eCGUI 编程指南

版权所有© **eCGUI** 开发小组

http://www.ecgui.com

商业销售合作伙伴：

大连天维科技有限公司

联系： ecgui.com@gmail.com

开发论坛 **http://www.ecgui.com/bbs**

最后更新日期：2009-9-6

目录

说明
前言

1.0 关于嵌入式 **GUI**
1.1 关于这个微型嵌入式 **GUI**

2.0 编写第一个应用程序
2.1 与 **Win32 API** 编程对比
2.2 总结

3.0 编程思想
3.1 对象在 **GUI** 中的应用
3.2 目前 **GUI** 中支持的控件对象类
3.3 GUI 的消息

4.0 控件编程
4.1 按钮
4.2 单行文本编辑框
4.3 多行文本编辑框
4.4 滚动条
4.5 进度条
4.6 文字标签

5.0 高级控件
5.1 下拉菜单
5.2 弹出式菜单
5.3 组合框
5.4 单选框
5.5 多选框
5.6 列表框

6.0 对话框
6.1 定时器
6.2 MessageBox 消息窗口

7.0 控件类创建机制
7.1 创建一个新的控件类
7.2 模块化部件的构造

8.0 图形设备
8.1 基本作图
8.2 关于剪裁
8.3 图象显示
8.4 缓冲

9.0 多任务和多线程
9.1 Linux,DOS,Windows,uC/OS-II 多任务比较
9.2 GUI 中对 多任务**/**多线程 的接口

10.0 GUI 移植性分析
10.1 GUI 输入**/**输出设备驱动
10.2 GUI 在多任务**/**多进程操作系统的移植

说明：

本指南为多平台通用的文档，所以在一些地方忽略了系统的差异性,如 **GUI** 的安装，开发环境的建立，应用程序的编译等。

这些信息将其它文档做详细介绍，需要时请仔细参考副送的文档。

文中的源代码可以从网站中 下载。

如遇到遇到无法解决的问题，请查阅网站上的常见问题解决办法。

网站：www.ecgui.com

前言

阅读本指南前，作者假设您对 C 语言是熟悉的，如果您对 C 语言还不了解，希望您先学习 C 语言 方面的知识。是否有图形界面应用程序编写经验不是必备的条件。如果您很熟悉诸如基于 Windows ，QT 等系统应用程序的编写，您可以更轻松的浏览本指南，因为许多思想和编程模式是各种 GUI 所共有的。

本指南使用的例子，您可以使用在自己的程序中，源代码可以在网站上下载。
网站：http://www.ecgui.com

1.0 关于嵌入式 GUI

GUI 是 Graphics User Interface 的缩写,意思是图形用户界面系统接口早期电脑操作主要通过键盘,使用大量命令,随着的科技的发展,和人们的需要具有图形操作界面的软件随之诞生,并很受大众欢迎,最具代表性的就是当时最成功的苹果机系统 ,人性化的操作界面,强大方便的鼠标操作,让使用电脑变得简单,有趣.由于历史的原因,随后出现的 Windows 系统迅速发展,拥有的了大量使用者,在相当长的时间里几乎有 90%以上的 PC(个人电脑)使用 Windows.目前 Windows 仍然是最主流的图形界面操作系统,并且根据不同需求出现多个版本和分支.如面向个人桌面应用的 Windows 9x,Me,XP 等 和 服务器应用的 Windows NT, 还有面向嵌入式应用的 Windows CE 和 Windows XP Embedded.

除了最常见 Windows 系统外,常用的操作系统为 Linux,Unix,FreeBSD,Solairs 等.其中 Linux 表现得比较活跃,应用范围不断增大,而且得到越来越多的厂商支持.相信 Linux 会有更好的表现!

Linux 在嵌入式领域备受关注,完全免费和开放源代码吸引了大量开发者,
基本上在各种领域中得到广泛应用.但是 Linux 本身并不包含 图形用户界面系统,即 GUI.目前在 Linux 运行的 GUI 主要是 X Window ,如像桌面应用中最流行的 GNOME ,KDE 等.但是由于 X Window 主要是为 PC 机资源较丰富设备设计,并不完全适合资源受限的嵌入式系统.所以又出现了不少为嵌入式系统设计的 GUI ,如著名的 QT/E.

在嵌入式应用中,除了广为人知的 Windows CE 和 Linux 外,还有像 uC/OS-II
eCOS,eRTOS VxWorks 等等操作系统.

如在教学中最常见的 uC/OS-II 系统,其实 uC/OS-II 是一个相当精简的操作系统,仅包含了最主要的 多任务管理 和相关部分.因此体积非常小,性能优越,得到非常广泛的应用,是一个很成熟的微系统.而且非常适合教学.并且源代码也是开放的.但是由于功能不够强大,无法进行要求较高的应用.

eCOS 也是一个非常受欢迎的嵌入式操作系统,支持多任务,相对 uC/OS 有更多更强的功能,目前由最流行的 Linux 发行版开发商 Red Hat 公司主持.现在 eCOS 有很多成功应用.

QNX 也是一个相当出色的嵌入式操作系统,并且本身具有小型的 GUI 系统,并且有像网络浏览器这样的关键应用程序, QNX 公司曾发布过一个演示软盘,在一个容量仅 1.44 MB 的软盘中实现了具有完整图形用户界面,多任务操作系统,和网络浏览器等应用.

除了已经提到的,目前还有大量的系统已经或正在应用到实际当中.

1.1 关于这个微型嵌入式 GUI

这个 微型嵌入式 **GUI** 由 **ecurb2006(**开发小组负责人**)**利用业余时间独立完成，拥有全部版权。开发历时四年左右，期间有多次中断.

目前由开发小组(**http://www.ecgui.com**)负责维护,升级等工作。

商业销售合作伙伴：大连天维科技有限公司。

2.0 编写第一个应用程序

我们将以经典的 **HelloWorld** 为我们的第一个应用程序的例子.
让我们回想一下,我们在字符界面的控制台是如何完成的.

```
#include <stdio.h>

int main(void)
{
  printf("Hello,World");
  return 0;
}
```

程序相当的简单,但确实是一个完成的程序,运行的结果是在屏幕显示 **Hello,World** 这个字符串.
在我们的 **GUI** 环境中如何完成呢?
* 应用程序入口函数为 **int gmain(void *data);**
- step1.c -

```
#include “gui.h”

void HelloGUI(HAND hd,MESSAGE msg)
{
if(msg.type == GM_SYSTEM)
  switch(msg.message)
  {
  case GM_Draw:
    {
    HDC hdc;
    hdc=efGDI->Start(hd);
    efGDI->SetColor(hdc,COLOR_BLACK);
    efGDI->DrawText(hdc,50,50,”Hello,World”);
    efGDI->End(hd,hdc);
    }
  return;
  default:return;
  }
}
int gmain(void *data)
{
efObj->New(0,MAINWINDOW,1,1,"HelloGUI",10,10,400,400,HelloGUI,data);
```

```
}
```

编译运行

2.1 与 Win32 API 编程对比

现在使用 **Win32 API** 完成类似的程序,代码主体由 **Dev-CPP** 生成，手动加入了显示字符串部分。

```
#include <windows.h>

/*  Declare Windows procedure  */
LRESULT CALLBACK WindowProcedure (HWND, UINT, WPARAM,
LPARAM);

/*  Make the class name into a global variable  */
char szClassName[ ] = "WindowsApp";

int WINAPI WinMain (HINSTANCE hThisInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszArgument,
                    int nFunsterStil)

{
    HWND hwnd;               /* This is the handle for our window */
    MSG messages;            /* Here messages to the application are
saved */
    WNDCLASSEX wincl;        /* Data structure for the windowclass */

    /* The Window structure */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = szClassName;
    wincl.lpfnWndProc = WindowProcedure;      /* This function is called by
windows */
    wincl.style = CS_DBLCLKS;                 /* Catch double-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Use default icon and mouse-pointer */
    wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
```

```
    wincl.lpszMenuName = NULL;                 /* No menu */
    wincl.cbClsExtra = 0;                      /* No extra bytes after the
window class */
    wincl.cbWndExtra = 0;                      /* structure or the window
instance */
    /* Use Windows's default color as the background of the window */
    wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;

    /* Register the window class, and if it fails quit the program */
    if (!RegisterClassEx (&wincl))
        return 0;

    /* The class is registered, let's create the program*/
    hwnd = CreateWindowEx (
           0,                   /* Extended possibilites for variation */
           szClassName,         /* Classname */
           "Windows App",       /* Title Text */
           WS_OVERLAPPEDWINDOW, /* default window */
           CW_USEDEFAULT,       /* Windows decides the position */
           CW_USEDEFAULT,       /* where the window ends up on the
screen */
           544,                 /* The programs width */
           375,                 /* and height in pixels */
           HWND_DESKTOP,        /* The window is a child-window to
desktop */
           NULL,                /* No menu */
           hThisInstance,       /* Program Instance handler */
           NULL                 /* No Window Creation data */
           );

    /* Make the window visible on the screen */
    ShowWindow (hwnd, nFunsterStil);

    /* Run the message loop. It will run until GetMessage() returns 0 */
    while (GetMessage (&messages, NULL, 0, 0))
    {
        /* Translate virtual-key messages into character messages */
        TranslateMessage(&messages);
        /* Send message to WindowProcedure */
        DispatchMessage(&messages);
    }

    /* The program return-value is 0 - The value that PostQuitMessage() gave
*/
```

```
    return messages.wParam;
}


/*  This function is called by the Windows function DispatchMessage()  */

LRESULT CALLBACK WindowProcedure (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)                  /* handle the messages */
    {
        case WM_PAINT:
        {
        HDC hdc;
        PAINTSTRUCT ps ;
        hdc = BeginPaint (hwnd, &ps) ;
        TextOut (hdc, 50, 50, "Hello,World",11) ;
        EndPaint (hwnd, &ps) ;

        }
        return 0;
        case WM_DESTROY:
            PostQuitMessage (0);       /* send a WM_QUIT to the message queue */
            break;
        default:                      /* for messages that we don't deal with */
            return DefWindowProc (hwnd, message, wParam, lParam);
    }

    return 0;
}
```

Win32 API 烦琐的结构，给应用程序的编写制造了一定的困难，不难发现使用 **Win32 API** 编程并不是件非常轻松的事，当然相对 **MFC** 这样的 **Framework,** 在程序执行效率上仍有一定优势。

编译运行:

2.2 总结

从上面的对比中可以看出，这个 **GUI** 在 **API** 方面更简单易用，与 **Win32**

API 这样庞大的 **API** 相比，更适合硬件资源受限的嵌入式应用。

3.0 编程思想

与在控制台编程不同的是,**GUI** 环境有很强的交互性,所以旧有的编程模式是无法完成任务的.我们需要采用新的编程模式,解决问题的方式.

如果您以前曾经在控制台编写应用程序,您可能已经习惯了在程序构造一个循环，不停的检测键盘,来判断用户是否按下了键盘中的某个按键,如果您想增加程序的功能,那么就意味着在这个循环中加入更多的代码和函数,这部分代码会变得庞大而且可能很难维护,如果用户希望可以使用鼠标,是的,在字符界面下也可以使用鼠标的,但对于您的程序来说,加入鼠标功能支持,就意味着需要编写更多的代码,完成更多烦琐的工作,程序复杂性加深,产生程序错误的几率增大,程序会变得不稳定.这些是我们所不希望看到的.

幸运的是,聪明的开发人员想出更好的解决方法,他们提出了新的编程思路,并开发出了许多成功的图形操作系统,如苹果公司的 **Apple/MAC** 系统,和微软公司的 **Windows** 系统.这些是最有代表性的图形操作系统,虽然他们都不是这相技术的第一发明者,关于这方面的历史请查阅相关资料.

图形操作系统的出现,彻底改变了应用程序的编写模式,并且引入诸如 面向对象 等新的编程理念,本指南中会大量的使用面向对象的思维模式,但本指南并非面向对象思想的专著,所以如果您希望了解更多面向对象的编程思想,请查阅相关书籍.但是就编写一般的应用程序而言,本指南讲解的面向对象的知识已经足够了.

首先您需要了解什么是前文提到的对象,简单来说,在图形用户界面系统中,一个窗口,一个按钮,一个文本框都可以看作是对象**(Object)**,每个对象都有自己的属性,私有的数据,和一些操作方法等内容.如何应用这种对象化的思想才是最关键的.

需要说明的是 **ANSI C** 本身并不明确声明支持面向对象的编程方法，目前一般普遍认为像 **C++,Java** 等这样的语言才是支持面向对象的编程语言，本指南认为 面向对象是一种思想，并不归属于特定的编程语言，表现形式可以是多种多样的。使用 **ANSI C** 编写图形用户界面的应用程序时，运用面向对象的编程思想是可以的。

在编写应用程序前，您还需要了解 事件驱动的编程模式，这是与在控制台编程的根本差异所在。

主要原理就是：系统负责检测 键盘/鼠标等输入设备(在嵌入式应用也可以是触摸屏)的状态变化，当有 按键按下时，系统将构造一个消息，并判断目前的活动的控件对象(比如一个正在录入字符的文本框)，然后将消息加入到消息列队当中，再继续检测其他事件是否发生，直到所有输入设备检查完毕，最后再依次将消息从列队中取出，发送给对应的对象，呼叫对象的消息处理函数。值得一提的是，在编写消息处理函数时，您只需处理您关注的消息，对于其它消息，不去理会就是了。

这样在编写应用程序时，思路清晰，程序能够实现模块化，程序更易与维护，稳定性也会较强。

一个简单的例子：实现对鼠标单击的响应

- step2.c -

```
#include "gui.h"
void     HelloTextbox(HAND hd,MESSAGE msg)
{
if(msg.type == GM_MOUSE)
switch(msg.message)
{
case GM_LeftDown:
   efTextBox->Set(hd,"OK!");
   break;
case GM_LeftUp:
   efTextBox->Set(hd,"Key words");
   break;
default:break;
}
}
void     HelloGUI(HAND hd,MESSAGE msg)
{
if(msg.type == GM_SYSTEM)
switch(msg.message)
{
case GM_Create:
  efObj->New(hd,TEXTBOX,1,1,"Key
words",20,40,120,60,HelloTextbox,NULL);
  efObj->New(hd,BUTTON,1,1,"Search",130,40,200,60,NULL,NULL);
  break;
default:break;
```

```
}
}
int gmain(void *data)
{
efObj->New(0,MAINWINDOW,1,1,"HelloGUI",10,10,400,400,HelloGUI,data);
}
```

以上代码中

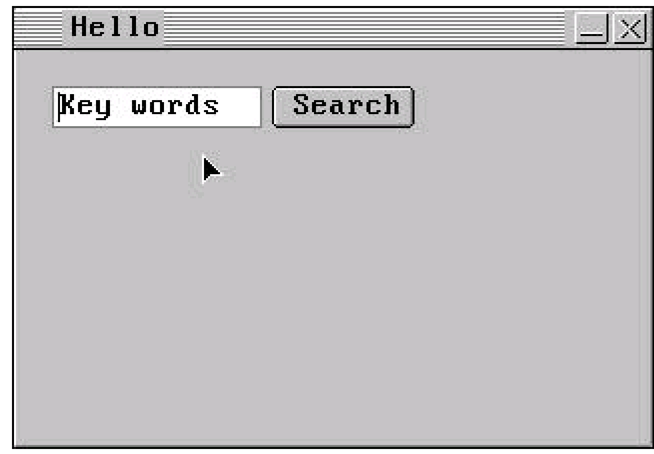
HAND	为对象的句柄，其本质是指针
MESSAGE	为消息的结构
GM_SYSTEM	为系统预先定义好的系统消息类，用常数表示,宏定义
GM_Create	为系统消息,在对象被创建时产生
GM_MOUSE	为系统预先定义好的鼠标消息类
GM_LeftDown	为消息，表示鼠标左键按下
GM_LeftUp	为消息，表示鼠标左键释放
efObj->New	函数用于创建 **GUI** 对象，这是一个非常重要的函数
TEXTBOX	表示单行文本框类，值为常数，是宏定义
BUTTON	表示按钮类，值为常数，是宏定义
efTextBox	为文本框的功能调用模块(*私有 API 子集*)，这是这个 **GUI** 的特色之一，将简化应用的编写，设计目标就是易用，实现少写代码多做事

关于私有 API

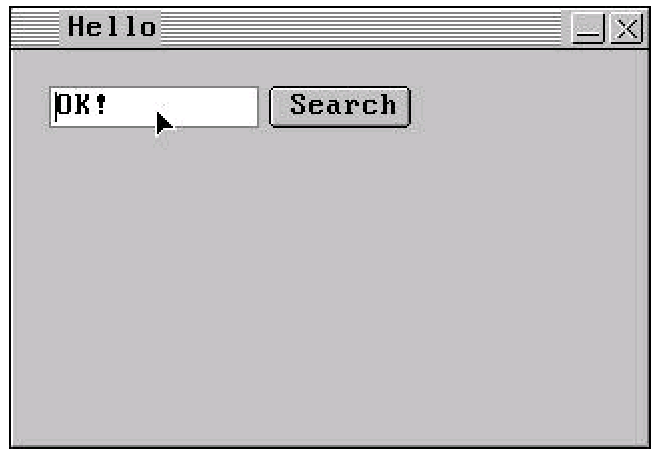
与目前的广为人知的 Win32 API 相比,这个 GUI 中 API 为私有化设计,即不与目前常用 Win32 API 兼容,众所周知, Win32 API 是一个非常庞大的 API 系统,并不完全适合资源受限的嵌入式系统,即使是面向嵌入式应用的 WinCE 系统,也对 Win32 API 进行大量的筛选,仅提供对部分 Win32 API 的支持,一般来说使用 Win32 API 编写的程序并不能直接在 WinCE 进行编译,必须重新编写或者将 WinCE 不支持的 API 去除.

虽然 Win32 API 是 Windows 编程的基础,但是使用 Win32 API 编写应用程序仍然是一个比较烦琐的过程,许多 Windows 程序员更喜欢用像 VCL,MFC 这样的 Framework .

编译运行



在文本框鼠标按下左键!



这样就完成了具有一定功能的应用程序。

更详细的介绍将在以后的章节中给出，这里只为了给出一个大概的编程形式。

3.1 对象在 **GUI** 中的应用

所有的控件**(BUTTON** 按钮**,TEXTBOX** 单行文本框等**)**都可以看作对象**(Object),**事实上**,**在 **GUI** 中**,**这些对象都是从同一个 **Object** 类中派生出来的**,**只是有不同的属性**,**私有数据**,**和操作方法**.**

要使用一个对象**(Object),**需要先创建它**,**设定的的属性等信息**.**完成这相工作直接调用系统的 **efObj->New** 函数就可以完成**.**

efObj->New 函数原形如下

```
HAND        GUIAPI efObj->New(HAND pHOST,
int16 guiclass,
int16 id,
int16 style,
char    *szTitle,
int16 left,int16 top,int16 right,int16 bottom,
void      (*Server)(HAND,MESSAGE),
void * pData);
```

其中

pHOST	上一级对象句柄(一般为窗口)

guiclass	表示对象的类别属性
id	表示对象的唯一数字编号，每个对象需要一个与其他对象不同的编号
style	表示对象的风格属性
szTitle	为对象的标题
left,top,right,bottom	为对象的区域大小，为相对坐标
void (*Server)(HAND,MESSAGE)	为对象的消息处理函数
pData	为附加数据，一般不用，设为 **NULL** 就可以了

系统提供许多常见的控件类，您可以在程序里通过调用 **efObj->New** 派生出具有控件类属性的新对象。

如在上一个例子中(***3.0 编程思想***),使用下面的语句，创建了一个文本框对象，并指定 **HelloTextbox** 为新对象的消息处理函数。

```
efObj->New(hd,TEXTBOX,1,1,"Key
words",20,40,120,60,HelloTextbox,NULL);
```

3.2 目前 GUI 中支持的控件对象类

BUTTON	按钮
MAINMENU	下拉式菜单
POPMENU	弹出式菜单
HSCROLLBAR	横向滚动条
VSCROLLBAR	竖向滚动条
TEXTBOX	单行文本编辑框
EDITBOX	多行文本编辑框
COMBOBOX	组合框
LISTBOX	列表框
CHECKBOX	多项选择框
CHOICEBOX	单项选择框
LABEL	文字标签
PAGEVIEW	标签页
SPEEDBAR	进度条

在创建这些控件对象时,必须从属一个窗口

MAINWINDOW	主窗口
DLGWINDOW	对话框窗口,从属于主窗口

主窗口也可以通过 **efObj->New** 函数创建,对话窗口使用其他函数即时创建，所以不要使用 **efObj->New** 函数创建对话窗口。

应用程序中可以使用系统已经提供的控件类,也可以创建新的控件类

3.3 GUI 的消息

和大多数 **GUI** 系统类似，这个 **GUI** 也采用了 消息循环和事件驱动 的模式，事件是以消息的形式发送给对象的。

MESSAGE 结构原形如下

```
typedef struct _MESSAGE{
HAND    pHOST;
uint32    type;
uint32    message;
int32      icode;
int32      jcode;
void      *pData;
}MESSAGE,*PMESSAGE;
```

其中

pHOST	接收消息的对象
type	消息类型
message	消息

icode	附加码 icode
jcode	附加码 jcode
pData	附加数据

目前 GUI 消息有以下几类

GM_SYSTEM	系统消息，包括 GM_Create 等
GM_MOUSE	鼠标类消息
GM_KEY	键盘类消息
GM_CTRL	控件类消息
GM_COMMAND	按钮事件消息，直接发送到从属窗口
GM_TIMECALL	定时器类消息

GUI 中消息以 **GM** 开头，表示 **GUI Message .**

约定：
类消息全部大写，如 **GM_SYSTEM** 。
消息以 **GM** 开头，用下划线分隔，大小写混合体表示消息，如 **GM_LeftDown** 。

以下是 **GM_SYSTEM** 类消息

GM_SYSTEM	
GM_Create	对象被创建时产生
GM_Destroy	对象被销毁前产生
GM_Draw	对象被显示时产生
GM_Close	窗口被关闭或用户选择退出时产生
GM_GotFocus	对象获得焦点时产生
GM_LostFocus	对象失去焦点时产生

GM_MOUSE 类消息（部分）

GM_MOUSE	
GM_MouseMove	鼠标移动
GM_MouseOver	鼠标移入对象区域内
GM_MouseAway	鼠标移出对象区域

GM_LeftDown	鼠标左键按下**(**在对象区域内**)**
GM_LeftUp	鼠标左键释放**(**在对象区域内**)**
GM_LeftAwayDown	鼠标左键在对象区域外按下
GM_LeftAwayUp	鼠标左键在对象区域外释放
GM_LeftDownMove	鼠标左键按下拖动
GM_LeftAwayDownMove	鼠标左键在对象区域外按下拖动

为了更好的理解消息储存形式，现在以鼠标按下左键事件引发的消息为例说明

```
……
if(msg.type = = GM_MOUSE)
 switch(msg.message)
 {
 case GM_LeftDown:
  {
  int x,y;
  x=msg.icode; /* mouse x */
  y=msg.jcode; /* mouse y */
  }
 return;
 default: break;
 }
……
```

GM_KEY 类消息

GM_KEY	
GM_KeyDown	键按下
GM_KeyUp	键释放

消息结构中 **icode** 储存按键的 **ASCII** 码

GM_CTRL 类消息（部分）

GM_CTRL	
GM_TextInsert	有字符串插入
GM_ScrollHith	滚动条值变大
GM_ScrollLow	滚动条值变小

GM_Scrolling	滚动条值变化
GM_MenuItem	选择框选择子项
GM_TextChange	编辑框内容改变

GM_COMMAND 类消息
按钮按下并释放时产生，消息发送给按钮从属的窗口，当窗口中按钮较多时，使用这个消息比较方便. 消息结构中 **message** 储存按钮的 **ID** 。

GM_TIMECALL 类消息
为定时器类消息，消息发送给定时器从属对象。
消息结构中 **message** 储存定时器 **ID,**一个对象可以创建多个定时器，需要分别设定不同的 **ID** ，定时器才可以正常工作。

4.0 控件编程

一般而来说，像窗口中的按钮，文本框这样用户交互对象，都可以称作控件，基于已有控件类，可以做出功能强大的图形用户界面交互式应用程序。您只需使用 **efObj->New** 函数派生出自己需要的控件对象就可以了，**GUI** 系统将负责维护控件的各种消息，如文本框会接受字符录入，您只需在对象的消息处理函数中关注文本框内容改变**(GM_TextChange)**的消息**,**就可以知道文本框的最新内容。

您也可以创建自己的控件类，系统提供统一的控件类创建机制。您甚至动态改变系统控件类的行为属性，当然如果没有特殊需求，没有必要这样做。

一个新的控件类建立好后，应用程序中就派生新的对象，新的对象拥有控件类的行为属性，却拥有自己的数据，如 多个文本框，各自之间的内容互不影响，
保证各个对象的独立性。

当使用 **efObj->New** 来创建新对象时，**left,top,right,bottom** 参数为对象在父对象中的相对坐标位置，如果超出了父对象的区域，那么 **GUI** 系统会自动剪裁超出的部分，这是可以接受的。比如说一个按钮的一部分在窗口外，显然不是很合适的。

(注：这里仅讨论基本的 **GUI** 界面设计，可能一些界面特效会把按钮的一部分放在窗口外面，想法不错！不过超出了本指南的讨论范围。您有兴趣的话，可以通过邮件或其他方式和作者进行交流。**)**

当控件需要创建自己的数据时,建议不要使用全局变量，**GUI** 提供一些 **API** 来实现数据的私有化。

主要是 **efObj->SetVar** 和 **efObj->Var** ，原形如下

```
void     GUIAPI   efObj->SetVar(HAND,void*);
void*    GUIAPI   efObj->Var(HAND);
```

参数为 **void*** ，所以可以存储任意类型的数据。

efObj->SetVar 函数可以设定对象的数据指针。

efObj->Var 函数可以获得对象的数据指针，使用时最好将返回值强制转换成程序中使用的数据类型，来避免编译器警告。

您可能要问，为什么需要这样做？因为当不使用全局变量时，一个函数通常可以成为的一个可重入函数，这是这个 **GUI** 实现对象的多次动态加载的重要基础。这也是这个 **GUI** 的特色之一。

如果不使用全局变量，就意味着需要通过其他途径来实现多个 **GUI** 控件访问同一数据。

推荐的做法是**:**将共享的数据指针储存在窗口里，其他控件通过得到窗口的句柄，再访问共享的数据。共享数据的储存空间可以窗口处理 **GM_Create** 消息创建。

实现数据共享的例子：

-step3.c-

```
#include "gui.h"
typedef struct _MyData{
int x,y;
char str[100];
}MyData,*PMyData;
void     HelloTextBox(HAND hd,MESSAGE msg)
{
if(msg.type == GM_SYSTEM)
 switch(msg.message)
 {
  case GM_Draw:
  {
```

```
   HAND host;
   PMyData mydata;
   host=GetObjHost(hd);
   mydata=(PMyData)efObj->Var(host);
   if(mydata == NULL) return; /* Not exist */
   efTextBox->Set(hd,mydata->str);
   }
   return;
  default:return;
  }
}
void     HelloGUI(HAND hd,MESSAGE msg)
{
PMyData mydata;
if(msg.type == GM_SYSTEM)
  switch(msg.message)
  {
  case GM_Create:
     mydata=(PMyData)Gmalloc(sizeof(MyData),"HelloGUI:Main");
     strcpy(mydata->str,"Object private data");
     efObj->SetVar(hd,(void*)mydata);
     efObj->New(hd,TEXTBOX,1,1,"",10,50,110,70,HelloTextBox,NULL);
     return;
  case GM_Destroy:
       mydata=efObj->Var(hd);
       Gfree(mydata,sizeof(MyData));
       return;
  default:return;
  }
}

int gmain(void *data)
{
efObj->New(0,MAINWINDOW,1,1,"HelloGUI",10,10,400,400,HelloGUI,data);
}
```

程序代码说明

程序首先创建了一个主窗口，设定消息处理函数为 **HelloGUI**，并处理 **GM_Create** 消息，需要说明的是使用 **GUI** 提供的内存函数，原形如下

```
void* GUIAPI Gmalloc(int16 size,char *msg);
uint8 GUIAPI Gfree(void *data,int16 size);
```

原由：作者在移植 **GUI** 到 **uC/OS-II** 这样的多任务嵌入式操作系统时，了解到这个操作系统仅提供基本的任务管理功能，没有提供 **malloc/free** 这样的系统函数，于是作者自己写了一套内存管理函数，供 **GUI** 使用。

char *msg 参数供调试时使用，无法成功得到内存空间时，会有提示。
在内存释放，参数 **size** 用于核对大小，这样可以避免一些非法的内存释放，有效提高程序健壮性。这也是这个 **GUI** 的特色之一。

** 目前提供免费下载的* ***SDK*** *开发包中，****Gmalloc/Gfree*** *只是简单调用了系统提供的* ***malloc/free*** *，没有核对和提示的功能，最新说明请查看开发包中的其他文档。*

请注意代码中的这一行

```
if(mydata == NULL) return; /* Not exist */
```

判断是否返回了空指针，可以避免程序进行非法指针访问。

数据的销毁工作

```
case GM_Destroy:
	mydata=efObj->Var(hd);
	Gfree(mydata,sizeof(MyData));
	return;
```

在编写程序时，可能经常创建了数据空间，却忘记了释放，造成内存泄露，在这里作者建议在使用 **GM_Create** 消息后，就接着处理 **GM_Destroy** 消息，避免错误的发生。在这里只是提醒一下，希望对您编写应用程序有帮助。

4.1 按钮

GUI 应用程序中，按钮可以说是最基本，最常用的控件了。
目前提供两风格的按钮

STY_BUTTON_NORMAL	普通按钮
STY_BUTTON_3D	三态按钮，鼠标在上空时，会突起

在窗口消息处理函数中**(GM_Create** 消息**)**创建了按钮后，当按钮被按下并释放后，**GUI** 会向窗口发送 **GM_COMMAND** 类消息，消息结构中 **message** 储存此按钮的 **ID** 值，这样就可以窗口消息处理函数中处理按钮功能了，在按钮较多，而且不需要处理按钮的其他功能时，比较有用，形式如下

```
……
```

```
#define MYBUTTON_OK_ID 101
void    HelloGUI(HAND hd,MESSAGE msg)
{
if(msg.type == GM_SYSTEM)
 switch(msg.message)
 {
 case GM_Create:
  efObj->New(hd,BUTTON,MYBUTTON_OK_ID,STY_BUTTON_NORMAL,
 "OK",50,50,120,70,NULL,NULL);
  return;
 default:return;
 }
if(msg.type == GM_COMMAND)
 switch(msg.message)
 {
 case MYBUTTON_OK_ID:
   /*
  Your code
  */
   return;
 default:return;
 }
}
……
```

完整例子：

```
/*
BUTTON - 按钮
www.ecgui.com
*/
#include "gui.h" /* GUI 的头文件 */

void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
if(msg.type == GM_SYSTEM) /* 判断消息类型，消息类型全部为大写 */
 switch(msg.message)        /* 判断消息 */
 {
 case GM_Draw: /* 对象被显示，或者重绘的时候会产生该消息 */
     {
```

```
      HAND hdc; /* HDC 句柄，指向的结构中保存作图时的一些数据，与文件操作句柄 FILE 类似 */
      hdc=efGDI->Start(hd);/* 开始作图，获得 HDC 句柄，与 fopen 函数类似 */
      efGDI->SetColor(hdc,COLOR_BLUE);/* 设置颜色 为颜色 */
       efGDI->DrawText(hdc,10,150,"DJGPP!");/* 输出字符串 */
      efGDI->End(hd,hdc);/* 完成作图，与 fclose 类似，hdc 为句柄*/
      }
        return;
 case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                    对象还没有显示。可以在这里进行一些初始化工作。
                    如果是窗口的 GM_Create 消息，可以创建新的对象，如 BUTTON 等 */
         CreatObject(hd,BUTTON,1,1,"BUTTON_A",10,25,150,45,0,0); /* 创建新的按钮对象 */
        /* efObj->New 函数原型如下
        HAND GUIAPI CreatObject( - 函数返回新创建的对象句柄
        HAND pHOST,   - 父对象句柄
        int16 guiclass,   - 要创建的对象类型
        int16 id, - 对象的 ID
        int16 style, - 对象的风格
        char   *szTitle, - 对象的 Title 标题
         int16 left,int16 top,int16 right,int16 bottom, - 对象的坐标
         void (*Server)(HAND,MESSAGE),void *pData); -对象的消息处理函数，附加数据指针
        */
         CreatObject(hd,BUTTON,2,3,"EXIT",10,120,150,140,0,0);/* 创建新的按钮对象 */
         return;
 case GM_Destroy:/* 对象将要被销毁时，产生该消息。 */
         /* 窗口中的 Button 等控件对象，系统会自动销毁 */
         return;
 case GM_Close:/* 一般情况下，是当用户点击了 窗口的关闭 按钮后 产生该消息 */
         QuitWindow(hd); /* 退出窗口 */
         return;
 default:return;
 }
if(msg.type == GM_COMMAND) /* 按钮被按下后产生的消息 */
     switch(msg.message) /* message 为按钮的 ID */
     {
     case 1:/* 按钮 "BUTTON_A" */
          return;
     case 2:/* "按钮 "BUTTON_B" */
```

```
            QuitWindow(hd); /* 退出窗口 */

            return;
      defualt:return;
      }
}

int gmain(void) /* GUI 程序 的入口函数 */
{
      /* 创建一个主窗口 */
      CreatObject(0,MAINWINDOW,1,1,"Button",10,10,630,300,MyWin,0);
}
```

4.2 单行文本编辑框

单行文本框的控件类型为 **TEXTBOX**,创建时系统默认最长可以支持 **256** 个字符**,**可以随需要动态改变。

现在介绍一下 **GUI** 提供的 **TEXTBOX** 类对象的功能模块或者可以看作是一些操作方法，作者认为可以称之为一种特殊的 组件，可能更恰当一些。

组件名称为 **efTextBox,**本质为结构指针**,**定义如下

```
PefTextBoxTag efTextBox;
```

结构原形如下

```
typedef struct _efTextBox
{
	void	GUIAPI	(*Set)(HAND,char*);
	void	GUIAPI	(*Text)(HAND,char *buf);
	void	GUIAPI	(*Insert)(HAND,char*);
	void	GUIAPI	(*SetAble)(HAND,uint8 op);
	int16	GUIAPI	(*Long)(HAND);
	void	GUIAPI	(*SetTextColor)(HAND,uint32);
	void	GUIAPI	(*SetLineColor)(HAND,uint32);
	void	GUIAPI	(*SetBackColor)(HAND,uint32);
	uint32	GUIAPI	(*TextColor)(HAND);
	uint32	GUIAPI	(*BackColor)(HAND);
	uint32	GUIAPI	(*LineColor)(HAND);
	void	GUIAPI	(*SetMax)(HAND,int16 max);
	int16	GUIAPI	(*Max)(HAND);
	int16	GUIAPI	(*GetX)(HAND);
	void	GUIAPI	(*SetX)(HAND);
}efTextBoxTag,*PefTextBoxTag;
```

这种结构定义，具有很大的灵活性，使用时也非常方便，如用过下面语句

```
efTextBox->Set(hd, "NewText");
```

就实现文本框内容的改变，您还可以这样做

```
{
PefTextBoxTag tb=efTextBox;
tb->Set(hd, "NewText");
…….
}
```

当大量使用 文本框操作时，这样简单的一行代码就避免不少重复的代码。
作者在设计 **API** 时，就希望能更好的提高开发人员的体验**(**用户体验**)**。
下面就介绍一下结构中各个函数的功能,其实从名称上就很容易得知，而且容易记忆

Set	设定内容
Text	将内容复制到 **buf** 指向的内存区域
Insert	插入内容
SetAble	**op** 为 **0** 时不可编辑，**1** 时可编辑
Long	返回内容的长度
SetTextColor	设定字体颜色
SetLineColor	设定光棒的颜色
SetBackColor	设定背景颜色

TextColor	返回字体颜色
LineColor	返回光棒的颜色
BackColor	返回背景颜色
SetMax	设定可录入字符最大长度
Max	返回可录入字符最大长度
GetX	返回光棒位置，初始值为 1
SetX	设定光棒位置

需要说明的是新版本中可能增加新的功能函数，但会尽保证兼容性。

在获得文本框内容,先通过 **efTextBox->Long** 确定内容长度，再通过 **efTextBox->Text** 将内容复制到指定的内存区域，需要说明的是，如果指定的内存空间小于内容长度，会造成溢出错误。

插入的字符串的位置由光棒位置决定

```
efTextBox->Insert(hd, "InsertText");
```

完整例子

```
/*
TextBox - 文本框
www.ecgui.com
*/
#include "gui.h" /* GUI 的头文件 */

#define BUTTON_Clear 2 /* Clear 按钮的 ID */
#define BUTTON_Get     3 /* Get 按钮的 ID */
#define BUTTON_OnOff 4 /* OnOff 按钮的 ID */
```

```
typedef struct _MyData
{
HAND text; /* 保存 TextBox 的句柄 */
char status; /* 定义一个变量来保存文本框的状态 */
}MyData,*PMyData;

void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
PMyData pWin;

if(msg.type == GM_SYSTEM) /* 判断消息类型，消息类型全部为大写 */
 switch(msg.message)      /* 判断消息 */
 {
 case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                  对象还没有显示。可以在这里进行一些初始化工作。
                 如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */
        {
        pWin=(PMyData)Gmalloc(sizeof(MyData),"");/*  分配内存 */

         efObj->SetVar(hd,(HAND)pWin);/* 设置 对象的附加数据指针 */

         pWin->text=CreatObject(hd,TEXTBOX,1,1,"Text",10,90,260,110,0,0); /*
创建新的按钮对象 */
        /* efObj->New 函数原型如下
        HAND GUIAPI CreatObject( - 函数返回新创建的对象句柄
        HAND pHOST,  - 父对象句柄
        int16 guiclass,  - 要创建的对象类型
        int16 id, - 对象的 ID
        int16 style, - 对象的风格
        char  *szTitle, - 对象的 Title 标题
         int16 left,int16 top,int16 right,int16 bottom, - 对象的坐标
         void (*Server)(HAND,MESSAGE),void *pData); -对象的消息处理函数，
附加数据指针
        */

CreatObject(hd,BUTTON,BUTTON_Clear,1,"Clear",10,120,120,140,0,0);     /*
创建新的按钮对象 */
         CreatObject(hd,BUTTON,BUTTON_Get,1,"Get",130,120,260,140,0,0);
/* 创建新的按钮对象 */
         efObj-
>New(hd,BUTTON,BUTTON_OnOff,1,"On/Off",10,160,120,180,0,0);/* 创建新
的按钮对象 */
```

```
        efTextBox->SetTextColor(pWin->text,COLOR_GREEN);/* 设定文本框字体的颜色 */
        efTextBox->SetBackColor(pWin->text,COLOR_BLACK);/* 设定文本框背景的颜色 */
        efTextBox->SetLineColor(pWin->text,COLOR_WHITE);/* 设定文本框边框及光标的颜色 */
        }
        return;
 case GM_Destroy:/* 对象将要被销毁时，产生该消息。 */
        {
        pWin=efObj->Var(hd);/* 获得 对象的附加数据指针 */
        Gfree(pWin,sizeof(MyData));/* 释放在 GM_Create 中分配的内存空间 */
        }
        /* 窗口中的 Button 等控件对象，系统会自动销毁 */
        return;
 case GM_Close:/* 一般情况下，是当用户点击了 窗口的关闭 按钮后 产生该消息 */
        QuitWindow(hd); /* 退出窗口 */
        return;
 default:return;
 }

pWin=efObj->Var(hd); /* 任何对象的消息处理函数，收到的第一个消息是 GM_Create，
                        所以收到其他消息时，pWin 句柄已经初始化完成 */
if(!pWin) return;   /* 如果得到空句柄，则不再进行后续操作 */

if(msg.type == GM_COMMAND) /* 按钮被按下后产生的消息 */
    switch(msg.message) /* message 为按钮的 ID */
    {
    case BUTTON_Clear:
        efTextBox->Set(pWin->text,"");/* 设定 文本框内容 为空，即清空文本 */
         SetObjFocus(pWin->text);/* 因为按下按钮后，焦点发生转移，这里重新 设定焦点 */
        return;
    case BUTTON_Get:
        {
        char *buf;
        int buf_long;
        buf_long=efTextBox->Long(pWin->text);/* 获得文本长度*/
        buf=(char*)Gmalloc(buf_long+1,"");/* 分配内存 */
        efTextBox->Text(pWin->text,buf);/* 获得文本框内容，复制到 buf*/
```

```
        efTextBox->Insert(pWin->text,buf);/* 将 buf 中的内容插入到文本框 光
标的位置 */
        SetObjFocus(pWin->text);/* 因为按下按钮后，焦点发生转移，这里重
新 设定焦点 */
        Gfree(buf,buf_long+1);/* 释放内存空间 */
        }
        return;
     case BUTTON_OnOff:
        {
        SetObjFocus(pWin->text);/* 因为按下按钮后，焦点发生转移，这里重
新 设定焦点 */
        if(pWin->status)/* 判断 */
            {
            efTextBox->SetAble(pWin->text,False);/* 使文本框不能编辑*/
            pWin->status=False;
            return;
            }
        else
            {
            efTextBox->SetAble(pWin->text,True);/* 回复正常编辑 */
             pWin->status=True;
            }
            /* 可以使用 pWin->status=efTextBox->Able(pWin->text); 来获得
当前的状态 */
        }
        return;
    default:return;
    }
}

int gmain(void) /* GUI 程序 的入口函数 */
{
    /* 创建一个主窗口 */
    CreatObject(0,MAINWINDOW,1,1,"TextBox",10,10,400,300,MyWin,0);
    return True;
}
```

4.3 多行文本编辑框

多行文本编辑框的控件类型为 **EDITBOX** ,组件名称为 **efEditBox.**
结构原形如下

```
typedef struct _efEditBoxTag
{
```

```
    uint8     GUIAPI    (*LoadFile)(HAND,char *filname);
    uint8     GUIAPI    (*SaveFile)(HAND,char *filename);
     void     GUIAPI    (*Insert)(HAND,char*);
     void     GUIAPI    (*SetAble)(HAND,uint8 op);
    int16     GUIAPI    (*GetX)(HAND);
    int16     GUIAPI    (*GetY)(HAND);
     void     GUIAPI    (*SetBackColor)(HAND,uint32 color);
    void     GUIAPI    (*SetTextColor)(HAND,uint32 color);
    void     GUIAPI    (*SetLineColor)(HAND,uint32 color);
    uint32    GUIAPI    (*BackColor)(HAND);
    uint32    GUIAPI    (*TextColor)(HAND);
    uint32    GUIAPI    (*LineColor)(HAND);
     int16    GUIAPI    (*LineNum)(HAND);
    int16     GUIAPI    (*LineLong)(HAND hd,int16 line);
     void     GUIAPI    (*LineText)(HAND hd,int16 line,char *buf);

}efEditBoxTag,*PefEditBoxTag;
```

功能介绍

LoadFile	读入指定文件**(filename)**的内容
SaveFile	将内容保存到文件**(filename)**
LineNum	返回行数
LineLong	返回指定行的长度
LineText	将指定行的内容复制到 **buf** 指向的内存区域

因为不少函数与 **TEXTBOX** 的功能相同，所以就不再重复介绍了

完整例子

```
/*
EditBox - 多行文本编辑框
www.ecgui.com
*/
```

```
#include "gui.h"
/*
GUI 中内置了内存分配管理模块，作用 malloc,free 相同，但功能更强大！
使用 GMemInit 函数，将一个储存空间交给 GUI 内存分配模块管理后，
再使用 Gmalloc,Gfree 函数时，GUI 可以检测内存是否正确释放，
当有释放错误时(释放大小不符，空指针释放)，或者 内存没有释放(退出窗口时)
会给出 提示窗口。
有助于开发更稳定的应用程序。
*/
char Gmemory[1024*10];/* 10kb 储存空间 */

void MyWin(HAND hd,MESSAGE msg) /* 消息处理函数 */
{
HAND editbox; /* 保存编辑框的句柄 */

    if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写 */
    switch(msg.message)/* 判断消息 */
    {
    case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                    对象还没有显示。可以在这里进行一些初始化工作。
                    如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */
        {
        GMemInit(Gmemory,1024*10); /* 储存空间 托管 */
        editbox=efObj->New(hd,EDITBOX,1,1,"",10,40,210,240,0,0);/* 创建
编辑框 */
        efObj->SetVar(hd,editbox); /* 将句柄保存到 自定义数据 ，方便访问 */
         efObj->New(hd,BUTTON,1,1,"统计",10,250,110,270,0,0);/* 创建按钮
*/
        efObj->New(hd,BUTTON,2,1,"提取第 2 行",120,250,210,270,0,0);/* 创
建按钮 */
        }
        return;
    case GM_Destroy:/* 对象将要被销毁时，产生该消息。 */
       /* 窗口中的 Button 等控件对象，系统会自动销毁 */

        return;
    case GM_Draw:
        return;
    default:
        return;
    }

editbox=efObj->Var(hd);
```

```
    if(msg.type ==GM_COMMAND)
     switch(msg.message)
     {
     case 1:/* "统计" 按钮 */
          {
          char buf[256];
          memset(buf,0,256);/* 清 0 */
           sprintf(buf,"共%d 行",(int)efEditBox->LineNum(editbox)); /* 生成制定格式的字符串 */
          MessageBox(0,"统计",buf,0); /* 消息窗口  */
          SetObjFocus(editbox);/* 设定 焦点 对象为编辑框 */
          }
          return;
      case 2:/* "提取第 2 行" 按钮  */
          {
           char *buf;
          char title[12]={"提取"};
          if(efEditBox->LineNum(editbox)<2) /* 获得编辑框行数 */
               {
               MessageBox(0,"Message","第二行不存在",0);/* 消息窗口  */
               SetObjFocus(editbox);/* 设定 焦点 对象为编辑框 */
               return;
               }
           buf=Gmalloc(efEditBox->LineLong(editbox,2)+1,""); /* 分配内存空间 */
          /*
           int16   GUIAPI   (*LineNum)(HAND);
           - 获得编辑框文本 行数
           int16   GUIAPI   (*LineLong)(HAND hd,int16 line);
           - 获得指定行的内容长度，如 efEditBox->LineLong(editbox,1); 为第一行长度
           void    GUIAPI   (*LineText)(HAND hd,int16 line,char *buf);
           - 获得指定行的内容,复制到 buf 指向的内存空间
          */
          efEditBox->LineText(editbox,2,buf);/* 获得指定行的内容 */
          memset(title+4,0,8);/* 清 0 */
          sprintf(title+4,"(%d)",efEditBox->LineLong(editbox,2));/* 生成制定格式的字符串 */
          MessageBox(0,title,buf,0);/* 消息窗口  */
          SetObjFocus(editbox);/* 设定 焦点 对象为编辑框 */
          // Gfree(buf,efEditBox->LineLong(editbox,2));

           /* 上面这行代码，为释放内存的，没有执行会造成 "内存泄漏" */
```

```
		}
		return;
	default:
		return;

	}
}

int gmain(void *data)
{
	/* 创建窗口 */
	efObj->New(0,MAINWINDOW,1,1,"EditBox",10,10,630,300,MyWin,0);
	return 0;
}
```

4.4 滚动条

GUI 中将滚动条分为两种：横向和竖向滚动条，即 **HSCROLLBAR** 和 **VSCROLLBAR**．但有趣的是，却可以使用一个组件 **efScrollBar**．

结构原形如下

```
typedef struct _efScrollBarTag
{
    void   GUIAPI   (*SetHost)(HAND,HAND host);
    void   GUIAPI   (*SetN)(HAND,int16 n);
    int16   GUIAPI   (*Value)(HAND);

}efScrollBarTag,*PefScrollBarTag;
```

与 **Windows** 中的滚动条复杂用法（**Win32 API**）相比，这个 **GUI** 提供的滚动条使用起来相当的简单方便。

功能介绍

SetHost	设定消息接受的对象，可以是窗口或者其他任意对象
SetN	设定项目数
Value	返回当前的位置值**(1- n)**

完整例子

```
/*
ScrollBar - 滚动条
www.ecgui.com
*/
#include "gui.h"

void show_value(HAND hd) /* 显示滚动条 数值*/
{
    char buf[10];
    HAND scroll;/* 句柄 */
    HDC hdc;/* 绘图句柄 */

    scroll=efObj->Var(hd);/* 获得 滚动条 句柄 */

    hdc=efGDI->Start(hd);/* 开始作图 */
    Bar(hdc,10,30,80,50,COLOR_BLUE);/* 画实心矩形,颜色为蓝色 */
    sprintf(buf,"%d\0",efScrollBar->Value(scroll));/* 生成 指定格式的字符串 */
    /* efScrollBar->Value(HAND); 返回的数值为 0~(n-1) ,n 为项目数 */
    SetColor(hdc,COLOR_WHITE);/* 设置颜色为白色 */
    DrawText(hdc,10,30,buf);/* 显示字符串 */
    efGDI->End(hd,hdc);/* 完成作图 */
}
```

```
void MyWin(HAND hd,MESSAGE msg) /* 消息处理函数 */
{
HAND scroll;/* 句柄 */
if(msg.type == GM_SYSTEM) /* 判断消息类型，消息类型全部为大写 */
      switch(msg.message)    /* 判断消息 */
      {
      case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                      对象还没有显示。可以在这里进行一些初始化工作。
                   如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */

                /* scroll=efObj->New(hd,VSCROLLBAR,1,1,"",260,50,280,170,0,0);
*/
             /* VSCROLLBAR 为竖向滚动条 HSCROLLBAR 为横向滚动条 */
             scroll=efObj->New(hd,HSCROLLBAR,1,1,"",90,50,260,70,0,0);
            efScrollBar->SetN(scroll,6); /* 设置项目数 n ,n >=2 */
            /* efScrollBar->SetN(scroll,99999999); */
            /* efScrollBar->SetN(scroll,999); */
              /* efScrollBar->SetN(scroll,10); */
              /* efScrollBar->SetN(scroll,2); */
            efScrollBar->SetHost(scroll,hd); /* 设置滚动消息(GM_Scrolling 等) 接
收对象 */
            efObj->SetVar(hd,scroll);/* 将句柄 保存到自定义数据，方便访问 */

            return;
     case GM_Draw:/* 对象被显示，或者重绘的时候会产生该消息 */
              show_value(hd);/* 调用 显示滚动条 数值的函数 */
           return;
      default:
            return;
      }
scroll=efObj->Var(hd);
if(msg.type == GM_CTRL)/* 控件类 消息*/
      switch(msg.message)
      {
      case GM_Scrolling:/* 滚动条发生滚动 */
             show_value(hd);/* 调用 显示滚动条 数值的函数 */
            return;
      default:
            return;
      }

}
```

```
int gmain(void *data)
{
    /* 创建窗口 */
    efObj->New(0,MAINWINDOW,1,1,"ScrollBar",10,10,630,300,MyWin,0);
    return 0;
}
```

4.5 进度条

进度条也是一个比较简单的控件，控件类型为 **SPEEDBAR**，组件名称为 **efSpeedBar .**

```
typedef struct _efScrollBarTag
{

    void    GUIAPI    (*SetN)(HAND,int16 n);
    void    GUIAPI    (*SetValue)(HAND,int16 value);

}efSpeedBarTag,*PefSpeedBarTag;
```

功能介绍

SetN	设定项目数
SetValue	设定当前值

完整例子

```
/*
```

```
SpeedBar  - 进度条
www.ecgui.com
*/
#include "gui.h" /* GUI 的头文件 */

void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
HAND button;/* 句柄 */
HAND speedbar;/* 句柄 */
if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写 */
    switch(msg.message)/* 判断消息 */
    {
    case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                    对象还没有显示。可以在这里进行一些初始化工作。
                  如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */
        {
        /* 创建 进度条 对象 */
         speedbar=efObj->New(hd,SPEEDBAR,1,1,"",60,100,280,120,0,0);
        efSpeedBar->SetN(speedbar,100);/* 设置进度条项目数 */

         /* 创建按钮对象 */
        button=efObj->New(hd,BUTTON,2,1,"0",60,130,100,150,0,0);
        efObj->SetVar(button,speedbar);/* 将 进度条对象 的句柄设置为 按钮
的自定义数据*/
         /* 创建按钮对象 */
         button=efObj->New(hd,BUTTON,3,1,"25",105,130,145,150,0,0);
        efObj->SetVar(button,speedbar);/* 将 进度条对象 的句柄设置为 按钮
的自定义数据*/
         /* 创建按钮对象 */
        button=efObj->New(hd,BUTTON,4,1,"50",150,130,190,150,0,0);
        efObj->SetVar(button,speedbar);/* 将 进度条对象 的句柄设置为 按钮
的自定义数据*/
         /* 创建按钮对象 */
        button=efObj->New(hd,BUTTON,5,1,"75",195,130,235,150,0,0);
        efObj->SetVar(button,speedbar);/* 将 进度条对象 的句柄设置为 按钮
的自定义数据*/
         /* 创建按钮对象 */
        button=efObj->New(hd,BUTTON,6,1,"100",240,130,280,150,0,0);
        efObj->SetVar(button,speedbar);/* 将 进度条对象 的句柄设置为 按钮
的自定义数据*/
        }
        return;
    default:return;
```

```
	}
if(msg.type == GM_COMMAND)/* 按钮被触发 */
	switch(msg.message)/* 按钮 ID*/
	{
	default:
		button=(HAND)msg.pData;/* msg.pData 中为按钮的句柄 */
		speedbar=efObj->Var(button);/* 获得 进度条 对象的句柄 */
		efSpeedBar->SetValue(speedbar,((int)msg.message-2)*25);/* 设置进度条的但前值 */
		return;
	}
}
int gmain(void *data)
{
	/* 创建窗口 */
	efObj->New(0,MAINWINDOW,1,1,"SpeedBar",10,10,630,300,MyWin,0);
	return 0;
}
```

4.6 文字标签

文字标签的控件类型为 **LABEL,**组件名称为 **efLabel.**

结构原形为

```
typedef struct _efScrollBarTag
{
	void	GUIAPI	(*SetLineColor)(HAND,uint32);
	void	GUIAPI	(*SetTextColor)(HAND,uint32);
	void	GUIAPI	(*SetBackColor)(HAND,uint32);
	void	GUIAPI	(*UnderLine)(HAND,uint8 op);

}efLabelTag,*PefLabelTag;
```

功能介绍

SetTextColor	设定字体颜色
SetBackColor	设定背景颜色
UnderLine	**op** 为 **1** 下划线 **2** 删除线 **0** 恢复无线状态

有趣的是可以使用这个组件实现简单的超连接模拟

```
……
if(msg.type == GM_MOUSE)
 switch(msg.message)
 {
 ……
 case GM_MouseOver:
    efLabel->SetLineColor(hd,COLOR_BLUE);
    efLabel->UnderLine(hd,1);
    return;
 case GM_MouseAway:
    efLabel->UnderLine(hd,0);
   return;
……
 }
……
```

完整例子

```
/*
Label  - 标签
www.ecgui.com
*/
#include "gui.h" /* GUI 的头文件 */

void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
HAND label;/* 句柄 */
HAND button;/* 句柄 */
if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写  */
     switch(msg.message)/* 判断消息 */
```

```
    {
    case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                     对象还没有显示。可以在这里进行一些初始化工作。
                     如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */
        {/* LABEL 对象创建时，会自动根据字符串长度，改变自身对象的长度*/
        label=efObj->New(hd,LABEL,1,1," 窗 口 的 消 息 处 理 函 数
",10,30,100,50,0,0);

        button=efObj->New(hd,BUTTON,2,1," 恢 复 默 认 ",10,60,100,82,0,0);/*
创建按钮 */
         efObj->SetVar(button,label);/* 将标签对象的句柄保存到 按钮的自定
义数据，方便访问 */

         button=efObj->New(hd,BUTTON,3,1,"      黑      底      白      字
",110,60,200,82,0,0);/* 创建按钮 */
         efObj->SetVar(button,label);/* 将标签对象的句柄保存到 按钮的自定
义数据，方便访问 */

        }
        return;
    default:return;
    }

if(msg.type == GM_COMMAND)/* 按钮被触发 */
    switch(msg.message)/* 按钮 ID*/
    {
    case 2:/* "恢复默认" 按钮*/
         {
        button=(HAND)msg.pData;/* msg.pData 中为按钮的句柄 */
        label=efObj->Var(button);/* 获得标签的句柄 */
        efLabel->SetBackColor(label,COLOR_LIGHTGRAY);/* 设定背景颜色
*/
        efLabel->SetTextColor(label,COLOR_BLACK);/* 设定字体颜色 */
        }
        return;
    case 3:/* "白底黑字" 按钮*/
        {
         button=(HAND)msg.pData;/* msg.pData 中为按钮的句柄 */
        label=efObj->Var(button);/* 获得标签的句柄 */
        efLabel->SetBackColor(label,COLOR_BLACK);/* 设定背景颜色 */
        efLabel->SetTextColor(label,COLOR_WHITE);/* 设定字体颜色 */
        }
        return;
```

```
    default:return;
    }
}

int gmain(void *data)
{
    /* 创建窗口 */
    efObj->New(0,MAINWINDOW,1,1,"Label",10,10,630,300,MyWin,0);
    return 0;
}
```

5.0 下拉菜单

下拉菜单是最常见的菜单，也可以称之为 主菜单，控件类型为 **MAINMENU,**组件名称为 **efMainMenu .**

结构原形为

```
typedef struct _efMainMenuTag
{

void    GUIAPI    (*Add)(HAND hd,PMENU Menu);

}efMainMenuTag,*PefMainMenuTag;
```

其中结构 **MENU** 原形为

```
typedef struct _MENU{
int16 id;
char * szTitle;
int16 key;                  /* Keyboard */
void *pData;
}MENU,*PMENU;
```

其中 **pData** 为附加数据，一般不用,以下是简单的例子

```
const MENU mymenu[]={
{0,"a-字母-a",0,0},
{0,"b-字母-b",0,0},
{0,"c-字母-c",0,0},
{0,"d-字母-d",0,0},
{0,"e-字母-e",0,0}
};
……
case GM_Create:
{
HAND hm=NULL;
int i=0;
hm=CreatObject(hw,MAINMENU,1,1,"File",10,30,80,50,NULL,NULL);
for(i=0;i<5;i++)
efMainMenu->Add(hm,&mymenu[i]);
}
……
```

一般来说，菜单的内容是不需要动态改变的，所以使用 **const** 关键字**.**
当菜单中的子项被选中后， **GUI** 会发送 **GM_CTRL** 消息，消息结构中 **message** 储存子项的 **ID** 值**.**

5.1 弹出式菜单

弹出式也是一种重要的菜单形式，其控件类型为 **POPMENU,**组件名称为 **efPopMenu.**

结构原形为

```
typedef struct _efPopMenuTag
{

void   GUIAPI   (*Add)(HAND hd,PMENU Menu);
void   GUIAPI   (*Pop)(HAND hd,int16 x,int16 y);

}efPopMenuTag,*PefPopMenuTag;
```

功能介绍

Add	添加菜单子项
Pop	在**(x,y)**处弹出菜单

用法举例

```
void    HelloGUI(HAND hd,MESSAGE msg)
{
```

```
 ……
 case GM_Create:
  {
   HAND hm;
   hm=CreatObject(hw,POPMENU,1,1,"",170,30,240,50,NULL,NULL);
    for(i=0;i<5;i++)
    AddMenu(hm,&mydmenu[i]);
   efObj->SetVar(hd,hm);
  }
 ……
case GM_RightDown:
  {
  HAND hm;
  hm=efObj->Var(hd);
  if(hm == NULL) return;
  efPopMenu->Pop(hm,msg.icode,msg.jcode); /* mouse x,y */
  }
……
}
```

右键弹出菜单

实现效果

a-字母-a
b-字母-b
c-字母-c
d-字母-d
a-字母-a
b-字母-b
c-字母-c

完整例子

```
/*
MainMenu，PopMenu - 菜单
www.ecgui.com
*/
#include "gui.h" /* GUI 的头文件 */

#define FILE_ITEM_NUM 5 /* 项目数*/
#define EDIT_ITEM_NUM 3
```

```
MENU mymenu_file[]={ /* 菜单项目 */
{0,"New",0,0},
{1,"Open",0,0},
{2,"Save",0,0},
{3,"Save as",0,0},
{4,"Exit",0,0}
};

MENU mymenu_edit[]={ /* 菜单项目 */
{0,"Copy",0,0},
{1,"Cut",0,0},
{2,"Paste",0,0}
};

void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
HAND box;/* 句柄 */
int i;
if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写 */
    switch(msg.message)/* 判断消息 */
    {
    case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                对象还没有显示。可以在这里进行一些初始化工作。
              如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */


              efObj-
>New(hd,BUTTON,1,1,"BUTTON_A",10,100,120,120,0,0);

                /* 创建一个下拉菜单(主菜单)*/
              box=efObj->New(hd,MAINMENU,1,1,"File",10,50,70,70,0,0);

              for(i=0;i<FILE_ITEM_NUM;i++)
               efMainMenu->Add(box,&mymenu_file[i]); /* 添加菜单项目*/

                /* 创建一个下拉菜单(主菜单)*/
                box=efObj-
>New(hd,MAINMENU,2,1,"Edit",80,50,150,70,0,0);
              for(i=0;i<EDIT_ITEM_NUM;i++)
               efMainMenu->Add(box,&mymenu_edit[i]); /* 添加菜单项目*/

                /* 创建一个弹出式菜单 */
```

```
                box=efObj->New(hd,POPMENU,3,1,"",80,50,150,70,0,0);
                for(i=0;i<EDIT_ITEM_NUM;i++)
                 efPopMenu->Add(box,&mymenu_edit[i]); /* 添加菜单项目*/
                /* 弹出式菜单创建后，并不显示。
                当使用 efPopMenu->Pop(HAND,int x,int y); 激活后，就会出现.
                一但鼠标点击了 菜单以外的区域，菜单就会马上消失。
                等待下次激活。
                */
                efObj->SetVar(hd,box);/* 句柄保存到 自定义数据 */

        return;
    default:return;
    }
if(msg.type == GM_CTRL && msg.message ==GM_MenuItem)
    {
      /* 菜单的某一项目被单击后，将产生 GM_CTRL 类 GM_MenuItem 消息,
         msg.icode 中保存 菜单对象的 ID, msg.jcode 中保存 菜单子项目的
ID */
        if((int)msg.icode==1)
        MessageBox(0,"菜单",mymenu_file[(int)msg.jcode].szTitle,0);

        if((int)msg.icode==2)
         MessageBox(0," 菜 单 ",mymenu_edit[(int)msg.jcode].szTitle,0);


        if((int)msg.icode==3)
         MessageBox(0," 弹 出 菜 单 ",mymenu_edit[(int)msg.jcode].szTitle,0);

        return;


    }
if(msg.type == GM_MOUSE && msg.message ==GM_RightDown)
    {
     box=efObj->Var(hd);
    efPopMenu->Pop(box,(int)msg.icode,(int)msg.jcode);
    }
}

int gmain(void *data)
{
    /* 创建窗口 */
    efObj->New(0,MAINWINDOW,1,1,"Menu",10,10,630,300,MyWin,0);
    return 0;
```

```
}
```

5.2 组合框

组合框是由文本编辑框，按钮还有列表显示框组合构成。控件类型为 **COMBOBOX,**组件名称为 **efComboBox.**

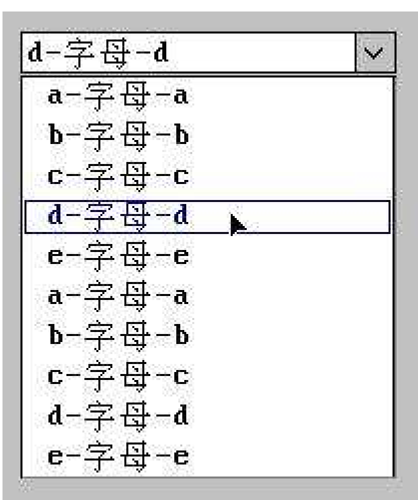
结构原形为

```
typedef struct _efCOMBOBOXTag
{

void    GUIAPI    (*Add(HAND hd,PMENU Menu);
int16    GUIAPI    (*Get)(HAND hd);

}efComboBoxTag,*PefComboBoxTag;
```

功能说明

Add	添加子项
Get	返回当前选中的子项 **ID** 值

实现效果



完整例子

```
/*
ComboBox - 组合框
www.ecgui.com
*/
#include "gui.h" /* GUI 的头文件 */

#define ITEM_NUM 3 /* 项目数*/

const MENU mymenu[]={
{0,"苹果",0,0},
{1,"香蕉",0,0},
{2,"西瓜",0,0}
};

void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
HAND box;/* 句柄 */
int i;
if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写 */
	switch(msg.message)/* 判断消息 */
	{
	case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
				对象还没有显示。可以在这里进行一些初始化工作。
			如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */

				box=efObj-
>New(hd,COMBOBOX,1,1,"Combobox",10,50,210,70,0,0);
			efObj->SetVar(hd,box);
			for(i=0;i<ITEM_NUM;i++)
			 AddMenu(box,&mymenu[i]);

		return;
	default:return;
	}
if(msg.type == GM_CTRL &&msg.message == GM_MenuItem && (int)msg.ico
```

```
de ==1)
        {
        PMENU pm=(PMENU)msg.pData;
        MessageBox(hd,"Message",pm->szTitle,0);
        }
if(msg.type == GM_MOUSE && msg.message ==GM_RightDown)
        {
        box=efObj->Var(hd);
        i=efComboBox->Get(box);
        MessageBox(hd,"Message",mymenu[i].szTitle,0);
        }
}

int gmain(void *data)
{
        /* 创建窗口 */
        efObj->New(0,MAINWINDOW,1,1,"ComboBox",10,10,630,300,MyWin,0);
        return 0;
}
```

5.3 单选框

单选框控件类为 **CHOICEBOX**，组件名称是 **efChoiceBox.**

结构原形

```
typedef struct _efChoiceBoxTag
{

void    GUIAPI    (*Add)(HAND hd,PMENU Menu);
int16    GUIAPI    (*Get)(HAND);

}efChoiceBoxTag,*PefChoiceBoxTag;
```

功能说明

Add	添加子项
Get	返回当前选中的子项 **ID** 值

一般来说，系统根据创建时对象的大小来判断选项是横向排列，还是竖向排列.

当 **right-left > bottom-top** 时 横向排列
当 **bottom-top > right-left** 时 竖向排列

实现效果

完整例子

```
/*
ChoiceBox  - 单选框
www.ecgui.com
*/
#include "gui.h" /* GUI 的头文件 */
#define ITEM_NUM 5
MENU choice_menu[]={
{0,"A. C 语言",0,0},
{1,"B. C++语言",0,0},
{2,"C. Java 语言",0,0},
{3,"D. C#  语言",0,0},
{4,"E. Pascal 语言",0,0},
{5,"F. Basic 语言",0,0}
};

void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
int i;
HAND choice;
if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写  */
    switch(msg.message)
    {
    case GM_Create:
```

```
            choice=efObj->New(hd,CHOICEBOX,1,1,"",10,30,200,200,0,0);

            for(i=0;i<ITEM_NUM;i++)
             efChoiceBox->Add(choice,&choice_menu[i]);/* 添加子项目 */

            efObj->SetVar(hd,choice);/* 将句柄保存到 自定义数据 ，方便访问 */

            efObj->New(hd,BUTTON,2,1,"统计",210,60,300,80,0,0);

            return;
      default:
            return;

      }
if(msg.type == GM_COMMAND)/* 按钮被按下后产生的消息 */
      switch(msg.message)/* message 为按钮的 ID */
      {
      case 2:
            {
            char buf[256];
            choice=efObj->Var(hd);/* 获得句柄 */
             i=efChoiceBox->Get(choice);/* 获得选择项目的 ID*/
            if(i==-1)
                  {
                  MessageBox(hd,"错误","程序错误！",0);
                  return;
                  }
            sprintf(buf,"您选择了%s",choice_menu[i].szTitle);
            MessageBox(hd,"Message",buf,0);
            }
            return;
      default:
            return;
      }
}
int gmain(void *data)/* GUI 程序 的入口函数 */
{
      /* 创建窗口 */
      efObj->New(0,MAINWINDOW,1,1,"ChoiceBox",10,10,630,300,MyWin,0);
      return 0;
}
```

5.4 多选框

与单选框不同的是，多选框每个选项都是相对独立，所以完全将每个选项当做一个单独的对象来处理。

控件类型为 **CHECKBOX** **,**组件名称为 **efCheckBox.**

结构原形为

```
typedef struct _efCheckBoxTag
{

uint8   GUIAPI (*Get)(HAND);

}efCheckBoxTag,*PefCheckBoxTag;
```

功能说明

Get	返回 **1** 选中 **0** 没选

实现效果如下

- [ ] Windows
- [x] Linux
- [ ] Unix
- [x] FreeBSD
- [x] Mac OS

完整例子

```
/*
CheckBox   - 多选框
www.ecgui.com
*/
```

```
#include "gui.h" /* GUI 的头文件 */
typedef struct _MyWinHand
{
HAND linux;
HAND windows;
HAND mac;
HAND dos;
}MyWinHand,*PMyWinHand;
void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
PMyWinHand winhand;
if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写 */
      switch(msg.message)/* 判断消息 */
      {
      case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                           对象还没有显示。可以在这里进行一些初始化工作。
                        如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */
            winhand=(PMyWinHand)Gmalloc(sizeof(MyWinHand),"");
            winhand->linux=efObj-
>New(hd,CHECKBOX,1,1,"Linux",10,50,100,70,0,0);
             winhand->windows=efObj-
>New(hd,CHECKBOX,2,1,"Windows",10,75,100,95,0,0);
             winhand->mac=efObj->New(hd,CHECKBOX,3,1,"Mac OS
X",10,100,100,120,0,0);
             winhand->dos=efObj-
>New(hd,CHECKBOX,4,1,"DOS",10,125,100,145,0,0);

            efObj->New(hd,BUTTON,5,1,"统计",10,160,100,180,0,0);
            efObj->SetVar(hd,winhand);
            return;
     case GM_Destroy:/* 对象将要被销毁时，产生该消息。 */
          /* 窗口中的 Button 等控件对象，系统会自动销毁 */
           winhand=efObj->Var(hd);
            Gfree(winhand,sizeof(MyWinHand));
           return;
      default:
            return;
      }
winhand=efObj->Var(hd);
if(msg.type == GM_COMMAND)/* 按钮被按下后产生的消息 */
      switch(msg.message)/* message 为按钮的 ID */
      {
      case 5:
```

```
            {
            char buf[256];
            int n=0;
              memset(buf,0,256);
            if(efCheckBox->Get(winhand->linux)) n++;
              if(efCheckBox->Get(winhand->windows)) n++;
            if(efCheckBox->Get(winhand->mac)) n++;
            if(efCheckBox->Get(winhand->dos)) n++;
            sprintf(buf,"您选择了%d 个操作系统",n);
            MessageBox(hd,"Message",buf,0);
            }
            return;
      default:
            return;
      }

}
int gmain(void *data)
{
      /* 创建窗口 */
      efObj->New(0,MAINWINDOW,1,1,"CheckBox",10,10,630,300,MyWin,0);
      return 0;
}
```

5.5 列表框

列表框也可以称之为列表选择框，控件类型为 **SELBOX**，组件名称为 **efSelBox.**

结构原形为

```
typedef struct _efCheckBoxTag
{
void     GUIAPI    (*Add)(HAND hd,PMENU Menu);
```

```
void     GUIAPI    (*Set)(HAND,uint8 op);
int16     GUIAPI    (*Get)(HAND);

}efSelBoxTag,*PefSelBoxTag;
```

功能介绍

Add	添加子项
Set	设定选择模式， **0** 单选，**1** 多选
Get	单选时，返回选择的子项 **ID**

列表框也有单选和多选之分，可以在处理 **GM_Create** 消息时设定。

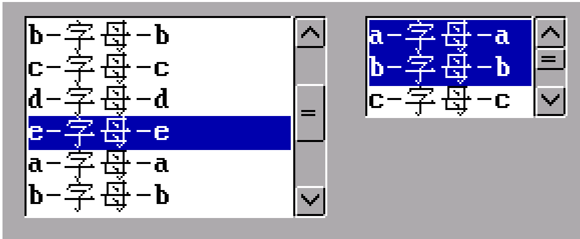
说要说明的是当设定为多选模式时，系统会修改子项的结构中的 **key** 值，所以在使用时，所以子项的储存模式必须为私有数据模式。
可以这样实现**(**部分代码**)**

```
……
case GM_Create:
 {
 PMENU pm;

 pm=(PMENU)Gmalloc(sizeof(MENU)*2, "SelBox");
 pm->id=201;
 pm->szTitle="No.1";
 pm++;
 pm->id=202;
 pm->szTitle="No.2";
 pm--;
 efObj->SetVar(hd,pm);
 }
 return;

case GM_Destroy:
 {
 PMENU pm;
 pm=(PMENU)efObj->Var(hd);
 if(pm == NULL) return;
 Gfree(pm,sizeof(MENU)*2);
 }
 return;
……
```

当选择模式为多选时，只需判断子项的 **key** 值，为 **1** 时被选中，为 **0** 未被选

中.

实现效果



完整例子

```
/*
SelBox - 选择框
www.ecgui.com
*/
#include "gui.h" /* GUI 的头文件 */

#define ITEM_NUM 6 /* 项目数*/

/*
GUI 中内置了内存分配管理模块，作用 malloc,free 相同，但功能更强大！
使用 GMemInit 函数，将一个储存空间交给 GUI 内存分配模块管理后，
再使用 Gmalloc,Gfree 函数时，GUI 可以检测内存是否正确释放，
当有释放错误时(释放大小不符，空指针释放)，或者 内存没有释放(退出窗口时)
会给出 提示窗口。
有助于开发更稳定的应用程序。
*/

char Gmemory[1024*10];/* 10kb 储存空间 */

const MENU mymenu[]={/* 选择框与菜单的子项结构相同，但内容使用上有差
异 */
{0,"苹果",0,0},
{1,"香蕉",0,0},
{2,"西瓜",0,0},
{3,"葡萄",0,0},
{4,"杏子",0,0},
{5,"菠萝",0,0}
};
```

```
void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
HAND box,button;/* 句柄 */
PMENU pm;
int i;
if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写 */
    switch(msg.message)/* 判断消息 */
    {
    case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                    对象还没有显示。可以在这里进行一些初始化工作。
                   如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */

                    GMemInit(Gmemory,1024*10); /* 储存空间 托管 */

                   {
                   /* 分配 ITEM_NUM*2 个菜单子项的空间，供两个 选择框
(SelBox) 使用 */
                   pm=(PMENU)Gmalloc(sizeof(MENU)*ITEM_NUM*2,"");

                     memcpy(pm,mymenu,sizeof(MENU)*ITEM_NUM);/* 复制内
容 */

memcpy(pm+ITEM_NUM,mymenu,sizeof(MENU)*ITEM_NUM);/* 复制内容 */

                     efObj->SetVar(hd,(HAND)pm);/* 设定为对象 自定义数据，
方便在 GM_Destroy 时释放 */

                   box=efObj->New(hd,SELBOX,1,1,"",10,50,210,100,0,0); /* 创
建 选择框 对象 */

                   for(i=0;i<ITEM_NUM;i++,pm++)
                    efSelBox->Add(box,pm); /* 添加子项 */

                     efSelBox->Set(box,1); /* 设置为多选，如果不设置，则默认
为单选 */

                   button=efObj->New(hd,BUTTON,3,1,"          统          计
A",10,110,210,130,0,0);/* 创建按钮 */
                   efObj->SetVar(button,box);/* 将选择框句柄设置为按钮的 自定
义数据 */

                     box=efObj->New(hd,SELBOX,2,1,"",220,50,310,100,0,0);/*
创建 选择框 对象 */
```

```
                    for(i=0;i<ITEM_NUM;i++,pm++)
                     AddMenu(box,pm);/* 添加子项 */

                      button=efObj->New(hd,BUTTON,4,1,"              统              计
B",220,110,310,130,0,0);/* 创建按钮 */
                    efObj->SetVar(button,box);/* 将选择框句柄设置为按钮的 自定
义数据 */
                     }
          return;
      case GM_Destroy:
          {
          void *pm;
          pm=efObj->Var(hd);/* 获得自定义数据指针 */
          Gfree(pm,sizeof(MENU)*ITEM_NUM*2); /* 释放占用的空间 */
          }
          return;
     default:return;
     }

pm=(PMENU)efObj->Var(hd);/* 获得自定义数据指针 */

if(msg.type ==GM_COMMAND)/* 按钮被触发 */
     switch(msg.message)/* 按钮 ID*/
     {
     case 3:/* "统计 A" 按钮*/
           {
          char buf[256];
          int j;
          memset(buf,0,256); /* 被选择的子项 key 值为 1 ，未选则为 0 */
          for(i=0,j=0;i<ITEM_NUM;i++,pm++)
               if(pm->key) j++; /* 统计选择的子项数 */
           sprintf(buf,"您选择了%d 项",j);/* 生成指定格式的字符串 */
          MessageBox(hd,"统计 A",buf,0);
          }
          return;
     case 4:/* "统计 B" 按钮*/
          {
          box=efObj->Var(msg.pData);

          i=efSelBox->Get(box);/* 返回选择的子项 ID,没有选或者出错返回 -1 */

          if(i==-1)
               {
```

```
                MessageBox(hd,"Message","没有选",0);
                return;
                }
            MessageBox(hd,"选中",(pm+i)->szTitle,0);

          }
          return;
     default:return;
     }
}

int gmain(void *data)
{
     /* 创建窗口 */
     efObj->New(0,MAINWINDOW,1,1,"SelBox",10,10,630,300,MyWin,0);
     return 0;
}
```

6.0 对话窗口

对话窗口是一种特殊的窗口，从属于主窗口，与主窗口在功能上基本相同。控件类型为 **DLGWINDOW,** 组件名称为 **efDialogWin .**

结构原形为

```
typedef struct _efDiallogWinTag
{

void GUIAPI (*Start)(HAND pHOST,
int16 id,
int16 style,
char    *szTitle,
int16 left,int16 top,int16 right,int16 bottom,
void (*Server)(HAND,MESSAGE),void *pData);

void GUIAPI (*Quit)(HAND dlg);

}efDialogWinTag,*PefDialogWinTag;
```

功能说明

Start	与 **efObj->New** 函数基本相同，少了一个 **guiclass** 参数
Quit	可以结束一个对话窗口

举例说明用法

```
/*
Dialog - 对话窗口及 消息窗口(MessageBox)
www.ecgui.com
*/
#include "gui.h"    /* GUI 的头文件 */

void MyDlg(HAND hd,MESSAGE msg)/* 窗口的消息处理函数 */
{
if(msg.type == GM_SYSTEM)/* 判断消息类型，消息类型全部为大写    */
switch(msg.message)/* 判断消息 */
    {
    case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                        对象还没有显示。可以在这里进行一些初始化工作。
                    如果是窗口的 GM_Create 消息， 可以创建新的对象， 如
BUTTON 等 */
          {
                efObj->New(hd,LABEL,1,1," 窗 口 的 消 息 处 理 函 数
",10,30,100,50,0,0);
          efObj->New(hd,BUTTON,1,1,"BUTTON_A",50,60,180,90,0,0);
        /* efObj->New 函数原型如下
        HAND GUIAPI CreatObject( - 函数返回新创建的对象句柄
        HAND pHOST,    - 父对象句柄
        int16 guiclass,    - 要创建的对象类型
```

```
        int16 id, - 对象的 ID
        int16 style, - 对象的风格
        char   *szTitle, - 对象的 Title 标题
         int16 left,int16 top,int16 right,int16 bottom, - 对象的坐标
        void (*Server)(HAND,MESSAGE),void *pData); -对象的消息处理函数，
附加数据指针
        */

          efObj->New(hd,BUTTON,2,1,"BUTTON_B",50,100,180,130,0,0);
          }
          return;
     default:return;
     }
if(msg.type == GM_COMMAND)
     switch(msg.message)
     {
     case 1:/* BUTTON_A 的 ID */
          {
          MessageBox(hd,"Command","BUTTON_A",0);
          /* MessageBox 函数原型如下
          uint32 MessageBox(HAND hd,char *title,char *string,uint32 type);
          - hd 为父对象的句柄，一般为主窗口或者对话窗口的句柄
          - title 消息窗口的标题
            - string 消息窗口中的内容
          - type 消息窗口的类型
            目前支持如下四种(也是最常用的:)
           MB_OK     只有一个 OK(或 确定) 按钮
            MB_OKCANCEL 有 OK(确定) 和 CANCEL(取消) 两个按钮
            MB_YESNOCANCEL 有 YES(是),NO(否),CANCEL(取消) 三个按钮
            MB_YESNO 有 YES(是),NO(否) 两个按钮

          函数返回：
          IDOK 按下 OK(确定)按钮
          IDYES 按下 YES(是)按钮
          IDNO 按下 NO(否)按钮
          IDCANCEL 按下 CANCEL(取消)按钮

          */
          }
          return;/* BUTTON_B 的 ID */
     case 2:
          {
          MessageBox(hd,"Command","BUTTON_B",0);
          }
```

```
            return;
      default:return;
      }
}
void MyWin(HAND hd,MESSAGE msg)
{
if(msg.type == GM_SYSTEM)
switch(msg.message)
      {
      case GM_Create:
            {
            efObj->New(hd,BUTTON,1,1,"Dialog",50,50,180,80,0,0);
            efObj->New(hd,BUTTON,2,1,"MessageBox",50,100,180,130,0,0);
            }
            return;
        case GM_Close:
            {
            if(MessageBox(hd,"Message","    确    定    要    退    出    么    ？
",MB_YESNO)==IDYES)
            QuitWindow(hd);/*  退出窗口  */
            return;
            }
      default:return;
      }
if(msg.type == GM_COMMAND)
      switch(msg.message)
      {
      case 1:/*  按钮  "Dialog" */
            {
            /*  启动一个对话窗口,参数与 efObj->New 基本相同,
            只却省  对象类别,因为类别已确定为：对话窗口  */
            efDialogWin->Start(hd,1,1,"Dialog",30,60,240,240,MyDlg,0);
            }
            return;
       case 2:/*  按钮  "MessageBox" */
            {
            MessageBox(hd,"Message","Hello,World!",0);
            }
      default:return;
      }
}

int gmain(void)
{
```

```
/* 窗口风格设置为 STY_WINDOW_CS 后,可以收到 GM_Close 消息,在确定是否退出 */
efObj->New(0,MAINWINDOW,1,STY_WINDOW_CS,"Dialog Test!",10,10,640,300,MyWin,0);
return 0;
}
```

6.1 定时器

定时器虽然不是一个 **GUI** 对象，但这里仍然提供一个组件，名称为 **efTimeCall.**

结构原形为

```
typedef struct _efTimeCallTag
{

int16 (*Add)(
	HAND hd,
	int16 id,
	void	(*Server)(),
	uint8	isclass,
	uint8	isontime,
	uint32	waittime);
int16 (*Del)(HAND hd,int16 id);
int16 (*Start)(HAND hd,int16 id);
int16 (*Stop)(HAND hd,int16 id);

}efTimeCallTag,*PefTimeCallTag;
```

功能说明

Add	添加一个定时器 **id** 每个定时器应有个不相同的 **ID** **isclass** 是否为对象类函数 **isontime** 暂时无效参数 **waittime** 时间间隔，值为 **100** 则表示 **1** 秒钟产生一次
Del	销毁一个定时器
Start	启动一个定时器
Stop	暂停一个定时器

举例说明

```
void    HelloTextBox(HAND hd,MESSAGE msg)
{
if(msg.type == GM_SYSTEM && msg.message == GM_Create)
     {
     efObj->SetVar(hd,0);
     }
if(msg.type == GM_TIMECALL)
     switch(msg.message)
     {
     case 501:
          {
          char text[2]={'0','\0'};
          int16 a;
          a=(int16)efObj->Var(hd);
          if(a == 9) efObj->SetVar(hd,0);
          else
               {
               a++;
               efObj->SetVar(hd,(void*)a);
               }
            text[0]+=a;
            efTextBox->Set(hd,text);
          }
          return;
     default:return;
     }
}

void    HelloGUI(HAND hd,MESSAGE msg)
{
if(msg.type == GM_SYSTEM)
  switch(msg.message)
     {
     case GM_Create:
       {
       HAND hdtext;                                                    hdtext=efObj-
       >New(hd,TEXTBOX,1,1,"0",10,50,100,70,HelloTextBox,NULL);
       efTimeCall->Add(hd,501,HelloTextBox,0,0,100);
      efTimeCall->Start(hd,501);
      }
     return;
    case GM_Destroy:
          {
```

```
            efTimeCall->Del(hd,501);
            }
            return;
      default:return;
      }
}
int gmain(void*data)
{
efObj->New(0,MAINWINDOW,1,1,"HelloGUI",10,10,400,400,HelloGUI,0);
}
```

6.2 MessageBox　消息窗口的使用

例子

```
/*
MessageBox - 消息窗口
www.ecurb2006.com
*/
#include "gui.h" /* GUI 的头文件 */

void MyWin(HAND hd,MESSAGE msg) /* 窗口的消息处理函数 */
{
int i;
if(msg.type == GM_SYSTEM) /* 判断消息类型，消息类型全部为大写 */
 switch(msg.message)          /* 判断消息 */
 {
 case GM_Create:/* 对象被创建后产生该消息，一般情况下，收到该消息时，
                    对象还没有显示。可以在这里进行一些初始化工作。
                 如果是窗口的 GM_Create 消息，可以创建新的对象，如
BUTTON 等 */
         {
              /* efObj->New 函数原型如下
         HAND GUIAPI CreatObject( - 函数返回新创建的对象句柄
         HAND pHOST,   - 父对象句柄
         int16 guiclass,   - 要创建的对象类型
         int16 id, - 对象的 ID
         int16 style, - 对象的风格
         char   *szTitle, - 对象的 Title 标题
          int16 left,int16 top,int16 right,int16 bottom, - 对象的坐标
         void (*Server)(HAND,MESSAGE),void *pData); -对象的消息处理函数，
附加数据指针
         */
              efObj->New(hd,BUTTON,1,1,"MB_OK",10,50,210,75,0,0);
                efObj-
>New(hd,BUTTON,2,1,"MB_OKCANCEL",10,80,210,105,0,0);
```

```
            efObj->New(hd,BUTTON,3,1,"MB_YESNO",10,110,210,135,0,0);
            efObj-
>New(hd,BUTTON,4,1,"MB_YESNOCANCEL",10,140,210,165,0,0);

        }
        return;
  default:return;
  }
if(msg.type == GM_COMMAND)/* 按钮被按下后产生的消息 */
    switch(msg.message)/* message 为按钮的 ID */
    {
    case 1:/* "MB_OK" 按钮 */
        MessageBox(hd,"Message","MB_OK",MB_OK);
    /* MessageBox 函数原型如下
        uint32 MessageBox(HAND hd,char *title,char *string,uint32 type);
        - hd 为父对象的句柄，一般为主窗口或者对话窗口的句柄
        - title 消息窗口的标题
         - string 消息窗口中的内容
        - type 消息窗口的类型
         目前支持如下四种(也是最常用的:)
         MB_OK     只有一个 OK(或 确定) 按钮
         MB_OKCANCEL 有 OK(确定) 和 CANCEL(取消) 两个按钮
         MB_YESNOCANCEL 有 YES(是),NO(否),CANCEL(取消) 三个按钮
         MB_YESNO 有 YES(是),NO(否) 两个按钮

        函数返回：
        IDOK 按下 OK(确定)按钮
        IDYES 按下 YES(是)按钮
        IDNO 按下 NO(否)按钮
        IDCANCEL 按下 CANCEL(取消)按钮

        */

        return;
    case 2:/* "MB_OKCANCEL" 按钮 */
        i=MessageBox(hd,"Message","MB_OKCANCEL",MB_OKCANCEL);
         switch(i)
        {
        case IDOK:
            return;
        case IDCANCEL:
            return;
        default:return;
        }
```

```
            return;
        case 3:/* "MB_YESNO"  按钮  */
            i=MessageBox(hd,"Message","MB_YESNO",MB_YESNO);
             switch(i)
            {
            case IDYES:
                return;
            case IDNO:
                return;
            default:
                return;
            }
            return;
        case 4:/* "MB_YESNOCANCEL"  按钮*/

        i=MessageBox(hd,"Message","MB_YESNOCANCEL",MB_YESNOCANC
EL);
             switch(i)
            {
            case IDYES:
                return;
            case IDNO:
                return;
            case IDCANCEL:
                return;
            default:return;
            }
            return;
        default:return;
        }
}

int gmain(void *data) /* GUI 程序  的入口函数  */
{
/*  创建一个主窗口  */
        CreatObject(0,MAINWINDOW,1,1,"MessageBox",10,10,400,300,MyWin,0)
;
        return True;
}
```

7.0 控件类创建机制

创建一个新的控件类需要在 **GUI** 中注册，才可以正常使用，这里也提供一个组件，名称为 **efClass.**

结构原形为

```
typedef struct _efClassTag
{

uint8 GUIAPI (*Get)(int16 guiclass,int16 zip);
int16 GUIAPI (*Creat)(int16 guiclass,int16 zip,CLASS_RPO (*Man)(MESSAGE));

}efClassTag,*PefClassTag;
```

其中

```
CLASS_PRO (*Man)(MESSAGE);
```

为对象类消息处理函数，与对象私有的消息处理函数不同，参数只有 **MESSAGE .**

其实，对象的私有消息处理函数参数 **hd** 与 **msg.pHOST** 值相同，加入参数 **HAND hd** 的原因有两个

1．与对象类消息处理函数相区别

2．加入 **hd** 参数后，可代替 **msg.pHOST**，使用起来更方便。

组件功能介绍

Get	探测一个对象类是否存在，存在返回 **True** 不存在返回 **False**
Creat	创建一个对象类

	guiclass 为对象类型或控件类型，建议值 **>1000** 避免与系统控件值冲突 **zip** 暂时无效参数

对象类创建后，由系统负责销毁工作。

对象类的消息处理函数与对象的消息处理函数是两个不同类型的函数，这点很重要，比如说 对象类 **TEXTBOX** **,**您可以在程序中通过 **efObj->New** 函数派生出两个新的文本编辑框，暂时命名为 **TextBoxA** 和 **TextBoxB**。

现在就来分析 **TextBoxA** 和 **TextBoxB** 的共同点和不同点，来更好的理解对象类与对象之间的关系。

共同点：都属于 **TEXTBOX** 这个对象类，都相同的行为属性，如录入字符。
不同点：数据储存在对象，对象类消息处理函数通过对象句柄获取私有数据，这样就可以保证 **TextBoxA** 和 **TextBoxB** 之间内容互不影响。

7.1 创建一个新的控件类

对象类消息处理函数通过对象句柄获取私有数据，读者可会有疑问，这样会不会和对象消息处理函数中使用的私有数据相互冲突**?**

系统提供了另外两个 **API** 来完成对象类消息处理函数的私有数据储存。原形如下

```
void    GUIAPI SetObjClassVar(HAND,void*);
void*    GUIAPI GetObjClassVar(HAND);
```

现在就来实践

```
#include "gui.h"

#define MYMOUSEBOX 1001

CLASS_RPO CLASS_MYMOUSEBOX(MESSAGE);

void     HelloGUI(HAND hd,MESSAGE msg)
{
if(msg.type == GM_SYSTEM && msg.message == GM_Create)
     efObj->New(hd,MYMOUSEBOX,1,1,"",10,10,100,100,NULL,NULL);
}

int gmain(void*data)
{
efClass->Creat(MYMOUSEBOX,0,CLASS_MYMOUSEBOX);
efObj->New(0,MAINWINDOW,1,1,"HelloGUI",10,10,400,400,HelloGUI,0);
```

```
}


CLASS_RPO CLASS_MYMOUSEBOX(MESSAGE msg)
{
HAND hd=msg.pHOST;
HDC hdc;
PefGDITag g=efGDI;
PefObject obj=efObject;
if(msg.type == GM_SYSTEM)
	switch(msg.message)
	{
	case GM_Draw:
		{
		hdc=g->Start(hd);
		g->Rectangle(hdc,0,0,obj->Width(hd),obj-
>Height(hd),COLOR_BLACK);
		g->End(hd,hdc);
		}
		return;
	default:return;
	}
if(msg.type == GM_MOUSE)
	switch(msg.message)
	{
	case GM_MouseOver:
		hdc=g->Start(hd);
						g->Rectangle(hdc,0,0,obj->Width(hd),obj-
>Height(hd),COLOR_GREEN);
		g->End(hd,hdc);
		return;
	case GM_MouseAway:
		 hdc=g->Start(hd);
						g->Rectangle(hdc,0,0,obj->Width(hd),obj-
>Height(hd),COLOR_BLACK);
		g->End(hd,hdc);
		 return;
	case GM_LeftDown:
		hdc=g->Start(hd);
						g->Rectangle(hdc,0,0,obj->Width(hd),obj-
>Height(hd),COLOR_BLUE);
		g->End(hd,hdc);
		return;
	case GM_LeftAwayUp:
```

```
    case GM_LeftUp:
         hdc=g->Start(hd);
                                                  g->Rectangle(hdc,0,0,obj->Width(hd),obj-
>Height(hd),COLOR_BLACK);
        g->End(hd,hdc);
        return;
    default:return;
    }
}
```

在上面的代码出现的新的 **API** ，开始作图了，并得到了对象的长，宽。
使用的组件 **efGDI** 作图组件，在下一章中详细介绍
efObject 为对象组件，提供基本的对象操作方法.
您可能已经发现了，下面的代码，不知道避免了多少烦琐代码的编写。

```
PefGDITag g=efGDI;
PefObject obj=efObject;
```

或者还可以

```
PefObject o=efObject;
```

代码就该写成

```
g->Rectangle(hdc,0,0,o->Width(hd),o->Height(hd),COLOR_BLACK);
```

“o->Width,o->Height” 是不是很有意思，但不建议这么做。

8.0 图形设备

GUI 系统为了忽略硬件上的差异，提供了 **HDC** 句柄 和 **efGDI** 组件.
efGDI 结构原形为**(**部分**)**

```
typedef struct _efGDITag{

HDC     GDIAPI   (*Start)(HAND);
void    GDIAPI   (*End)(HAND,HDC);
void    GDIAPI   (*Circle)(HDC,int16 x,int16 y,int16 r);
uint32  GDIAPI   (*Getpixel)(HDC,int16 x,int16 y);
void    GDIAPI   (*Line)(HDC,int16 x1,int16 y1,int16 x2,int16 y2,uint32 color);
void    GDIAPI   (*Rectangle)(HDC,int16 x1,int16 y1,int16 x2,int16 y2,uint32
color);
void    GDIAPI   (*Putpixel)(HDC,int16 x,int16 y,uint32 color);
void    GDIAPI   (*Bar)(HDC,int16 left,int16 top,int16 right,int16 bottom,uint32
color);
void    GDIAPI   (*SetColor)(HDC,uint32 color);
uint16  GDIAPI   (*DrawText)(HDC,uint16 x,uint16 y,char *str,...);
…….
}efGDITag,*PefGDITag;
```

如果要进行图形处理，如作图等操作，必须先获得对象的 **HDC** 句柄.

功能介绍

Start	返回对象的 **HDC** 句柄，标志着开始进行作图
End	结束作图，释放资源
Circle	圆
Getpixel	读取对象区域内 **(x,y)** 处像素值，相对坐标
Line	画线
Rectangle	画矩形
Putpixel	画点
Bar	画实心矩形
SetColor	设置颜色
DrawText	显示字符串，与 **printf** 功能对应

系统预先定义的 **16** 种基本色

COLOR_BLACK		**COLOR_DARKGRAY**	
COLOR_BLUE		**COLOR_LIGHTBLUE**	
COLOR_GREEN		**COLOR_LIGHTGREEN**	
COLOR_CYAN		**COLOR_LIGHTCYAN**	
COLOR_RED		**COLOR_LIGHTRED**	
COLOR_MAGENTA		**COLOR_LIGHTMAGENTA**	
COLOR_BROWN		**COLOR_YELLOW**	
COLOR_LIGHTGRAY		**COLOR_WHITE**	

8.1 基本作图

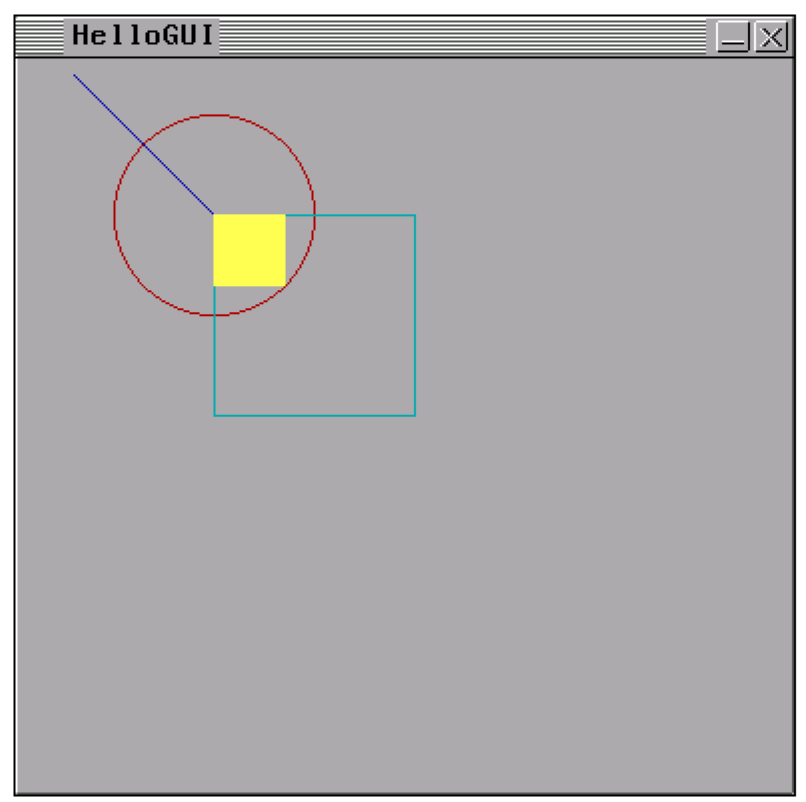
由于这个 **GUI** 还可以在 **DOS** 下运行，所以 **16** 色的定义与 **Borland** 公司的 **BGI** 图形接口标准基本相同，这是一个相当古老的标准。

简单作图的例子

```
void    HelloGUI(HAND hd,MESSAGE msg)
{
 if(msg.type == GM_SYSTEM)
 switch(msg.message)
 {
 case GM_Draw:
  {
  HDC hdc;
  PefGDITag g=efGDI;
  hdc=g->Start(hd);
  g->SetColor(hdc,COLOR_RED);
  g->Circle(hdc,100,100,50);
  g->Line(hdc,30,30,100,100,COLOR_BLUE);
```

```
    g->Rectangle(hdc,100,100,200,200,COLOR_CYAN);
    g->Bar(hdc,100,100,135,135,COLOR_YELLOW);
    g->End(hd,hdc);
    }
  return;
  default:return
  }

}
int gmain(void*data)
{
CreatObject(0,MAINWINDOW,1,1,"HelloGUI",10,10,400,400,HelloGUI,0);
}
```

运行结果



8.2 关于剪裁

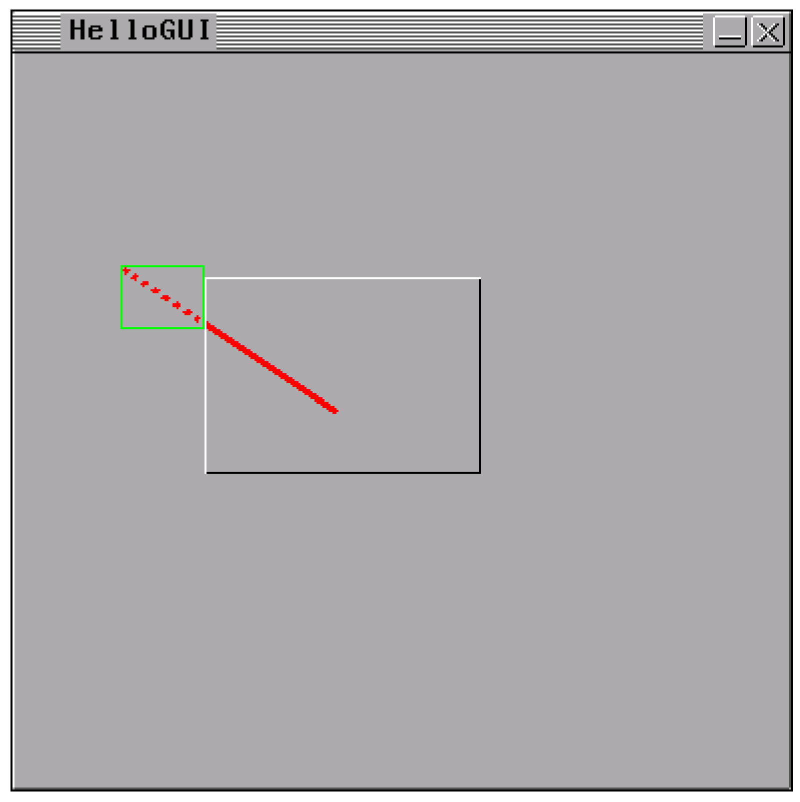
关于剪裁的实现方法曾一度是桌面图形用户界面系统讨论的热点技术之

一，诸如 窗口 Z 轴，有效区域等概念似乎是 GUI 系统开发人员所必须熟知的。

在流行的 Windows 系统，剪裁区域有 矩形，多边形甚至椭圆这样复杂的区域，可谓功能强大。但是在嵌入式系统中，情形就完全不同了，受屏幕和资源等众多因素的限制，进行复杂的剪裁运算是没有必要，举个简单的例子，您需要在您的手机里移动窗口嘛？显然完全没有这种必要，所以避免复杂的剪裁算法，可以有效提高对嵌入式系统的适应性。

在这个 GUI 中，没有提供复杂的剪裁功能支持，但 GUI 系统会自动剪裁超出对象本身范围的图象输出，这样就不会破坏其他对象的正常显示。这是非常简单的剪裁功能，效率也非常高。

具体剪裁表现可以从下图中得知



其中绿色区域为超出对象本身范围的作图行为，系统会阻止这部分越界作图，而在对象本身区域的输出正常进行。

8.3 图象显示

图象显示组件为 **efImage .**

结构原形为

```
typedef struct _efImageTag
{
void (*Load)(PIMAGE);
void (*Save)(PIMAGE);
vdoi (*Show)(PIMAGE,int16 x,int16 y);

}efImageTag,*PefImageTag;
```

功能说明

Load	加载图象文件信息
Save	保存（暂不支持）
Show	显示图象

其中 **PIMAGE** 结构定义如下

```
typedef struct _IMAGE
{
HDC hdc;
char *filename;
uint16 type; /* bmp,pcx */
uint32 width;
uint32 height;
uint8   color_bit;
uint32   colors;
uint32   cut_color;
uint32 sys_offset;
}IMAGE,*PIMAGE;
```

图形格式

```
#define IMAGE_TYPE_BMP 1
#define IMAGE_TYPE_PCX 2
#define IMAGE_TYPE_GIF 3
#define IMAGE_TYPE_PNG 4
```

用法举例

```
……
case GM_Draw:
{
  HDC hdc;
  PIMAGE img;
  img=(PIMAGE)Gmalloc(sizeof(IMAGE), "Image");
  if(img == NULL) return;
  hdc=efGDI->Start(hd);
  img->hdc=hdc;
  img->filename="image.bmp";
  img->type=IMAGE_TYPE_BMP;
  efImage->Load(img);
  efImage->Show(img,10,10);
  efGDI->End(hd,hdc);
}
……
```

目前 这个 **GUI** 还支持 **JPG** 图形格式。

8.4 缓冲

为提高显示速度，可以先将要显示的图形信息储存在内存中，在复制到显示缓冲区，这样可以实现教高速度的显示。目前 **PC** 机显卡内存都达到 **32MB,64MB** 甚至更高，如 **Windows** 就支持将图形信息储存在显卡内存中，通过改变显示区活动页，实现非常高速的图形显示。

因为硬件加速，与系统相关性很高，这个 **GUI** 支持根据不同系统进行差异化定制。

缓冲组件 **efImageBuf**

```
typedef struct _efImageBuf
{
void (*LoadImage)(PIMAGE,PIMAGEBUF);
void (*Show)(HDC,PIMAGEBUF);
void (*Copy)(HDC,PIMAGEBUF);

}efImageBuf,*PefImageBuf;
```

其中 **PIMAGEBUF** 结构原形如下

```
typedef struct _IMAGEBUF
{
RECT rect;
void *data;
}IMAGEBUF,*PIMAGEBUF;
```

其中 **RECT** 结构原形

```
typedef struct _RECT
{
int16 left;
int16 top;
int16 right;
int16 bottom;
}RECT,*PRECT;
```

IMAGEBUF 中 **rect** 表示在对象内显示区域，或复制时的复制区域。
rect 中 **left,top,right,bottom** 均为相对坐标

缓冲区域由应用程序提供

缓冲组件功能说明

LoadImage	将图形文件中的内容复制到 缓冲区域
Show	根据 **HDC** 句柄区域信息，显示缓冲区域内容
Copy	根据 **HDC** 句柄区域信息，复制显示区内容到缓冲区

缓冲组件属于 **GUI** 附加组件，可能在一些系统中不支持。

9.0 多任务和多线程

多任务与多线程基本可以看作同一个概念，但又有所不同，在 **uC/OS-II** 这样的系统中，更习惯使用多任务这样的概念，当然有一定的原因**,**诸如实现方法不同，还有历史的缘故。

多线程现在已经被大多数操作系统支持，如 **UNIX/Linux,Windows** 还有 **solaris** 等，**solaris** 曾一度是这样方面的佼佼者。

UNIX/Linux 现在已经普遍支持 **POSIX** 标准，**Windows** 支持部分 **POSIX** 标准。

简单来说 多线程或者多任务（以下统一为多线程）可以看作是一个程序**(**进程**)**中多个函数并发执行**.**

9.1 Linux, DOS,Windows,uC/OS-II 多任务比较

Linux 中提供一个名为 **pthread** 的库，来提供对多线程的支持。

头文件：**pthread.h**
库：**libpthread.a**

使用 **pthread_create** 函数可以创建一个线程

```
void newthread(void)
{
printf("I am in a new thread");
}
int main(void)
{
pthread_t id;
pthread_create(&id,NULL,(void *) newthread,NULL);
printf("main:creat a new thread");
pthread_join(id,NULL);
return 0;
}
```

每个线程拥有自己的堆栈，操作系统负责维护工作。

当多个线程读写同一块内存数据时，必须进行同步措施，**Linux** 系统提供互斥锁来避免线程之间的错误。

Windows 中提供 **CreateThread** 函数来创建新的线程，并且提供 **CRITICAL_SECTION** 结构和 **EnterCriticalSection** 等函数来实现线程同步。这些与 **Linux** 提供的功能基本相同，只是接口和形式不同。

DOS 一般被认为是典型的单任务操作系统，为了应对这样的缺陷，出现 **TSR** 技术，如著名的 **SideKick** 工具软件，就是利用 **TSR** 技术。拦截时钟，在后台运行程序，确实在一定程度上实现了"多线程"。因为 **TSR** 技术没有统一的标准，导致多个 **TSR** 程序之间互相冲突，后又因 **Windows** 提供更好的解决方案，**TSR** 技术可以认为已经消失。

有趣的是公认的多任务操作系统 **uC/OS-II** 在 **x86** 上的移植版本，就使用通过修改 **DOS** 环境下的时钟来实现多任务切换的。

uC/OS-II 系统在创建新任务(对应 新线程)时必须指定堆栈区域，而 **Windows** 则可以根据需要动态增加堆栈大小.而且 **uC/OS-II** 提供的任务(对应线程)同步机制与 **Linux,Windows** 均不同，但仍实现类似的功能。

9.2 GUI 中对 多任务/多线程 的接口

为了提高 **GUI** 的适应性，作者在设计一套虚拟的接口层，来适应不同的操作系统。

10.0 GUI 移植性分析

这个 **GUI** 对设备要求非常抵，**CPU: 25 MHz** 内存：**1 MB (**可以更低**)** 就可以运行，对操作系统没有要求，可以 单任务模式运行，也可以 多任务**/**多线程 模式运行。

关于 **C** 库，**GUI** 中使用了 **memcpy,memset,strcpy** 这样的 **C** 标准函数，但是这些都在 **GUI** 中有自己的实现，所以可以不需要 **C** 标准库的支持。

GUI 已经成功移植 **Linux,DOS,uC/OS-II,Windows** 。

10.1 GUI 输入/输出设备驱动

GUI 输入设备主要有 鼠标和键盘，但是 **GUI** 本身是通过虚拟的消息来工作的，所以只要构造一些函数**(**设备驱动**)**，产生这些虚拟消息，就可以驱动 **GUI** 运行。

GUI 会在消息循环中不断调用以下函数**,**原形为

```
extern void low_drivers(void);
```

移植时，需要完成这个函数，在这个函数中检测 输入设备，如果检测到事件发生，就可以调用对应函数**,**来完成虚拟消息的创建

```
void low_mouse_leftdown(int x,int y);
void low_mouse_leftup(int x,int y);
void low_mouse_rightdown(int x,int y);
void low_mouse_rightup(int x,int y);
void low_mouse_middleup(int x,int y);
void low_mouse_middledown(int x,int y);
void low_mouse_move(int x,int y);
……
```

```
void low_key_down(char key);
void low_key_up(char key);
……
```

对于输出设备，如果有 **FrameBuffer** 支持，那么就直接使用 **FrameBuffer** 作为 **GUI** 的输出显示驱动。

即使没有 **FrameBuffer** 支持，也可以直接编写 **GUI** 的显示驱动
形式如下

```
typedef struct low_gdidrv {
uint8 (*init)(void);
void   (*exit)(void);
void (*putpixel)(int x,int y,uint32 color);
 ……
}low_gdidrv,*Plow_gdidrv
```

只需填充这个结构也可以完成驱动的编写工作。

10.2 GUI 在多任务**/**多进程操作系统的移植

在其它多任务操作系统的移植可以参考 **uC/OS-II** 或 **Linux** 上的移植代码完成。

目前还不支持多进程模式。

后记

如果遇到开发问题，请先到 **eCGUI** 开发论坛 http://www.ecgui.com/bbs/ 获得最新资讯，或者发帖求助。开发团队会及时解答！