(MMI_CATEGORY888_ID, history_buffer))
    {
        fixed_list_goto_item_no_redraw(MMI_fixed_list_menu.highlighted_item);
    }
    else
    {
        fixed_list_goto_item_no_redraw(highlighted_item);
    }
    ……

    ExitCategoryFunction = ExitCategory888Screen;
    RedrawCategoryFunction = dm_redraw_category_screen;  //DM 绘制模板函数
    GetCategoryHistory = dm_get_category_history;        //DM 获取模板历史函数
    GetCategoryHistorySize = dm_get_category_history_size; // DM 获取模板历史记录大小函数
    dm_data.s32ScrId = (S32) GetActiveScreenId();        //当前屏幕 ID，EntryNewScreen(SCR_XXX, …)时自动保存
    dm_data.s32CatId = MMI_CATEGORY888_ID;               //当前屏幕模板的 ID
    dm_data.s32flags = DM_NO_FLAGS;                      //当前模板的 flags，详细标志见下表
    dm_setup_data(&dm_data);                             //将模板参数传入 DM
    dm_redraw_category_screen();                         //由 DM 绘制模板
}
```

代码 20.6

如此修改后运行效果与上一章完全相同，而且我们也可以对整个屏幕进行触摸屏操作了。

按设计初衷，模板的初始化及绘制应该都在 DM 中进行，但是因为早期 CategoryScreen 与 WGUI Control 都没有考虑到触摸屏的情况，现在要完全移植进 DM 工程将极大，所以造成了现在 CategoryScreen 与 DM 共存的局面。

一般情况下，我们都在 CategoryScreen 中初始化，在 DM 中显示。

下面是 dm_data.s32flags 所用到的一些标志：

Flag	描述
DM_NO_FLAGS	默认标志
DM_NO_SOFTKEY	不显示软键
DM_NO_TITLE	不显示标题条
DM_NO_STATUS_BAR	不显示状态条
DM_NO_POPUP_BACKGROUND	不显示弹出提示框背景
DM_LEFT_ALIGN_TITLE	标题条文本左对齐(默认居中显示)
DM_SCROLL_TITLE	标题条文本过长时自动滚动
DM_CLEAR_SCREEN_BACKGROUND	清理整个屏幕背景(为新屏幕绘制作准备)
DM_SHOW_VKPAD	显示虚拟键盘
DM_SUB_LCD_CONTEXT	此屏幕模板用于副屏显示
DM_NO_BLT	DM 不自动合并层(默认会自动合并)

图 20.7

自绘制控件

如果现有控件中找不到符合要求的怎么办？用 DM_CATEGORY_CONTROLLED_AREA。此控件由你自己绘制，触摸屏幕操作也由控件自己处理。
下面我们为新模板加上一个自画区域，并将此区域的触摸屏操作用文本显示出来。
修改模板数据库：

```
const U8 category888[] =
{
    4,
    DM_BASE_LAYER_START,
    DM_CATEGORY_CONTROLLED_AREA,      //将自画区域放在最前面，这样就不防碍后面控件显示及操作
    DM_LIST1,
    DM_BUTTON_BAR1,
};
const S16 coordinate_set888[] =
{
    DM_FULL_SCREEN_COORDINATE_FLAG,
    DM_FULL_SCREEN_COORDINATE_FLAG,   //此控件为全屏大小
    20, MMI_CONTENT_Y + 5, 136, MMI_CONTENT_HEIGHT - 40, DM_NO_FLAGS,
    DM_DEFAULT_BUTTON_BAR_FLAG, MMI_SOFTKEY_WIDTH
};
```

代码 20.6

修改 CategoryScreen：

```
//将当前触摸屏操作以文本形式显示出来
void DrawCate888PenStatus(U8* event_type, mmi_pen_point_struct point)
{
    S32 x, y;
    color text_color = {255, 0, 0, 100};

    gdi_layer_lock_frame_buffer();
    gui_reset_clip();
    gui_set_text_color(text_color);
    x = 20;
    y = 170;
    gui_move_text_cursor(x, y);

    gui_fill_rectangle(0, y - 3, UI_device_width - 1, y + 20 + 3, UI_COLOR_WHITE);
    gui_printf((UI_string_type)"%s {%d, %d}",event_type, point.x, point.y);

    gdi_layer_unlock_frame_buffer();
    gui_BLT_double_buffer(0, y, UI_device_width - 1, y + 20);
}

//绘制自画区域
```

```
void DrawCate888CategoryControlArea(dm_coordinates *coordinate)
{
    mmi_pen_point_struct point = {-1, -1};
    DrawCate888PenStatus("Pen None", point);
}

//触摸笔按下响应函数
MMI_BOOL Cate888CategoryControlAreaPenDownHandler(mmi_pen_point_struct point)
{
    DrawCate888PenStatus("Pen Down", point);
    return TRUE;
}

//触摸笔放开响应函数
MMI_BOOL Cate888CategoryControlAreaPenUpHandler(mmi_pen_point_struct point)
{
    DrawCate888PenStatus("Pen Up", point);
    return TRUE;
}

//触摸笔移动响应函数
MMI_BOOL Cate888CategoryControlAreaPenMoveHandler(mmi_pen_point_struct point)
{
    DrawCate888PenStatus("Pen Move", point);
    return TRUE;
}

//触摸笔重复响应函数
MMI_BOOL Cate888CategoryControlAreaPenRepeatHandler(mmi_pen_point_struct point)
{
    DrawCate888PenStatus("Pen Repeat", point);
    return TRUE;
}

//触摸笔长按响应函数
MMI_BOOL Cate888CategoryControlAreaPenLongTapHandler(mmi_pen_point_struct point)
{
    DrawCate888PenStatus("Pen LongTap", point);
    return TRUE;
}

//触摸笔中止响应函数
MMI_BOOL Cate888CategoryControlAreaPenAbortHandler(mmi_pen_point_struct point)
{
    DrawCate888PenStatus("Pen Abort", point);
    return TRUE;
```

```
}

void ShowCategory888Screen(
    U16 left_softkey, U16 left_softkey_icon, U16 right_softkey, U16 right_softkey_icon,
    S32 number_of_items, U16 *list_of_items, U16 *list_of_icons,
    S32 highlighted_item, U8 *history_buffer)
{
    ……
    //初始化自画区域
    dm_register_category_controlled_callback(DrawCate888CategoryControlArea);     //注册自画区域绘制函数

    wgui_register_category_screen_control_area_pen_handlers(//注册自画区域触摸笔按下响应函数
        Cate888CategoryControlAreaPenDownHandler, MMI_PEN_EVENT_DOWN);
    wgui_register_category_screen_control_area_pen_handlers(//注册自画区域触摸笔放开响应函数
        Cate888CategoryControlAreaPenUpHandler, MMI_PEN_EVENT_UP);
    wgui_register_category_screen_control_area_pen_handlers(//注册自画区域触摸笔移动响应函数
        Cate888CategoryControlAreaPenMoveHandler, MMI_PEN_EVENT_MOVE);
    wgui_register_category_screen_control_area_pen_handlers(//注册自画区域触摸笔重复响应函数
        Cate888CategoryControlAreaPenRepeatHandler, MMI_PEN_EVENT_REPEAT);
    wgui_register_category_screen_control_area_pen_handlers(//注册自画区域触摸笔长按响应函数
        Cate888CategoryControlAreaPenLongTapHandler, MMI_PEN_EVENT_LONG_TAP);
    wgui_register_category_screen_control_area_pen_handlers(//注册自画区域触摸笔中止响应函数
        Cate888CategoryControlAreaPenAbortHandler, MMI_PEN_EVENT_ABORT);

    gdi_layer_unlock_frame_buffer();
    ExitCategoryFunction = ExitCategory888Screen;
    ……
    dm_redraw_category_screen();
}
```

代码 20.7

运行后显示效果如下：

图 20.8

至此，普通模板向高级模板的转化已基本完成，剩下的流程都由 TouchScreen 与 DM 自行完成。