# MiniGUI 特性说明书

本文档说明 MiniGUI 所支持的功能特性。

## 一、概述

MiniGUI 是 Linux 控制台上运行的，基于 SVGALib 和 LinuxThread 库的多窗口图形用户界面支持系统。MiniGUI 采用了类 Win32 的 API 接口，实现了简化的类 Windows 98 风格的图形用户界面。

图形用户界面在许多情况下都优于字符界面，其最大的优点是使应用程序的操作简单易学。

在 MiniGUI 中，图形用户界面包括如下基本元素：

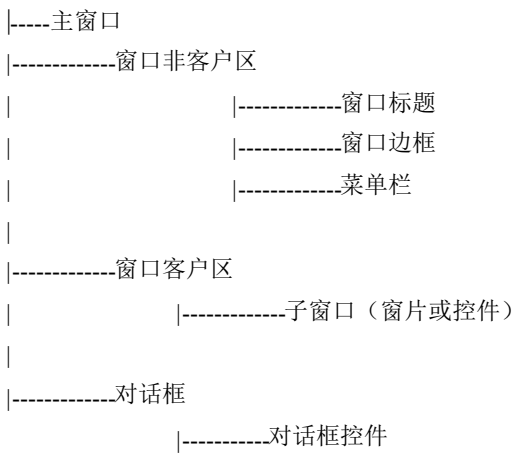


图 1.1 图形用户界面的基本元素

### 1.1 窗口

MiniGUI 中的窗口基本分四类，分别为主窗口、对话框、控件和主窗口中的窗片。

MiniGUI 中的主窗口和 Windows 应用程序的主窗口概念类似，但略微有些不同，MiniGUI 中的每个主窗口对应于一个单独的线程，通过函数调用可建立主窗口以及对应的线程。每个线程有一个消息队列，主窗口从这一消息队列中获取消息并由窗口过程（回调函数）进行处理。

MiniGUI 中的对话框是一种特殊的窗口，对话框一般和控件一起使用，这两个概念和 Windows 的相关概念是类似的。MiniGUI 支持的控件类型有：

- ✓ 静态框：文本、图标或矩形框等。这种控件的属性一般不会在运行时发生变化。
- ✓ 文本框：单行或多行的文本编辑框。
- ✓ 按钮：单选钮、复选框和一般按钮等。
- ✓ 其他特殊控件。

窗片是 MiniGUI 所特有的，窗片实际是主窗口的子窗口，只存在于主窗口中。为了

处理上的方便，主窗口的子窗口只以平铺的形式出现，因此我们将这种子窗口称为“窗片”或“窗格”。窗片可以是私有的控件类型，也可以是标准的控件类型。

## 1.2 消息和消息循环

在 Windows 系列操作系统中，广泛使用了消息驱动的概念。在 MiniGUI 中，我们也使用了消息驱动作为应用程序的创建构架。

在消息驱动的应用程序中，计算机外设发生的事件，例如键盘键的敲击、鼠标键的按击等，都由支持系统收集，将其以事先的约定格式翻译为特定的消息。应用程序一般包含有自己的消息队列，系统将消息发送到应用程序的消息队列中。应用程序可以建立一个循环，在这个循环中读取消息并处理消息，一直处理到特定的消息传来为止。这样的循环称为消息循环。一般地，消息由代表消息的一个整型数和消息的附加参数组成。例如，鼠标左键的按下消息，可能由 133 这个数来表示，其附加参数可能包含按下时的鼠标所在位置信息。例如，MiniGUI 中如下定义消息：

```
typedef struct
{
  HWND        hwnd;
  int         message;
  WPARAM      wParam;
  LPARAM      lParam;
  ...
}MSG;
```

message 指定了特定的消息类型，wParam 是以 unsigned int 类型定义的消息的短参数，lParam 是以 long 类型定义的消息长参数。

应用程序一般要提供一个处理消息的标准函数。在消息循环中，系统可以调用此函数，应用程序在此函数中处理相应消息。

图 1.2 是一个消息驱动的应用程序的简单构架示意。

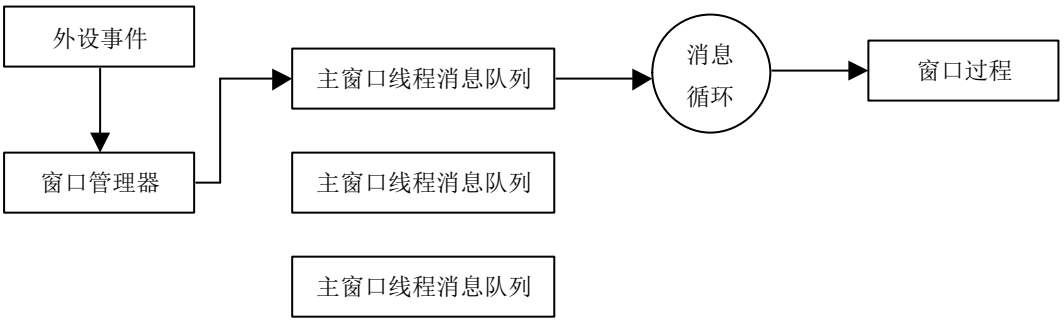


图 1.2 消息驱动的应用程序的简单构架

在 MiniGUI 中，消息分为如下几种类型：

- ✓ 系统消息，为系统内部管理使用。
- ✓ 鼠标消息，鼠标的点击、移动等产生的消息。
- ✓ 键盘消息，键盘的按键消息。
- ✓ 窗口消息，窗口管理消息。
- ✓ 菜单消息，菜单管理消息。

✓ 命令消息等。

## 1.3 窗口过程和窗口类

窗口过程是用来处理窗口消息的函数过程。对于同一类型的控件，其窗口过程一般是一样的。因此，系统一般利用窗口的窗口类名来区分不同的窗口类并调用不同的窗口过程。由于几乎每一个主窗口均和其他窗口有着不同的窗口过程，因此，在 MiniGUI 中，窗口类的概念只存在于控件和窗片中。对于主窗口来说，其窗口过程在建立主窗口时指定，而对控件和窗片来说，则在注册窗口类时指定，而在建立窗片或控件时指定所属窗口类。

## 1.4 句柄

句柄是 MiniGUI 用来标识对象的标识符。句柄和指针概念类似，但它不一定是指针值。利用句柄，MiniGUI 将系统变量从应用项目中分离了出来，因为程序员只能通过句柄访问对象，因而就没有利用指针是可能发生的因非法访问而导致的数据不一致问题。

在 MiniGUI 中，窗口、控件、设备环境、菜单、图标等均使用句柄访问。

# 二、窗口

## 2.1 应用程序和主窗口

我们将基于 MiniGUI 的一个会话（session）称为一个应用项目，而其中每个单独的线程或线程组称为应用。每个应用项目可建立多个应用。主窗口是建立在 MiniGUI 基础上的应用的主界面。MiniGUI 为每个主窗口建立单独的消息队列，在该主窗口基础上派生出的窗片、对话框及其控件均使用同一消息队列。在 MiniGUI 中，每个应用对应于一个线程。理论上讲，每个应用可以具备多个主窗口，但在 MiniGUI 中，主窗口均以单独的线程实现。但多个主窗口对应单一线程的情况也是可以在 MiniGUI 中实现的。

每个应用项目有一个 MiniGUIMain 函数，在这个函数中，可建立初始的应用线程。在调用 MiniGUIMain 之前，MiniGUI 启动自己的桌面窗口（Desktop）。桌面窗口作为 MiniGUI 的窗口管理器而存在。下面的代码段在 MiniGUIMain 中启动了三个主窗口线程:

```
int MiniGUIMain(int args, char* arg[])
{
   pthread_t thread, thread2, thread3;

   CreateThreadForMainWindow(&thread,  NULL, TestWindowMain, 0);

   CreateThreadForMainWindow(&thread2,  NULL, TestWindowMain2, 0);

   CreateThreadForMainWindow(&thread3,  NULL, TestWindowMain3, 0);

   return 0;
}
```

CreateThreadForMainWindow 函数为主窗口建立线程，并返回线程标识符。

其中的第三个参数是线程的入口函数地址。如下的代码段定义了上述代码中第一个主窗口线程的入口函数：

```
void InitCreateInfo(PMAINWINCREATE  pCreateInfo)
{
  pCreateInfo->dwStyle  = WS_THICKFRAME;
  pCreateInfo->spCaption  = "The first main window" ;
  pCreateInfo->hMenu  = 0;
  pCreateInfo->hCursor  = GetSystemCursor(2);
  pCreateInfo->hIcon  = LoadIconFromFile("res/table.ico");
  pCreateInfo->MainWindowProc  = TestMainWinProc;
  pCreateInfo->lx  = 50;
  pCreateInfo->ty  = 50;
  pCreateInfo->rx  = 300;
  pCreateInfo->by  = 480;
  pCreateInfo->iBkColor  = COLOR_lightwhite;
  pCreateInfo->dwAddData1  = 0;
  pCreateInfo->dwAddData2  = 0;
}

void* TestWindowMain(void*  data)
{
  MSG Msg;

  MAINWINCREATE  CreateInfo;
  HWND hMainWnd;

  InitCreateInfo(&CreateInfo);

  if( !(hMainWnd = CreateMainWindow(&CreateInfo))  )
    return NULL;

  ShowWindow(hMainWnd,  SW_SHOWNORMAL);

  while( GetMessage(&Msg, hMainWnd) ) {
    DispatchMessage(&Msg);
  }

  MainWindowThreadCleanup(hMainWnd);
  return NULL;
}
```

在上面的代码段中，该线程首先调用 CreateMainWindow 建立了主窗口，然后调用 ShowWindow 显示了主窗口，最后启动了消息循环。当消息循环因为接收到 MSG_QUIT 消息而终止时，该函数调用了 MainWindowThreadCleanup 清除了相关的线程数据。

从上述代码中可看出主函数不支持窗口类，在调用 CreateMainWindow 函数时直接指定主窗口的窗口过程地址。我们也可以从中看到主窗口所支持的其他属性：

1．窗口风格。表 2.1 给出了所支持的窗口风格

表 2.1　MiniGUI 支持的主窗口风格

| 风格 | 描述 |
|---|---|
| WS_BORDER | 创建一个具有单线边框的窗口 |
| WS_THICKFRAME | 创建一个具有宽边框的窗口 |
| WS_THINFRAME | 创建一个具有细边框的窗口 |
| WS_CAPTION | 创建一个具有标题栏的窗口 |
| WS_HSCROLL | 创建一个具有水平滚动条的窗口 |
| WS_MAXMIZEBOX | 创建一个具有最大化框的窗口 |
| WS_MINIMIZEBOX | 创建一个具有最小化框的窗口 |
| WS_SYSMENU | 创建一个具有系统菜单的窗口 |
| WS_VSCROLL | 创建一个具有垂直滚动条的窗口 |

| | |
|---|---|
| WS_DISABLED | 创建一个初始为禁止的窗口 |
| WS_MAXIMIZE | 创建一个初始最大化的窗口 |
| WS_MINIMIZE | 创建一个初始最小化的窗口 |
| WS_VISIBLE | 创建一个初始可见的窗口 |
| WS_EX_TOPMOST | 创建一个顶层窗口，这是一个 Win32 的扩展风格 |

2．窗口标题。

3．窗口菜单。

4．窗口图标。

5．窗口背景色。

## 2.2 主窗口过程

主窗口过程实际是一个回调函数，一般由 DispatchMessage 函数调用，用来处理应用的消息。主窗口过程一般如下定义：

```
int TestMainWinProc(HWND  hWnd, int message, WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  int x, y;
  RECT client;
  static BITMAP bitmap;
  static BOOL fValid = FALSE;
  static int paintCount = 0;

  switch (message) {
    case MSG_CREATE:
      SetTimer (hWnd, 100, 20);
      if(LoadBitmap(&bitmap, "res/mnls.bmp") < 0)
        fprintf (stderr, "Test Main Win: Loading bitmap failure!\n");
      else
        fValid = TRUE;
    break;

    case MSG_PAINT:
      hdc = BeginPaint (hWnd);
      testdc (hdc, fValid?&bitmap:NULL);
      EndPaint (hWnd, hdc);
    break;

    case MSG_LBUTTONDBLCLK:
      hdc = GetClientDC (hWnd);
      TextOut(hdc, 0, 0, "Left button double clicked");
      ReleaseDC (hdc);
      PostMessage (hWnd, MSG_CLOSE, 0, 0);
    return 0;

    case MSG_TIMER:
      if (ISINBACKGROUND)
        break;
      paintCount++;
      if (paintCount % 10 != 0) {
        hdc = GetClientDC (hWnd);
        GetClientRect (hWnd, &client);
        x = random() % (RECTW (client));
        y = random() % (RECTH (client));
        SetBrushColor(hdc, RGB2Index (hdc, random() % 256,
                        random() % 256,
                        random() % 256));
        FillBox(hdc, x, y, random() % RECTW (client),
          random() % RECTH (client));
```

```
            ReleaseDC (hdc);
        }
        else
            InvalidateRect (hWnd, NULL, FALSE);
      break;

      case MSG_CLOSE:
        KillTimer (hWnd, 100);
        UnloadBitmap (&bitmap);
        DestroyMainWindow (hWnd);
        PostQuitMessage (hWnd);
      return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}
```

主窗口有四个入口参数，分别是消息的目标窗口句柄、消息、消息的 WPARAM (即 unsigned int) 型参数和 LPARAM (即 long int) 型参数。在 32 位系统中，消息的两个参数实际是等长度的。

主窗口的窗口过程处理应用感兴趣的消息，将其他消息传递给 DefaultMainWindProc 函数处理。

## 2.3 桌面的应用项目接口

当用户在桌面上单击鼠标右键时可弹出浮动式菜单，应用项目可通过桌面提供的接口在该菜单中添加菜单项，从而提供一定的灵活性。要利用桌面的应用项目接口，应用项目要实施两个函数：

```
void CustomizeDesktopMenu (HMENU hmnu, int iPos);
int CustomDesktopCommand (int id);
```

通过 CustomizeDesktopMenu 函数，应用项目可在桌面的浮动式菜单中添加菜单及菜单项。当用户选择了定制的菜单项时，MiniGUI 将调用 CustomDesktopCommand 函数，这时，应用项目就可以处理该菜单命令。

## 2.4 对话框和控件

对话框实际是一种特殊的主窗口。一般而言，对话框中包含有许多系统提供的控件。对话框和控件的设计目标是提供和 Win32 在源代码级上的完全兼容。但某些复杂的控件不打算支持，对话框页也不支持。MiniGUI 支持的控件有：

- ✓ 静态控件。
- ✓ 按钮控件。
- ✓ 编辑控件。
- ✓ 列表控件。
- ✓ 滚动条控件。
- ✓ 组合框控件。

同时，MiniGUI 将提供标准的公共对话框：

- ✓ 文件打开对话框。
- ✓ 文件保存对话框。

### 2.6 消息框

MiniGUI 将提供和 Win32 完全兼容的消息框调用接口。

### 2.6 窗片

窗片是主窗口中子窗口的简化支持，为了避免子窗口之间的互相剪切，而采用了平铺式的子窗口实现，不考虑子窗口的互相剪切，因而也就不提供多文档接口。也因为这个原因，我们将这种子窗口称为窗片或窗格。窗片的建立和控件的建立方法一致。

### 2.7 滚动支持

将提供和 Win32 完全兼容的主窗口或子窗口的滚动支持。

## 三、对话框、窗片和控件

该部分功能尚未实现，其目标是提供和 WIN32 的 80% 兼容。需要提供的接口分为如下几类（注，我们将窗片和控件统称为窗口，并严格区分“主窗口”和“窗口”这两个名词）：

- ✓ 窗口类的注册和注销。
- ✓ 窗口的创建和销毁。
- ✓ 对话框和控件管理。
- ✓ 常见控件消息、操作接口等的实现。

## 四、消息及消息队列

在 MiniGUI 中，基本实现了 Win32 的消息处理函数。

消息可通过如下函数发送：

- ✓ 通过 PostMessage 发送。消息发送到消息队列后立即返回。这种发送方式称为“邮寄”消息。如果消息队列中的邮寄消息缓冲区满，则该函数返回错误值。
- ✓ 通过 PostSyncMessage 发送。该函数用来向不同于调用该函数的线程消息队列邮寄消息，并且只有该消息被处理之后，该函数才能返回，因此这种消息称为“同步消息”。
- ✓ 通过 SendMessage 发送。该函数可以向任意一个窗口发送消息，消息处理完成之后，该函数返回。如果目标窗口所在线程和调用线程是同一个线程，该函数直接调用窗口过程，如果处于不同的线程，则利用 PostSyncMessage 函数发送同步消息。
- ✓ 通过 SendNotifyMessage 发送。该函数向指定的窗口发送通知消息，将消息放入消息队列后立即返回。由于这种消息和邮寄消息不同，是不允许丢失的，因此，系统以链表的形式处理这种消息。
- ✓ 通过 SendAsyncMessage 发送。利用该函数发送的消息称为“异步消息”，系统直接调用目标窗口的窗口过程。

# 五、图形设备接口

图形设备接口，即 GDI， MiniGUI 中用来实现图形输出的模块。

## 5.1 基本概念

### 5.1.1 图形设备

在 MiniGUI 中，采用了在 Windows 和 X Window 中普遍采用的图形设备概念。每个图形设备定义了计算机显示屏幕上的一个矩形输出区域。

在调用图形输出函数时，均要求指定经初始化，或经建立的图形设备上下文，或设备环境（DC）。

每个图形输出均局限在图形设备指定的矩形区域内。

在多窗口系统中，各个图形设备之间的输出互相剪切,以避免图形输出之间互相影响。

### 5.1.2 剪切域

剪切域就是在图形设备上定义的一个区域，所有在该图形设备上进行的图形输出，超过剪切域的部分，均被裁剪。只有在剪切域上的图形输出才是可见的输出。

MiniGUI 中的剪切域，定义为矩形剪切域的集合。

### 5.1.3 映射模式

映射模式指定了特定图形输出的坐标值如何映射到图形设备的坐标值。

图形设备的坐标系原点定义为图形设备矩形区域的左上角。向右为正 X 坐标轴方向；向下为正 Y 坐标轴方向。这一坐标系称为设备坐标系。

通过 GDI 模块的映射模式操作函数，可定义自己的逻辑坐标系。逻辑坐标系可以是设备坐标系的水平或垂直反转，缩放，或者偏移。

多数 GDI 输出函数指定的是逻辑坐标系。

默认情况下，逻辑坐标系和设备坐标系是重合的。

## 5.2 GDI 功能特性的分类说明

### 5.2.1 图形设备能力

通过调用函数 GetGDCapability，可获得图形设备的如下能力：

- ✓ 颜色数目；
- ✓ 水平和竖直方向的象素点数；
- ✓ 设备坐标系中可见点的最大 x 和 y 坐标值。

### 5.2.2 创建、销毁或获取、释放图形设备接口

和 Win32 类似， MiniGUI 中也有一个 DC 的缓冲区，应用可调用 GetDC 或 GetClientDC 函数从 DC 缓冲区中获取图形设备环境，在结束使用 DC 之后，应当调用 ReleaseDC 函数释放 DC。

应用也可以建立自己私有的 DC，这种 DC 可以是全局有效的 DC，由于免除了获取和释放以及初始化等工作，因此，利用这种 DC 可加速图形显示。当应用不再使用私有 DC

时，应当利用 DeletePrivateDC 删除私有 DC。下面的代码即利用了这种 DC：

```
int TestMainWinProc2(HWND  hWnd, int message, WPARAM wParam, LPARAM lParam)
{
  static HDC hdc;
  HDC hPaintDC;
  int x, y;
  RECT client;
  static int count = 0;
  static BITMAP bitmap;
  static BOOL fValid = FALSE;
  static int paintCount = 0;

  switch (message) {
    case MSG_CREATE:
      SetTimer (hWnd, 100, 20);
      SetTimer (hWnd, 200, 5);
      if(LoadBitmap(&bitmap,  "res/j11b.BMP")  < 0)
        fprintf (stderr, "Test Main Win: Loading bitmap failure!\n");
      else
        fValid = TRUE;
    break;

    case MSG_SHOWWINDOW:
      if (wParam == SW_SHOWNORMAL)
        hdc = CreatePrivateClientDC  (hWnd);
    break;

    case MSG_PAINT:
      hPaintDC = BeginPaint (hWnd);
      testdc (hPaintDC, fValid?&bitmap:NULL);
      EndPaint (hWnd, hPaintDC);
    break;

    case MSG_LBUTTONDBLCLK:
    return 0;

    case MSG_TIMER:
      if (ISINBACKGROUND)
        break;

      if (wParam == 100) {
        paintCount ++;

        if (paintCount % 10 != 0) {
          GetClientRect  (hWnd, &client);
          x = random() % (RECTW (client));
          y = random() % (RECTH (client));
          SetPenColor(hdc,  RGB2Index (hdc, random() % 256,
                        random() % 256,
                        random() % 256));
          LineTo(hdc, x, y);
        }
        else
          InvalidateRect  (hWnd, NULL, TRUE);
      }
      else if (wParam == 200) {
        if (count < 5)
        {
          Ping ();
          count ++;
        }
        else
          KillTimer (hWnd, 200);
      }
    break;
```

```
        case MSG_CLOSE:
            KillTimer (hWnd, 100);
            KillTimer (hWnd, 200);
            UnloadBitmap (&bitmap);
            DeletePrivateDC (hdc);
            DestroyMainWindow (hWnd);
            PostQuitMessage (hWnd);
        return 0;
    }

    return DefaultMainWinProc(hWnd, message, wParam, lParam);
}
```

上述代码在应用主窗口建立时创建了私有 DC，然后在定时器消息中利用私有 DC 进行绘制，最后在关闭窗口时删除了私有 DC。

MiniGUI 也支持内存中的虚拟 DC，可通过 CreateCompatibleDC 建立内存 DC，利用这种 DC，可加速绘制过程，并减少绘制过程中闪烁现象。应用可使用 DeleteCompatibleDC 函数删除内存 DC。

上述 DC 的创建或获取与 Win32 API 有较明显的不同。

### 5.2.3 一般绘制属性

这类功能用来设置 DC 的绘制属性，这些属性及其影响的绘图操作在表 5.1 中列出。

表 5.1 一般绘图属性

| 绘制属性 | 所影响的绘图操作 | 备注 |
|---|---|---|
| 背景色 | 文本输出 | GetBkColor, SetBkColor |
| 背景模式 | 文本输出 | GetBkMode, SetBkMode |
| 文本颜色 | 文本输出 | GetTextColor, SetTextColor |
| 画笔类型 | 线条输出 | 只支持实型画笔<br>GetPenType, SetPenType |
| 画笔颜色 | 线条输出 | GetPenColor, SetPenColor |
| 画刷类型 | 填充操作 | 只支持实型画刷<br>GetBrushType, SetBrushType |
| 画刷颜色 | 填充操作 | GetBrushColor, SetBrushColor |

### 5.2.4 一般绘图支持

这类功能用来完成基本的绘图功能，这些功能包括：画点、直线、圆、矩形等。另外还有一些颜色转换方面的功能，可以将特定的 RGB 转换为最接近的 256 色调色板中的颜色索引值。接口函数由表 5.2 给出。

表 5.2 一般绘图支持

| 函数 | 功能说明 |
|---|---|
| SetPixel | 设置指定的象素颜色，颜色值以调色板索引给出。 |
| SetPixelRGB | 设置指定的象素颜色，颜色值以 RGB 值给出。 |
| GetPixel | 获取指定点的象素颜色，以调色板索引给出。 |
| GetPixelRGB | 获取指定点的象素颜色，以 RGB 值给出。 |
| RGB2Index | 完成 RGB 值到调色板索引值的转换。 |
| MoveTo, LineTo | 绘制直线。 |

| | |
|---|---|
| Circle | 绘制圆。 |
| Rectangle | 绘制矩形。 |

### 5.2.5 文本输出支持

这类功能用来利用系统字体输出文本。可实现无格式文本（TextOut）和有格式文本（TabbedTextOut）的输出。作为辅助函数，应用还可以利用 GetTabbedTextExtent 函数来获得格式化文本字符串的输出尺寸。由于系统字体是等宽字体，无格式文本的输出长度就等于单字节字符串个数乘以单个字符的宽度。利用 GetCharWidth 以及 GetCCharWidth 可获取单字节字符以及双字节字符（汉字）的宽度；利用 GetCharHeight 可获取字符的高度。

### 5.2.6 映射函数支持

MiniGUI 支持的映射方式有两种，一种和 Win32 的 MM_TEXT 映射方式一样，即设备坐标系和逻辑坐标系是一致的；一种和 Win32 的 MM_ANISOTROPIC 映射方式一样，逻辑 x 和 y 轴可以以任意的比例映射到对应的坐标轴上。其他的 Win32 映射方式不被支持。

通过函数 SetMapMode 设定映射方式。

利用 SetWindowExt 和 SetViewportExt 可设置 MM_ANISOTROPIC 映射方式的坐标轴方向、比例等。利用 SetWindowOrg 和 SetViewPortOrg 可以设定原点位置。

和上述函数的对应的 Get 函数组可用来获取设定值。

### 5.2.7 坐标转换

这类函数用来实现设备坐标到逻辑坐标的转换。DPtoLP 将设备坐标转换为逻辑坐标，LPtoDP 将逻辑坐标转换为设备坐标。

### 5.2.8 剪切支持

这类函数用来实现对 DC 剪切域的操作。和 Win32 不同的是，MiniGUI 的剪切域只支持矩形剪切域。

ExcludeClipRect 可用来在当前剪切域中排除指定的矩形区域。

IncludeClipRect 可用来在当前剪切域中包含指定的矩形区域。

CliprectIntersect 可用来将当前剪切域和指定矩形相交。

SelectClipRect 将剪切域设置为指定矩形。

GetBoundsRect 获取包含当前剪切域的最大矩形。

PtVisible 可判断给定点是否处于剪切域。

RectVisible 可判断给定矩形是否和剪切域相交。

### 5.2.9 位图支持

这类函数用来实现填充操作。

利用 FillBox 可以以当前的画刷类型和颜色填充指定矩形。

利用 FillBoxWithBitmap 可以用指定的位图填充指定矩形，如果有需要，该函数可进行位图的缩放。

利用 BitBlt 函数可在两个 DC 之间复制图象。

利用 StretchBlt 函数可两个 DC 之间复制图象，并完成缩放。

利用函数 LoadBitmap 可从 Windows BMP 文件中装入位图信息。UnloadBitmap 则卸载位图信息。

### 5.2.10 图标支持

MiniGUI 的图标支持和 Win32 API 大致相同，包含如下几种操作：

创建和销毁图标：LoadIconFromFile、CreateIcon、DestroyIcon。

图标绘制支持：DrawIcon。

根据需要 MiniGUI 将提供系统图标集，及相应的操作函数。

### 5.2.11 矩形支持

这类函数提供了实现矩形相交，合并等一般性的操作，达到 Win32 的 95% 兼容。下面是这些函数的说明。

#### 1) SetRect

```
void GUIAPI SetRect(RECT* prc, int left, int top, int right, int bottom);
```

该函数将矩形 prc 设定为参数 left, top, right, bottom 指定的大小。

#### 2) SetRectEmpty

```
void GUIAPI SetRectEmpty(RECT* prc);
```

该函数将矩形 prc 设置为空矩形. 空矩形就是面积为 0 的矩形. 该函数将矩形的 left, top, right, bottom 值均设置为 0。

#### 3) CopyRect

```
void GUIAPI CopyRect(RECT* pdrc, const RECT* psrc);
```

该函数将矩形 psrc 复制到 pdrc 中。

#### 4) IsRectEmpty

```
BOOL GUIAPI IsRectEmpty(const RECT* PRC);
```

该函数判断指定矩形是否为空矩形。若为空矩形，则函数返回 TRUE，否则返回 FALSE。

#### 5) EqualRect

```
BOOL GUIAPI EqualRect(const RECT* prc1, const RECT* prc2);
```

该函数判断指定的两个矩形是否是相等的矩形。相等的矩形其 left, top, right, bottom 值均相等。

#### 6) NormalizeRect

```
void GUIAPI NormalizeRect(RECT* pRect);
```

该函数将指定的矩形 pRect 进行正规化处理。矩形的正规化指满足如下条件的矩形：

left <= right 并且 top <= bottom。

### 7) IntersectRect

```
BOOL GUIAPI IntersectRect(RECT*  pdrc,
                  const RECT* psrc1, const RECT* psrc2);
```

该函数求两个矩形 psrc1 和 psrc2 的相交区域, 并在 pdrc 中返回相交矩形。

如果 psrc1 和 psrc2 相交，函数返回为 TRUE。

如果 psrc1 和 psrc2 不相交，函数返回为 FALSE，pdrc 为空矩形

### 8) DoesIntersect

```
BOOL GUIAPI DoesIntersect(const  RECT* psrc1, const RECT* psrc2);
```

该函数判断两个矩形是否相交。

如果 psrc1 和 psrc2 相交，函数返回为 TRUE。

如果 psrc1 和 psrc2 不相交，函数返回为 FALSE。

### 9) UnionRect

```
BOOL GUIAPI UnionRect(RECT* pdrc, const RECT* psrc1, const RECT* psrc2);
```

该函数求两个矩形 psrc1 和 psrc2 相并矩形，并在 pdrc 中返回相并矩形。

如果 psrc1 和 psrc2 能够进行相并操作，函数返回 TRUE。

如果 psrc1 和 psrc2 不能进行相并操作，函数返回 FALSE，pdrc 为空矩形。

### 10) SubtractRect

```
BOOL GUIAPI SubtractREct(RECT* pdrc, const RECT* psrc1, const RECT* psrc2);
```

该函数求两个矩形 psrc1 和 psrc2 相减的矩形， 并在 pdrc 中返回结果矩形。

如果 psrc1 和 psrc2 能够进行相减操作， 函数返回 TRUE。

如果 psrc1 和 psrc2 不能进行相减操作，函数返回 FALSE，pdrc 为空矩形。

### 11) OffsetRect

```
void GUIAPI OffsetRect(RECT* prc, int x, int y);
```

该函数将矩形 prc 偏移指定的偏移量 x, y。

### 12) InflateRect

```
void GUIAPI InflateRect(RECT* prc, int cx, int cy);
```

该函数将矩形 prc 的长和宽分别增加 cx 和 cy。

### 13) PtInRect

```
BOOL GUIAPI PtInRect(const  RECT* prc, int x, int y);
```

该函数判断指定点（x, y）是否处于矩形 prc 中。

如果指定点在矩形中，函数返回 TRUE；否则返回 FALSE。

# 六、菜单

MiniGUI 中的菜单接口和 Win32 90% 兼容，但有少许差别。在外观上，MiniGUI 可为弹出式菜单定义一个菜单标题，可支持空的弹出式菜单；在内部结构和概念上也有一些小的差别。主要区别在于子菜单的定义上，子菜单即可以是一个完整的弹出式菜单，也可以是一个没有弹出式菜单标题的子菜单。从另一个角度讲，弹出式菜单是由标题以及一系列子菜单项组成的。

MiniGUI 提供用来操作菜单的函数有：

```
HMENU GUIAPI CreateMenu ();
HMENU GUIAPI CreatePopupMenu ( PMENUITEMINFO pmii);
HMENU GUIAPI CreateSystemMenu ();
int GUIAPI InsertMenuItem (HMENU hmnu, int item,
                BOOL flag, PMENUITEMINFO pmii);
int GUIAPI RemoveMenu (HMENU hmnu, int item, UINT flags);
int GUIAPI DeleteMenu (HMENU hmnu, int item, UINT flags);
int GUIAPI DestroyMenu (HMENU hmnu);
```

上述这些函数用来操作菜单数据，可实现菜单的创建、销毁，菜单项的添加、删除等功能。

```
int GUIAPI IsMenu (HMENU hmnu);
```

该函数可用来判断给定句柄是否为菜单句柄。

```
HMENU GUIAPI SetMenu (HWND hwnd, HMENU hmnu);
HMENU GUIAPI GetMenu (HWND hwnd);
```

上述函数用于获取或设置主窗口的菜单。

```
void GUIAPI DrawMenuBar (HWND hwnd);
int GUIAPI TrackMenuBar (HWND hwnd, int pos);
int GUIAPI TrackPopupMenu (HMENU hmnu, UINT uFlags, int x, int y,
                HWND hwnd;
HMENU GUIAPI GetMenuBarItemRect (HWND hwnd, int pos, RECT* prc);
BOOL GUIAPI HiliteMenuBarItem (HWND hwnd, int pos, UINT flag);
```

上述函数用于显示并跟踪菜单。

```
int GUIAPI GetMenuItemCount (HMENU hmnu);
int GUIAPI GetMenuItemID (HMENU hmnu, int pos);
int GUIAPI GetMenuItemInfo (HMENU hmnu, int item,
                BOOL flag, PMENUITEMINFO pmii);
int GUIAPI GetMenuItemRect (HWND hwnd, HMENU hmnu, int item, PRECT prc);
HMENU GUIAPI GetPopupSubMenu (HMENU hpppmnu);
HMENU GUIAPI GetSubMenu (HMENU hmnu, int pos);
int GUIAPI GetSystemMenu (HWND hwnd, BOOL flag);
UINT GUIAPI EnableMenuItem (HMENU hmnu, int item, UINT flags);
int GUIAPI CheckMenuRadioItem (HMENU hmnu, int first, int last,
                int checkitem, UINT flags);
int GUIAPI SetMenuItemBitmaps (HMENU hmnu, int item, UINT flags,
                PBITMAP hBmpUnchecked, PBITMAP hBmpChecked);
int GUIAPI SetMenuItemInfo (HMENU hmnu, int item,
                BOOL flag, PMENUITEMINFO pmii);
```

上述这些函数用于获取或设置菜单属性。

# 七、定时器

在 MiniGUI 中应用项目可用的定时器总共可有 16 个，而每个应用最多只能定义 8 个定时器。

MiniGUI 中的定时器和 Win32 中使用定时器的方法是一样的，但不支持定时器回调函数。MiniGUI 提供来操作定时器的函数有：

```
BOOL GUIAPI SetTimer (HWND hWnd, int id, int speed);
BOOL GUIAPI KillTimer (HWND hWnd, int id);
BOOL GUIAPI SetTimerSpeed (HWND hWnd, int id, int speed);
```

如下的代码段建立了标识号为 100 的定时器：

```
SetTimer (hWnd, 100, 20);
```

然后在适当的时候删除了定时器：

```
KillTimer (hWnd, 100);
```

注意，在 Win32 中，创建定时器时指定定时器的时间间隔，而在 MiniGUI 中指定的是速度。

# 八、鼠标光标

MiniGUI 提供了和 Win32 基本兼容的鼠标光标操作，其中有光标的创建和销毁，系统光标支持，光标剪切，光标位置以及光标的显示和隐藏等。

下面的函数可用来创建、销毁鼠标光标，或获取系统鼠标光标：

```
HCURSOR GUIAPI LoadCursorFromFile(const  char* filename);
HCURSOR GUIAPI CreateCursor(int  xhotspot, int yhotspot, int w, int h,
        const BYTE* pANDBits, const BYTE* pXORBits, int colornum);
BOOL GUIAPI DestroyCursor(HCURSOR  hcsr);
HCURSOR GUIAPI GetSystemCursor(int  csrid);
```

函数 GetCurrentCursor 则可以返回当前的鼠标光标：

```
HCURSOR GUIAPI GetCurrentCursor(void);
```

下面的函数剪切鼠标光标的活动范围：

```
void GUIAPI ClipCursor(const  RECT* prc);
void GUIAPI GetClipCursor(RECT*  prc);
```

下面的函数同步获取或设置鼠标光标的位置，需要给定屏幕坐标：

```
void GUIAPI GetCursorPos(POINT*  ppt);
void GUIAPI SetCursorPos(int  x, int y);
```

SetCursor 函数可用来设置鼠标光标形状：

```
HCURSOR GUIAPI SetCursor(HCURSOR  hcsr);
```

ShowCursor 可隐藏或显示鼠标光标：

```
int GUIAPI ShowCursor(BOOL  fShow);
```

# 九、插入符

MiniGUI 将完全实现 Win32 对插入符的管理功能。包括如下函数：

```
CreateCaret
DestroyCaret
```

上述函数用来创建和销毁插入符。

```
GetCaretBlinkTime
GetCaretPos
HideCaret
SetCaretBlinkTime
SetCaretPos
ShowCaret
```

上述函数用来显示、隐藏或操作插入符属性。

# 十、键盘和鼠标输入

这部分内容应保持和 Win32 的 80% 兼容，在现有基础上还需要完成如下工作：

鼠标输入信息中键盘的标志信息，即 Ctrl、Alt、Shift 等键的信息。

键盘从原始键码到 ASCII 或 GB2312 码的翻译。

# 十一、键盘快捷键

这部分内容将保持和 Win32 的 90% 兼容，包含如下函数：

```
CopyAcceleratorTable
CreateAcceleratorTable
DestroyAcceleratorTable
LoadAccelerators  *
AddAccelerator  *
RemoveAccelerator  *
```

上述函数用来操作快捷键数据结构。因为没有资源文件，因此，LoadAccelerators 是实现从指定文件中装载快捷键的函数，AddAccelerator 和 RemoveAccelerator 函数用来向已有的快捷键中添加或删除快捷键。

```
TranslateAccelerator
```

该函数将快捷键翻译为对应的菜单命令。