

第1章 μ C/GUI的介绍

μ C/GUI

μ C/GUI 是一种用于嵌入式应用的图形支持软件。它被设计用于为任何使用一个图形 LCD 的应用提供一个有效的不依赖于处理器和 LCD 控制器的图形用户接口。它能工作于单任务或多任务的系统环境下。 μ C/GUI 适用于使用任何 LCD 控制和 CPU 的任何尺寸的物理和虚拟显示。它的设计是模块化的，由在不同的模块中的不同的层组成。一个层，称作 LCD 驱动程序，包含了对 LCD 的全部访问。 μ C/GUI 适用于所有的 CPU，因为它 100%由的 ANSI 的 C 语言编写的。

μ C/GUI 很适合大多数的使用黑色/白色和彩色 LCD 的应用程序。它有一个很好的颜色管理器，允许它处理灰阶。 μ C/GUI 也提供一个可扩展的 2D 图形库和一个视窗管理器，在使用一个最小的 RAM 时能支持显示窗口。

本文档的目的

本指南描述如何安装，配置和在嵌入式应用中使用 μ C/GUI 图形用户界面。它也说明了软件的内部结构。

假设

本指南假定你对 C 编程语言已经具有一个扎实的认识。

如果你觉得你对 C 语言的认识不是很充分的话，我们推荐该由 Kernighan 和 Richie 编写的“C 语言编程语言”给你，它描述了程序设计标准，而在新版中，也包含了 ANSI 的 C 语言标准。汇编语言编程的知识不需要。

1.1 需求

在你使用 μ C/GUI 进行软件开发时，并不需要一个目标系统；只需要使用模拟器，大多数软件就能够进行开发。然而，最后的目的通常是能够在一个目标系统上运行该软件。

目标系统（硬件）

你的目标系统必须：

- 有一个 CPU（8/16/32/64 位）
- 有最少的 RAM 和 ROM
- 有一个完全的图形 LCD（任何类型和任何分辨率）

内存需求的变化取决于软件的哪些部分被使用以及你的目标编译程序的效率有多高。所以指定精确值是不可能的，但是下面的数值适合典型系统。

小的系统（没有视窗管理器）

- RAM：100 字节
- 堆栈：500 字节
- ROM：10~25KB（取决于使用的功能）

大的系统（包括视窗管理器和控件）

- RAM：2~6KB（取决于所需窗口的数量）
- 堆栈：1200 字节
- ROM：30~60KB（取决于使用的功能）

注意，如果你的应用程序使用许多字体的话，ROM 的需求将增加。以上所有的数值都是粗略的估计，不能得到保证。

开发环境（编译程序）

使用什么样的 CPU 并不重要；仅仅需要一个与 ANSI 兼容的 C 编译器。如果你的编辑器有一些限制，请告知我们，我们将通知你在编译软件时是否会带来问题。我们所知道的任何用于 16/32/64 位 CPU 或者 DSP 的编译器都可以使用；大多数的 8 位编译器也可以使用。

一个 C++编译器并不需要，不过可以使用。因此，如果想要的话，应用程序也可以用 C++

语言来编制。

1.2 μ C/GUI 的特点

μ C/GUI 被设计用于给使用一个图形 LCD 的任何应用程序提供一个高效率的，与处理器和 LCD 控制器无关的图形用户界面。它适合于单一任务和多任务环境，专用的操作系统或者任何商业的实时操作系统（RTOS）。 μ C/GUI 以 C 源代码形式提供。它可以适用于任何尺寸的物理和虚拟显示，任何 LCD 控制器和 CPU。其特点包括下列这些：

一般特点

- 任何 8/16/32 位 CPU；只需要一个与 ANSI 兼容的 C 编译器。
- 任何控制器支持（如果有合适的驱动程序）的任何（单色的，灰度级或者彩色）LCD。
- 在较小显示屏上，可以不要 LCD 控制器工作。
- 使用配置宏可以支持任何接口。
- 显示屏大小可配置。
- 字符和位图可能是写在 LCD 上的任一点，而不仅仅局限于偶数的字节的地址。
- 程序对大小和速度都进行了最优化。
- 允许编译时的切换以获得不同的优化。
- 对于较慢的 LCD 控制器，LCD 能够被存储到内存当中，减少访问的次数使其最小，从而得到非常高的速度。
- 清晰的结构。
- 支持虚拟显示；虚拟显示能够比实际的显示表现更大尺寸的内容。

图库

- 支持不同颜色深度的位图。
- 有效的位图转换器。
- 绝对没有使用浮点运算。
- 快速线/点绘制（没有使用浮点运算）。
- 非常快的圆/多边形的绘制。
- 不同的绘画模式。

字体

- 为基本软件提供了不同种类的字体：4*6，6*8，6*9，8*8，8*9，8*16，8*17，8*18，24*32，以及 8，10，13，16 等几种高度（以像素为单位）的均衡字体。更多的信息，参见第 25 章：“标准字体”。

- 可以定义和简便地链接新的字体。
- 只有用于应用程序的字体才实际上与执行结果链接，这样保证了最低的 ROM 占用。
- 字体可以分别在 X 轴和 Y 轴方向上充分地缩放。
- 提供有效的字体转换器，任何在你的主系统（即 Microsoft Windows）上的有效字体都可以转换。

字符串/数值输出程序

- 程序支持任何字体的十进制，二进制，十六进制的数值显示。
- 程序支持任何字体的十进制，二进制，十六进制的数值编辑。

视窗管理器（WM）

- 完全的窗口管理器包括剪切在内。一个窗口的外部区域的改写是不可能的。
- 窗口能够移动和缩放。
- 支持回调函数（可选择用法）。
- WM 使用极小的 RAM（大约每个窗口 20 字节）。

可选择用于PC外观的控件

- 控件（窗口对象）有效。它们一般自动运行，并且易于使用。

触摸屏和鼠标支持

- 对于比如按钮控件之类的窗口对象， μ C/GUI 提供触摸屏和鼠标支持。

PC工具

- 模拟器及观察器。
- 位图转换器。
- 字体转换器。

范例和演示

关于 μ C/GUI 能做什么，为了给你一个更好的概念，我们准备有不同的演示作为可运行的仿真程序，在目录 sample\EXE 下。范例程序的源代码位于 Sample 目录。文件夹 Sample\GUIDemo 包括一个展示大部分 μ C/GUI 特点的应用程序。

1.3 估价板

一个完全的评估板包括一个带有 LCD 的演示板，一个 C 语言编译器和一个有效的范例工程。它已经设计好，主要测试和验证 μ C/GUI，并且它可用于熟悉这个软件。

评估板

评估板包括 Mitsubishi M30803 CPU 和 SED13705 LCD 控制器（包括原理图和技术资料）。

LCD（320×240 像素）或者单色的 LCD，1/4VGA 彩色显示 LCD 或者 TFT。

更详细的资料，请访问我们的网站：www.micrium.com。



1.4 如何使用本手册

该手册说明了如何安装，配置和使用 μ C/GUI。它描述了软件的内部结构和 μ C/GUI 提供的所有的功能（应用程序接口，或者 API）。

在实际上使用 μ C/GUI 之前，你应该阅读或者至少浏览本手册，对该软件做到耳濡目染。推荐下列步骤：

- 拷贝 μ C/GUI 文件到你的电脑。
- 仔细研究第 2 章：“入门指南”。
- 使用模拟器多熟悉一理这个软件能作什么（参考第 3 章：“仿真器”）。
- 使用本手册其余部分的参考资料扩展你的程序。

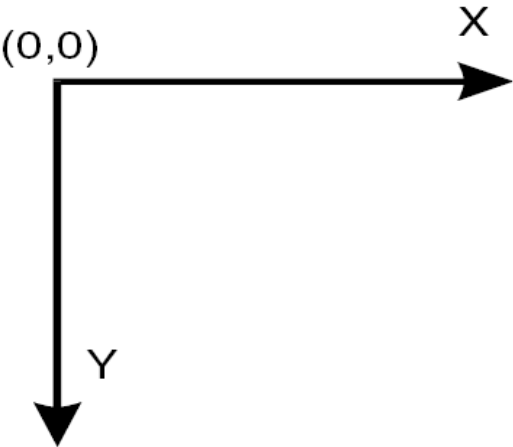
排版上的语约定

本手册使用下列印刷惯例：

类型	用于
Body	正文文字。
Keyword	那些你在命令-提示中输入的文字，或者那些能在显示屏上看得见的文字（即系统函数，文件或者路径名）。
Parameter	API 函数中的参数。
Sample	在程序范例中的范例代码。
New Sample	那些已经被加到一个已存在有程序范例中的范例代码。

1.5 屏幕和坐标

屏幕由能够被单独控制的许多点组成。这些被称作像素。大部分 μ C/GUI 在它的 API 中向用户程序提供的文本和绘图函数能够在任何指定像素上写或绘制。



水平刻度被称作 X 轴，而垂直刻度被称作 Y 轴。一个二维坐标用 X 轴和 Y 轴坐标表示，即值 (X, Y)。在程序中需要用到 X 和 Y 坐标时，X 坐标总在前面。显示屏（或者一个窗口）的左上角为一默认的坐标 (0, 0)。正的 X 值方向被总是向右；正的 Y 值方向总是向下。上图说明该坐标系和 X 轴和 Y 轴的方向。所有传递到一个 API 函数的坐标总是以像素为单位所指定。

1.6 如何连接LCD到微控制器

μ C/GUI 处理所有的 LCD 访问。事实上任何 LCD 控制器都能够被支持，不取决于它是如何访问的。至于细节，请参阅第 20 章：“低层配置”。此外，如果你的 LCD 控制器不被支持的话，请与我们联系。我们目前为全部有销售 LCD 控制器编写驱动程序，对于你打算使用

的 LCD 控制器已经有一个经过验证的驱动程序提供。在你的应用程序中写这样的用于访问 LCD 的程序（或者宏）通常非常简单。如果你的目标硬件有需要的话，Micrium 公司可以提供定制的服务。

LCD 如何与系统连接并不真的重要，只要它通过软件以某种方式达到，可能是按多种方式完成的。大多数这些接口通过一个提供源代码方式的驱动程序来支持。这些驱动程序通常不需要修改，但是用于你的硬件，要通过修改文件 LCDConf.h 进行配置。有关根据需要如何定制一个驱动程序到你的硬件在第 22 章：“LCD 驱动程序”中说明。最通用的访问 LCD 的方式如下所描述。如果你只是想领会如何使用 μ C/GUI，你可以跳过本节。

带有存储映像LCD控制器的LCD

LCD 控制器直接连接到系统的数据总线，意思是能够如同访问一个 RAM 一样访问控制器。这是一个很有效的访问 LCD 控制器方法，最值得推荐。LCD 地址被定义为段 LCDSEG，为了能访问该 LCD，连接程序/定位器只需要告知这些段位于什么地方。该位置必须与物理地址空间中访问地址相吻合。驱动程序对于这类接口是有效的，并且能用于不同的 LCD 控制器。

带有LCD控制器的LCD连接到端口/缓冲区

对于在快速处理器上使用的较慢的 LCD 控制器，端口-连线的使用可能是唯一的方案。这个访问 LCD 的方法有稍微比直接总线接口慢一些的缺点，但是，特别是使用一个减少 LCD 访问次数的高速缓存的情况，LCD 刷新并不会有很大的延迟。所有那些需要处理的是定义程序或者宏，设置或者读取与 LCD 连接的硬件端口/缓冲区。这类接口也被用于不同的 LCD 控制器的不同的驱动程序所支持。

特殊方案：没有LCD控制器的LCD

LCD 可以不需要 LCD 控制器而进行连接。在这种情况下，LCD 数据通常通过控制器经由一个 4 或 8 位移位寄存器直接提供。这些特殊的硬件方案有价格便宜的优点，但是使用上的缺点是占用了大部分有效的计算时间。根据不同的 CPU，这会占到 CPU 的开销的 20%到几乎 100% 之间；对于较慢的 CPU，它根本是极不合理的。这类接口不需要一个特殊的 LCD 驱动器，因为 μ C/GUI 简单地将所有显示数据放入 LCD 高速缓存中。你自己必须写硬件相关部分软件，周期性地数据从高速缓存的内存传递到你的 LCD。

对于 M16C 和 M16C/80，传递图像到显示屏中的范例代码可以用“C”和优化的汇编程序实现。

1.7 数据类型

因为 C 语言并不提供与所有平台相吻合的固定长度的数据类型，大多数情况下， μ C/GUI 使用它自己的数据类型，如下表所示：

数据类型	定义	说明
I8	signed char	8 位有符号值
U8	unsigned char	16 位无符号值
I16	signed short	16 位有符号值
U16	unsigned short	16 位无符号值
I32	signed long	32 位有符号值
U32	unsigned long	32 位无符号值
I16P	signed short	16 位（或更多）有符号值
U16P	unsigned short	16 位（或更多）无符号值

对于大多数 16/32 位控制器来说，该设置将工作正常。然而，如果你在你的程序的其它部分中有相似的定义，你可能想对它们进行修改或者重新配置。一个推荐的位置是置于配置文件 LCDConf.h 中。

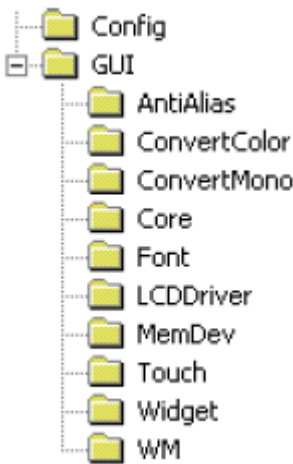
第2章 入门指南

这一章提供一个在你的目标系统上设置和配置 μ C/GUI的基本处理过程的概述。同时也包括了一个简单的范例程序。

请注意，大多数主题在后面的章节会有更详细的描述。在你开始更复杂的编程之前，你很有必要参阅本手册的其它部分。

2.1推荐的结构

使μC/GUI 和你的应用文件分离，这是我们推荐的。在工程文件的“root”目录的 GUI 子目录下保留所有的程序文件（包括头文件），这是一个好的习惯。目录结构应该和下图相似。这种习惯有一个好处，就是很容易升级更新版本的μC/GUI，只需要替换 GUI 目录就可以。



子目录

下表显示了 GUI 所有子目录的内容

目 录	内 容
Config	配置文件
GUI/AntiAlias	抗锯齿支持 *
GUI/ConvertMono	用于 B/W（黑白两色）及灰度显示的彩色转换程序
GUI/ConvertColor	用于彩色显示的彩色转换的程序
GUI/Core	μC/GUI 内核文件
GUI/Font	字体文件
GUI/LCDDriver	LCD 驱动
GUI/Mendev	存储器件支持 *
GUI/Touch	触摸屏支持 *
GUI/Widget	视窗控件库 *
GUI/WM	视窗管理器 *

（带“*”标志的为可选项）

“Include” 目录

确认你的 Include 路径包括有以下目录（包括的先后顺序并不重要）：

- Config
- GUI/Core
- GUI/Widget（如果使用视窗控件库）
- GUI/WM（如果使用视窗管理器）

警告：你必须确认你在每个文件中只使用了一个版本的 μ C/GUI

2.2 向目标程序加入 μ C/GUI

你主要是在这两者之间做一个选择，一是将你要在你的工程中使用的源文件包括进来，然后进行编译和连接；或者建立一个库并连接这个库文件。如果你的链接工具支持“智能化”连接（仅仅连接那些使用到的模块而不是那些没有使用到的模块），那么就完全没有必要建立一个库，因为只是要求将函数和数据结构进行连接。如果你的工具链接不支持“智能化”连接，建立一个库就很有意义了，否则如果将每样东西都要进行连接的话，程序会变得非常大。对于一些 CPU 来说，我们能提供有效的范例工程帮助你开始使用。

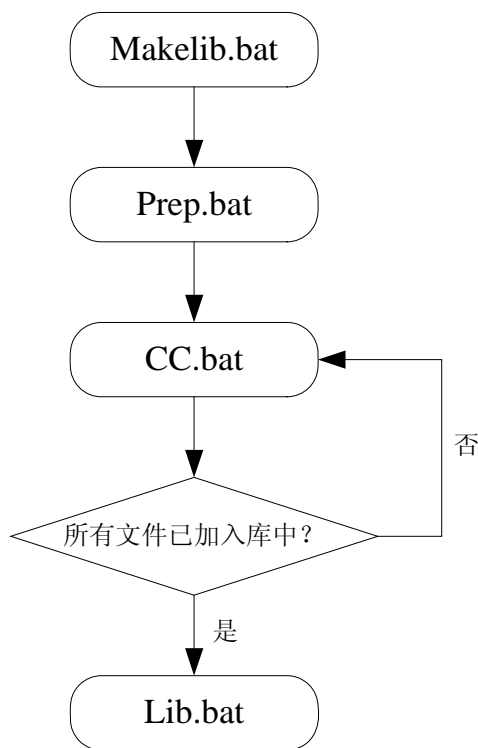
2.3 建立一个库

从源程序建立一个库是一个简单的流程。第一步是拷贝批处理文件（位于“Sample\Makelib”目录下）到根目录下。然后，做一些必要的修改。总共有四个批处理文件需要拷贝，如下表描述的那样。主文件“Makelib.bat”在所有的系统中都是一样的，所以无需修改。在你的目标系统上建立一个库，正常情况下你需要对其它三个比较小的文件做一些微小的改动。最后执行“Makelib.bat”文件建立库。批处理文件假定你的 GUI 和配置子目录已经如前面所推荐的那样建立起来了。

文 件	说 明
Makelib.bat	主批处理文件，不需要修改
Prep.bat	由 Makelib.bat 调用，建立用于链接工具的工作环境
CC.bat	由 Makelib.bat 调用，对库所用到的文件进行处理，为这些目标文件建立一个列表，该列表在下一步中由 lib.bat 中的
lib.bat	由 Makelib.bat 调用，将列表中的目标文件置入一个库当中

建立库的流程如下图所示。Makelib.bat 文件首先调用 Prep.bat 准备用于链接工具的环境。然后调用 CC.bat 处理库当中所包括的每一个文件，做完这些工作需要一些时间。CC.bat

将这些目录文件加入一个列表，这个列表是 lib.bat 要使用的。当所有加入到库当中的文件已经写入列表后，Makelib.bat6 调用 lib.bat，使用一个库管理工具将列表中的目标文件置入一个活动的库当中。



我们假设一个微软编译器已经安装到它的默认位置。如果所有的批处理文件都拷贝到根目录（GUI 的上一级目录），并且不作任何修改，将会产生一个用于 μ C/GUI 仿真的仿真库。无论如何，要建立一个目标库的话，必须要对 Prep.bat、CC.bat 和 lib.bat 三个文件进行修改。

2.4 将 μ C/GUI的“C”文件加入工程中

通常说法，你需要加入 μ C/GUI 的核心“C”文件，LCD 驱动，你显示屏所使用的字体文件及其它你定制可选择的模块：

- 目录 GUI\Core、GUI\ConvertColor 及 GUI\ConvertMono 下的所有“C”文件
- 你的显示屏用到的字体（位于目录 GUI\Font 下）

附加的软件包

如果你的显示屏使用附加的可选的模块，你必须也要包括相关的“C”文件

2.5 配置 μ C/GUI

配置目录应该包含与你的要求相匹配的配置文件。文件 LCDConf.h 通常包含所有的需要的定义，使你能够为你的 LCD 使用 μ C/GUI，这是开始配置 μ C/GUI 的主要任务。了解更多的细节，请参阅第 20 章“底层配置”。

如果因为你没有选择正确的显示方案或选择了错误的 LCD 控制器而导致 μ C/GUI 没有正确配置，LCD 可能不会显示任何东西，或者显示些不是你所期望的内容。因此，要注意修改你所需要的 LCDConf.h。

配置宏的类型

下面是一些配置宏的类型：

二进制开关“B”

这个开关的数值是“0”或“1”，“0”表示不激活，而“1”表示激活（除了“0”以外的数值都可以激活，但是使用“1”使配置文件更易于阅读）。这些开关能够启用或禁止某一个功能或行为。开关是配置宏中最简单的格式。

数值“N”

数值有代码中某些地方使用，以替代数值常量。在 LCD 配置方案中有一个典型的例子。

选择开关“S”

选择开关用于从多个选项中选择一项（只能选中一项）。典型的例子是用于所使用的 LCD 控制器的选择，选择的数值指示调用相应源代码（相应的 LCD 驱动）产生目标代码。

别名“A”

一个类似于简单的文本替代这样操作的宏。一个典型例子是定义 U8，预处理程序会用“unsigned char”代替“U8”。

函数替换“F”

该宏基本被视为一个正常的函数，尽管有某些应用上的限制，宏依旧被放入代码当中，就象文本代换的例子一样。函数替换主要用于给一个高度依赖硬件的模块增加一些特殊的函

数（例如 LCD 的访问），这类宏通常使用括弧（与可选择参数一起）来声明。

2.6 初始化μC/GUI

程序 GUI_Init() 初始化 LCD 和μC/GUI 的内部数据结构，在其它μC/GUI 函数运行之前必须被调用。这通过将下面一行放入你的程序序列的开始来做到：

```
GUI_Init();
```

如果忽略了这个调用，整个图形系统将不会得到初始化，从而无法准备下一步的动作。

2.7 在目标硬件上使用μC/GUI

下面所陈述是只是我们使用μC/GUI 进行编程的一些基本的步骤要点。这些步骤更详细的解释在以后的章节介绍。

第一步：定制μC/GUI

通常第一步是通过修改头文件 LcdConf.h 来定制μC/GUI。你必须定义一些基本数据类型（U8，U16 等），及有关显示方案和所使用的 LCD 控制器的开关配置。

第二步：定义访问地址和访问规则

对于使用存储器映象的 LCD，仅仅需要在 LcdConf.h 中定义访问地址。

对于端口/缓冲的 LCD，必须定义接口程序，在 Samples\Lcd_x 目录下，或是在我们 Web 站点的下载区的中，有一些所需的接口程序的范例代码可供参考。

第三步：编译、连接和测试范例程序

μC/GUI 带有一些单任务和多任务环境下的范例程序，编译、连接和测试这些范例程序，直到你感觉已经熟悉它们了。

第四步：修改范例程序

对范例程序进行简单的修改，增加些额外的命令，诸如在显示时显示不同尺寸的文字，显示一条线等等。

第五步：多任务应用：适应你的操作系统（如果需要的话）

如果多任务允许同时访问显示器，则宏 GUI_MAXTASK 和 GUI_OS 与文件 GUITask.cg 一道开始运行。更详细的内容及范例程序的修改请参考第 21 章：高层次配置。

第六步：使用μC/GUI 编写你的应用程序

到现在，你应该对如何使用μC/GUI 应该有一个清楚的了解。考虑如何去构建你的应用要求的程序，通过调用适当的程序来使用μC/GUI。参考本手册后面相关的章节，这些章节讨论特殊的μC/GUI 函数和配置有效的宏。

2.8 “Hello World” 范例程序

在早些时候，一个“Hello World”程序被做为 C 语言编程的入门程序，因为它本质上是一个能写出的最简单的程序。μC/GUI 的“Hello World”程序的名称是 Hello.c，如下所示。在μC/GUI 所带的范例中的它的名称为 Basic_HelloWorld.c。

该程序的目的是在显示器的左上角写“Hello World”，为了能够实现这个功能，应用硬件，LCD 和 GUI 必须首先要初始化。μC/GUI 的初始化通过在程序开始调用 GUI_Init() 来实现，就象先前所描述的那样。在本程序中，我们假设应用硬件的初始化已经完成。

```

/*-----
文件：      BASIC_HelloWorld.c
目的：      绘制“Hello world”的简单范例
-----*/

#include "GUI.H"

/*****
*                               *
*****/

void main(void)
{
/* 要做的事：确认硬件首先初始化了！ */
    GUI_Init();
    GUI_DispString("Hello world!");
    while(1);

```

```
}
```

给“Hello Word”程序增加功能

我们的小程序能做的工作实在太少，现在我们对它扩展一点功能：在显示“Hello World”后，我们希望程序开始计数以估计能够获得多快的输出速度（至LCD）。我们在主程序末尾的仅仅增加一点点代码进行循环，本质上是调用一个显示十进制形态数值的函数。

```
/*-----
文件:      BASIC>HelloWorld1.c
目的:      绘制“Hello world”的简单范例
-----*/

#include "GUI.H"

/*****
*                               *
*                               * 主函数 *
*                               *
*****/

void main(void)
{
    int i=0;
    /* 要做的事：确认硬件首先初始化了！ */
    GUI_Init();
    GUI_DisString("Hello world!");
    while(1)
    {
        GUI_DisDecAt( i++, 20, 20, 4);
        if(i>9999) i=0;
    }
}
```


第3章 仿真器

μ C/GUI的PC仿真器允许你在Windows下编译相同的“C”源程序。PC使用一个本地编译器（一般是微软所提供的）并建立一个用于你自己应用的可执行文件。这样做可能完成：

- 在你的PC上进行用户接口设计（不需要硬件支持）
- 调试你的用户接口程序
- 建立你的应用的演示，可以用于描述用户接口

最终的可执行文件能够很容易通过E-Mail传送。



3.1 理解仿真器

μ C/GUI仿真器使用微软Visual C++（6.0或更高版本）及其所带的集成开发环境（IDE）。你能够在PC屏幕上看到你的LCD仿真效果，一旦正确配置你的LCD后仿真效果能提供与你的LCD在X轴和Y轴上相同的分辨率及同样精确的颜色。

仿真的整个图形库API和视窗管理API与你的目标系统是一样的；所有函数运行与在目标硬件上运行高度一致，因为仿真时使用了与目标系统同样的“C”源代码。唯一不同是在软件的底层：LCD驱动。PC仿真使用一个仿真的驱动写入一个位图，以代替实际的LCD驱动。在你的屏幕上显示的位图使用第二个仿真线程。第二个线程在实际应用并不存在，它只是在LCD程序被直接写屏时运行。

3.2 在评估版 μ C/GUI中使用仿真器

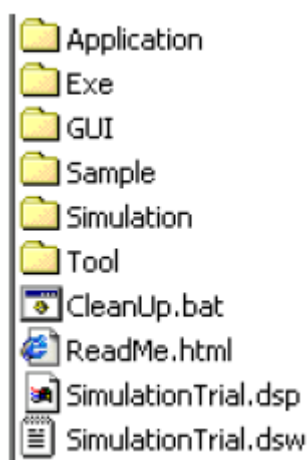
版本

μ C/GUI 评估版包括一个完全的库，允许你评估 μ C/GUI的所有特性。同时也包括 μ C/GUI观察器（用于调试应用程序）及字体转换器和位图转换器的演示版。

请记住这些，做为一个评估版本，你不能改变任何配置或者察看源代码，但是你仍然能够熟悉 μ C/GUI的使用。

目录结构

评估版仿真器的目录结构如下图所示。



目录“Application”包括演示程序的源代码。

目录“Exe”包括一个“ready-to-use”演示程序。

目录“GUI”包括库文件和库使用的包含文件。

目录“Sample”包括仿真范例及其源代码。

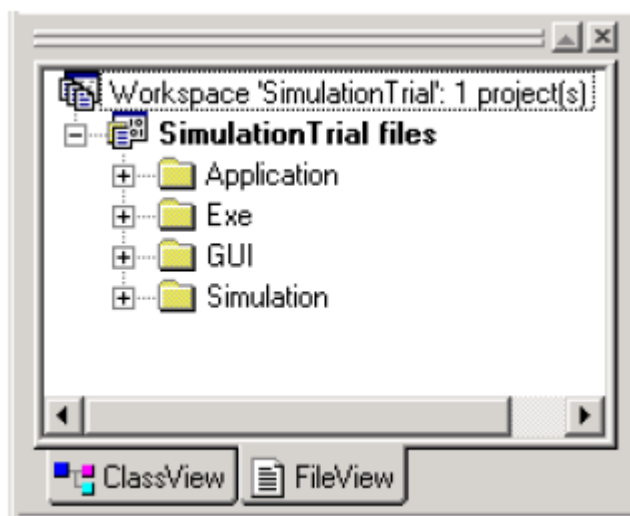
目录“Simulation”包括仿真所需的文件。

目录“Tool”包括 μ C/GUI观察器，一个演示版的位图转换器和一个演示版的字体转换器。

Visual C++工作区

上面所示的根目录包括微软Visual C++工作区（Simulation-Trial.dsw）及项目文件（Simulation-Trial.dsp）。双击工作区文件可以打开微软IDE。

Visual C++工作区的目录结构如下图所示。



编译演示程序

位于应用目录下的演示程序源文件是一个“ready-to-go”仿真，意思是你仅仅需要建立和启动它。请注意，如果需要建立可执行文件，你必须先安装微软Visual C++（6.0或以上的版本）。

- 第一步：双击Simulation-Trial.dsw 文件打开Visual C++工作区。
- 第二步：在菜单中选择“Build/Rebuild All”（或按“F7”键）重建项目。

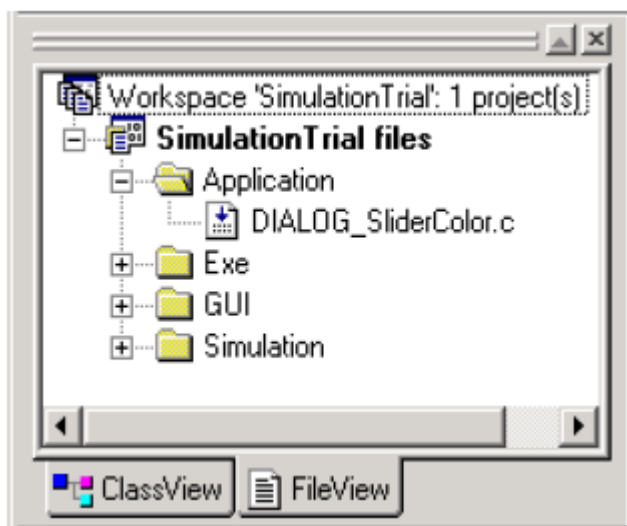
- 第三步：在菜单中选择“Build/Start Debug/Go”（或按“F5”键）开始仿真。

演示项目开始运行，在任意时候可能通过单击右键并选择“Exit”退出。

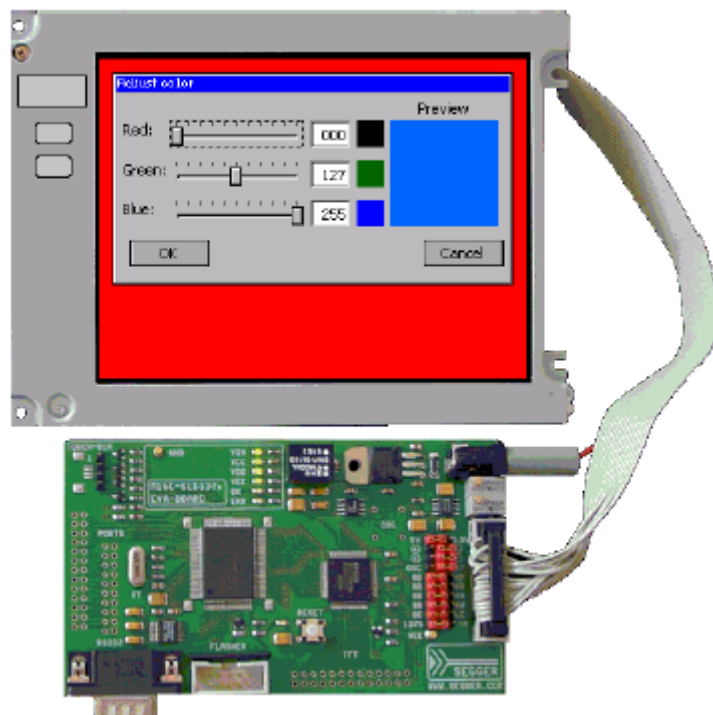
编译范例程序

目录“Sample”包括“ready-to-go”范例程序，可以示范 μ C/GUI的不同特性及提供它们的典型应用的例子。为了建立这些可执行文件，它们的C源代码必须加入项目中。通过下面的步骤很容易做到：

- 第一步：双击Visual C++工作区的“Application”文件夹。演示文件会出现在它下面。
- 第二步：选择“Application”文件夹下的所有文件，按下“Delete”键将它们删除。这些文件并不是真是被删除了，只是从项目中移走。
- 第三步：现在你有了一个空的“Application”文件夹。在其上面单击右键，选择需加入的文件加入到文件夹，出现一个对话框。
- 第四步：双击“Sample”文件夹，选择里面的一个范例文件。你的工作区目录应该如下图所示。当然，文件名可以不一样；在这里，很重要的一件事是“Application”文件夹只能包含你所想编译的范例的C文件，而不能是其它种类的文件。



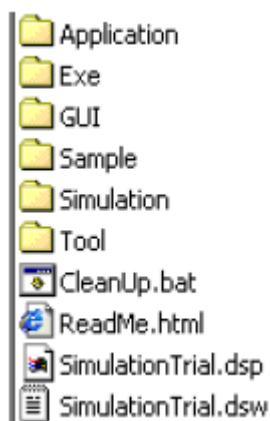
- 第五步：在菜单下选择“Build/Rebuild All”（或按“F7”键）重建范例文件。
- 第六步：在菜单中选择“Build/Start Debug/Go”（或按“F5”键）开始仿真。上面所选择范例的仿真结果如下图所示：



3.3 使用 μ C/GUI 源代码的仿真器

目录结构

仿真器的根目录可以在PC上的任意位置，例如：C:\work\GSCSim。目录结构如下图所示，该目录结构与我们推荐的用于目标应用的目录结构很相似（参阅第2章：“入门指南”以获得更多信息）。子目录GUI包括 μ C/GUI程序文件，用于目标（交叉）编译器的同名目录中也要有同名的文件。你不应对GUI子目录做任何改变，因为这样会使 μ C/GUI升级到新的版本变得困难。配置目录包括需要修改的配置文件，以反映你的目标硬件设置（主要是LCD尺寸和能够显示的颜色）。

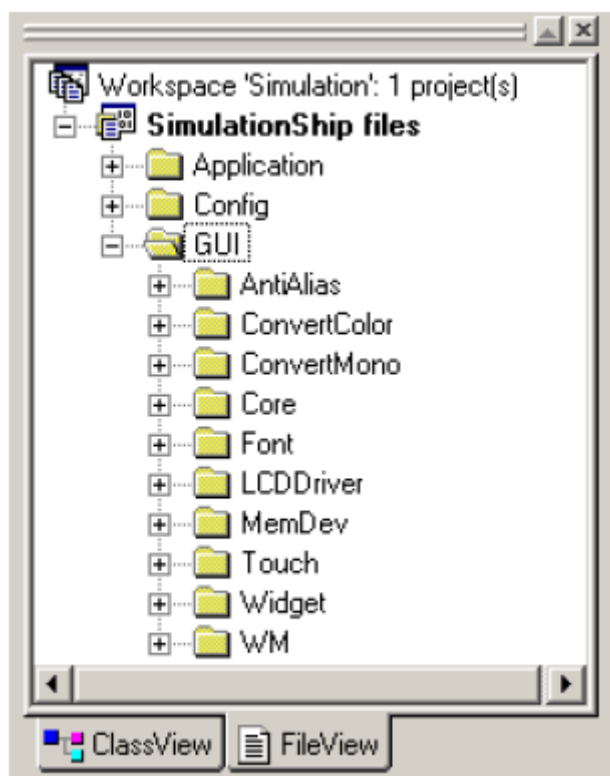


Visual C++工作区

上面所显示的根目录包括微软Visual C++工作区（Simulation.dsw）和项目文件（Simulation.dsp）。

工作区允许你在编译应用程序将其用于目标系统之前进行修改及调试。

Visual C++ 工作区的目录结构与下图所示的相似。在这，GUI文件夹是打开的，显示 μ C/GUI子目录。请注意，你的GUI目录可能与下图并不完全一样，这取决于你所拥有的 μ C/GUI的附加特性。文件夹“Core”，“Font”及“LCDDriver”是基本 μ C/GUI软件包的一部分，总在工作区目录中显示。



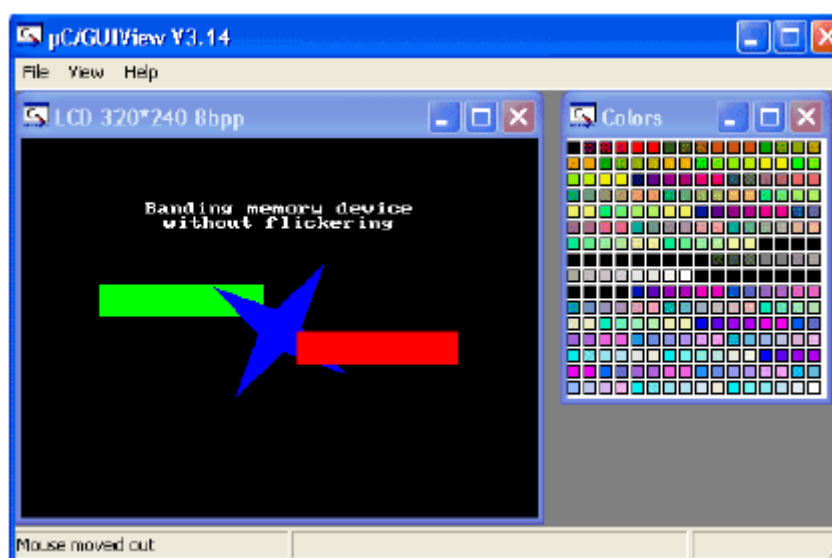
编译你的应用程序

仿真演示包括一个或多个可以修改的C文件（在“Application”目录下），你也可以在项目中增加或删除文件。最典型的是你至少应该把位图修改成为你公司的徽标或所选择的图片。你应该在Visual C++ 工作区中重建程序以进行测试及调试。一旦你获得一个满意的结果并打算在你的应用中使用该程序，你应该能够在目标系统中编译这些同样的文件，得目标显示上得到同样的结果。使用仿真器的通常的处理步骤如下所述：

- 第一步：双击Simulation.dsw 文件打开Visual C++工作区。
- 第二步：在菜单下选择“Build/Rebuild All”（或按“F7”键）编译项目。
- 第三步：在菜单中选择“Build/Start Debug/Go”（或按“F5”键）开始仿真。
- 第四步：使用你的徽标或图片代替原有的位图。
- 第五步：如果有需要，对应用程序进行更大的修改，这通过编辑源代码和增加/删除文件来完成。
- 第六步：在Visual C++中编译及运行应用程序测试结果，根据你的需要进行继续修改和调试。
- 第七步：在你的目标系统上编译及运行应用程序。

3.4 观察器

如果你使用仿真器调试你的应用程序，当你对源代码执行单步调试时无法看到LCD输出。观察器可能解决这个问题，它能显示仿真的LCD窗口和色彩窗口。观察器的执行文件是“Tool\μC-GUI-View.exe”。



使用仿真器和观察器

如果你在调试应用程序之前或正在调试时，想启动观察器，同时使用仿真器及观察器是你的选择。我们建议：

- 第一步：启动观察器。在仿真开始前，没有LCD或色彩窗口出现。
- 第二步：打开Visual C++ 工作区。
- 第三步：编译和运行应用程序。

- 第四步：如先前描述的一样调试应用程序。

优点是，当你使用单步操作时，能在LCD窗口中显示所有相应绘图操作。观察器窗口默认的位置总是在顶部，你可能通过在菜单中选择“View\Always on top”修改这个行为。

3.5 设备仿真及其它高级特性

警告：设备仿真和以其为基础的其它特性，是高级特性，可以对仿真器源代码提出要求以使其能工作于目标系统。通常这些源代码不随 μ C/GUI一起提供。请和我们联系得到更多的信息。

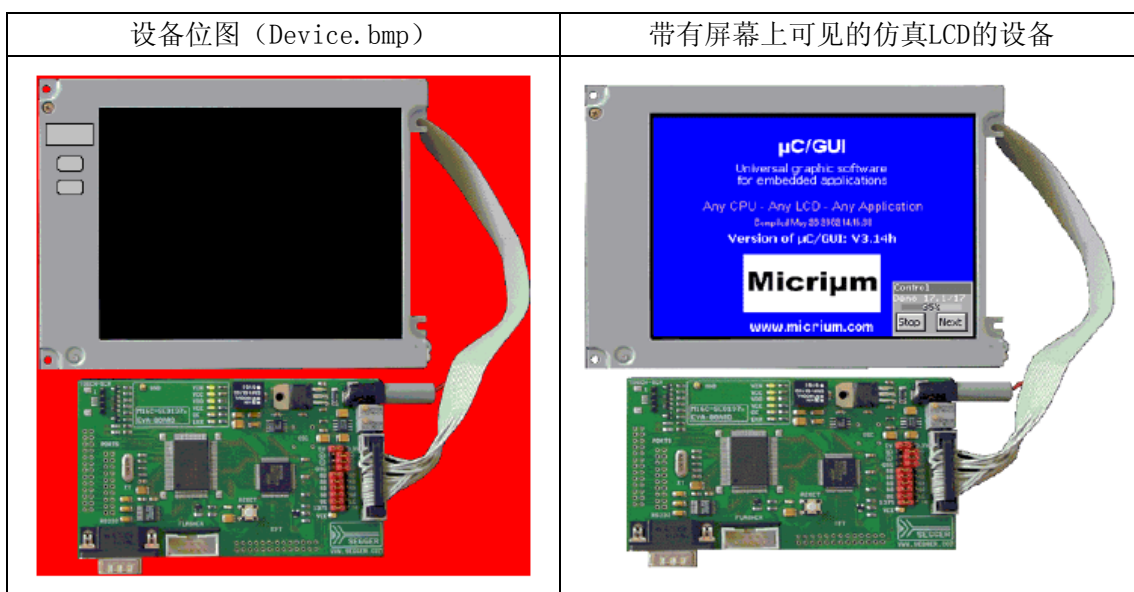
仿真器能够在你所选择的一张位图中显示仿真的LCD，例如你的目标设备的图片。该位图能够在屏幕上任意拖动，在某些应用中，可以用于仿真整个目标设备的行为。

为了仿真设备的外观，需要一幅位图。该位图通常是设备的照片（顶视图），必须命名为Device.bmp。它可以是一个独立文件（在同一目录下作为一个可执行文件），或是做为一个资源包括进应用程序当中，下面一行表示该位图文件包括在资源文件（extension.rc）中：

```
145 BITMAP DISCARDABLE "Device.bmp"
```

更多的信息，请参考Win32文档资料。

位图的尺寸应该是这样的，位图中LCD区域在屏幕上显示的尺寸应该等于仿真LCD的分辨率。在下面的例子中这是最好的显示效果：



红色区域自动成为透明区。透明区没有必要是矩形；它们可以是任意的形状（甚至可以

是你的操作系统所限制的复杂形状，但是一般的就足够了）。亮红色（0xFF0000）是默认的透明区域的颜，主要因为在大多数位图中很少用到这种颜色。如果位图中含有亮红色，你可以在函数SIM_SetTransparentColor中改变默认的透明色。

Hardkey 仿真

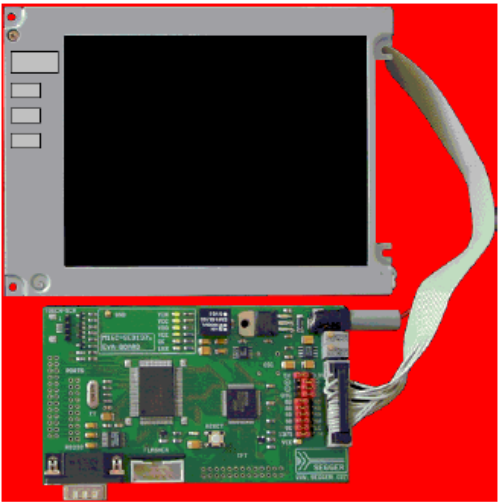
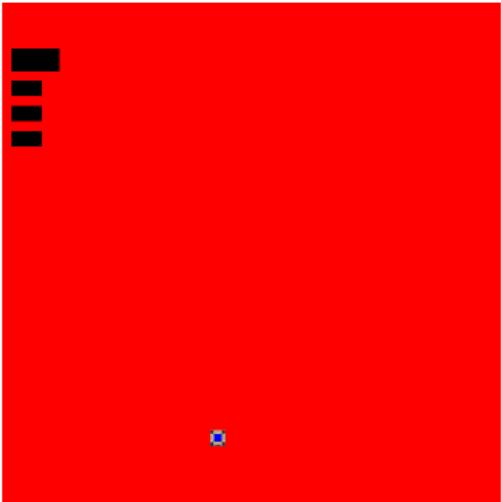
Hardkey也可以作为设备的一部分进行仿真，它可以由鼠标指针选择。意思是能区别在仿真设备中的一个键或一个按钮是否被按下或没有按下。当鼠标指针位于一个Hardkey上方，并且鼠标按键保持按下状态，则该Hardkey被认为是按下。当鼠标按键释放或指针移开Hardkey，表示该Hardkey“没有按下”。在“按下”和“非按下”之间的切换行为也可以在程序SIM_HARDKEY_SETmode中说明。

为了仿真Hardkey，你需要第二幅设备位图，除了按键自己（按下状态）以外，都是透明的。该位图也作为目录里的一个独立文件或包括在可执行文件中做为一个资源。

文件名需要定为“Device1.bmp”，典型的，下面两行表示两们位图文件包括在资源文件（extension.rc）中：

```
145 BITMAP DISCARDABLE "Device.bmp"
146 BITMAP DISCARDABLE "Device1.bmp"
```

尽管Hardkey可以是任意的形状，但有一点很重要，就是两幅位图尺寸必须是一样的，即在像素上一样，这样在Device1.bmp上的Hardkey就能够正确地覆盖在Device.bmp相应的位置上。下面的例子说明了这种情况：

设备位图：Hardkey非按下的状态 (Device.bmp)	设备Hardkey位图：Hardkey按下的状态 (Device1.bmp)
	

当一个键被鼠标“按下”，hardkey 位图（Device1.bmp）的相应部分将覆盖设备位图以显示该键处于其按下状态。

键可以周期性的轮询以确定它们的状态（按下/非按下）是否已经改变以及它们是否需要更新。二者选一，当一个hardkey的状态改变时，一个回调函数会被设置以触发一个自带的特殊动作。

3.6 仿真器API

所有仿真器的API函数在设置阶段必须被调用。调用应该从函数SIM_X_Init()内部被完美执行，该程序位于文件SIM_X.c中。下面的例子表示在设置中调用SIM_SetLCDPos()：

```
*/
#include <windows.h>
#include <stdio.h>
#include "SIM.h"
void SIM_X_Init()
{
    SIM_SetLCDPos(0,0);    // 定义LCD在位图中的位置
}
```

下表列出了与仿真相关有用的函数，在各自的类型中按字母进行排列。函数的详细描述在后面列出。

函 数	说 明
设备仿真	
SIM_SetLCDPos()	在目标设备位图中设置仿真LCD的位置
SIM_SetTransparentColor()	设置用于透明区域的颜色
Hardkey仿真	
SIM_HARDKEY_GetNum()	返回一个有效的Hardkey的号码
SIM_HARDKEY_GetState()	返回一个指定的Hardkey的状态（0=非按下，1=按下）
SIM_HARDKEY_GetCallback()	设置当指定Hardkey状态改变时要执行的回调函数
SIM_HARDKEY_SetMode()	设置一个指定Hardkey的行为（默认=0，不切换）
SIM_HARDKEY_SetState()	设置一个指定Hardkey的状态

SIM_SetLCDPos()

描述

在目标设备位图中设置仿真LCD的位置。

函数原型

```
void SIM_SetLCDPos(int x, int y);
```

参 数	含 意
x	仿真LCD左上角（单位：像素）的 X 轴坐标
y	仿真LCD左上角（单位：像素）的 Y 轴坐标

附加信息

X和Y坐标相对于目标设备位图，因此坐标（0,0）表示位图左上角（原点）而非你的实际LCD。只有仿真屏幕的原点需要指定；你的显示器的分辨率应该已经反映在配置目录下的配置文件中。

SIM_SetTransparentColor()

描述

设置用于设备或Hardkey位图的透明区域的颜色

函数原型

```
I32 SIM_SetTransparentColor(I32 Color);
```

参 数	含 意
Color	颜色的RGB数值

附加信息

透明色的默认设置为亮红（0xFF0000）。如果你的位图包括有同样色调的红色时，你才需要这个典型的设置。

SIM_HARDKEY_GetNum()

描述

返回一个有效的Hardkey的号码

函数原型

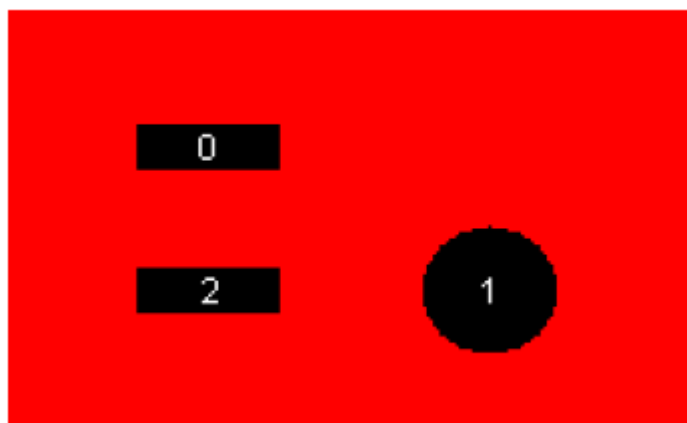
```
int SIM_HARDKEY_GetNum(void);
```

返回值

在位图中找到的有效的Hardkey的号码。

附加信息

Hardkey的序号遵循阅读顺序（从左到右，从上到下）。因此一个Hardkey最顶端的像素首先被发现，而不管它的水平位置如果。例如，在下面的位图中， Hardkey被标号，而在其它的函数中它们通过[KeyIndex](#)参数进行引用：



推荐调用该函数对一幅位图是否被完全截入进行校验。

SIM_HARDKEY_GetState()

描述

返回指定的Hardkey的状态

函数原型

```
int SIM_HARDKEY_GetState(unsigned int KeyIndex);
```

参 数	含 意
KeyIndex	Hardkey的标签（0=第一个键的标签）

返回值

指定Hardkey的数值：

- 0: 非按下
- 1: 按下

SIM_HARDKEY_SetCallback()

描述

设置当指定Hardkey状态改变时要执行的回调函数

函数原型

```
SIM_HARDKEY_CB* SIM_HARDKEY_SetCallback ( unsigned int KeyIndex,
                                           SIM_HARDKEY_CB* pfCallback);
```

参 数	含 意
KeyIndex	Hardkey的标签（0=第一个键的标签）
pfCallback	回调函数的指针

返回值

先前的回调函数的指针。

附加信息

回调函数原型必须如下所示：

函数原型

```
typedef void SIM_HARDKEY_CB(int KeyIndex, int State);
```

参 数	含 意
KeyIndex	Hardkey的标签（0=第一个键的标签）
State	指定Hardkey的状态（如下所示）

参数State允许的数值：

- 0: 非按下

1: 按下

SIM_HARDKEY_SetMode()

描述

设置指定Hardkey的行为

函数原型

```
int SIM_HARDKEY_SetMode(unsigned int KeyIndex, int Mode);
```

参 数	含 意
KeyIndex	Hardkey的标签（0=第一个键的标签）
Mode	行为模式（如下表所示）

参数[Mode](#)允许的数值

0: 正常行为（默认）

1: 切换行为

附加信息

正常（默认）Hardkey行为意思指当鼠标指针位于一个键上方，并且鼠标按键保持按下状态，则该键被认为是按下。当鼠标按键释放或指针移开Hardkey，该键被认为是非按下。

而切换行为，每一次单击鼠标对一个Hardkey进行一次“按下”或“非按下”状态的切换。这意思是如果你在一个Hardkey上方单击一下鼠标，它变成“按下”，这个状态一直保持到你再次单击鼠标。

SIM_HARDKEY_SetState()

描述

设置指定Hardkey的状态。

函数原型

```
int SIM_HARDKEY_SetState(unsigned int KeyIndex, int State);
```

参 数	含 意
-----	-----

KeyIndex	Hardkey的标签（0= 第一个键的标签）
State	指定Hardkey的状态（如下所示）

参数State允许的数值：

0: 非按下

1: 按下

附加信息

该函数只有在 SIM_HARDKEY_SetMode 被设为 1（切换模式）时才有效。

第4章 文本显示

使用 μ C/GUI 显示字体是很容易的。仅仅需要很少的的函数知识就能让我们在任何有效的字体当中进行文本书字，然后显示在任何一个位置。我们首先对显示字体进行简短的介绍，然后是分别对所用的函数进行更详细的说明。

4.1 基本函数

为了在LCD上显示文本，可以简单地调用函数GUI_DispString()，把你所希望显示的文本作为其参数，例如：

```
GUI_DispString("Hello world!");
```

上面的代码将会在当前文本坐标显示文本“Hello world!”。

然而，正如你所看到的，有很多函数用于显示不同字体的文本或都在不同的坐标显示文本。另外，它不仅能写字符串，而且能写十进制数，十六进制数和二进制数用于显示。即使图形显示通常是以字节为导向，文本能够定位在显示屏上的任何像素上，不仅仅是按字节定位。

控制字符

控制字符是一个小于32的字符代码。控制字符被定义为ASCII码的一部分。μC/GUI忽略所有除了下表所列出的以外的控制字符：

字符代码	ASCII代码	“C”	含义
10	LF	\n	换行，改变当前文本坐标到下一行，即： X=0； Y += 字体-距离（单位：像素）（如函数 GUI_GetFontDistY()所讨论的那样）
13	CR	\r	回车，改变当前文本坐标到当前行的开始处，即：x=0

控制字符LF的用法在字符串中非常方便。换行能将一个字符串拆开几部分，这样，只需要调用一个函数就能将这个字符串就能变成几行显示。

在一个选定坐标放置文本

这个功能可能通过调用函数GUI_GotoXY()来实现，如下面例子所示：

```
GUI_GotoXY(10, 10);           // 设置坐标（以像素为单位）
GUI_DispString("Hello world!"); // 显示文本
```

4.2 文本API

下表列出了与文本处理相关的函数，在各自的类型中按字母顺序进行排列。函数的详细

描述后面列出。

函 数	说 明
显示文本的函数	
GUI_DisPChar()	在当前坐标显示单个字符
GUI_DisPCharAt()	在指定坐标显示单个字符
GUI_DisPChars()	按指定重复次数显示一个字符
GUI_DisPString()	在当前坐标显示字符串
GUI_DisPStringAt()	在指定坐标显示字符串
GUI_DisPStringAtCEOL()	在指定坐标显示字符串，并清除到行末
GUI_DisPStringInRect()	在指定矩形区域内显示字符串
GUI_DisPStringLength()	在当前坐标显示指定字符数量的字符串
选择文本绘图模式	
GUI_SetTextMode()	设置文本绘图模式
选择文本对齐方式	
GUI_GetTextAlign()	返回当前文本对齐模式
GUI_SetLBorder()	设置换行后的左边界
GUI_SetTextAlign()	设置文本对齐模式
设置当前文本坐标	
GUI_GotoX()	设置当前X坐标
GUI_GotoXY()	设置当前X、Y坐标
GUI_GotoY()	设置当前Y坐标
找回当前文本坐标	
GUI_GetDispPosX()	返回当前X坐标
GUI_GetDispPosY()	返回当前Y坐标
清除视窗或其部分的函数	
GUI_Clear()	清除活动视窗（如果背景是活动视窗，则是清除整个屏幕）
GUI_DisPCEOL()	清除从当前坐标到行末的显示内容

4.3 显示文本的函数

GUI_DisPChar()

描述

在当前视窗使用当前字体在当前文本坐标处显示单个字符。

函数原型

```
void GUI_DispChar(U16 c);
```

参 数	含 意
<code>c</code>	显示的字符

附加信息

这是显示字符的基本函数。所有其它显示函数（`GUI_DispCharAt()`，`GUI_DispString()`等）都要调用这个函数输出单个字符。

字符是否有效取决于所选择的字体，如果在当前字体中该字符无效，则不会有任何显示。

范例

在屏幕上显示一个大写“A”：

```
GUI_DispChar('A');
```

相关主题

`GUI_DispChars()`，`GUI_DispCharAt()`

GUI_DispCharAt()

描述

在当前视窗使用当前字体在指定坐标处显示单个字符。

函数原型

```
void GUI_DispCharAt(U16 c, I16P x, I16P y);
```

参 数	含 意
<code>c</code>	显示的字符
<code>x</code>	写到客户窗口经X轴坐标（以像素为单位）
<code>y</code>	写到客户窗口经Y轴坐标（以像素为单位）

附加信息

显示字符的左上角在指定的（X, Y）坐标。

使用函数`GUI_DispChar()`写字符。

如果在当前字体中该字符无效，则不会有任何显示。

范例

在屏幕左上角显示一个大写“A”：

```
GUI_DisCharAt('A', 0, 0);
```

相关主题

`GUI_DisChar()`，`GUI_DisChars()`

GUI_DisChars()

描述

在当前视窗使用当前字体在当前文本坐标显示一个字符，并指定重复显示的次数。

函数原型

```
void GUI_DisChars(U16 c, int Cnt);
```

参 数	含 意
<code>c</code>	显示的字符
<code>Cnt</code>	重复的次数 ($0 \leq Cnt \leq 32767$)

附加信息

使用函数`GUI_DisChar()`写字符。

如果在当前字体中该字符无效，则不会有任何显示。

范例

在屏幕上显示一行“*****”：

```
GUI_DisChars('*', 30);
```

相关主题

`GUI_DisChar()`，`GUI_DisCharAt()`

GUI_DisString()

描述

在当前视窗的当前坐标，使用当前字体显示作为参数的字符串。

函数原型

```
void GUI_DisString(const char GUI_FAR *s);
```

参 数	含 意
<code>s</code>	显示的字符串

附加信息

字符串包括控制字符“\n”。该控制字符把当前文本坐标移到下一行的开始处。

范例

在屏幕上显示“Hello world”及在下一行显示“Next line”：

```
GUI_DisString("Hello world");    // 显示文本
GUI_DisString("\nNext line");    // 显示文本
```

相关主题

`GUI_DisStringAt()`，`GUI_DisStringAtCEOL()`，`GUI_DisStringLen()`

GUI_DisStringAt()

描述

在当前视窗，使用当前字体在指定坐标显示作为参数的字符串。

函数原型

```
void GUI_DisStringAt(const char GUI_FAR *s, int x, int y);
```

参 数	含 意
<code>s</code>	显示的字符串
<code>x</code>	写到客户视窗的X轴坐标（以像素为单位）

y	写到客户视窗的Y轴坐标（以像素为单位）
---	---------------------

范例

在屏幕上坐标（50,20）处显示“Position 50,20”

```
GUI_DispStringAt("Position 50,20", 50, 20);    // 显示文本
```

相关主题

GUI_DispString(), GUI_DispStringAtCEOL(), GUI_DispStringLen()

GUI_DispStringAtCEOL()

描述

该函数使用的参数与GUI_DispStringAt()完全一致。它也执行同样的操作：在指定坐标显示所给出的字符串。但是，完成这步操作后，它会调用GUI_DispCEOL函数清除本行剩下部分内容直至行末。如果字符串覆盖了其它的字符串，同时该字符串长度比原先的字符串短的时候，使用该函数就会很方便。

GUI_DispStringInRect()

描述

在当前视窗，使用当前字体在指定坐标显示作为参数的字符串。

函数原型

```
void GUI_DispStringInRect( const char GUI_FAR *s,
                           const GUI_RECT *pRect,
                           int Align);
```

参 数	含 意
s	显示的字符串
pRect	写像素的客户窗口的矩形区域
Align	垂直对齐：GUI_TA_TOP, GUI_TA_BOTTOM, GUI_TA_VCENTER; 水平对齐：GUI_TA_LEFT, GUI_TA_RIGHT, GUI_TA_HCENTER;

范例

在当前视窗的水平及垂直对中的坐标显示字“Text”：

```
GUI_RECT rClient;  
GUI_GetClientRect(&rClient);  
GUI_DispStringInRect("Text", &rClient, GUI_TA_HCENTER | GUI_TA_VCENTER);
```

附加信息

如果指定的矩形太小，文本会被裁剪。

相关主题

GUI_DispString(), GUI_DispStringAtCEOL(), GUI_DispStringLen()

GUI_DispStringLen()

描述

在当前视窗，使用当前字体在指定坐标显示作为参数的字符串，指定显示字符的数量。

函数原型

```
void GUI_DispStringLen(const char GUI_FAR *s, int Len);
```

参 数	含 意
s	显示的字符串，应该以一个“\0”作为8位字符排列的结束标记。允许用NULL作为参数。
Len	显示的字符数量

附加信息

如果字符串的字符少于指定的数量，则用空格填满。如果多于指定的数量，则显会显示指定数量的字符。

文本信息可能以不同语言显示（自然长度会不一样）时，该函数十分有用，但是只有某些字符能够显示。

相关主题

GUI_DispString(), GUI_DispStringAt(), GUI_DispStringAtCEOL()

4.4 选择文本的绘制模式

通常，在当前文本坐标，使用所选择的字体，在选择视窗中以正常文本模式定入文本。正常文本意思是指，文本覆盖已经显示的任何东西，在这种情况下，在字符屏蔽中被设定的位在屏幕上被设定。在这种模式下，活动的位使用前景色写，而非活动的位用背景色写。然而，在一些场合，需要改变这些默认的行为。为了这个目的， μ C/GUI提供四种标识（一种默认加上三种修改值），它们可以组合使用：

正常文本

文本可能正常显示，此时模式标识应指定为GUI_TEXTMODE_NORMAL 或 0。

反转文本

文本反转显示，模式标识应指定为GUI_TEXTMODE_REVERSE。通常在黑色上显示白色变成在白色上显示的黑色。

透明文本

透明文本意思是文本写在已经在屏幕上可见的任何东西上面。不同的是，屏幕上原有的内容仍然能够看得见，与正常文本相比，背景色被擦除了。

模式标识指定为GUI_TEXTMODE_TRANS表示显示透明文本。

异或文本

通常情况下，用白色绘制的（实际字符）显示是反相的。如果背景颜色是黑色，效果与正常模式（正常文本）是一样的。如果背景是白色，输出与反转文本一样。如果你使用彩色，一个反相的像素由下式计算：

$$\text{新像素颜色} = \text{颜色的值} - \text{实际像素颜色} - 1$$

透明反转文本

作为透明文本，它不覆盖背景，作为反转文本，文本显示是反转的。

文本通过指定标识GUI_TEXTMODE_TRANS | GUI_TEXTMODE_REVERSE来实现这种效果。

范例

显示正常，反转，透明，异或及透明反转文本：

```
GUI_SetFont(&GUI_Font8x16);
GUI_SetFont(&GUI_Font8x16);
GUI_SetBkColor(GUI_BLUE);
GUI_Clear();
GUI_SetPenSize(10);
GUI_SetColor(GUI_RED);
GUI_DrawLine(80, 10, 240, 90);
GUI_DrawLine(80, 90, 240, 10);
GUI_SetBkColor(GUI_BLACK);
GUI_SetColor(GUI_WHITE);
GUI_SetTextMode(GUI_TM_NORMAL);
GUI_DispStringHCenterAt("GUI_TM_NORMAL", 160, 10);
GUI_SetTextMode(GUI_TM_REV);
GUI_DispStringHCenterAt("GUI_TM_REV", 160, 26);
GUI_SetTextMode(GUI_TM_TRANS);
GUI_DispStringHCenterAt("GUI_TM_TRANS", 160, 42);
GUI_SetTextMode(GUI_TM_XOR);
GUI_DispStringHCenterAt("GUI_TM_XOR", 160, 58);
GUI_SetTextMode(GUI_TM_TRANS | GUI_TM_REV);
GUI_DispStringHCenterAt("GUI_TM_TRANS | GUI_TM_REV", 160, 74);
```

下图为上面范例程序执行结果的的屏幕截图



GUI_SetTextMode()

描述

按照指定的参数设置文本模式。

函数原型

```
int GUI_SetTextMode(int TextMode);
```

参 数	含 意
TextMode	设置的文本模式，可以是文本模式标识的任意组合

参数TextMode允许的数值（可以用“OR（或）”进行组合）

GUI_TEXTMODE_NORMAL	设置正常文本，这是默认的设置，该数值等同于0
GUI_TEXTMODE_REVERSE	设置反转文本
GUI_TEXTMODE_TRANSPARENT	设置透明文本
GUI_TEXTMODE_XOR	反相显示的文本

返回值

选择的文本模式

范例

屏幕上坐标（0,0）处显示“The value is”，设置文本模式为反转模式，再将其设回正常模式：

```
int i = 20;
GUI_DispStringAt("The value is", 0, 0);
GUI_SetTextMode(GUI_TEXTMODE_REVERSE);
GUI_DispDec(20, 3);
GUI_SetTextMode(GUI_TEXTMODE_NORMAL);
```

4.5 文本对齐的选择

GUI_GetTextAlign()

描述

返回当前文本对齐模式。

函数原型

```
int GUI_GetTextAlign(void);
```

GUI_SetLBorder()

描述

设置在当前视窗换行后的左边界

函数原型

```
void GUI_SetLBorder(int x)
```

参 数	含 意
x	新的左边界（以像素为单位，0表示视窗左边界）

GUI_SetTextAlign()

描述

设置文本对齐模式，用于当前视窗的字符串输出。

函数原型

```
int GUI_SetTextAlign(int TextAlign);
```

参 数	含 意
TextAlign	设定的文本对齐模式。可以是水平和垂直对齐标志的组合。

参数[TextAlign](#)允许的数值（水平和垂直标识以“OR”组合）

水平对齐	
GUI_TA_LEFT	X轴方向左对齐（默认）
GUI_TA_HCENTER	X轴方向对中
GUI_TA_RIGHT	X轴方向右对齐（默认）
垂直对齐	
GUI_TA_TOP	在字符Y轴方向顶部对齐（默认）
GUI_TA_VCENTER	在Y轴方向对中
CUI_TA_BOTTOM	在字体Y轴底部线像素对齐

返回值

所选择的文本对齐模式

附加信息

GUI_SetTextAlign() 不影响以源于GUI_Dispatch的字符输出函数。

范例

在坐标（100, 100）处显示数值1234，采用对中模式：

```
GUI_SetTextAlign(GUI_TA_HCENTER | GUI_TA_VCENTER);
GUI_Dispatch(1234, 100, 100, 4);
```

4.6 设置当前文本坐标

每个任务都有一个当前文本坐标，该坐标以视窗的原点（通常是(0, 0)）为参考，如果调用了文本输出函数，下一个字符会写在这个坐标上。初始化时，该坐标是（0, 0），即当前视窗的左上角。在三个函数可能用来设置当前文本坐标。

GUI_GotoXY(), GUI_GotoX(), GUI_GotoY()

描述

设置当前写文本的坐标。

函数原型

```
char GUI_GotoXY(int x, int y);
char GUI_GotoX(int x);
```

```
char GUI_GotoY(int y);
```

参 数	含 意
x	新的X轴坐标（以像素为单位，0为视窗左边界）
y	新的Y轴坐标（以像素为单位，0为视窗顶部边界）

返回值

通常为0。如果返回数值非0，则当前文本坐标超出视窗范围（到了右边或下边），这样紧接着的写操作可能被忽略。

附加信息

GUI_GotoXY() 对当前视窗文本坐标的X坐标和Y坐标两部分同时设置。

GUI_GotoX() 只对当前视窗文本坐标的X坐标部分进行设置，Y坐标保持不变。

GUI_GotoY() 只对当前视窗文本坐标的Y坐标部分进行设置，X坐标保持不变。

范例

在屏幕上坐标（20, 20）处显示 “(20, 20)”：

```
GUI_GotoXY(20, 20)
GUI_DispString("The value is");
```

4.7 找回当前文本坐标

GUI_GetDispPosX()

描述

返回当前X坐标

函数原型

```
int GUI_GetDispPosX(void);
```

GUI_GetDispPosY()

描述

返回当前Y坐标

函数原型

```
int GUI_GetDispPosY(void);
```

4.8 清除一个视窗或其部分的函数

GUI_Clear()

描述

清除当前视窗。

函数原型

```
void GUI_Clear(void);
```

附加信息

如果没有定义视窗，当前视窗为整个显示区。这样的话，整个显示区都会被清除。

范例

在屏幕上显示“Hello world”，等待1秒种，然后清除显示内容：

```
GUI_DispStringAt("Hello world", 0, 0);    // 显示文本
GUI_Delay(1000);                          // 等待1秒钟（非μC/GUI部分）
GUI_Clear();                              // 清屏
```

GUI_DispCEOL()

描述

清除当前视窗（或屏幕）从当前文本坐标到行末显示区域的内容，高度为当前字体高度。

函数原型

```
void GUI_DispCEOL(void);
```

范例

在屏幕上显示“Hello world”，等待1秒钟，然后在同步坐标显示“Hi”，代替原先显示的字符：

```
GUI_DispStringAt("Hello world", 0, 0);           // 显示文本
Delay(1000);
GUI_DispStringAt("Hi", 0, 0);
GUI_DispCEOL();
```

第5章 显示数值

前面章节说明了如何在屏幕上显示字符串。当然你也可以用字符串和标准C库函数显示数值。

然而，有时候这会是件困难的事。通常最容易（最有效）的是调用一个函数显示你所需要结构的数值。 μ C/GUI提供了不同的十进制，十六进制和二进制输出函数。在这一章对这些函数进行单独的说明。

所有的函数不需要使用浮点库，并对速度和大小进行了优化。当然“Sprintf”可以用于任何系统。使用本章所述的函数有时能在 ROM 的存储空间和执行时间上进行精简。

5.1 数值API

下表列出了与数值处理相关的函数，在各自的类型中按字母顺序进行排列。函数的详细描述后面列出。

函 数	说 明
显示十进制数值	
GUI_DisDec()	在当前坐标显示指定数量字符的十进制数值
GUI_DisDecAt()	在指定坐标显示指定数量字符的十进制数值
GUI_DisDecMin()	在当前坐标显示最少数量字符的十进制数值
GUI_DisDecShift()	在当前坐标显示指定数量字符的长型十进制数值
GUI_DisDecSpace()	在当前坐标显示指定数量字符的十进制数值，用空格代替首位的0
GUI_DispsDec()	在当前坐标显示指定数量字符的十进制数值及显示符号
GUI_DispsDecShift()	在当前坐标显示指定数量字符的长型十进制数值及显示符号
显示浮点数值	
GUI_DisFloat()	在当前坐标显示指定数量字符的浮点数值
GUI_DisFloatFix()	显示浮点数值，指定小数点右边数字数量
GUI_DisFloatMin()	在当前坐标显示最少数量字符的浮点数值
GUI_DispsFloatFix()	显示浮点数值，指定小数点右边数字数量及显示符号
GUI_DispsFloatMin()	在当前坐标显示最少数量字符的浮点数值及显示符号
显示二进制数值	
GUI_DisBin()	在当前坐标显示二进制数值
GUI_DisBinAt()	在指定坐标显示二进制数值
显示十六进制数值	
GUI_DisHex()	在当前坐标显示十六进制数值
GUI_DisHexAt()	在指定坐标显示十六进制数值

5.2 显示十进制数值

GUI_DisDec()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个十进制数值，指定显示字符的数量。

函数原型

```
void GUI_DisDec(I32 v, U8 Len);
```

参 数	含 意
<code>v</code>	用于显示的数值。 最小值为：-2147483648 (-2^{31}) 最大值为：2147483647 ($2^{31}-1$)
<code>Len</code>	显示的数字的数量（最大为9）

附加信息

不支持首位为0的格式（如0）。

如果数值为负，则会显示一个减号。

范例

```
// 以分秒的格式显示时间
GUI_DisString("Min:");
GUI_DisDec(Min, 2);
GUI_DisString("Sec:");
GUI_DisDec(Sec, 2);
```

相关主题

`GUI_DisSDec()`，`GUI_DisDecAt()`，`GUI_DisDecMin()`，`GUI_DisDecSpace()`

GUI_DisDecAt()

描述

在当前视窗的当前文本坐标，使用当前字体显示十进制数值，指定显示字符的数量。

函数原型

```
void GUI_DisDecAt(I32 v, I16P x, I16P y, U8 Len);
```

参 数	含 意
<code>v</code>	用于显示的数值。 最小值为：-2147483648 (-2^{31}) 最大值为：2147483647 ($2^{31}-1$)
<code>x</code>	写入客户视窗的X坐标（以像素为单位）
<code>y</code>	写入客户视窗的Y坐标（以像素为单位）
<code>Len</code>	显示的数字的数量（最大为9）

附加信息

不支持首位为0的格式。如果数值为负，则会显示一个减号。

范例

```
// 在左上角更新秒
GUI_DisDecAT(Sec, 200, 0, 2);
```

相关主题

GUI_DisDec(), GUI_DisSDec(), GUI_DisDecMin(), GUI_DisDecSpace()

GUI_DisDecMin()

描述

在当前视窗的当前文本坐标，使用当前字体显示十进制数值。不需要指定长度；自动使用最小的长度值。

函数原型

```
void GUI_DisDecMin(I32 v);
```

参 数	含 意
v	用于显示的数值。 最小值为：-2147483648 (-2^{31}) 最大值为：2147483647 ($2^{31}-1$) 能显示的数字的最大数量为9。

附加信息

如果数值必须要对齐，但是数字的数量不一样，使用该函数不是一个好的选择，应该使用一个能够指定数字数量的函数。

范例

```
// 显示结果
GUI_DisString("The result is :");
GUI_DisDecMin(Result);
```

相关主题

GUI_DispNetDec(), GUI_DispNetDecAt(), GUI_DispNetSDec(), GUI_DispNetDecSpace()

GUI_DispNetDecShift()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个长型十进制数值（用小数点作分隔符），指定显示字符的数量及使用小数点。

void GUI_DispNetDecShift(I32 v, U8 Len, U8 Shift);

参 数	含 意
v	用于显示的数值。 最小值为：-2147483648 (-2^{31}) 最大值为：2147483647 ($2^{31}-1$)
Len	显示的数字的数量（最大为9）
Shift	小数点右边数字的数量

附加信息

注意显示的最大字符数量为9（包括符号及小数点）

GUI_DispNetDecSpace()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个十进制数值，禁止首位的0（用空格代换）。

函数原型

void DispNetDecSpace(I32 v, U8 MaxDigits);

参 数	含 意
v	用于显示的数值。 最小值为：-2147483648 (-2^{31}) 最大值为：2147483647 ($2^{31}-1$)

MaxDigits	显示的数字数量，包括首位空格。 最大显示的数字的数量为9（包括首位空格）
-----------	---

附加信息

如果数值必须要对齐，但是数字的数量不一样，使用该函数是一个好的选择。

范例

// 显示结果

```
GUI_DisString("The result is :");
GUI_DisDecSpace(Result, 200);
```

相关主题

GUI_DisDec(), GUI_DisDecAt(), GUI_DisSDec(), GUI_DisDecMin()

GUI_DisSDec()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个十进制数值（包括符号），并指定显示字符的数量。

函数原型

```
void GUI_DisSDec(I32 v, U8 Len);
```

参 数	含 意
V	用于显示的数值。 最小值为：-2147483648 (-2^{31}) 最大值为：2147483647 ($2^{31}-1$)
Len	显示的数字的数量（最大为9）

附加信息

不禁止首位为0的格式。

该函数与GUI_DisDec类似，但是在显示数值的前面总带有符号，即使这个数值是正的。

相关主题

GUI_DispDec(), GUI_DispDecAt(), GUI_DispDecMin(), GUI_DispDecSpace()

GUI_DispSDecShift()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个长型十进制数值（包括符号，用小数点作分隔符），指定数字的数量及使用小数点。

函数原型

```
void GUI_DispSDecShift(I32 v, U8 Len, U8 Shift);
```

参 数	含 意
v	用于显示的数值。 最小值为：-2147483648 (-2^{31}) 最大值为：2147483647 ($2^{31}-1$)
Len	显示的数字的数量（最大为9）
Shift	小数点右边数字的数量

附加信息

在数值前面总带有一个符号。注意显示的最大字符数量为9（包括符号及小数点）。

范例

```
void DemoDec(void)
{
    long l = 12345;
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x8);
    GUI_DispStringAt("GUI_DispDecShift:\n", 0, 0);
    GUI_DispSDecShift(l, 7, 3);
    GUI_SetFont(&GUI_Font6x8);
    GUI_DispStringAt("Press any key", 0, GUI_VYSIZE-8);
    WaitKey();
}
```

下图为上面范例程序运行结果的屏幕截图



5.3 显示浮点数

GUI_DispNetFloat()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个浮点数，指定显示字符数量。

函数原型

```
void GUI_DispNetFloat(float v, char Len);
```

参 数	含 意
<code>v</code>	用于显示的数值。 最小值为：1.2E-38 最大值为：3.4E38
<code>Len</code>	显示的数字的数量（最大为9）

附加信息

不支持首位为0的格式。小数点当作一个字符处理。如果数值为负数，会显示一个减号。

范例

```
/* 浮点数显示的所有特点的演示 */
void DemoFloat(void)
{
    float f = 123.45678;
    GUI_Clear()
    GUI_SetFont(&GUI_Font8x8);
    GUI_DispNetStringAt("GUI_DispNetFloat:\n", 0, 0);
    GUI_DispNetFloat (f, 9);
}
```

```

GUI_GotoX(100);
GUI_DisFloat (-f, 9);
GUI_DisStringAt("GUI_DisFloatFix:\n", 0, 20);
GUI_DisFloatFix (f, 9, 2);
GUI_GotoX(100);
GUI_DisFloatFix (-f, 9, 2);
GUI_DisStringAt("GUI_DisSFloatFix:\n", 0, 40);
GUI_DisSFloatFix (f, 9, 2);
GUI_GotoX(100);
GUI_DisSFloatFix (-f, 9, 2);
GUI_DisStringAt("GUI_DisFloatMin:\n", 0, 60);
GUI_DisFloatMin (f, 3);
GUI_GotoX(100);
GUI_DisFloatMin (-f, 3);
GUI_DisStringAt("GUI_DisSFloatMin:\n", 0, 80);
GUI_DisSFloatMin (f, 3);
GUI_GotoX(100);
GUI_DisSFloatMin (-f, 3);
GUI_SetFont(&GUI_Font6x8);
GUI_DisStringAt("Press any key", 0, GUI_VYSIZE-8);
WaitKey();
}

```

下图为上面范例程序运行结果的屏幕截图



```

GUI_DisFloat:
123.45678      -123.4568
GUI_DisFloatFix:
000123.46      -00123.46
GUI_DisSFloatFix:
+00123.46      -00123.46
GUI_DisFloatMin:
123.457        -123.457
GUI_DisSFloatMin:
+123.457        -123.457

Press any key

```


GUI_DispFloatFix()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个浮点数，指定总的显示字符的数量及小数点右边字符的数量。

函数原型

```
void GUI_DispFloatFix (float v, char Len, char Decs);
```

参 数	含 意
v	用于显示的数值。 最小值为：1.2E-38 最大值为：3.4E38
Len	显示的所有数字的数量（最大为9）
Decs	小数点右边数字的数量

附加信息

不禁止首位为0的格式。如果数值为负，则显示一个减号。

GUI_DispFloatMin()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个浮点数，小数点右边十进制数的数量为一个最小值。

函数原型

```
void GUI_DispFloatMin(float f, char Fract);
```

参 数	含 意
v	用于显示的数值。 最小值为：1.2E-38 最大值为：3.4E38
Fract	显示字符的最小数量

附加信息

不支持首位为0的格式。如果数值为负，则显示一个减号。

长度不需要指定，自动使用最小的长度。如果数值必须要对齐，但是数字的数量不一样，使用该函数不是一个好的选择，应该使用一个能够指定数字数量的函数。

GUI_DispSFloatFix()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个浮点数（包括符号），指定总的显示字符的数量及小数点右边字符的数量。

函数原型

```
void GUI_DispSFloatFix(float v, char Len, char Decs);
```

参 数	含 意
<code>v</code>	用于显示的数值。 最小值为：1.2E-38 最大值为：3.4E38
<code>Len</code>	显示的所有数字的数量（最大为9）
<code>Decs</code>	小数点右边数字的数量

附加信息

不禁止首位为0的格式。数值的前面总会显示一个符号。

GUI_DispSFloatMin()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个浮点数（包括符号），小数点右边数字使用最小数量。

函数原型

```
void GUI_DispSFloatMin(float f, char Fract);
```

参 数	含 意
<code>v</code>	用于显示的数值。 最小值为：1.2E-38 最大值为：3.4E38
<code>Fract</code>	显示数字的最小数量

附加信息

不支持首位为0的格式

数值的前面总会显示一个符号。

长度不需要指定，自动使用最小的长度。如果数值必须要对齐，但是数字的数量不一样，使用该函数不是一个好的选择，应该使用一个能够指定数字数量的函数。

5.4 显示二进制数值

GUI_DispBin()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个二进制数。

函数原型

```
void GUI_DispBin(U32 v, U8 Len);
```

参 数	含 意
<code>v</code>	用于显示的数值，32位。
<code>Len</code>	显示的数字的数量（包括首位的0）

附加信息

与十进制及十六进制一样，最低有效位在最右边。

范例

```
// 显示二进制数“7”，结果为：000111
U32 Input = 0x7;
GUI_DispBin(Input, 6);
```

相关主题

`GUI_DispBinAt()`

GUI_DispBinAt()

描述

在当前视窗的指定的文本坐标处，使用当前字体显示一个二进制数。

函数原型

```
void DispBinAt(U32 v, I16P y, I16P x, U8 Len);
```

参 数	含 意
<code>v</code>	用于显示的数值，32位。
<code>x</code>	写入客户视窗的X轴坐标（以像素为单位）
<code>y</code>	写入客户视窗的Y轴坐标（以像素为单位）
<code>Len</code>	显示的数字的数量（包括首位的0）

附加信息

十进制及十六进制一样，最低有效位在最右边。

```
// 显示二进制输入状态
GUI_DispBinAt(Input, 0, 0, 8);
```

相关主题

GUI_DispBin(), GUI_DispHex()

5.5 显示十六进制数值

GUI_DispHex()

描述

在当前视窗的当前文本坐标，使用当前字体显示一个十六进制数值。

函数原型

```
void GUI_DispHex(U32 v, U8 Len);
```

参 数	含 意
<code>v</code>	用于显示的数值，16位。

Len	显示的数字的数量
-----	----------

附加信息

与十进制及二进制一样，最低有效位在最右边。

范例

```
/* 显示一个AD转换器的数值 */
GUI_DispHex(Input, 4);
```

相关主题

```
GUI_DispDec(), GUI_DispBin(), GUI_DispHexAt()
```

GUI_DispHexAt()

描述

在当前视窗的指定的文本坐标处，使用当前字体显示一个十六进制数。

函数原型

```
void GUI_DispHexAt(U32 v, I16P x, I16P y, U8 Len);
```

参 数	含 意
v	用于显示的数值，32位。
x	写入客户视窗的X轴坐标（以像素为单位）
y	写入客户视窗的Y轴坐标（以像素为单位）
Len	显示的数字的数量（包括首位的0）

附加信息

与十进制及二进制一样，最低有效位在最右边。

范例

```
// 在指定坐标显示一个AD转换器数值
GUI_DispHexAt(Input, 0, 0, 4);
```

相关主题

GUI_DispDec(), GUI_DispBin(), GUI_DispHex()

5.6 μ C/GUI的版本

GUI_GetVersionString()

描述

返回一个字符串包含当前 μ C/GUI的版本

函数原型

```
const char * GUI_GetVersionString(void);
```

范例

```
// 在当前光标位置显示当前版本  
GUI_DispString(GUI_GetVersionString());
```

第6章 2-D图形库

μ C/GUI包括有一个完整的2-D图形库，在大多数场下应用是足够了。 μ C/GUI提供的函数既可以与裁剪区一道使用也可以脱离裁剪区使用（参考第12章“视窗管理器”），这些函数基于快速及有效率的算法建立。

目前，只有绘制圆弧函数要求浮点运算支持。

6.1 API参考：图形

下表列出了与图形处理相关的函数，在各自的类型中按字母顺序进行排列。函数的详细描述后面列出。

函 数	说 明
绘图模式	
GUI_SetDrawMode()	设置绘图模式。
基本绘图函数	
GUI_ClearRect()	使用背景颜色填充一个矩形区域。
GUI_DrawPixel()	绘一个单像素点。
GUI_DrawPoint()	绘一个点。
GUI_FillRect()	绘一个填充的矩形。
GUI_InvertRect()	反转一个矩形区域。
绘制位图	
GUI_DrawBitmap()	绘制一幅位图。
GUI_DrawBitmapExp()	绘制一幅位图。
GUI_DrawBitmapMag()	绘制一幅放大的位图。
GUI_DrawStreamedBitmap()	从一个位图数据流的数据绘制一幅位图。
绘线	
GUI_DrawHLine()	绘一根水平线。
GUI_DrawLine()	绘一根线。
GUI_DrawLineRel()	从当前坐标到端点绘一根线，该端点由X轴距离及Y轴距离指定。
GUI_DrawLineTo()	从当前坐标到端点(X, Y)绘一根线。
GUI_DrawPolyLine()	绘折线。
GUI_DrawVLine()	绘一根垂直线。
绘多边形	
GUI_DrawPolygon()	绘一个多边形。
GUI_EnlargePolygon()	对一个多边形进行扩边。
GUI_FillPolygon()	绘一个填充的多边形。
GUI_MagnifyPolygon()	放大一个多边形。
GUI_RotatePolygon()	按指定角度旋转一个多边形。
绘圆	
GUI_DrawCircle()	绘一个圆。
GUI_FillCircle()	绘一个填充的圆。
绘椭圆	
GUI_DrawEllipse()	绘一个椭圆。
GUI_FillEllipse()	绘一个填充的椭圆。

绘圆弧	
GUI_DrawArc()	绘一个圆弧

6.2 绘图模式

μC/GUI提供两种绘图模式，NORMAL模式及XOR模式。默认为NORMAL模式，即显示屏的内容被绘图所完全覆盖。在XOR模式，当绘图覆盖在上面时，显示屏的内容反相显示。

与GUI_DRAWMODE_XOR有关的限制

- XOR模式通常用于在活动视窗或屏幕中使用两种颜色进行显示的场合。
- 一些μC/GUI的绘图函数并不能正确地工作在这种模式。通常情况下，这模式只是工作于一个像素大小的笔尖尺寸。这意味着在使用类似GUI_DrawLine，GUI_DrawCircle，GUI_DrawRect等等这样的函数之前，你必须确定在XOR模式下，笔尖尺寸已经设为1。
- 当使用颜色的深度大于1位/像素（bpp）进行位图绘制，该模式无效。
- 当使用诸如GUI_DrawPolyLine这样的函数或多次调用GUI_DrawLineTo函数，转角点会反相两次。结果是这些像素保持背景颜色。

GUI_SetDrawMode

描述

选择指定的绘图模式

函数原型

```
GUI_DRAWMODE GUI_SetDrawMode(GUI_DRAWMODE mode);
```

参 数	含 意
mode	设置的绘图模式。可以是任意设置绘图模式的函数的返回值或是下表中的任一个。

参数mode允许的数值

GUI_DRAWMODE_NORMAL	默认：绘点，线，区域，位图
GUI_DRAWMODE_XOR	当在屏幕上另一个物体上用颜色覆盖时对点，线，区域进行反相显示

返回值

所选择的绘图模式

附加信息

作为设置绘图模式的附加功能，该函数也可以用于恢复原先被修改的绘图模式。

如果使用颜色，一个反相的像素由下式算出：

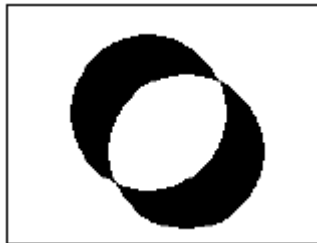
新像素颜色 = 颜色的数值 - 实际像素颜色 - 1

范例

// 显示两个圆，其中第二个以XOR模式与第一个结合

```
GUI_Clear();
GUI_SetDrawMode(GUI_DRAWMODE_NORMAL);
GUI_FillCircle(120, 64, 40);
GUI_SetDrawMode(GUI_DRAWMODE_XOR);
GUI_FillCircle(140, 84, 40);
```

上面范例程序运行结果的屏幕截图



6.3 基本绘图函数

基本绘图函数允许在显示屏上的任何位置进行单独的点，水平和垂直线段和形状的绘制。使用任何有效的绘图模式。因为这些函数在大多数应用中被频繁调用，因此它们已经被尽量地优化以获得尽可能快的速度。例如，水平和垂直线段绘制函数不需要使用单个点绘制函数。

GUI_ClearRect

描述

在当前视窗的指定位置通过向一个矩形区域填充背景色来清除它。

函数原型

```
void GUI_ClearRect(int x0, int y0, int x1, int y1);
```

参 数	含 意
x0	左上角X坐标
y0	左上角Y坐标
x1	右下角X坐标
y0	右下角Y坐标

相关主题

[GUI_InvertRect](#), [GUI_FillRect](#)

GUI_DrawPixel

描述

在当前视窗的指定坐标绘一个像素点。

函数原型

```
void GUI_DrawPixel(int x, int y);
```

参 数	含 意
x	像素点的X坐标
y	像素点的Y坐标

相关主题

[GUI_DrawPoint](#)

GUI_DrawPoint

描述

在当前视窗使用当前尺寸笔尖绘一个点。

函数原型

```
void GUI_DrawPoint(int x, int y);
```

参 数	含 意
x	点的X坐标
y	点的Y坐标

相关主题

[GUI_DrawPixel](#)

GUI_FillRect

描述

在当前视窗指定的位置绘一个矩形填充区域。

函数原型

```
void GUI_FillRect(int x0, int y0, int x1, int y1);
```

参 数	含 意
x0	左上角X坐标
y0	左上角Y坐标
x1	右下角X坐标
y0	右下角Y坐标

附加信息

使用当前的绘图模式，通常表示在矩形内的所有像素都被设置。

相关主题

[GUI_InvertRect](#), [GUI_ClearRect](#)

GUI_InvertRect

描述

在当前视窗的指定位置绘一反相的矩形区域。

函数原型

```
void GUI_InvertRect(int x0, int y0, int x1, int y1);
```

参 数	含 意
x0	左上角X坐标
y0	左上角Y坐标
x1	右下角X坐标
y0	右下角Y坐标

相关主题

GUI_FillRect, GUI_ClearRect

6.4 绘制位图

GUI_DrawBitmap

描述

在当前视窗的指定位置绘一幅位图。

函数原型

```
void GUI_DrawBitmap(const GUI_BITMAP*pBM, int x, int y);
```

参 数	含 意
pBM	需显示位图的指针
x	位图在屏幕上位置的左上角X坐标
y	位图在屏幕上位置的左上角Y坐标

附加信息

位图数据必须定义为像素×像素。每个像素等同于一位。最高有效位（MSB）定义第一个像素；图片数据以位流进行说明，以第一个字节的MSB作为起始。新的一行总是在一个偶数地址开始，而位图的第N行在地址偏移量n* BytesPerLine处开始。位图可以在客户区中任意一点显示，位图转换器用于产生位图。

范例

```
extern const GUI_BITMAP bmMicriumLogo;      /* 声明外部位图 */  
void main()
```

```

{
    GUI_Init();
    GUI_DrawBitmap(&bmMicriumLogo, 45, 20);
}

```

上面范例程序运行结果的屏幕截图



GUI_DrawBitmapExp

描述

与GUI_DrawBitmap函数具有相同功能，但是带有扩展参数设置。

函数原型

```

void GUI_DrawBitmapExp( int x0, int y0,
                        int XSize, int YSize,
                        int XMul, int YMul,
                        int BitsPerPixel,
                        int BytesPerLine,
                        const U8* pData,
                        const GUI_LOGPALETTE* pPal);

```

参 数	含 意
<code>x</code>	位图在屏幕上位置的左上角X坐标
<code>y</code>	位图在屏幕上位置的左上角Y坐标
<code>Xsize</code>	水平方向像素的数量，有效范围：1~255
<code>Ysize</code>	垂直方向像素的数量，有效范围：1~255
<code>XMul</code>	X轴方向比例因数
<code>YMul</code>	Y轴方向比例因数
<code>BitsPerPixel</code>	每像素的位数
<code>BytesPerLine</code>	图形每行的字节数
<code>pData</code>	实际图形的指针，该数据定义位图的外表特征
<code>pPal</code>	指向GUI_LOGPALETTE结构的指针

GUI_DrawBitmapMag

描述

该函数使在屏幕上放缩一幅位图成为可能。

函数原型

```
void GUI_DrawBitmapMag( const GUI_BITMAP* pBM,
                        int x0, int y0,
                        int XMul, int YMul);
```

参 数	含 意
pBM	所显示位图的指针
x0	位图在屏幕上位置的左上角X坐标
y0	位图在屏幕上位置的左上角Y坐标
XMul	X轴方向比例因数
YMul	Y轴方向比例因数

GUI_DrawStreamedBitmap

描述

该函数从一个位图数据流的数据绘制一幅位图。

函数原型

```
void GUI_DrawStreamedBitmap (  const GUI_BITMAP_STREAM *pBMH,
                               int x,
                               int y);
```

参 数	含 意
pBMH	指向数据流的指针
x	位图在屏幕上位置的左上角X坐标
y	位图在屏幕上位置的左上角Y坐标

附加信息

你可以使用在后面的章节描述的位图转换器建立位图数据流。这些数据流的格式与BMP文件的格式不一样。

6.5 绘线

绘图函数使用频率最高的是那些从一个点到另一个点的绘线函数。

GUI_DrawHLine

描述

在当前视窗从一个指定的起点到一个指定的终点，以一个像素厚度画一条水平线。

函数原型

```
void GUI_DrawHLine(int y, int x0, int x1);
```

参 数	含 意
y	Y轴坐标
x0	起点的X轴坐标
x1	终点的X轴坐标

附加信息

对于大多数LCD控制器，该函数执行速度非常快，因为多个像素能马上布置好，无需进行计算。很明显在绘水平线方面，该函数执行比GUI_DrawLine函数还要快。

GUI_DrawLine

描述

在当前视窗的指定始点到指定终点绘一条直线。

函数原型

```
void GUI_DrawLine(int x0, int y0, int x1, int y1);
```

参 数	含 意
x0	X轴开始坐标
y0	Y轴开始坐标
x1	X轴结束坐标
y1	Y轴结束坐标

附加信息

如果线的一部分是不可见的，因为它不在当前视窗内，或者如果当前视窗的一部分是不可见的，由于裁剪的原因，这些部分将不会绘出。

GUI_DrawLineRel

描述

在当前视窗从当前坐标(X, Y)到一个端点绘一条直线，指定X轴距离和Y轴距离。

函数原型

```
void GUI_DrawLineRel(int dx, int dy);
```

参 数	含 意
dx	到所绘直线末端X轴方向的距离
dy	到所绘直线末端Y轴方向的距离

GUI_DrawLineTo

描述

在当前视窗从当前坐标(X, Y)到一个端点绘一条直线，指定端点的X轴，Y轴坐标。

函数原型

```
void GUI_DrawLineTo(int x, int y);
```

参 数	含 意
x	终点的X轴坐标
y	终点的Y轴坐标

GUI_DrawPolyLine

描述

在当前视窗中用直线连接一系列预先确定的点。

函数原型

```
void GUI_DrawPolyLine(const GUI_POINT* pPoint, int NumPoints, int x, int y);
```

参 数	含 意
<code>pPoint</code>	指向所显示的折线的指针
<code>NumPoints</code>	点系列中指定点的数量
<code>x</code>	原点的X轴坐标
<code>y</code>	原点的Y轴坐标

附加信息

折线的起点和终点不要求重合。

GUI_DrawVLine

描述

在当前视窗从一个指定的起点到一个指定的终点，以一个像素厚度画一条垂直线。

函数原型

```
void GUI_DrawVLine(int x, int y0, int y1);
```

参 数	含 意
<code>x</code>	X轴坐标
<code>y0</code>	起点的Y轴坐标
<code>y1</code>	终点的Y轴坐标

附加信息

如果`y1 < y0`，则不会有任何显示。

对于大多数LCD控制器，该函数执行速度非常快，因为多个像素能马上布置好，无需进行计算。很明显在绘垂线方面，该函数执行比GUI_DrawLine函数还要快。

6.6 绘多边形

在绘矢量符号时绘多边形函数很有用。

GUI_DrawPolygon

描述

在当前视窗中绘一个由一系列点定义的多边形的轮廓。

函数原型

```
void GUI_DrawPolygon(const GUI_POINT* pPoint, int NumPoints, int x, int y);
```

参 数	含 意
pPoint	显示的多边形的指针
Numpoints	在点的序列中指定点的数量
x	原点的X轴坐标
y	原点的Y轴坐标

附加信息

所绘的折线通过连接起点和终点而自动闭合。

GUI_EnlargePolygon

描述

通过指定一个以像素为单位的长度，对多边形的所有边进行放大（扩边，与对多边形进行放大的概念不一样）。

函数原型

```
void GUI_EnlargePolygon ( GUI_POINT* pDest,
                          const GUI_POINT* pSrc,
                          int NumPoints,
                          int Len);
```

参 数	含 意
pPoint	目标多边形的指针
pSrc	源多边形的指针
Numpoints	在点的序列中指定点的数量
Len	对多边形进行放大的以像素为单位的长度

附加信息

确认目标点的阵列等于或大于源阵列。

范例

```
#define countof(Array) (sizeof(Array) / sizeof(Array[0]))
const GUI_POINT aPoints[] = {
    {0, 20},
    {40, 20},
    {20, 0}
};
GUI_POINT aEnlargedPoints[countof(aPoints)];
void Sample(void)
{
    int i;
    GUI_Clear();
    GUI_SetDrawMode(GUI_DM_XOR);
    GUI_FillPolygon(aPoints, countof(aPoints), 140, 110);
    for (i = 1; i < 10; i++)
    {
        GUI_EnlargePolygon( aEnlargedPoints, aPoints,
                           countof(aPoints), i * 5);
        GUI_FillPolygon(aEnlargedPoints, countof(aPoints), 140, 110);
    }
}
```

上面范例程序运行结果的屏幕截图



GUI_FillPolygon

描述

在当前视窗中绘一个由一系列点定义的填充多边形。

函数原型

```
void GUI_FillPolygon(const GUI_POINT* pPoint, int NumPoints, int x, int y);
```

参 数	含 意
<code>pPoint</code>	显示的填充多边形的指针
<code>Numpoints</code>	在点的序列中指定点的数量
<code>x</code>	原点的X轴坐标
<code>y</code>	原点的Y轴坐标

附加信息

绘制的折线通过连接起点与终点而自动闭合。终点必须不能与多边形的轮廓接触。

GUI_MagnifyPolygon

描述

通过指定一个因数对多边形进行放大。

函数原型

```
void GUI_MagnifyPolygon ( GUI_POINT* pDest,
                          const GUI_POINT* pSrc,
                          int NumPoints,
                          int Mag);
```

参 数	含 意
<code>pDest</code>	目标多边形的指针
<code>pSrc</code>	源多边形的指针
<code>Numpoints</code>	在点的序列中指定点的数量
<code>Mag</code>	多边形的放大因数

附加信息

确认目标点的阵列等于或大于源阵列。注意对多边形进行扩边与放大的不同。设置参数Len为1将会对多边形的所有边进行1个像素的扩大，而设置参数Mag为1，则对多边形没有任何影响。

范例

```
#define countof(Array) (sizeof(Array)/sizeof(Array[0]))
const GUI_POINT aPoints[] = {
    {0, 20},
    {40, 20},
    {20, 0}
};
GUI_POINT aMagnifiedPoints[countof(aPoints)];
void Sample(void)
{
    int Mag, y = 0, Count = 4;
    GUI_Clear();
    GUI_SetColor(GUI_GREEN);
    for (Mag = 1; Mag <= 4; Mag *= 2, Count /= 2)
    {
        int i, x = 0;
        GUI_MagnifyPolygon( aMagnifiedPoints, aPoints,
                           countof(aPoints), Mag);
        for (i = Count; i > 0; i--, x += 40 * Mag)
        {
            GUI_FillPolygon(aMagnifiedPoints, countof(aPoints), x, y);
        }
        y += 20 * Mag;
    }
}
```

上面范例程序运行结果的屏幕截图



GUI_RotatePolygon

描述

按指定角度旋转一个多边形。

函数原型

```
void GUI_RotatePolygon( GUI_POINT* pDest,
                        const GUI_POINT* pSrc,
                        int NumPoints,
                        float Angle);
```

参 数	含 意
<code>pDest</code>	目标多边形的指针
<code>pSrc</code>	源多边形的指针
<code>Numpoints</code>	在点的序列中指定点的数量
<code>Angle</code>	多边形旋转的角度（以弧度为单位）

附加信息

确认目标点的阵列等于或大于源阵列。

范例

下面的范例显示如何画一个多边形。该范例源程序文件是“Samples\Misc\DrawPolygon.c”。

```
/*-----
文件:      DrawPolygon.c
```

 * 箭头的点 *

```

/*****
*                                     绘制一个多边形                                     *
*****/

```


参数含意

pDest: 目标多边形的指针。
pSrc: 源多边形的指针
NumPoints: 在一个点的序列中指定点的数量
Angle: 旋转多边形的角度（以弧度为单位）

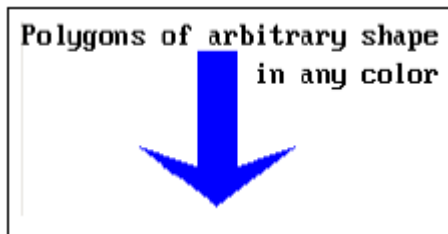
```

/*****
*                                     *
*****/

void main(void)
{
    GUI_Init();
    DrawPolygon();
    while(1)
        GUI_Delay(100);
}

```

上面范例程序运行结果的屏幕截图



6.7 绘圆

GUI_DrawCircle

描述

在当前视窗指定坐标以指定的尺寸绘制一个圆。

函数原型

```
void GUI_DrawCircle(int x0, int y0, int r);
```

参 数	含 意
<code>x0</code>	在客户视窗中圆心的X轴坐标（以像素为单位）
<code>y0</code>	在客户视窗中圆心的Y轴坐标（以像素为单位）
<code>r</code>	圆的半径（直径的一半）。 最小值：0（结果是一个点） 最大值：180

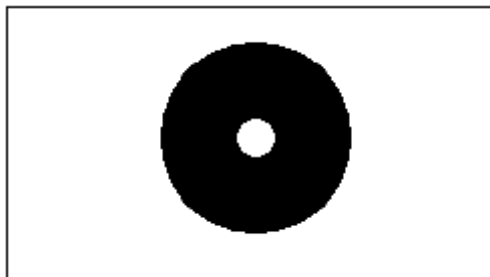
附加信息

该函数不能处理超过180的半径，因为那样将使用到导致溢出错误的整数运算。但是，对于大多数嵌入式应用，这不算是一个问题，因为一个直径为360的圆无论如何都会比显示屏要大。

范例

```
// 画同心圆
void ShowCircles(void)
{
    int i;
    for (i=10; i<50; i++)
        GUI_DrawCircle(120, 60, i);
}
```

上面范例程序执行结果的屏幕截图



GUI_FillCircle

描述

在当前视窗指定坐标以指定的尺寸绘制一个填充圆。

函数原型

```
void GUI_FillCircle(int x0, int y0, int r);
```

参 数	含 意
<code>x0</code>	在客户视窗中圆心的X轴坐标（以像素为单位）
<code>y0</code>	在客户视窗中圆心的Y轴坐标（以像素为单位）
<code>r</code>	圆的半径（直径的一半）。 最小值：0（结果是一个点） 最大值：180

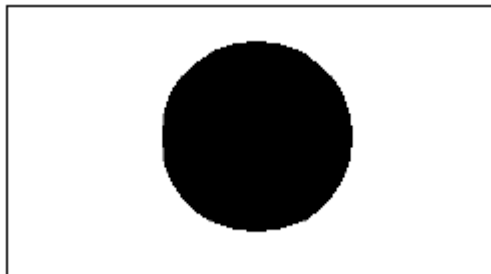
附加信息

该函数不能处理超过180的半径。

范例

```
GUI_FillCircle(120, 60, 50);
```

上面范例程序执行结果的屏幕截图



6.8 绘椭圆

GUI_DrawEllipse

描述

在当前视窗的指定坐标以指定的尺寸绘一个椭圆。

函数原型

```
void GUI_DrawEllipse (int x0, int y0, int rx, int ry);
```

参 数	含 意
<code>x0</code>	在客户视窗中圆心的X轴坐标（以像素为单位）
<code>y0</code>	在客户视窗中圆心的Y轴坐标（以像素为单位）
<code>rx</code>	椭圆的X轴半径（直径的一半）。 最小值：0，最大值：180
<code>ry</code>	椭圆的Y轴半径（直径的一半）。 最小值：0，最大值：180

附加信息

该函数处理的rx/ry参数不能超过180，因为那样将使用到导致溢出错误的整数运算。

范例

参考GUI_FillEllipse函数的范例

GUI_FillEllipse

描述

按指定的尺寸绘一个填充椭圆。

函数原型

```
void GUI_FillEllipse(int x0, int y0, int rx, int ry);
```

参 数	含 意
<code>x0</code>	在客户视窗中圆心的X轴坐标（以像素为单位）
<code>y0</code>	在客户视窗中圆心的Y轴坐标（以像素为单位）
<code>rx</code>	椭圆的X轴半径（直径的一半）。 最小值：0，最大值：180
<code>ry</code>	椭圆的Y轴半径（直径的一半）。 最小值：0，最大值：180

附加信息

该函数处理的rx/ry参数不能超过180。

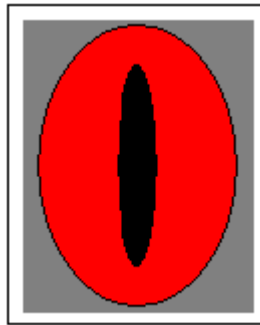
范例

```

/* 椭圆范例 */
GUI_SetColor(0xff);
GUI_FillEllipse(100, 180, 50, 70);
GUI_SetColor(0x0);
GUI_DrawEllipse(100, 180, 50, 70);
GUI_SetColor(0x000000);
GUI_FillEllipse(100, 180, 10, 50);

```

上面范例程序执行结果的屏幕截图



6.9 绘制圆弧

GUI_DrawArc

描述

在当前视窗的指定坐标按指定尺寸绘一段圆弧，一段圆弧就是一个圆的一部分轮廓。

函数原型

```
void GL_DrawArc (int xCenter, int yCenter, int rx, int ry, int a0, int a1);
```

参 数	含 意
<code>xCenter</code>	客户视窗中圆弧中心的水平方向坐标（以像素为单位）
<code>yCenter</code>	客户视窗中圆弧中心的垂直方向坐标（以像素为单位）
<code>rx</code>	X轴半径（像素）。
<code>ry</code>	Y轴半径（像素）。
<code>a0</code>	起始角度（度）
<code>a1</code>	终止角度（度）

限制

现在不使用参数ry，取而代之的是参数rx。

附加信息

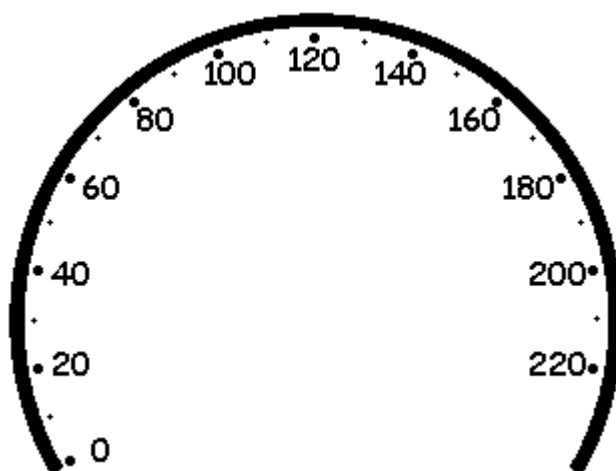
GUI_DrawArc使用浮点库。处理的参数rx/ry不能超过180，因为那样将使用到导致溢出错误的整数运算。

范例

```
void DrawArcScale(void)
{
    int x0 = 160;
    int y0 = 180;
    int i;
    char ac[4];
    GUI_SetBkColor(GUI_WHITE);
    GUI_Clear();
    GUI_SetPenSize( 5 );
    GUI_SetTextMode(GUI_TM_TRANS);
    GUI_SetFont(&GUI_FontComic18B_ASCII);
    GUI_SetColor( GUI_BLACK );
    GUI_DrawArc( x0,y0,150, 150,-30, 210 );
    GUI_Delay(1000);
    for (i=0; i<= 23; i++)
    {
        float a = (-30+i*10)*3.1415926/180;
        int x = -141*cos(a)+x0;
        int y = -141*sin(a)+y0;
        if (i%2 == 0)
            GUI_SetPenSize( 5 );
        else
            GUI_SetPenSize( 4 );
        GUI_DrawPoint(x,y);
        if (i%2 == 0)
        {
```

```
x = -123*cos(a)+x0;  
y = -130*sin(a)+y0;  
sprintf(ac, "%d", 10*i);  
GUI_SetTextAlign(GUI_TA_VCENTER);  
GUI_DispStringHCenterAt(ac, x, y);  
}  
}  
}
```

上面范例程序执行结果的屏幕截图



第7章 字体

随 μ C/GUI 一起提供的普通字体大部分是标准字体。实际上，你或许会发现这些字体对于你的应用已完全足够。对单独的字体，想了解更详细的情况，请参考第 25 章：标准字体，这一章描述了 μ C/GUI 中所有的字体，及这些字体在屏幕上显示时的所有字符。

μ C/GUI 支持 ASCII，ISO 8859-1 及 Unicode。通常， μ C/GUI 按 8 位字符进行编译，允许最大为 256 的不同的字符代码，32 之前的编码除外，这部分字符作为控制字符保留。字符是否有效取决所选择的字体（即该字体是否包括有所需的字符）。

7.1 有效字体

当前 μ C/GUI 版本提供 4 种字体：等宽位图字体，比例位图字体，带有 2bpp (bit/pixel) 用于建立反混淆信息的比例位图字体，带有 4bpp (位/像素) 用于建立反混淆信息的反混淆字体。（更多关于抗锯齿字体的信息请参考第 15 章：“抗锯齿”）

所有的字体都会与你的应用相连接，而字体的选择在 GUIConf.h 中定义。我们推荐编译所有的字体并将它们作为一个库模块进行连接，或者将所有的字体工程文件放入一个你能与你的应用相连接的库当中。这种方法可以让你确定那些你在应用中需要的字体被真正连接。字体转换器（在一本独立的手册中描述）用于创建附加的字体。

为了能在你的应用中使用一种字体，你必须要做到下面几点：

- 字体在与 μ C/GUI 规范相兼容的“C”文件，工程文件或库这三种文件中任一种当中。
- 字体文件与你的应用链接。
- 字体的描述要包含在 GUIConf.h 中（这很必要，这是为了避免由于没有声明的外部常量而产生的编译警告）

增加字体

一旦你连接过一个如上面所描述的字体文件，将要连接的字体声明为一个外部常量 GUI_FONT，如下面范例所显示的那样：

范例

```
extern const GUI_FONT GUI_FontNew;

int main (void)
{
    GUI_Init () ;
    GUI_Clear();
    GUI_SetFont(&GUI _FontNew);
    GUI_DispString("Hello world\n");
    return 0;
}
```

选择字体

μ C/GUI 提供不同的字体，总会有其中的一种被选中。可以通过调用函数 GUI_SetFont()

改变所选择的字体，该函数选择字体用于伴随当前任务的文字输出。

如果在你的应用中没有字体被选择，则使用默认字体。该默认值由 GUIConf.h 配置，可以进行修改。你应该确认默认字体是你的应用中真正用到的字体，因为默认字体会与你的应用连接，因此可能会耗尽 ROM 存储空间。

μC/GUI 的兼容性

老版本的μC/GUI 使用一个不同的字体概念，字体在一个字体表中列出，它们在表格中的位置通过一个整数选择。由于缺乏灵活性，这种概念改变了。新的概念较原来的概念提高了一个等级，字体标识符（例如 F6x8）依然有效。

7.2 字体API

下表列出了与字体处理相关的函数，在各自的类型中按字母顺序进行排列。函数的详细描述后面列出。

函 数	说 明
字体的选择	
GUI_GetFont()	返回当前选择字体的指针
GUI_SetFont()	设置当前字体
字体相关函数	
GUI_GetCharDistX()	返回当前字体中指定字符的宽度（X 轴，以像素为单位）
GUI_GetFontDistY()	返回当前字体 Y 轴方向间距
GUI_GetFontInfo()	返回一个包含字体信息的结构
GUI_GetFontSizeY()	返回当前字体的高度（Y 轴，以像素为单位）
GUI_GetStringDistX()	返回一个使用当前字体的文本的 X 轴尺寸
GUI_GetYDistOfFont()	返回一个特殊字体的 Y 轴间距
GUI_GetYSizeOfFont()	返回一个特殊字体的 Y 轴尺寸
GUI_IsInFont()	估计一个指定的字符是否在一种特殊字体里面

7.3 一种字体的选择

GUI_GetFont()

描述

返回当前选择字体的指针。

函数原型

```
const GUI_FONT * GUI_GetFont(void)
```

GUI_SetFont()

描述

设置用于文字输出的字体。

函数原型

```
const GUI_FONT * GUI_SetFont(const GUI_FONT * pNewFont) ;
```

参数	含义
<code>pFont</code>	所选择及使用字体的指针

返回值

返回先前所选择字体的指针，这样你可以在稍后一点恢复原先使用的字体。

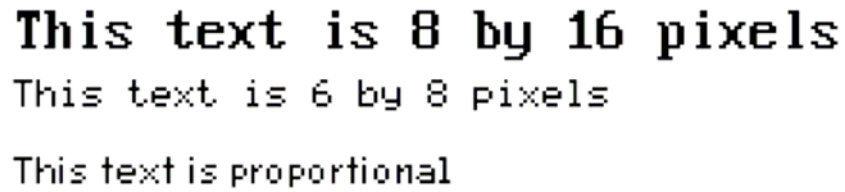
范例

用 3 种不同尺寸显示样本文字，然后恢复原先的字体：

void DispText(void)

```
{
    const GUI_FONT GUI_FLASH* OldFont=GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringAt("This text is 8 by 16 pixels", 0, 0);
    GUI_SetFont(&GUI_Font6x8);
    GUI_DispStringAt("This text is 6 by 8 pixels", 0, 20);
    GUI_SetFont(&GUI_Font8);
    GUI_DispStringAt("This text is proportional", 0, 40);
    GUI_SetFont(OldFont);           // 恢复字体
}
```

上面范例程序运行结果的屏幕截图：



This text is 8 by 16 pixels

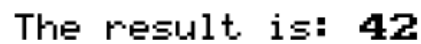
This text is 6 by 8 pixels

This text is proportional

用不同的字体显示文字和数值：

```
GUI_SetFont(&GUI_Font6x8);
GUI_DispString("The result is:");    // 显示文字
GUI_SetFont(&GUI_Font8x8);
GUI_DispDec(42, 2);                  // 显示数值
```

上面范例程序运行结果的屏幕截图：



The result is: 42

7.4 字体相关函数

GUI_GetCharDistX()

描述

返回当前选择字体中用于显示一个指定字符的宽度（X 轴，以像素为单位）

函数原型

```
int GUI_GetCharDistX(U16 c);
```

参数	含义
<code>c</code>	需计算宽度的字符

GUI_GetFontDistY()

描述

返回当前选择字体 Y 轴方向间距

函数原型

```
int GUI_GetFontDistY(void);
```

附加信息

Y 轴方向间距是一个以像素为单位在两个文字相邻的线之间的垂直距离。返回值是当前选择字体入口 Y 轴方向距离数值。该返回值对于比例字体及等宽字体都有效。

GUI_GetFontInfo()

描述

计算指向一个特殊字体的 GUI_FONTINFO 结构的指针。

函数原型

```
void GUI_GetFontInfo(const GUI_FONT* pFont, GUI_FONTINFO* pfi);
```

参数	含义
pFont	指向字体的指针
pfi	指向一个 GUI_FONTINFO 结构的指针

附加信息

GUI_FONTINFO 结构的定义如下：

```
typedef struct
{
    U16 Flags;
}GUI_FONTINFO;
```

变量标志的成员使用以下数值：

```
GUI_FONTINFO_FLAG_PROP
GUI_FONTINFO_FLAG_MONO
GUI_FONTINFO_FLAG_AA
GUI_FONTINFO_FLAG_AA2
GUI_FONTINFO_FLAG_AA4
```

范例

获得 GUI_Font6x8 字体的信息。计算后，FontInfo.Flags 包含 GUI_FONTINFO_FLAG_MONO 标志。

```
GUI_FONTINFO FontInfo;  
GUI_GetFontInfo(&GUI_Font6x8, &FontInfo);
```

GUI_GetFontSizeY()

描述

返回当前选择字体的高度（Y 轴，以像素为单位）

函数原型

```
int GUI_GetFontSizeY(void);
```

附加信息

返回值是当前选择字体入口 Y 轴方向大小数值。该值小于或等于通过执行 GUI_GetFontDistY() 获得的返回值——Y 轴方向间距。

该返回值对于比例字体及等宽字体都有效。

GUI_GetStringDistX()

描述

返回在当前选择的字体中用于显示一个指定字符串的 X 轴尺寸。

函数原型

```
int GUI_GetStringDistX(const char GUI_FAR *s);
```

参数	含义
s	字符串的指针

GUI_GetYDistOfFont()

描述

返回一种特殊字体的 Y 轴方向间距。

函数原型

```
int GUI_GetYDistOfFont(const GUI_FONT* pFont);
```

参数	含义
pFont	字体的指针

附加信息

参考 GUI_GetFontDistY()。

GUI_GetYSizeOfFont()

描述

返回一种特殊字体的 Y 轴方向尺寸。

函数原型

```
int GUI_GetYSizeOfFont(const GUI_FONT* pFont);
```

参数	含义
pFont	字体的指针

附加信息

参考 GUI_GetFontSizeY()

GUI_IsInFont()

描述

估计一个指定的字符是否在一种特殊字体里面

函数原型

```
char GUI_IsInFont(const GUI_FONT*pFont, U16 c);
```

参数	含义
<code>pFont</code>	字体的指针
<code>c</code>	搜索的字符

附加信息

如果指针 `pFont` 设置为 0，则使用当前选择字体。

范例

估计字体 `GUI_FontD32` 是否包含 “X”：

```
if (GUI_IsInFont(&GUI_FontD32, 'X') == 0)
{
    GUI_DisString("GUI_FontD32 does not contains 'X' ");
}
```

7.5 字符设置

ASCII

μC/GUI 支持所有的 ASCII 字符。下表是从 32 到 127 共 96 个字符：

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2x		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

很不幸，因为 ASCII 立足于美国用于信息互换的标准，它为美国的需要而制定。它不包括用于欧洲语言的任何特殊的字符，诸如 Ä, Ö, Ü, á, à 等等。没有单独的标准适合这些 ASCII 字符的“欧洲扩展”。现在已经有几个不同的标准，其中一个用于互联网并且被大多数 Windows 应用程序所接受的标准是 ISO8859-1，一个 ASCII 字符的扩展集。

ISO 8859-1 西文、拉丁文字符设置

μC/GUI 支持 ISO 8859-1，字符的定义如下表所示：

代码	描述	字符
160	non-breaking space	
161	inverted exclamation	¡
162	cent sign	¢
163	pound sterling	£
164	general currency sign	₭
165	yen sign	¥
166	broken vertical bar	
167	section sign	§
168	umlaut (dieresis)	¨
169	copyright	©
170	feminine ordinal	ª
171	left angle quote, guillemot left	«
172	not sign	¬
173	soft hyphen	
174	registered trademark	®
175	macron accent	—
176	degree sign	°
177	plus or minus	±
178	superscript two	²
179	superscript three	³
180	acute accent	´
181	micro sign	μ
182	paragraph sign	¶
183	middle dot	•
184	cedilla	,
185	superscript one	¹
186	masculine ordinal	º
187	right angle quote, guillemot right	»
188	fraction one-fourth	¼
189	fraction one-half	½
190	fraction three-fourth	¾
191	inverted question mark	¿
192	capital A, grave accent	À
193	capital A, acute accent	Á

194	capital A, circumflex accent	Â
195	capital A, tilde	Ã
196	capital A, dieresis or umlaut mark	Ä
197	capital A, ring	Å
198	capital A, diphthong (ligature)	Æ
199	capital C, cedilla	Ç
200	capital E, grave accent	È
201	capital E, acute accent	É
202	capital E, circumflex accent	Ê
203	capital E, dieresis or umlaut mark	Ë
204	capital I, grave accent	Ì
205	capital I, acute accent	Í
206	capital I, circumflex accent	Î
207	capital I, dieresis or umlaut mark	Ï
208	Eth, Icelandic	Ð
209	N, tilde	Ñ
210	capital O, grave accent	Ò
211	capital O, acute accent	Ó
212	capital O, circumflex accent	Ô
213	capital O, tilde	Õ
214	capital O, dieresis or umlaut mark	Ö
215	multiply sign	×
216	capital O, slash	Ø
217	capital U, grave accent	Ù
218	capital U, acute accent	Ú
219	capital U, circumflex accent	Û
220	capital U, dieresis or umlaut mark	Ü
221	capital Y, acute accent	Ý
222	THORN, Icelandic	Þ
223	sharp s, German (s-z ligature)	ß
224	small a, grave accent	à
225	small a, acute accent	á
226	small a, circumflex accent	â
227	small a, tilde	ã
228	small a, dieresis or umlaut mark	ä
229	small a, ring	å
230	small ae diphthong (ligature)	æ
231	cedilla	ç
232	small e, grave accent	è

233	small e, acute accent	é
234	small e, circumflex accent	ê
235	small e, dieresis or umlaut mark	ë
236	small i, grave accent	ì
237	small i, acute accent	í
238	small i, circumflex accent	î
239	small i, dieresis or umlaut mark	ï
240	small eth, Icelandic	ð
241	small n, tilde	ñ
242	small o, grave accent	ò
243	small o, acute accent	ó
244	small o, circumflex accent	ô
245	small o, tilde	õ
246	small o, dieresis or umlaut mark	ö
247	division sign	÷
248	small o, slash	ø
249	small u, grave accent	ù
250	small u, acute accent	ú
251	small u, circumflex accent	û
252	small u, dieresis or umlaut mark	ü
253	small y, acute accent	ý
254	small thorn, Icelandic	þ
255	small y, dieresis or umlaut mark	ÿ

Unicode

Unicode 是最终的字符编码，它是一个基于 ASCII 和 ISO8859-1 的国际标准，UNICODE 要求 16 位的字符，因为所有的字符都有它们的固有码。目前，已经定义了超过 30,000 个不同的字符。不过，不是所有的字符图像能在 μ C/ GUI 中定义。定义这些附加的字符是使用者的工作。请联系 Micrium 公司，或你的发行人，因为我们可能有你所需要的字符。

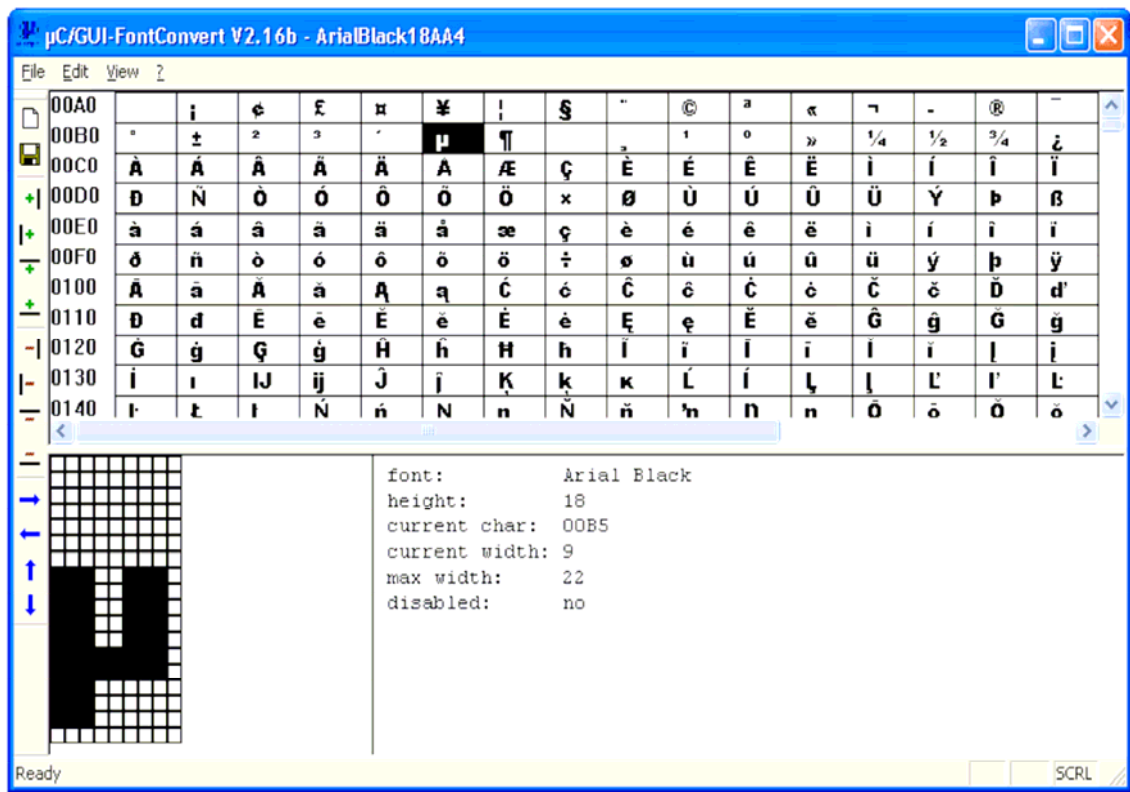
7.6 字体转换器

能用于 μ C/ GUI 的字体必须在“C”中定义为 GUI_FONT 结构。该结构或者由这些结构引用的与之相当的字体数据，可能相当大。手工产生这些字体时耗时巨大且效率很低。因此我们推荐使用字体转换器，它能够从字体自动产生“C”文件。

字体转换器是一个简单的 Windows 程序。你只需在程序载入一种 Windows 安装字体，如果有需要则对其进行编辑，然后将其保存为“C”文件。该“C”文件可以进行编译，你需要

的字体就可以随 μ C/GUI 一道在屏幕上显示。

默认情况下字符代码 0x00~0x1F 及 0x80~0x9F 是被禁止的。下面为字体转换器截入一种字体的范例的屏幕截图：



字体转换器在一份单独的资料中描述，你可以与 Micrium 公司联系以获得这份资料。

第8章 位图转换器

能用于 μ C/GUI 的位图通常定义为“C”的 GUI_BITMAP 结构。该结构或者由这些结构引用的相当的图片数据，可能相当大。在手工产生这些位图时，耗时巨大且效率很低，特别是你在处理相当大的图片及多级灰度梯度或颜色时。因此，我们推荐使用位图转换器。

位图转换器的一个很容易使用的 Windows 程序。仅仅载入一幅位图（.bmp 格式）到程序中，如果需要则转换该位图，然后将结果保存为一个“C”文件，供 μ C/GUI 使用，这样就能在屏幕上显示这幅位图了。

8.1 介绍

位图转换器作为一个工具，将位图从 PC 格式转换成“C”文件。同时也能够进行色彩转换，这样，最终得到的“C”代码必然很大。你应当减少像素深度，这样是为了减少内存的消耗。位图转换器能够显示所要转换的位图。

位图转换器中可以执行有限数量的一些简单函数，包括水平或垂直反转，旋转，转换位图索引或颜色（这些特性都可以在“Image”菜单下找到）。更进一步的图像处理要使用一个位图处理软件，诸如 Adobe Photoshop 或 Corel Photopaint。通常，使用这些软件进行图像处理非常有意义，使用位图转换器仅仅达到转换的目的。

8.2 支持的输入格式

一幅图像必须首先以一个 .bmp 格式文件的位图形式载入，直接载入或通过剪贴板都行。以下类型的 .bmp 文件可以在程序中载入：

- 带调色板的每像素 (bpp) 1, 4 或 8 位格式；
- 无调色板的 24bpp (RGB/全彩色模式)
- RLE4 和 RLE8 格式

其它的图像格式可以拷贝到剪贴板，转换成一个 .bmp 文件，然后再载入位图转换器中。

8.3 支持的输出格式

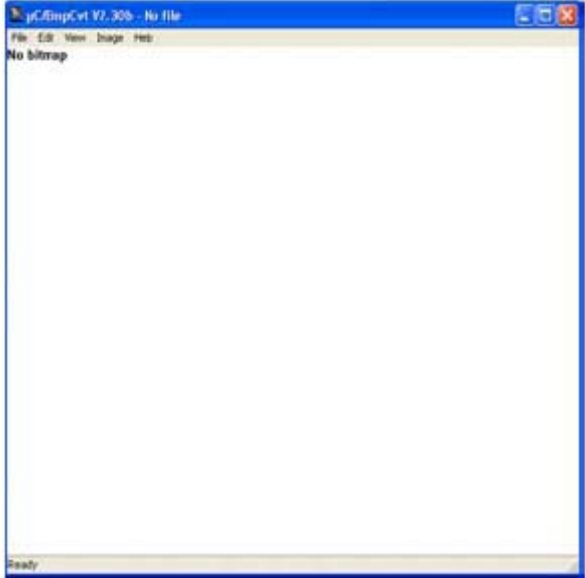

转换后的位图可以保存为一个 .bmp 文件（可以再次载入及使用或用其它位图处理软件载入）或一个“C”文件。一个“C”文件作为你的“C”编译器的输入文件使用。它可以包括一个调色板（器件无关位图，即 DIB）或不包括（器件有关位图，即 DDB）。推荐使用 DIB，因为它们可以在任何 LCD 上正确显示；一个 DDB 只能在一个使用与位图同样调色板的 LCD 上正确的显示。

8.4 从位图产生“C”文件

位图转换器的主要功能是将 PC 格式的位图转换成一个能被 μ C/GUI 所使用的“C”文件。然而，在做这件事之前，通常希望能修改一幅图像的调色板，以使产生的“C”文件不至于太过庞大。对于全彩色位图，很有必要将其转换成调色板格式的位图，因为位图转换器不能从一个 RGB 模式的位图生成“C”文件。

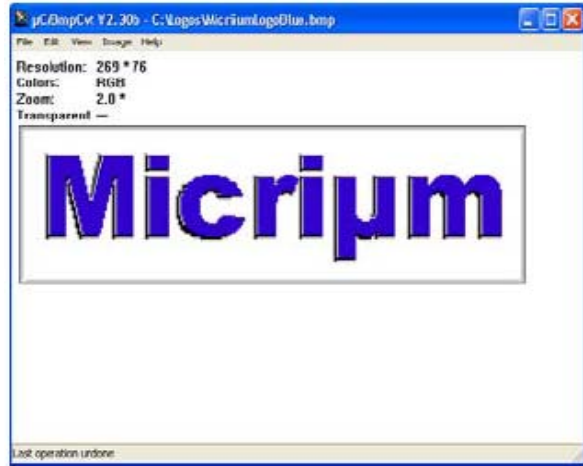
可以生成的“C”文件可以是“带调色板的C”，“不带调色板的C”，“压缩的带调色板的C”，“压缩的不带调色板的C”。想了解更多关于压缩的信息，请参考“压缩的位图”部分及本章末的范例。

位图转换器的基本的操作过程如下表中的插图描述的步骤：

过程	屏幕截图
<p>第一步：开始应用</p> <p>位图转换器以一个空窗口的形式打开。</p>	
<p>第二步：载入一幅位图到位图转换器中</p> <p>选择“File/Open”</p> <p>查找到你所想打开的文件，点击“Open”（必须是一个.bmp 文件）。</p> <p>在这个例子中，选择的位图文件是 MicriumLogo.bmp。</p>	

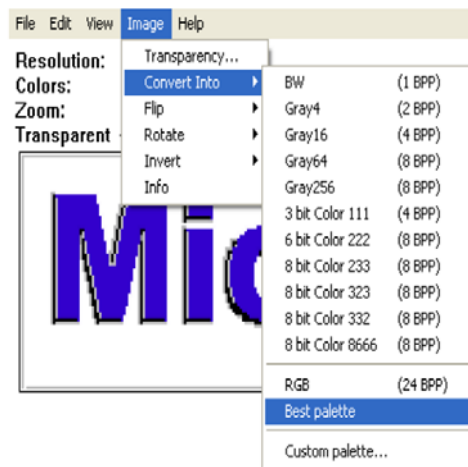
第三步：位图转换器显示载入的（源）位图

在这个例子中，源位图是全彩色（RGB）模式，因此在产生一个“C”文件之前，必须将它转换成一个调色板格式。



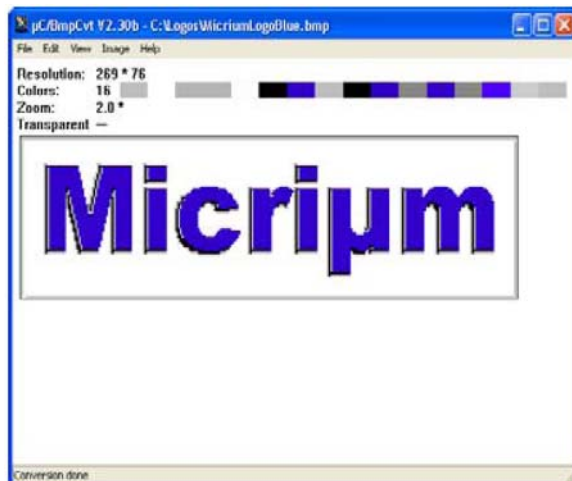
第四步：如果需要，转换位图模式

选择“Image/Convert Into.”再选择所希望的调色板，在该范例当中，我们选择“Best palette（最佳调色板）”



第五步：位图转换器显示转换后的位图

在这个例子中，在显示方面该图像没有改变，但是仅使用了一个 16 色的调色板代替原先的全彩色模式后，内存的使用会少了很多。显示这幅特定的位图所用到的实际颜色都包括在这 16 种颜色当中。

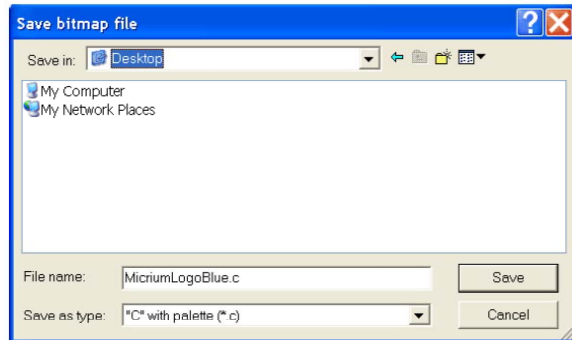


第六步：将位图保存为一个“C”文件

选择“File/Save”，为这个“C”文件起一个名字并选择一个保存位置。选择文件类型，在本范例中，文件以“C with palette”类型保存。

点击“Save”。

位图转换器将在指定目录建立一个独立的文件，该文件包括位图的 C 代码。



8.5 色彩转换

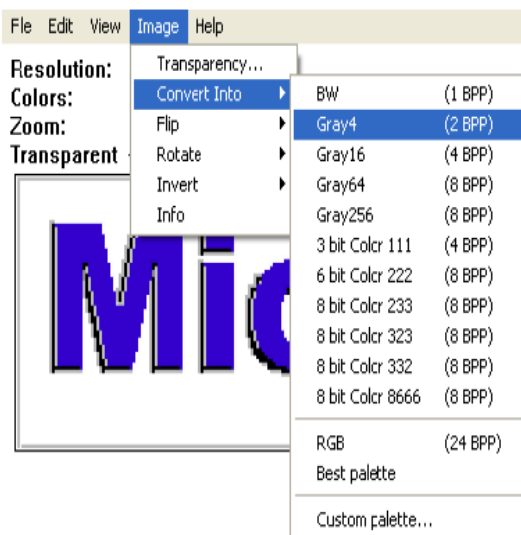
转换一幅位图的颜色格式的主要目的是为了减少存储器的消耗以生成一个效率更高的“C”文件。

实现这个目的最普通的办法是如上面范例所述的使用“Best palette”选项，这是给专门的位图定制调色板，包含只用于位图的颜色。这对于全彩色位图特别有用，这可以使调色板尽可能小，而依然能完全支持位图。一旦一个位图文件在位图转换器中打开，只需简单的在菜单中选择“Image/Convert Into/Best palette”就可实现。

对于某些应用，使用一个固定的调色板更有效，在菜单“Image/Convert Into.”下选择。例如，假设一幅全彩色模式的位图在屏幕上显示，而屏幕只支持四级灰度。这幅位图会浪费存储间以保持位图的原始格式，尽管它在屏幕上只以四级灰度显示。转换的过程如下所示：

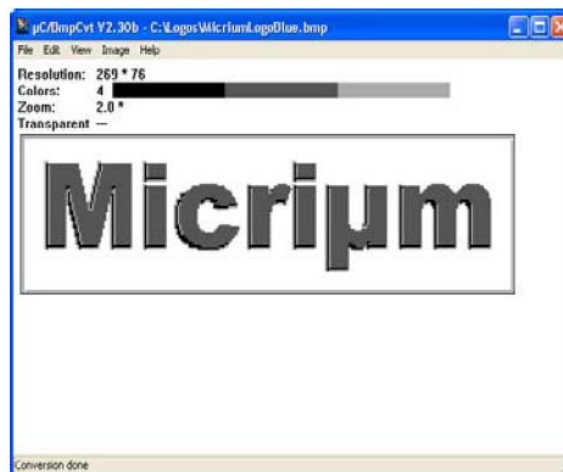
过程	屏幕截图
<p>位图转换器显示载入的（源）位图，如先前的范例一样（全彩色模式）。</p>	

选择 “Image/Convert Into/Gray4”



位图转换器显示转换后的位图。

在本范例中，因为使用了一个只有四级灰度的调色板代替全彩色模式，位图使用的内存空间减小了。如果目标屏幕仅支持四级灰度，则使用一个更高的像素深度是没有作用的，只会浪费内存。



8.6 压缩的位图

位图转换器及µC/GUI 在结果的源代码文件中支持位图的 run-length encoding (RLE, 行程长度编码) 压缩方式。如果你的位图包括很多相等颜色的像素序列的话，RLE 压缩方法是行之有效的。一幅有效率压缩位图将保存为在意义数量的空间。然而，对于摄影图片并不推荐使用压缩，因为它们通常并不含有相同的像素序列。同时也应该注意到压缩位图在显示时使用的时间会稍长一些，因为它首先要进行解压缩。

如果你想使用 RLE 压缩方式保存一幅位图，你可以这样做，当你将位图为 “C” 时，选择一种压缩的输出格式，如 “压缩的带调色板的 C 文件” 或 “压缩的不带调色板的 C 文件”。没有专门的函数用于显示压缩位图，压缩位图的显示与非压缩位图的处理是一样的。

8.7 使用定制调色板

在某些环境下，进行转换时可能要使用一个定制调色板才能获得理想的效果。既然如此（通常只是你有一个使用特殊的定制调色板的彩色显示器，而你希望位图也使用同一个调色板），应必须用到一个定制调色板。你可以选择菜单中的“Image/Convert Into/Custom palette”在实现这个功能。

定制调色板的文件格式

定制调色板文件是一个简单的文件，定义用于转换的有效颜色，它们包括这些内容：

- 头（8 字节）
- 颜色数 NumColors（U32，4 字节）
- 0（4 字节）
- U32 颜色[NumColors]（NumColors*4 字节，GUI_COLOR 类型）

因此整个文件大小为： $16 + (\text{NumColors} * 4)$ 个字节。一个 8 种颜色的定制调色板将会占用： $16 + (8 * 4) = 48$ 字节。在这点上，可以使用一个二进制编辑器建立这样一个文件。

支持的最大颜色数为 256，最小为 2。

定制调色板范例文件

该范例文件定义一个调色板，包含两种颜色：红与白：

```
0000: 65 6d 57 69 6e 50 61 6c 02 00 00 00 00 00 00 00
0010: ff 00 00 00 ff ff ff 00
```

8 字节的头构成第一行的头 8 个字节。接下来的是 U32，表示为后面的 4 个字节，其保存方式是 LSB（最低有效位）在前（大端模式），再紧接着是 4 个字节 0。颜色保存为每种基色 1 个字节，第 4 个字节为 0：RRGGBB00。因此代码的第二行定义该范例用到的两种颜色。

8.8 命令行方式

位图转换器也能够使用命令行提示进行工作。位图转换器菜单下的所有有效的转换函数在命令行下也同样有效。用于一幅位图处理的很多函数都可以用单行的命令行完成。命令输入格式如下所示：

BmpCvt<文件名>.bmp <-命令>

（如果用到多个命令，则在每两个命令之间要用一个空格隔开）

例如，一幅名为“MicriumLlogo”的位图要转换成最佳调色板格式并存为一个名为“logo2”的“C”文件，同时完成这两步操作可以如下所示在命令提示行中键入：

```
BmpCvt MicriumLogo200.bmp -convertintobestpalette -saveaslogo2,1 -exit
```

注意，当文件载入位图转换器总是包括它的.bmp 扩展名，而-saveas 命令则不包括扩展名。用一个整数指定的所需要获得的文件类型。在上面-saveas 命令中的数字“1”表示“带调色板的C”。-exit 命令表示操作完成后自动关闭程序。更多的信息请参考下表。

有效的命令行选项

下表列出了所有允许的位图转换器的命令。你也可以随时通过在命令提示行中键入命令“BmpCvt /?”获得这些内容。

命 令	说 明
-convertintobw	转换为黑白两色
-convertintogray4	转换为 4 级灰度 (Gray4)
-convertintogray16	转换为 16 级灰度 (Gray16)
-convertintogray64	转换为 64 级灰度 (Gray64)
-convertintogray256	转换为 256 级灰度 (Gray256)
-convertinto111	转换为 111
-convertinto222	转换为 222
-convertinto233	转换为 233
-convertinto323	转换为 323
-convertinto332	转换为 332.
-convertinto8666	转换为 8666.
-convertintoRGB	转换为 RGB.
-convertintobestpalette	转换为最佳调色板
-convertintocustompalette- <filename>	转换为一个定制调色板 参数: filename: 用户指定的所希望的定制调色板的文件名
-fliph	水平翻转位图
-flipv	垂直翻转位图
-rotate90cw	顺时针方向 90 度旋转位图
-rotate90cc	逆时针方向 90 度旋转位图
-rotate180	180 度旋转位图

<code>-invertindices</code>	反转索引 Invert indices.
<code>-saveas<filename, type></code>	指定文件名保存文件 参数 filename : 用户指定的文件名, 不包括文件扩展名。 参数 type : 必须是如下所示的 1~6 当中的一个整数: 1. 带调色板的 “C” 文件 (.c 文件) 2. 不带调色板的 “C” 文件 (.c 文件) 3. 压缩的带调色板的 “C” 文件 (.c 文件) 4. 压缩的不带调色板的 “C” 文件 (.c 文件) 5. 流 (.dta 文件) 6. Windows 的位图文件 (.bmp 文件)
<code>-exit</code>	自动中止 PC 程序
<code>-help</code>	显示帮助对话框
<code>-?</code>	显示这个对话框

8.9 位图转换范例

使用位图转换器的典型例子是将你公司的标识转换成一个 “C” 的位图。看一下范例的位图描绘:



这幅位图载入位图转换器, 然后转换成最佳调色板格式, 保存为 “带调色板的 C” 文件。最终的 “C” 源代码如下所示 (一些数据省略掉了)。

最终的 C 代码 (由位图转换器产生)

```
/*
C -file generated by µC/BmpCvt V2.30b, compiled May 8 2002, 10: 05: 37

(c) 2002 Micrium, Inc. www.micrium.com

(c) 1998-2002 Segger Microcontroller Systeme GmbH www.segger.com

Source file: MicriumLogoBlue Dimensions: 269 * 76
```

第 10 页 μC/GUI 中文手册

我们可能使用同一幅位图映像建立一个压缩的C文件，如前所描述，简单地将位图载入及

转换，将其保存为“压缩的带调色板的C”格式文。源代码如下所示（一些数据省略掉了）：

位图中用到的总像素的数量为 $269 \times 76 = 20444$ 。

因为每个像素能用所需的 16 种颜色中的任一种构成位图，每个像素占用 4 位。每字节能存储两个像素，则图像尺寸未压缩为 $20444 \div 2 = 10222$ 字节。

在下面代码最后，总的压缩图像尺寸可以用 4702 字节表示 20444 个像素。

因此可以算出压缩比率： $10222 \div 4702 = 2.17$ 。

最终的压缩 C 代码（由位图转换器产生）

```

/*
C-file generated by µC/BmpCvt V2.30b, compiled May 8 2002, 10:05:37
(c) 2002 Micrium, Inc.
www.micrium.com
(c) 1998-2002 Segger
Microcontroller Systeme GmbH
www.segger.com
Source file: LogoCompressed
Dimensions: 269 * 76
NumColors: 10
*/

#include "stdlib.h"
#include "GUI.H"

/* Palette
The following are the entries of the palette table.
Every entry is a 32-bit value (of which 24 bits are actually used)
the lower 8 bits represent the Red component,
the middle 8 bits represent the Green component,
the highest 8 bits (of the 24 bits used) represent the Blue component
as follows: 0xBBGGRR
*/

const GUI_COLOR ColorsLogoCompressed[] = {
    0xBFBBBF, 0xFFFFFFFF, 0xB5B5B5, 0x000000
    , 0xFF004C, 0xB5002B, 0x888888, 0xCF0038
    , 0xCFCFCF, 0xC0C0C0

```



```

};

const GUI_LOGPALETTE PalLogoCompressed = {
    10, /* number of entries */
    0, /* No transparency */
    &ColorsLogoCompressed[0]
};

const unsigned char acLogoCompressed[] = {
    /* RLE: 270 Pixels @ 000,000*/ 254, 0x00, 16, 0x00,
    /* RLE: 268 Pixels @ 001,001*/ 254, 0x01, 14, 0x01,
    /* RLE: 001 Pixels @ 000,002*/ 1, 0x00,
    /* RLE: 267 Pixels @ 001,002*/ 254, 0x01, 13, 0x01,
    /* ABS: 002 Pixels @ 268,002*/ 0, 2, 0x20,
    ...
    /* ABS: 002 Pixels @ 268,073*/ 0, 2, 0x20,
    /* RLE: 267 Pixels @ 001,074*/ 254, 0x01, 13, 0x01,
    /* ABS: 003 Pixels @ 268,074*/ 0, 3, 0x20, 0x10,
    /* RLE: 267 Pixels @ 002,075*/ 254, 0x02, 13, 0x02,
    0}; /* 4702 for 20444 pixels */

const GUI_BITMAP bmLogoCompressed = {
    269, /* XSize */
    76, /* YSize */
    135, /* BytesPerLine */
    GUI_COMPRESS_RLE4, /* BitsPerPixel */
    acLogoCompressed, /* Pointer to picture data (indices) */
    &PalLogoCompressed /* Pointer to palette */
    ,GUI_DRAW_RLE4
};

/* *** End of file *** */

```

第9章 颜色

μ C/GUI 支持黑/白，灰度（不同亮度的单色）及彩色显示屏。同一个用户程序可以用于不同的显示屏，只需要改变 LCD 配置。色彩管理设法为应该显示的任何色彩找到最相近的匹配。

逻辑颜色：在应用中处理的颜色。一种逻辑颜色总是定义为一个 RGB 数值，这是一个 24 位的数值，其中每个基色 8 位，如：0xBBGGRR。因此，白色应该为 0xFFFFFF，黑色应该为 0x000000，大红为 0xFF0000。

物理颜色：显示屏实际显示的颜色。与逻辑颜色一样，同样定义为一个 24 位的 RGB 数值。在实际运行的时候，逻辑颜色映射到物理颜色。

对于显示色彩较少的显示屏（例如单色显示屏或 8/16 色 LCD）， μ C/GUI 通过一个优化版本的“最小平方偏移搜索”对它们进行转换。它对显示的颜色（逻辑颜色）及 LCD 实际能显示（物理颜色）的所有有效颜色进行比较，然后使用 LCD 度量认为最接近的颜色。

9.1 预定义颜色

除自定义颜色之外，在 μ C/GUI 中预先定义了一些标准的颜色，如下表所示：

定义	颜色	说明
GUI_BLACK	黑	0x000000
GUI_BLUE	蓝	0xFF0000
GUI_GREEN	绿	0x00FF00
GUI_CYAN	青	0xFFFF00
GUI_RED	红	0x0000FF
GUI_MAGENTA	洋红	0x8B008B
GUI_BROWN	褐	0x2A2AA5
GUI_DARKGRAY	深灰	0x404040
GUI_GRAY	灰	0x808080
GUI_LIGHTGRAY	浅灰	0xD3D3D3
GUI_LIGHTBLUE	淡蓝	0xFF8080
GUI_LIGHTGREEN	淡绿	0x80FF80
GUI_LIGHTCYAN	淡青	0x80FFFF
GUI_LIGHTRED	淡红	0x8080FF
GUI_LIGHTMAGENTA	淡洋红	0xFF80FF
GUI_YELLOW	黄	0x00FFFF
GUI_WHITE	白	0xFFFFFF

范例

```
/* 将背景色设为洋红 */
GUI_SetBkColor(GUI_MAGENTA);
GUI_Clear();
```

9.2 色彩条测试程序

下面的色彩条程序用于显示如下所示的 13 种颜色条：

黑 -> 红, 白 -> 红, 黑 -> 绿, 白 -> 绿, 黑 -> 蓝, 白 -> 蓝, 黑 -> 白, 黑 -> 黄, 白 -> 黄, 黑 -> 青, 白 -> 青, 黑 -> 洋红 及 白 -> 洋红。

这个小程序可以以任何颜色格式用于所有显示屏。当然，结果依赖于显示屏能够显示的颜色。该程序要求一个尺寸为 320*240 的显示屏以显示所有颜色。该程序用于证明对于显示屏不同颜色设定的效果。它同时也可能用作一个校验显示屏功能的测试程序，检查有效的颜

色及灰度，同时纠正颜色转换。屏幕截图来自 Windows 仿真器，如果你的设置和硬件是适当的，它看起来和你的显示屏实际输出正确的吻合。该程序在随 μ C/GUI 一道提供的范例中的名字为：COLOR_ShowColorBar.c。

```

/*-----
文件:      COLOR _ShowColorBar.c
目的:      绘制一个色彩条的例子
-----*/

#include "GUI.H"

/*****
*                      绘制 13 种颜色的色彩条                      *
*****/

void ShowColorBar(void)
{
    int x0 = 60, y0 = 40, yStep = 15, i;
    int NumColors = LCD_GetDevCap(LCD_DEVCAP_NUMCOLORS);
    int xsize = LCD_GetDevCap(LCD_DEVCAP_XSIZE) - x0;
    GUI_SetFont(&GUI_Font13HB_1);
    GUI_DispStringHCenterAt("μC/GUI-sample:Show color bars", 160, 0);
    GUI_SetFont(&GUI_Font8x8);
    GUI_SetColor(GUI_WHITE);
    GUI_SetBkColor(GUI_BLACK);
    #if(LCD_FIXEDPALETTE)
        GUI_DispString("Fixed palette: ");
        GUI_DispDecMin(LCD_FIXEDPALETTE);
    #endif
    GUI_DispStringAt("Red", 0, y0 + yStep);
    GUI_DispStringAt("Green", 0, y0 + 3 * yStep);
    GUI_DispStringAt("Blue", 0, y0 + 5 * yStep);
    GUI_DispStringAt("Grey", 0, y0 + 6 * yStep);
    GUI_DispStringAt("Yellow", 0, y0 + 8 * yStep);
    GUI_DispStringAt("Cyan", 0, y0 + 10 * yStep);
    GUI_DispStringAt("Magenta", 0, y0 + 12 * yStep);
    for(i=0; i < xsize; i++)
    {

```

```
U16 cs =(255 *(U32)i) / xsize;
U16 x = x0 + i; ;

/* 红色 */
GUI_SetColor(cs);
GUI_DrawVLine(x, y0, y0 +yStep - 1);
GUI_SetColor(0xff +(255 - cs) * 0x10100L);
GUI_DrawVLine(x, y0 +yStep, y0 + 2 * yStep - 1);

/* 绿色 */
GUI_SetColor(cs<<8);
GUI_DrawVLine(x, y0 + 2 * yStep, y0 + 3 * yStep - 1);
GUI_SetColor(0xff00 +(255 - cs) * 0x10001L);
GUI_DrawVLine(x, y0 + 3 * yStep, y0 + 4 * yStep - 1);

/* 蓝色 */
GUI_SetColor(cs * 0x10000L);
GUI_DrawVLine(x, y0 + 4 * yStep, y0 + 5 * yStep - 1);
GUI_SetColor(0xff0000 +(255 - cs) * 0x101L);
GUI_DrawVLine(x, y0 + 5 * yStep, y0 + 6 * yStep - 1);

/* 灰色 */
GUI_SetColor((U32)cs * 0x10101L);
GUI_DrawVLine(x, y0 + 6 * yStep, y0 + 7 * yStep - 1);

/* 黄色 */
GUI_SetColor(cs * 0x101);
GUI_DrawVLine(x, y0 + 7 * yStep, y0 + 8 * yStep - 1);
GUI_SetColor(0xffff +(255 - cs) * 0x10000L);
GUI_DrawVLine(x, y0 + 8 * yStep, y0 + 9 * yStep - 1);

/* 青色 */
GUI_SetColor(cs * 0x10100L);
GUI_DrawVLine(x, y0 + 9 * yStep, y0 + 10 * yStep - 1);
GUI_SetColor(0xffff00 +(255 - cs) * 0x1L);
GUI_DrawVLine(x, y0 + 10 * yStep, y0 + 11 * yStep - 1);

/* 洋红 */
GUI_SetColor(cs * 0x10001);
GUI_DrawVLine(x, y0 + 11 * yStep, y0 + 12 * yStep - 1);
GUI_SetColor(0xff00ff +(255 - cs) * 0x100L);
GUI_DrawVLine(x, y0 + 12 * yStep, y0 + 13 * yStep - 1);
}
```

```

/*****
*
*                               主函数
*
*****/

void main(void)
{
    GUI_Init() ;
    ShowColorBar();
    while(1)
        GUI_Delay(100);
}

```

9.3 固定的调色板模式

下表列出了有效的固定调色板颜色模式及必须的“#define”（需要在文件 LCDConf.h 中定义以取得这些模式）。

颜色模式	有效的颜色数	LCD_FIXEDPALETTE	LCD_SWAP_RB
1	2（黑和白）	1	0
2	4（灰度）	2	0
4	16（灰度）	4	0
111	8	111	0
222	64	222	0
233	256	233	0
-233	256	233	1
323	256	323	0
-323	256	323	1
332	256	332	0
-332	256	332	1
444	4096	444	0
555	32768	555	0
-555	32768	555	1
565	65536	565	0
-565	65536	565	1
8666	232	8666	0

详细的描述如下：

<p>1 模式: 1 bpp (黑和白)</p> <p>该模式必须用于每像素 1 位的单色显示屏。</p> <p>有效颜色数量: 2</p>	
<p>2 模式: 2 bpp (4 级灰度)</p> <p>该模式必须用于每像素 2 位的单色显示屏。</p> <p>有效颜色数量: $2 \times 2 = 4$.</p>	
<p>4 模式: 4 bpp (16 级灰度)</p> <p>该模式必须用于每像素 4 位的单色显示屏。</p> <p>有效颜色数量: $2 \times 2 \times 2 \times 2 = 16$</p>	
<p>111 模式: 3 bpp (每种基色 2 级)</p> <p>如果基本的 8 种颜色够用的话可以选择这种模式。如果你的硬件只支持每像素每基色 1 位, 或者你没有足够的视频内存用于更高的色彩浓度采用这种模式。</p> <p>有效颜色数量: $2 \times 2 \times 2 = 8$</p>	

222 模式：6 bpp （每种基色 4 级）

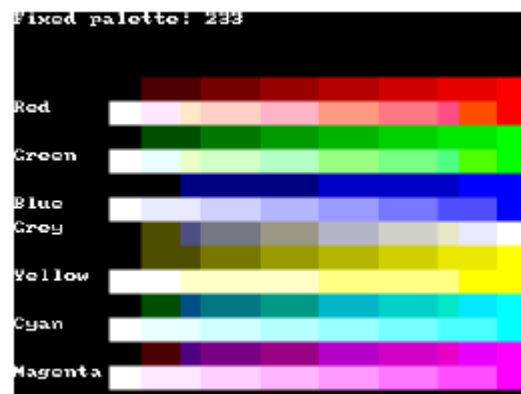
如果你的硬件没有一个调色板用于每种单独的颜色，该模式是一种很好的选择。每像素的每种基色保留 2 位；通常一个像素保存为一个字节。

有效颜色数量： $4 \times 4 \times 4 = 64$

**233 模式：2 位蓝色，3 位绿色，3 位红色**

该模式支持 256 种颜色。3 位用于颜色的红色和绿色部分，2 位用于蓝色部分。如图所示，结果是绿色和红色为 8 级，而蓝色为 4 级。有效颜色数量： $4 \times 8 \times 8 = 256$

基色顺序：BBGGRRRR

**-233 模式：2 位红色，3 位绿色，3 位蓝色，红蓝互换**

该模式支持 256 种颜色。3 位用于颜色的绿色和蓝色部分，2 位用于红色部分。有效的颜色与 332 模式是一样的。结果绿色和蓝色为 8 级，而红色为 4 级。

有效颜色数量： $4 \times 8 \times 8 = 256$

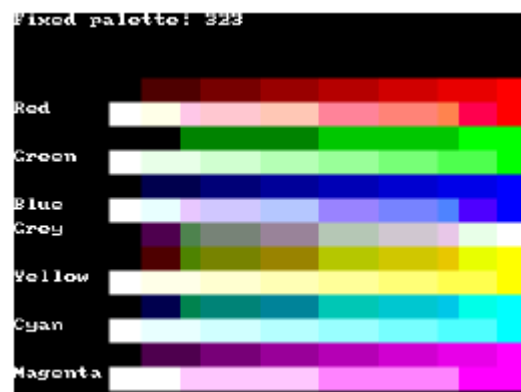
基色顺序：RRGGBBBB

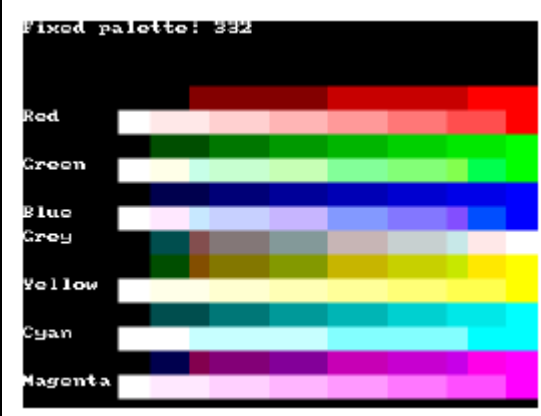
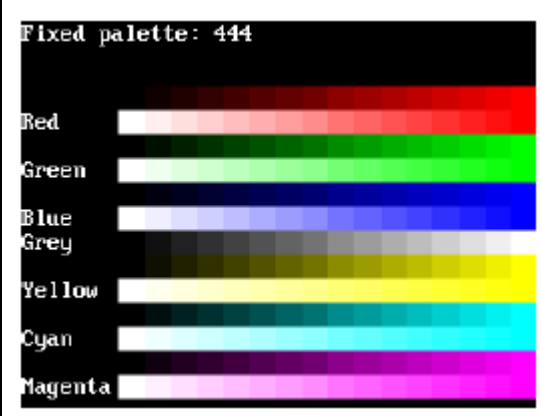
323 模式：3 位蓝色，2 位绿色，3 位红色

该模式支持 256 种颜色。3 位用于颜色的红色和蓝色部分，2 位用于绿色部分。如图所示，结果是蓝色和红色为 8 级，而绿色为 4 级。

有效颜色数量： $8 \times 4 \times 8 = 256$

基色顺序：BBBGGRRR



<p>-323 模式：3 位红色，2 位绿色，3 位蓝色，红蓝互换</p> <p>该模式支持 256 种颜色。3 位用于颜色的红色和蓝色部分，2 位用于绿色部分。有效的颜色与 323 模式是一样的。结果红色和蓝色为 8 级，而绿色为 4 级。</p> <p>有效颜色数量：8×4×8=256</p> <p>基色顺序：RRRGGBBB</p>	
<p>332 模式：3 位蓝色，3 位绿色，2 位红色</p> <p>该模式支持 256 种颜色。3 位用于颜色的蓝色和绿色部分，2 位用于红色部分。如图所示，结果是蓝色和绿色为 8 级，而红色为 4 级。</p> <p>有效颜色数量：8×8×4=256</p> <p>基色顺序：BBBGGGRR</p>	 <p>The image shows a color palette titled "Fixed palette: 332". It displays 8 levels of Red, Green, and Blue. The Red row shows 8 levels from black to red. The Green row shows 8 levels from black to green. The Blue row shows 8 levels from black to blue. The Grey row shows 8 levels from black to white. The Yellow row shows 8 levels from black to yellow. The Cyan row shows 8 levels from black to cyan. The Magenta row shows 8 levels from black to magenta.</p>
<p>-332 模式：3 位红色，3 位绿色，2 位蓝色，红蓝互换</p> <p>该模式支持 256 种颜色。3 位用于颜色的红色和绿色部分，2 位用于蓝色部分。有效的颜色与 233 模式是一样的。结果红色和绿色为 8 级，而蓝色为 4 级。</p> <p>有效颜色数量：8×8×2=256</p> <p>基色顺序：RRRGGBB</p>	
<p>444 模式：4 位红色，4 位绿色，4 位蓝色</p> <p>红，绿，蓝每部分平均分配 4 位。</p> <p>有效颜色数量：16×16×16=4096</p> <p>基色顺序：BBBBGGGRRRR</p>	 <p>The image shows a color palette titled "Fixed palette: 444". It displays 16 levels of Red, Green, and Blue. The Red row shows 16 levels from black to red. The Green row shows 16 levels from black to green. The Blue row shows 16 levels from black to blue. The Grey row shows 16 levels from black to white. The Yellow row shows 16 levels from black to yellow. The Cyan row shows 16 levels from black to cyan. The Magenta row shows 16 levels from black to magenta.</p>

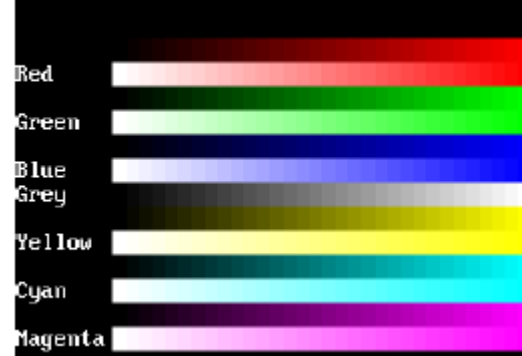
555 模式：5 位红色，5 位绿色，5 位蓝色

使用该模式需要一个支持15bpp的RGB颜色深度的LCD控制器（诸如SED1356或SED13806）。红，绿，蓝每部分平均分配5位。

有效颜色数量： $32 \times 32 \times 32 = 32768$

基色顺序：BBBBBGGGGRRRRR

Fixed palette: 555

**-555 模式：5 位蓝色，5 位绿色，5 位红色，红蓝互换**

使用该模式需要一个支持 15bpp 的 RGB 颜色深度的 LCD 控制器。红，绿，蓝每部分平均分配 5 位。有效的颜色与 555 模式是一样的。

有效颜色数量： $32 \times 32 \times 32 = 32768$.

基色顺序：RRRRRGGGGBBBBB

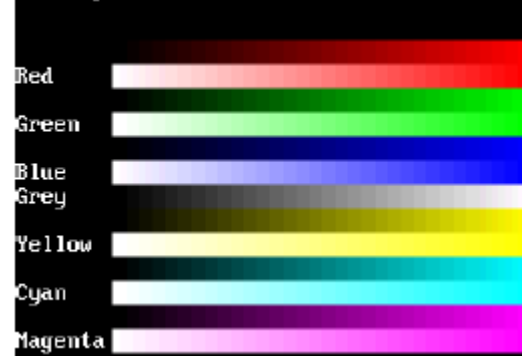
565 模式：5 位红色，6 位绿色，5 位蓝色

使用该模式需要一个支持 16bpp 的 RGB 颜色深度的 LCD 控制器红，绿，蓝每部分平均分配 5 位。一位未使用到。

有效颜色数量： $32 \times 64 \times 32 = 65536$.

基色顺序：BBBBBGGGGRRRRR

Fixed palette: 565

**-565 模式：5 位蓝色，6 位绿色，5 位红色，红蓝互换**

使用该模式需要一个支持 16bpp 的 RGB 颜色深度的 LCD 控制器红，绿，蓝每部分平均分配 5 位。一位未使用到。有效的颜色与 565 模式是一样的。

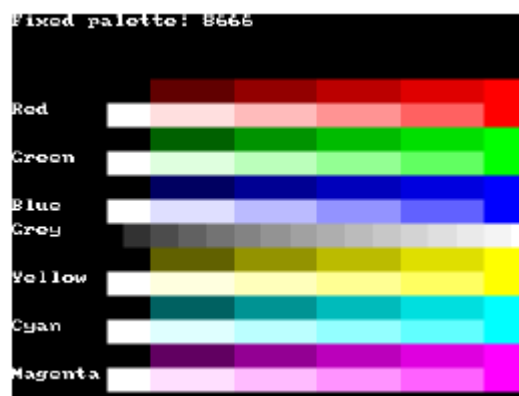
有效颜色数量： $32 \times 64 \times 32 = 65536$.

基色顺序：RRRRRGGGGBBBBB

8666 模式：8bpp，每种基色 6 级+ 16 级灰度

该模式多数用于一个可编程的颜色查询表（LUT），支持一个调色板总共 256 种可能的颜色。屏幕截图给出了一种有效颜色的意见，该模式包括一般目的应用的最好选择。每种基色 6 级有效亮度，附加 16 级灰度。

有效颜色数量： $6 \times 6 \times 6 + 16 = 232$



9.4 定制调色板模式

$\mu\text{C}/\text{GUI}$ 能处理一个定制的硬件调色板。定制调色板仅仅列出为硬件所使用的同一序列的所有有效的颜色。意思是无论如何你的 LCD 控制器/显示屏组件都能显示这些颜色。 $\mu\text{C}/\text{GUI}$ 能在 PC 仿真器上仿真这种应用，从而正确的在你的目标系统中处理这些颜色。

为了定义一个定制调色板，你应该在配置文件 LCDConf.h 中这样做。

范例

下面例子（LCDConf.h 部分）将定义一个定制调色板，其中包括 4 种颜色，它们全都是灰度色：

```
#define LCD_FIXEDPALETTE 0
#define LCD_PHYS_COLORS 0xffffffff, 0xaaaaaaaa, 0x55555555, 0x00000000
```

9.5 在运行时修改颜色查询表

每个像素的颜色信息既可以保存为 RGB 模式（每像素保持红、绿、蓝三部分），也可以保存为索引色模式（一个叫颜色索引的单个数字保存每个像素颜色信息）。每种颜色索引都反映一个查询表的条目，或者颜色映射，定义一个指定的 RGB 数值。

如果你的 LCD 控制器让一个颜色查询表（LUT）起作用，它完全可能通过 $\mu\text{C}/\text{GUI}$ 在初始化阶段（GUI_Init() \rightarrow LCD_Init() \rightarrow LCD_InitLUT() \rightarrow LCD_LO_SetLUTEntry()）进行初始化。然而，它可能需要在运行时修改 LUT（因为不同的理由）。一些可能的理由包括：

- 为了补偿显示屏问题（非线性）或进行伽马修正而进行颜色修正
- 显示屏的反相显示

- 需要使用比硬件能显示颜色（在一个时间段）更多的颜色（在不同的时间）

如果你仅仅在运行的时候修改 LUT，颜色转换函数将不会意识到这种改变，依然假定 LUT 是初始化时的原始状态。

使用不同的颜色

颜色表的缺省内容在对配置文件 GUIConf.h (LCD_PHYS_COLORS) 进行编译时被定义。为了尽可能减少 RAM 的浪费，这个数据通常声明为常量，因此它能保存在 ROM 中。为了能修改它，需要将它调入 RAM 中。这可能通过激活配置开关 LCD_LUT_IN_RAM 来完成。如果这激活了，API 函数 GUI_SetLUTColor() 会有效，就能用于在同一时间修改颜色表和 LUT 的内容。

调用 LCD_InitLUT() 函数将会恢复原始(默认)设置。

9.6 颜色API

下表列出了与颜色处理相关的函数，在各自的类型中按字母顺序进行排列。函数的详细描述后面列出。

函 数	说 明
基本颜色函数	
GUI_GetBkColor()	返回当前背景颜色
GUI_GetBkColorIndex()	返回当前背景颜色的索引
GUI_GetColor()	返回当前前景颜色
GUI_GetColorIndex()	返回当前前景颜色的索引
GUI_SetBkColor()	设置当前背景颜色
GUI_SetBkColorIndex()	设置当前背景颜色索引
GUI_SetColor()	设置当前前景颜色
GUI_SetColorIndex()	设置当前前景颜色索引
索引色及全彩色转换	
GUI_Color2Index()	将全彩色转换成索引色
GUI_Index2Color()	将索引色转换成全彩色.
查询表 (LUT) 类	
GUI_InitLUT()	初始化 LUT (硬件)
GUI_SetLUTColor()	设置一种索引色的颜色 (硬件及颜色表)
GUI_SetLUTEntry()	向 LUT 写入一个数值 (硬件)

9.7 基本颜色函数

GUI_GetBkColor()

描述

返回当前背景颜色。

函数原型

```
GUI_COLOR GUI_GetBkColor(void);
```

返回值

当前的背景颜色色。

GUI_GetBkColorIndex()

描述

返回当前背景颜色的索引

函数原型

```
int GUI_GetBkColorIndex(void);
```

返回值

当前背景颜色的索引

GUI_GetColor()

描述

返回当前前景颜色。

函数原型

```
GUI_COLOR GUI_GetColor(void);
```

返回值

当前前景颜色。

GUI_GetColorIndex()**描述**

返回当前前景色索引

函数原型

```
int GUI_GetColorIndex(void);
```

返回值

当前前景色索引

GUI_SetBkColor()**描述**

设置当前背景颜色。

函数原型

```
GUI_COLOR GUI_SetBkColor(GUI_COLOR Color);
```

参 数	含 意
Color	背景颜色，24 位 RGB 数值。

返回值

选择的背景颜色

GUI_SetBkColorIndex()**描述**

设置当前背景色的索引值

函数原型

```
int GUI_SetBkColorIndex(int Index);
```

参 数	含 意
Index	要使用的颜色的索引值。

返回值

所选择背景颜色的索引值

GUI_SetColor()

描述

设置当前前景色

函数原型

```
void GUI_SetColor(GUI_COLOR Color);
```

参 数	含 意
Color	前景颜色，24 位 RGB 数值。

返回值

所选择的前景颜色

GUI_SetColorIndex()

描述

设置当前前景色的索引值

函数原型

```
void GUI_SetColorIndex(int Index);
```

参 数	含 意
Index	要使用的颜色的索引值。

返回值

所选择前景颜色的索引值

9.8 索引和彩色转换

GUI_Color2Index()

描述

返回一种指定的 RGB 颜色数值的索引值

函数原型

```
int GUI_Color2Index(GUI_COLOR Color)
```

参 数	含 意
Color	要转换的颜色的 RGB 值。

返回值

颜色的索引值

GUI_Index2Color()

描述

返回一种指定的索引颜色的 RGB 颜色值

函数原型

```
int GUI_Index2Color(int Index)
```

参 数	含 意
Index	要转换的颜色的索引值

返回值

RGB 颜色值

9.9 查询表（LUT）类

这些函数是可选择的，而且只能在 LCD 控制器硬件支持的情况下工作。要求一个带 LUT 硬件的 LCD 控制器。请参考你所使用的 LCD 控制器的手册以获得关于 LUT 的更多信息。

GUI_InitLUT()

描述

初始化 LCD 控制器的查询表

函数原型

```
void LCD_InitLUT(void) ;
```

附加信息

该函数需要激活查询表（通过宏 LCD_INITCONTROLLER）才有效果

GUI_SetLUTColor()

描述

修改颜色表和 LCD 控制器的 LUT 中的单个条目

函数原型

```
void GUI_SetLUTColor(U8 Pos, GUI_COLOR Color);
```

参 数	含 意
Pos	查询表中的位置。应该小于颜色数（例如，对于 2bpp 为 0~3，4bpp 为 0~15，8bpp 为 0~255）
Color	24 位 RGB 数值

附加信息

最接近的可能数值将用于 LUT。如果一个颜色 LUT 已经初始化，所有三个部分基色都会用到。在单色模式下，用到绿色部分，但是仍然推荐（为了更好的理解程序代码）将三种基色设置为同一个数值（例如 0x555555 或 0xa0a0a0）。

该函数需要激活查询表（通过宏 LCD_INITCONTROLLER）才有效果。该函数总是有效的，但是只有在下面两个条件下才有效果：

- 如果 LUT 已经使用
- 颜色表位次于 RAM 中（LCD_PHYSCOLORS_IN_RAM）

GUI_SetLUTEntry()

描述

修改 LCD 控制器的 LUT 的中的单个条目

函数原型

```
void GUI_SetLUTEntry(U8 Pos, GUI_COLOR Color);
```

参 数	含 意
Pos	查询表中的位置。应该小于颜色数（例如，对于 2bpp 为 0~3，4bpp 为 0~15，8bpp 为 0~255）
Color	24 位 RGB 数值

附加信息

最接近的可能数值将用于 LUT。如果一个颜色 LUT 已经初始化，所有三个部分基色都会用到。在单色模式下，用到绿色部分，但是仍然推荐（为了更好的理解程序代码）将三种基色设置为同一个数值（例如 0X555555 或 0Xa0a0a0）。

该函数需要激活查询表（通过宏 LCD_INITCONTROLLER）才有效果。该函数常常用于确保实际显示的颜色与逻辑颜色相匹配（线性化）。

范例

```
// 线性化一个 4 级灰度 LCD 的调色板
GUI_SetLUTEntry(0, 0x000000);
GUI_SetLUTEntry(1, 0x777777);
// GUI_SetLUTEntry(2, 0xbbbbbb)线性化结果是 555555;
// GUI_SetLUTEntry(3, 0xffffffff)线性化结果是 aaaaaa;
```

第10章 存储设备

存储设备可以用在多种情况下，主要防止显示屏在有对象重叠的绘图操作时的闪烁现象。基本的思路很简单。没有使用存储设备时，绘图操作直接写屏。屏幕在绘图操作在执行时更新，当不同的更新在执行时会产生闪烁。例如，如果你想绘一幅位图作为背景，以一些透明的文字作为前景，你首先必须绘位图，然后是文字，最终结果文字会是闪烁的。


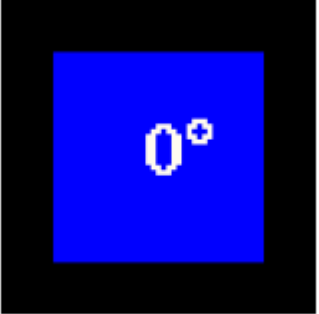
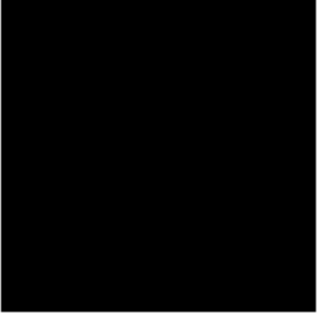
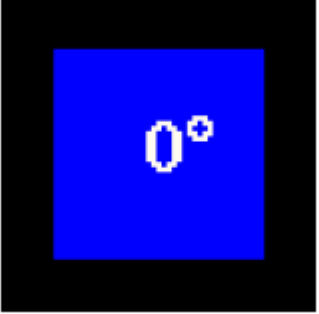
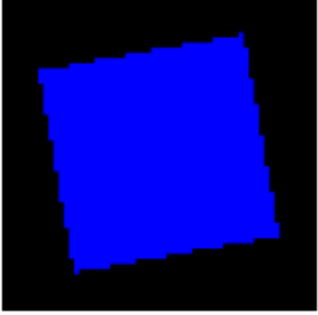
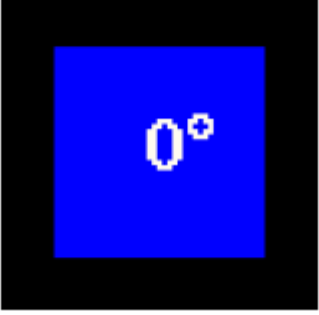
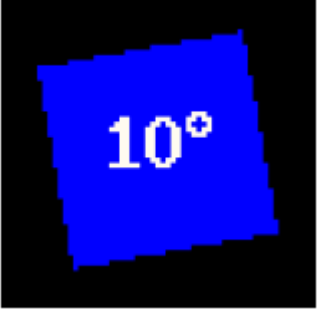
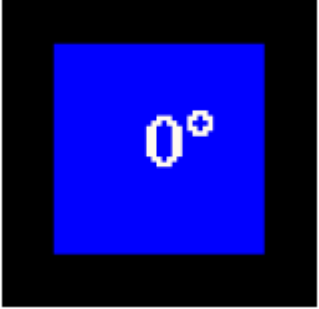
然而，如果这样的操作使用一个存储设备的话，所有的操作在存储设备内执行。只有在所有的操作执行完毕后最终结果才显示在屏幕上，具有无闪烁的优点。二者的不同会在下面的范例中列出，该范例图解一系列绘图操作在没有使用存储设备和使用存储设备的效果。

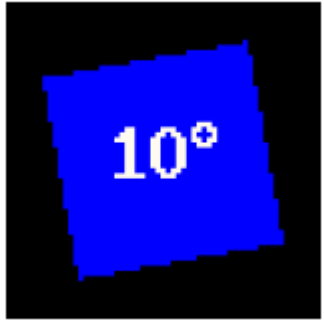
二者的区别总结如下：如果不使用存储设备，绘图的操作的效果看起来是一步一步的，带来闪烁的缺点。而使用存储设备，所有程序执行的效果看起来象单步操作，没有中间步骤可见。优势在于，如上说明，显示屏的闪烁完全消除，而这常常是希望看到的。

存储设备是一个附加（可选）的软件项目，不随 μ C/GUI 的基本软件包一起发布。存储设备的软件包位于子目录 GUI\Memdev 下。

10.1 图解存储设备的使用

下表是屏幕截图显示使用存储设备和不使用存储设备完成同样操作。两个例子的目的是是一样的：旋转一个工件，标注各自的旋转角度（在此处是 10 度）。在第一个例子（不使用存储设备），必须清屏，然后在新的位置重绘矩形和以新的标号写字符串。在第二个例子（使用一个存储设备），同样的操作在存储器执行，但是屏幕在这个时候并没有更新。唯一的更新出现在调用 GUI_MEMDEV_CopyToLCD 函数时，这样的更新立即反映了所有操作。注意两种处理方法的状态初始化和最后的输出是不同的。

API 函数	不使用存储设备	使用存储设备
步骤 1: 状态初始化		
步骤 2: GUI_Clear		
步骤 3: GUI_DrawPolygon		
步骤 4: GUI_DisString		

步骤 5: GUI_MEMDEV_CopyToLCD （只有在使用存储设备时有 效）		
---	--	---

10.2 基本函数

下面的函数是那些在使用存储设备时通常被调用的。基本用法相当简单：

1. 建立存储设备（使用 GUI_MEMDEV_Create）；
2. 激活它（使用 GUI_MEMDEV_Select）；
3. 执行绘图操作；
4. 将结果拷贝到显示屏（使用 GUI_MEMDEV_CopyToLCD）；
5. 如果你不再需要存储设备，删除它（使用 GUI_MEMDEV_Delete）。

10.3 为了能使用存储设备……

默认情形下，存储设备是被激活的。为了优化软件的性能，对存储设备的支持可以在配置文件 GUIConf.h 中加入下面一行而关闭：

```
#define GUI_SUPPORT_MEMDEV 0
```

如果这一行出现在配置文件里，而你又需要使用存储设备，可以删除这一行或将其定义改为 1。

10.4 存储设备 API 函数

下表列出了 μ C/GUI 的存储器 API 函数，所有函数在各自的类型中按字母顺序进行排列。函数的详细描述后面列出。

函 数	说 明
基本函数	
GUI_MEMDEV_Create ()	建立存储设备（第一步）
GUI_MEMDEV_CopyToLCD ()	将存储设备的内容拷贝到 LCD
GUI_MEMDEV_Delete ()	存储设备释放使用的存储空间

GUI_MEMDEV_Select ()	选择一个存储设备作为目标用于绘图操作
高级特性	
GUI_MEMDEV_Clear ()	将存储设备的内容标志为未改变 M
GUI_MEMDEV_CopyFromLCD ()	将 LCD 的内容拷贝到存储设备
GUI_MEMDEV_CopyToLCDAA ()	以反锯齿方式拷贝存储设备的内容
GUI_MEMDEV_GetYSize ()	返回存储设备的 Y 轴尺寸
GUI_MEMDEV_ReduceYSize ()	减少存储设备的 Y 轴尺寸
GUI_MEMDEV_SetOrg ()	在 LCD 上改变存储设备的原点
分片存储设备	
GUI_MEMDEV_Draw ()	使用一个存储设备进行绘图
自动设备对象函数	
GUI_MEMDEV_CreateAuto ()	建立一个自动设备对象。
GUI_MEMDEV_DeleteAuto ()	删除一个自动设备对象。
GUI_MEMDEV_DrawAuto ()	使用一个 GUI_AUTODEV 对象进行绘图

GUI_MEMDEV_Create()

描述

建立一个存储设备

函数原型

```
GUI_MEMDEV_Handle GUI_MEMDEV_Create(int x0, int y0, int XSize, int YSize)
```

参 数	含 意
x0	存储设备的 X 轴坐标
y0	存储设备的 Y 轴坐标
XSize	存储设备的 X 轴尺寸
YSize	存储设备的 Y 轴尺寸

返回数值

建立的存储设备的句柄，如果函数执行失败，返回值为 0。

GUI_MEMDEV_CopyToLCD()

描述

将一个存储设备的内容从内存拷贝到 LCD。

函数原型

```
void GUI_MEMDEV_CopyToLCD(GUI_MEMDEV_Handle hMem)
```

参 数	含 意
hMem	存储设备的句柄

GUI_MEMDEV_Delete()**描述**

删除一个存储设备。

函数原型

```
void GUI_MEMDEV_Delete(GUI_MEMDEV_Handle hMem);
```

参 数	含 意
hMem	存储设备的句柄

返回值

删除的存储设备的句柄

GUI_MEMDEV_Select()**描述**

激活一个存储设备(或如果句柄为 0 则激活 LCD)

函数原型

```
void GUI_MEMDEV_Select(GUI_MEMDEV_Handle hMem)
```

参 数	含 意
hMem	存储设备的句柄

10.5 高级特性

GUI_MEMDEV_Clear()

描述

将存储设备的所有内容标志为“未改变的”。

函数原型

```
void GUI_MEMDEV_Clear(GUI_MEMDEV_Handle hMem);
```

参 数	含 意
hMem	存储设备的句柄

附加信息

使用 GUI_MEMDEV_CopyToLCD 的下一步绘图操作是，只有在 GUI_MEMDEV_Clear 和 GUI_MEMDEV_CopyToLCD 之间有字节改变的情况才进行写操作。

GUI_MEMDEV_CopyFromLCD()

描述

从 LCD 数据（视频存储器）拷贝一个存储设备的内容到存储设备。换句话说，回读 LCD 的内容到存储设备。

函数原型

```
void GUI_MEMDEV_CopyFromLCD(GUI_MEMDEV_Handle hMem);
```

参 数	含 意
hMem	存储设备的句柄

GUI_MEMDEV_CopyToLCDAA()

描述

从存储区域拷贝存储设备的内容（反锯齿）到 LCD。

函数原型

```
void GUI_MEMDEV_CopyToLCDAA(GUI_MEMDEV_Handle hMem);
```

参 数	含 意
hMem	存储设备的句柄

附加信息

器件的数据处理为反锯齿数据。一个 2×2 像素矩阵转换 1 个像素。最终结果的像素强度取决于在多少像素在矩阵中调整。

范例

建立一个存储设备并选择它作为输出。设置一个大的字体，然后向存储设备写入一个文本：

```
GUI_MEMDEV_Handle hMem = GUI_MEMDEV_Create(0, 0, 60, 32);
GUI_MEMDEV_Select(hMem);
GUI_SetFont(&GUI_Font32B_ASCII);
GUI_DispString("Text");
GUI_MEMDEV_CopyToLCDAA(hMem);
```

范例执行的屏幕截图



GUI_MEMDEV_GetYSize()

描述

返回一个存储设备的 Y 轴尺寸。

函数原型

```
int GUI_MEMDEV_GetYSize(GUI_MEMDEV_Handle hMem);
```

参 数	含 意
hMem	存储设备的句柄

GUI_MEMDEV_ReduceYSize()

描述

减小一个存储设备的 Y 轴尺寸。

函数原型

```
void GUI_MEMDEV_ReduceYSize(GUI_MEMDEV_Handle hMem, int YSize);
```

参 数	含 意
<code>hMem</code>	存储设备的句柄
<code>YSize</code>	存储设备新的 Y 轴尺寸

附加信息

改变存储设备的尺寸比删除然后重建它更有效。

GUI_MEMDEV_SetOrg

描述

改变存储设备在 LCD 上的原点。

函数原型

```
void GUI_MEMDEV_SetOrg(GUI_MEMDEV_Handle hMem, int x0, int y0);
```

参 数	含 意
<code>hMem</code>	存储设备的句柄
<code>x0</code>	水平坐标（左上角像素）
<code>y0</code>	垂直坐标（左上角像素）

附加信息

这个函数在同一个器件用于不同的屏幕区域或存储设备的内容被拷贝到不同区域时非常有用。

修改存储设备的原点比删除然后重建它更有效。

使用存储设备的简单例子

该范例展示一个基本存储设备的使用。向存储设备写入一个文本，然后再拷贝到 LCD。该范例见 Source\Misc\MemDev.c 文件。

```

/*-----
文件:      MemDev.c
目的:      展示如何使用存储设备的简单例子
-----*/

#include "GUI.H"

/*****
*                      展示存储设备的使用                      *
*****/

static void DemoMemDev(void)
{
    GUI_MEMDEV_Handle hMem;
    while(1)
    {
        /* 建立存储设备..... */
        hMem = GUI_MEMDEV_Create(0, 0, 110, 18);
        /* .....然后选择其用于绘图操作 */
        GUI_MEMDEV_Select(hMem);
        /* 向存储设备绘一个文本*/
        GUI_SetFont(&GUI_FontComic18B_ASCII);
        GUI_DispStringAt("Memory device", 0, 0);
        /* 将存储设备的内容拷贝到 LCD */
        GUI_MEMDEV_CopyToLCD(hMem);
        /* 选择 LCD 并清除存储设备 */
        GUI_MEMDEV_Select(0);
        GUI_MEMDEV_Delete(hMem);
        GUI_Delay(1000);
        GUI_Clear();
        GUI_Delay(500);
    }
}

```

```

/*****
*                               *
*****

```

```

void main(void)
{
    GUI_Init();
    DemoMemDev();
}

```

10.6 分片存储设备

一个存储设备首先通过执行指定的绘图函数进行内容填充。设备填充完毕后，其内容写入 LCD。有些情况下，可能会没有足够的有效存储器空间能够立刻用于所有输出区域的存储，这依赖于你的配置（参考第 21 章“高层次配置”的 GUI_ALLOC_SIZE 配置宏）。一个分片存储设备将绘制区域分成几个片段，在每一片段里面用尽可能多占用当前可用的存储空间。

GUI_MEMDEV_Draw()

描述

防止显示屏闪烁的基本函数

函数原型

```

int GUI_MEMDEV_Draw (    GUI_RECT* pRect,
                          GUI_CALLBACK_VOID_P* pfDraw,
                          void* pData,
                          int NumLines,
                          int Flags)

```

参 数	含 意
pRect	所使用的 LCD 的一个 GUI_RECT 结构的指针。
pfDraw	执行绘图操作的一个回调函数的指针。
pData	作为回调函数参数使用的一个数据结构的指针。
NumLines	0（推荐）或者是存储设备的分段数量。
Flags	0 或者 GUI_MEMDEV_HASTRANS。

返回数值

如果成功返回 0，如果函数执行失败则返回 1。

附加信息

如果参数 NumLines 为 0，则每段中的线段数量由函数自动计算。通过移动存储设备的原点，函数一段又一段地重复改写输出区域。

使用分片存储设备的范例

下面的范例展示了一个分片存储设备的使用。其源文件是 Source\Misc\BandingMemdev.c。

```

/*-----
文件:      BandingMemdev.c
目的:      展示如何使用分片存储设备的例子
-----*/

#include "GUI.H"
static const GUI_POINT aPoints[] =
{
    {-50, 0}, {-10, 10}, {0, 50}, {10, 10}, {50, 0}, {10, -10}, {0, -50}, {-10, -10}
};
#define SIZE_OF_ARRAY (Array) (sizeof(Array)/sizeof(Array[0]))
typedef struct
{
    int XPos_Poly, YPos_Poly; int XPos_Text, YPos_Text; GUI_POINT aPointsDest[8];
} tDrawItContext;

/******
*                               *
*****/

static void DrawIt(void * pData)
{
    tDrawItContext * pDrawItContext = (tDrawItContext *)pData;
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x8);

```

```

GUI_SetTextMode(GUI_TM_TRANS);

/* 绘背景 */
GUI_SetColor(GUI_GREEN);
GUI_FillRect ( pDrawItContext->XPos_Text,
               pDrawItContext->YPos_Text - 25,
               pDrawItContext->XPos_Text + 100,
               pDrawItContext->YPos_Text - 5);

/* 绘多边形 */
GUI_SetColor(GUI_BLUE);
GUI_FillPolygon (pDrawItContext->aPointsDest,
                 SIZE_OF_ARRAY(aPoints),
                 160, 120);

/* 绘前景 */
GUI_SetColor(GUI_RED);
GUI_FillRect( 220 - pDrawItContext->XPos_Text,
              pDrawItContext->YPos_Text + 5,
              220 - pDrawItContext->XPos_Text + 100,
              pDrawItContext->YPos_Text + 25);
}

/*****
*                                     *
*                               展示分片存储设备                               *
*                                     *
*****/

#define USE_BANDING_MEMDEV (1)

/* 如设为 0，则定义不使用分片存储设备进行绘图 */
void DemoBandingMemdev(void)
{
    int i;
    int XSize = LCD_GET_XSIZE();
    int YSize = LCD_GET_YSIZE();
    tDrawItContext DrawItContext;
    GUI_SetFont(&GUI_Font8x9);
    GUI_SetColor(GUI_WHITE);
    GUI_DispStringHCenterAt ( "Banding memory device\nwithout flickering",
                             XSize/40)

    DrawItContext.XPos_Poly = XSize/2;
    DrawItContext.YPos_Poly = YSize/2;

```

```

    DrawItContext.YPos_Text = Ysize/2-4;
    for(i = 0; i < (XSize - 100); i++)
        float angle = i * 3.1415926 / 60;
    DrawItContext.XPos_Text = i;
    /* 旋转多边形 */
    GUI_RotatePolygon ( DrawItContext.aPointsDest,
                        aPoints,
                        SIZE_OF_ARRAY(aPoints),
                        angle);
    #if USE_BANDING_MEMDEV
    {
        GUI_RECT Rect = { 0, 70, 320, 170} ;
        /* 使用分片存储设备进行绘图 */
        GUI_MEMDEV_Draw(&Rect, &DrawIt, &DrawItContext, 0, 0);
    }
    #else
    /* 不使用存储设备的简单绘图 */
    DrawIt((void *)&DrawItContext);
    #endif
    #ifdef WIN32
        GUI_Delay(20); /* 只有在仿真时才使用一段短延时 */
    #endif
}

/* *****
*                                     主函数
* ***** */

void main (void)
{
    GUI_Init () ;
    while(1)
    {
        DemoBandingMemdev();
    }
}

```

范例执行结果的屏幕截图



10.7 自动设备对象

当显示屏必须更新以反映其对象的移动或改变时，存储设备非常有用，因此在防止 LCD 闪烁这样一个应用方面是很重要的。一个自动设备对象是基于分片存储设备建立的，它可以在某些应用方面更有效，例如移动标志，这种情况下，在一段时间内显示屏只有一小部分要更新。

该设备能自动识别显示屏的哪一部分包含固定的对象，哪一部分包含移动或改变的对象（必须更新）。当绘图函数第一次被调用时，所有的对象都被绘制出。而以后的函数调用只更新需要移动或改变的物体。实际的绘图操作使用分片存储设备机制，但只在需要的的空间内使用。使用一个自动存储设备（与直接使用分片存储设备相比）的主要优点是节省了计算时间，因为它需要更新整个显示屏。

GUI_MEMDEV_CreateAuto()

描述

建立一个自动设备对象。

函数原型

```
int GUI_MEMDEV_CreateAuto(GUI_AUTODEV * pAutoDev);
```

参 数	含 意
<code>pAutoDev</code>	一个 GUI_AUTODEV 对象的指针

返回数值

通常是 0，保留供以后使用。

GUI_MEMDEV_DeleteAuto()**描述**

删除一个自动设备对象。

函数原型

```
void GUI_MEMDEV_DeleteAuto(GUI_AUTODEV * pAutoDev);
```

参 数	含 意
<code>pAutoDev</code>	一个 GUI_AUTODEV 对象的指针

GUI_MEMDEV_DrawAuto()**描述**

使用一个分片存储设备执行一个指定的绘图函数。

函数原型

```
int GUI_MEMDEV_DrawAuto ( GUI_AUTODEV * pAutoDev,
                           GUI_AUTODEV_INFO * pAutoDevInfo,
                           GUI_CALLBACK_VOID_P * pfDraw, void * pData);
```

参 数	含 意
<code>pAutoDev</code>	一个 GUI_AUTODEV 对象的指针。
<code>pAutoDevInfo</code>	一个 GUI_AUTODEV_INFO 对象的指针。
<code>pfDraw</code>	用户定义要执行的绘图函数的指针。
<code>pData</code>	一个由绘图函数传递的数据结构的指针。

返回数值

如果成功返回 0，函数执行失败返回 1

附加信息

GUI_AUTODEV_INFO 结构包含有哪些对象必须要由用户函数绘制的信息:

```
typedef struct
{
    char DrawFixed;
} GUI_AUTODEV_INFO;
```

如果所有的对象都要绘制, DrawFixed 设为 1。当只有被移动或改变的物体才需要绘制的时候, 设为 0。当使用这个特性时, 我们推荐使用下面的程序:

```
typedef struct
{
    GUI_AUTODEV_INFO AutoDevInfo;      /* 哪些内容需要绘制的信息 */
    /* 给用户函数添加使用的数据 */
    .....
} PARAM;

static void Draw(void * p)
{
    PARAM * pParam = (PARAM *)p;
    if (pParam->AutoDevInfo.DrawFixed)
    {
        /* 绘固定的背景 */
        .....
    }
    /* 绘制移动的物体 */
    .....
    if (pParam->AutoDevInfo.DrawFixed)
    {
        /* 绘固定的前景（如果需要） */
        .....
    }
}

void main (void)
{
```

```

PARAM Param;                                /* 绘图函数的参数*/
GUI_AUTODEV AutoDev;                        /* 分片存储设备的对象 */
/* 绘图函数的 设置/修改 信息 */
.....
GUI_MEMDEV_CreateAuto(&AutoDev);           /* 建立 GUI_AUTODEV 对象 */
GUI_MEMDEV_DrawAuto (   &AutoDev,          /* 使用 GUI_AUTODEV 对象用于绘图 */
                        &Param.AutoDevInfo,
                        &Draw, &Param);
GUI_MEMDEV_DeleteAuto(&AutoDev);           /* 删除 GUI_AUTODEV 对象 */
}

```

使用一个自动设备对象的范例

下面的范例展示了一自动设备对象的使用。其源文件为：Source\Misc\AutoDev.c。在背景上绘一个带有可转动指针的刻度盘，在前景上绘一段小的文字。指针使用 μ C/GUI 的抗锯齿特性绘制。在这里使用高分辨率抗锯齿以增强转动的指针的外观效果。对于抗锯齿的更多信息，请参阅第 15 章：抗锯齿。

```

/*-----
文件:      AutoDev.c
目的:      展示 GUI_AUTODEV 对象用法的例子
-----*/

#include "GUI.H"
#include <math.h>
#include <stddef.h>
#ifndef WIN32
    #include "rtos.h"
#endif
#define countof (Obj) (sizeof(Obj)/sizeof (Obj[0]))
#define DEG2RAD (3.1415926f/180)

/*-----
*                               缩放位图                               *
*-----*/

static const GUI_COLOR ColorsScaleR140[] =
{
    0x000000, 0x00AA00, 0xFFFFF, 0x0000AA, 0x00FF00, 0xAEAEAE, 0x737373,

```

```

    0xD3D3D3, 0xDFDFDF, 0xBBDFBB, 0x6161DF, 0x61DF61, 0BBBBBDF, 0xC7C7C7,
    0x616193
};

static const GUI_LOGPALETTE PalScaleR140 =
{
    15,                      /* number of entries */
    0,                      /* 不透明 */
    &ColorsScaleR140[0]
};

static const unsigned char acScaleR140[] =
{
    /* ..... 像素数据不在菜单中显示。 请参考范例源文件..... */
};

static const GUI_BITMAP bmScaleR140 =
{
    200,                    /* X 轴尺寸 */
    73,                    /* Y 轴尺寸 */
    100,                   /* 每行字节数 */
    4,                     /* 每像素位数 */
    acScaleR140,           /* 图片数据的指针 (像素) */
    &PalScaleR140          /* 调色板的指针 */
};

/*****
*                               *
******/

#define MAG 3
static const GUI_POINT aNeedle[] =
{
    {MAG * (0) , MAG * (0 + 125)},
    {MAG * (-3), MAG * (-15 + 125)},
    {MAG * (-3), MAG * (-65 + 125)},
    {MAG * (3), MAG * (-65 + 125)},
    {MAG * (3), MAG * (-15 + 125)},
};

```

```

/*****
*
*                               包括绘图函数信息的结构
*
*****/

typedef struct
{
    /* Information about what has to be displayed */
    GUI_AUTODEV_INFO AutoDevInfo;
    /* 多边形数据 */
    GUI_POINT aPoints[7] ;
    float Angle;
} PARAM;

/*****
*
*                               获得角度
*
*****/

/* 这个函数返回数值进行显示。在实际应用中，该数值以某种方式进行测量 */
static float GetAngle(int tDiff)
{
    if (tDiff < 15000)
    {
        return 225 - 0.006 * tDiff ;
    }
    tDiff -= 15000;
    if (tDiff < 7500)
    {
        return 225 - 90 + 0.012 * tDiff ;
    }
    tDiff -= 7000;
    return 225;
}

/*****
*
*                               绘图函数
*
*****/

```

```

static void Draw(void * p)
{
    PARAM * pParam = (PARAM *)p;
    /* 固定前景 Fixed background */
    if (pParam->AutoDevInfo.DrawFixed)
    {
        GUI_ClearRect ( 60,
                        50 + bmScaleR140.YSize,
                        60 + bmScaleR140.XSize - 1,
                        150);

        GUI_DrawBitmap(&bmScaleR140, 60, 50);
    }
    /* 绘指针 */
    GUI_SetColor(GUI_WHITE);
    GUI_AA_FillPolygon(pParam->aPoints, countof(aNeedle), MAG*160, MAG*190);
    /* 固定前景 Fixed foreground */
    if (pParam->AutoDevInfo.DrawFixed)
    {
        GUI_SetTextMode(GUI_TM_TRANS);
        GUI_SetColor(GUI_RED);
        GUI_SetFont (&GUI_Font24B_ASCII);
        GUI_DispStringHCenterAt("RPM/1000", 160, 110);
    }
}

/*****
*                               *
*               使用分片存储设备显示一个带指针的刻度盘               *
*                               *
*****/

static void DemoScale(void)
{
    int Cnt;
    int tDiff, t0 = GUI_GetTime();
    PARAM Param;                /* 绘图函数的参数 */
    GUI_AUTODEV AutoDev;        /* 分片存储设备对象 */
    /* 显示消息 */
    GUI_SetColor(GUI_WHITE);

```

```

GUI_SetFont(&GUI_Font8x16);
GUI_DispStringHCenterAt("Scale using GUI_AUTODEV-object", 160, 0);
/* 启动高分辨率用于抗锯齿 */
GUI_AA_EnableHiRes();
GUI_AA_SetFactor(MAG);
/* 建立 GUI_AUTODEV 对象 */
GUI_MEMDEV_CreateAuto(&AutoDev);
/* 显示在一个固定时间上的指针 */
for (Cnt = 0; (tDiff = GUI_GetTime() - t0) < 24000; Cnt++)
{
    /* 获得数值用于显示一个多边形来表示指针 */
    Param.Angle = GetAngle(tDiff)* DEG2RAD;
    GUI_RotatePolygon ( Param.aPoints,
                        aNeedle,
                        countof(aNeedle),
                        Param.Angle);
    GUI_MEMDEV_DrawAuto(&AutoDev, &Param.AutoDevInfo, &Draw, &Param);
}
/* 显示 "milliseconds / picture" */
GUI_SetColor(GUI_WHITE);
GUI_SetFont(&GUI_Font8x16);
GUI_DispStringHCenterAt("Milliseconds / picture:", 160, 180);
GUI_SetTextAlign(GUI_TA_CENTER);
GUI_SetTextMode(GUI_TM_NORMAL);
GUI_DispNextLine();
GUI_GotoX(160);
GUI_DispFloatMin((float)tDiff/(float)Cnt, 2);
/* 删除 GUI_AUTODEV 对象 */
GUI_MEMDEV_DeleteAuto(&AutoDev);
}

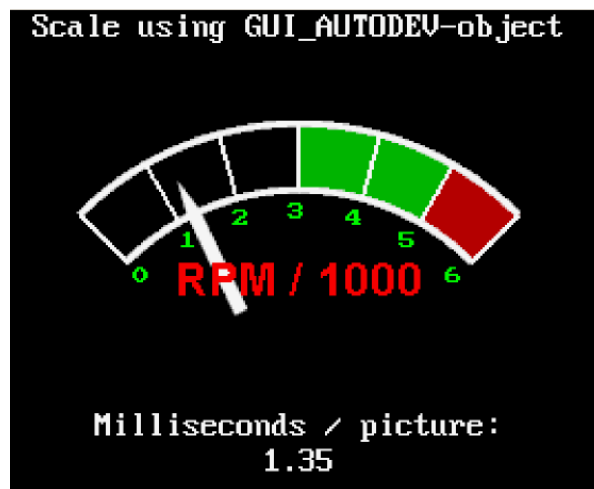
/*****
*                               主函数                               *
*****/

void main(void)
{

```

```
#ifndef WIN32
    OS_InitKern();
    OS_InitHW();
#endif
GUI_Init();
while(1)
    DemoScale();
}
```

范例程序执行结果的屏幕截图



第11章 运行模型：单任务/多任务

μ C/GUI 从一开始就被设计成够适应不同的环境类型。它能工作在单任务和多任务应用中，包括私有的操作系统或任何商业 RTOS，例如 embOS 或 μ C/OS。

11.1 支持的运行模型

我们主要区别一下三种不同的执行模型：

单任务系统（超级循环）

所有的程序运行在一个超级循环中。通常情况下，所有软件单元被有秩序地调用。既然没用到实时内核，软件的实时功能必须要依靠中断来完成。

μ C/GUI 多任务系统：一个任务调用 μ C/GUI

使用了一个实时内核（RTOS），但是其中只有一个任务调用 μ C/GUI 函数。从图形软件的观点来看，它和单任务系统中的使用是一样的。

μ C/GUI 多任务系统：多个任务调用 μ C/GUI

使用了一个实时内核（RTOS），其中多个任务调用 μ C/GUI 函数。这样的工作避免了软件造成的线程保护问题，该项工作通过在配置中使能多任务支持及配合内核接口函数来完成。对于普遍的的内核，内核接口函数易于使用。

11.2 单任务系统（超级循环）

描述

所有的程序运行在一个超级循环中。通常情况下，所有软件单元被有秩序地调用。没有使用实时内核，因此软件的实时功能必须要依靠中断来完成。这类系统起初用于更小的系统或如果实时行为并不是很必须的情况下。

超级循环范例（没有使用 μ C/GUI）

```
void main(void)
{
    HARDWARE_Init();
    /* 初始化软件单元 */
    XXX_Init( ) ;
    YYY_Init( ) ;
    /* 超级循环：有秩序地调用所有软件单元 */
}
```

```
while(1)
{
    /* 执行所有的软件单元 */
    XXX_Exec();
    YYY_Exec();
}
}
```

优点

没有使用实时内核（占用更小的 ROM 空间，只有一个任务，用于堆栈的 RAM 单元也很少），不存在 优先权/同步 问题。

缺点

超级循环程序如果超过一定的大小，可能变得很难维护。实时行为极有限，因为一个软件单元不能被其它的单元所打断（只有通过中断）。这意思是一个软件单元的反应时间依赖于这个系统中所有其它单元的执行时间。

使用 μ C/GUI

关于 μ C/GUI 的使用，没有真的约束。如通常情况一样，GUI_Init() 在你使用这个软件前必须要先调用，在这基础上，可以使用任何 API 函数。如果用到视察管理器的回调机制，一个 μ C/GUI 更新函数必须定期被调用。在从超级循环中调用 GUI_Exec() 是一种典型做法。单元化函数例如 GUI_Delay() 和 GUI_ExecDialog()，不应在循环中使用，因为它们会妨碍其它软件模块。

使用默认配置，它并不支持多任务系统使用（#define GUI_MT 0）；不需要内核接口函数。

超级循环范例（使用 μ C/GUI）：

```
void main(void)
{
    HARDWARE_Init();
    /* 初始化软件单元 */
    XXX_Init();
    YYY_Init();
}
```

```
GUI_Init();          /* 初始化  $\mu$ C/GUI */  
/* 超级循环：有秩序地调用所有的软件单元*/  
while(1)  
{  
    /* 运行所有的软件单元 */  
    XXX_Exec();  
    YYY_Exec();  
    GUI_Exec();       /* 功能性运行  $\mu$ C/GUI 如更新窗口*/  
}  
}
```

11.3 μ C/GUI 多任务系统：一个任务调用 μ C/GUI

描述

使用一个实时内核（RTOS）。用户程序被分割成不同的部分，运行在不同的任务中，具有典型不同的优先级别。通常情况下，实时的临界任务（要求某一个反应时间）具有最高级别。一个单个任务调用 μ C/GUI 函数，用于用户界面。这个任务通常在系统中的任务级别最低，至少是最低的级别之一（一些统计任务或简单的空闲处理的任务级别甚至可能更低）。中断可以用于软件的实时部分，但不是必须的。

优点

系统实时行为极佳。任务的实时行为只受运行在更高级别任务影响。这意思是换一个运行在低级别任务的程序单元的一点也不会影响实时行为。如果用户界面在一个低级别任务中运行，这意味着换到用户界面不会影响实时行为。这种系统使分配不同的软件单元到不同的开发组变得很容易，这能给予彼此工作非常高的独立性。

缺点

你需要一个实时内核（RTOS），这需要花费金钱，耗尽 ROM 和 RAM（用于堆栈）。另外，你要考虑到任务同步和如何从一个任务传输信息到另一个任务。

使用 μ C/GUI

如果用到视察管理器的回调机制，一个 μ C/GUI 更新函数（典型是 GUI_Exec()，GUI_Delay()）必须定期地从调用 μ C/GUI 的任务中调用。因为 μ C/GUI 只被一个任务调用，

对于 μ C/GUI 来说，这和单任务系统中使用是一样的。

使用默认配置，它并不支持多任务系统的使用（`#define GUI_MT 0`）；不需要内核接口函数。你可以使用任何实时内核，包括商业的或私有的。

11.4 μ C/GUI 多任务系统：多个任务调用 μ C/GUI

描述

使用一个实时内核。用户程序被分割成不同的部分，运行在不同的任务中，具有典型不同的优先级别。通常情况下，实时的临界任务（要求某一个反应时间）具有最高级别。多个任务用于用户界面，调用 μ C/GUI 函数。这些任务在系统中具有典型的低级别，因此它们不会影响系统的实时行为。

中断可以用于软件的实时部分，但不是必须的。

优点

系统实时行为极佳。任务的实时行为只受运行在更高级别任务影响。这意思是换一个运行在低级别任务的程序单元的一点也不会影响实时行为。如果用户界面在一个低级别任务中运行，这意味者换到用户界面不会影响实时行为。这种系统使分配不同的软件单元到不同的开发组变得很容易，这能给予彼此工作非常高的独立性。

缺点

你需要一个实时内核（RTOS），这需要花费金钱，耗尽 ROM 和 RAM（用于堆栈）。另外，你要考虑到任务同步和如何从一个任务传输信息到另一个任务。

使用 μ C/GUI

如果用到视察管理器的回调机制，一个 μ C/GUI 更新函数（典型是 `GUI_Exec()`，`GUI_Delay()`）必须定期地从一个或多个调用 μ C/GUI 的任务中调用。

默认配置并不支持多任务系统的使用（`#define GUI_MT 0`），在此不能使用。在配置中，需要启用多任务支持，定义调用 μ C/GUI 的任务的最大数量（引用自 `GUIConf.h`）：

```
#define GUI_MT 1           // 启用多任务支持
#define GUI_MAX_TASK 5     // 能调用 $\mu$ C/GUI 的任务的最大数量
```

需要用到内核接口函数，需要与使用的内核相匹配。你可以使用任何实时内核，包括商业的或私有的。在下面的章节，会对宏和函数二者进行论述。

建议

(1) 仅仅从一个任务调用 μ C/GUI 更新函数（即 GUI_Exec(), GUI_Delay()）。这对保持程序结构清晰有帮助。如果你的系统有足够的 RAM，专门使用一个任务（最低级别）更新 μ C/GUI。该任务将不断地调用 GUI_Exec(), 不做其它事情，就象下面例子显示的一样。

(2) 保持你的实时任务（决定你的系统行为，关于 I/O，接口，网络等等）与调用 μ C/GUI 的任务分开。这对保证获得最佳的实时性能有很大帮助。

(3) 如果可能的话，你的用户界面只使用一个任务。这利于保持程序结构简洁，易于调试（但是，这不是必需的，在一些系统也可能是不合适的。）

范例

该引用显示专门的 dedicated μ C/GUI 更新函数。它从范例 MT_Multitasking 中拿来，这个范例包括在随 μ C/GUI 发布的范例当中：

```

/*****
*                               GUI 背景处理                               *
*****/

/* 该任务进行背景处理。主要工作是更新有效窗口，其它的事情，例如测试鼠标或
* 触摸屏输入也可以处理 */

void GUI_Task(void)
{
    while(1)
    {
        GUI_Exec();           /* 做背景工作……更新窗口等等 */
        GUI_X_ExecIdle();     /* 剩下暂时不做什么事情……空闲处理 */
    }
}

```

11.5 多任务支持的 GUI 配置宏

下表显示了用于一个多个任务调用 μ C/GUI 多任务系统的配置宏：

类型	宏	默认值	说明
N	GUI_MAXTASK	4	当多任务支持启用时（如下），定义调用 μ C/GUI 最大任务数量。
B	GUI_OS	0	激活多任务支持的启用。

GUI_MAXTASK

描述

定义调用 μ C/GUI 访问显示屏的最大任务的数量。

类型

数值

附加信息

该功能只有在 GUI_OS 激活情况下才相应有效。

GUI_OS

描述

通过激活 GUITask 组件启用多任务支持。

类型

二进制开关

0: 停止，多任务支持禁止（默认值）

1: 激活，多任务支持启用

11.6 内核接口函数 API

一个 RTOS 通常提供一个机制，称为资源旗语。在它的里面，使用一个特定资源的一个

任务在实际使用这个资源之前要声明这个资源。显示屏是一个需要和资源旗语一起被保护的资源的例子。 μ C/GUI 使用宏 `GUI_USE` 在访问显示屏之前或使用一个临界内部数据之前调用函数 `GUI_Use()`。类似的方法，它在访问显示屏之后或使用一个临界内部数据之后调用函数 `GUI_Unuse()`。这在模块 `GUITask.c` 中实现。

`GUITask.c` 依次使用下表所示的 GUI 内核接口函数。这些函数的前缀为 `GUI_X_`，因为它们是高层函数（与硬件相关）。它们必须与所使用的实时内核相匹配，以构造 μ C/GUI 任务（或线程）保护。详细的函数描述在后面，还有范例说明如何与不同的内核匹配。

函数	说明
GUI_X_InitOS()	初始化内核接口模块（建立一个资源“旗语/互斥”）。
GUI_X_GetTaskId()	返回一个当前任务（线程）的唯一的 32 位标识符。
GUI_X_Lock()	锁定 GUI（阻塞资源“旗语/互斥”）。
GUI_X_Unlock()	解锁 GUI（解锁资源“旗语/互斥”）。

GUI_X_InitOS()

描述

建立资源旗语或互斥(体)，最典型是由 `GUI_X_Lock()` 和 `GUI_X_Unlock()` 使用。

函数原型

```
void GUI_X_InitOS(void)
```

GUI_X_GetTaskID()

描述

返回当前任务的唯一 ID。

函数原型

```
U32 GUI_X_GetTaskID(void);
```

返回值

当前任务的 ID 是一个 32 位整数。

附加信息

与实时操作系统一起使用。

返回哪个数值没有关系，只要对于每个使用 μ C/GUI API 的任务/线程来说，它是唯一的，以及只要对于每个特定的线程来说，该数值总是相同的。

GUI_X_Lock()

描述

锁定 GUI.

函数原型

```
void GUI_X_Lock(void);
```

附加信息

在访问显示屏之前或在使用临界内部数据结构之前，该函数被 GUI 所调用。它从插入相同的临界区处阻塞其它线程，直到调用 GUI_X_Unlock() 函数，而该插入使用一个资源“旗语/互斥”完成。当使用一个实时操作系统时，你通常必需增加一个计算资源旗语。

GUI_X_Unlock()

描述

解锁 GUI.

函数原型

```
void GUI_X_Unlock(void);
```

附加信息

这个函数在访问显示屏后或使用一个临界内部数据以后被 GUI 调用。

当使用一个实时操作系统时，你通常必须消耗一个计算资源旗语。

范例

用于 μ C/OS-II 的内核接口函数

下面范例展示与 uC/OS-II 相匹配的四个函数的（引用自文件 GUI_X_uCOS-II.c）：

```
#include "INCLUDES.H"
static OS_EVENT * DispSem;

U32 GUI_X_GetTaskId(void)
{
    return((U32) (OSTCBCur->OSTCBPrio));
}

void GUI_X_InitOS(void)
{
    DispSem = OSemCreate(1);
}

void GUI_X_Unlock(void)
{
    OSemPost(DispSem);
}

void GUI_X_Lock(void)
{
    INT8U err;
    OSemPend(DispSem, 0, &err);
}
```

Win32 的内核接口函数

以下内容引用自对 μ C/GUI 的 Win32 仿真。（当使用 μ C/GUI 仿真时，没有必要加入这些函数，因为它已经放在库里。）

注意：为了清晰起见，清除代码被省略了。

```
/*
 *           $\mu$ C/GUI 多任务接口，用于 Win32
 */
*/
```

以下部分由 4 个 Win32 用于构造 μ C/GUI 线程保护的函数组成：

```
static HANDLE hMutex;
void GUI_X_InitOS(void)
{

```

```
    hMutex = CreateMutex(NULL, 0, "μC/GUISim - Mutex");  
}  
unsigned int GUI_X_GetTaskId(void)  
{  
    return GetCurrentThreadId();  
}  
void GUI_X_Lock(void)  
{  
    WaitForSingleObject(hMutex, INFINITE);  
}  
void GUI_X_Unlock(void)  
{  
    ReleaseMutex(hMutex);  
}
```

第12章 视窗管理器（WM）

使用 μ C/GUI 管理器（WM）时，在显示屏上显示的的所有内容包括在一个窗口里面——屏幕上的一块区域，该区域作为一个绘制或显示对象的用户接口部件。窗口可以是任意大小，你可能在屏幕上同时显示多个窗口，甚至在其它窗口的上面部分或完全地显示。

视窗管理器提供了一套函数，使你能很容易地对许多窗口进行创建，移动，调整大小及其它操作。它也可以提供更低层的支持，这通过管理显示屏上的窗口的层，及通过给你的应用程序发送信号以显示影响它的窗口的修改来完成。

μ C/GUI 的视窗管理器是一个独立的（可选的）的软件项目，它没有包括进 μ C/GUI 基本软件包里。视窗管理器的软件位于子目录“GUI\WM”下。

12.1 术语解释

窗口外形是矩形，由它们的原点（左上角的 X 和 Y 坐标）及它们的 X 和 Y 尺寸（分别是宽和高）所定义。 μ C/GUI 中一个窗口：

- 是一个矩形
- 有一个 Z 坐标
- 可能是隐藏的或可见的
- 可能拥有有效/或无效区域
- 可以或者不可以有透明区域
- 可以或者不可以有一个回调函数

活动窗口

当前正在使用进行绘图操作的窗口被当作活动窗口。与最顶层窗口一样，它不是必需的。

回调函数

回调函数在用户程序中定义，当一个指定的事件发生时，通知图形系统调用指定的函数。通常应用于一个窗口内容改变时自动重绘的场合。

子/父窗口，同胞

一个子窗口的定义是相对于另一个窗口，该窗口称为父窗口。无论什么时候，一个父窗口移动了，它的子窗口会相应随之移动。一个子窗口总是完全包含在它的父窗口里面，如果需要，它会被剪切。从属于同一个父窗口的多个子窗口相互间的关系称为“同胞”。

客户区

一个窗口的客户区简单地说是它的可使用区。如果一个窗口包括一个边框或标题栏，则客户区是内部的矩形区域。如果没有这样一个边框，则客户区等同于窗口本身。

剪切，剪切区域

剪切是一种限制窗口或它的部分输出的操作。

剪切区域是一个窗口的原有可见区域。由于被更高 Z 序列的同胞窗口遮挡，或者不在父

窗口可见区域范围之内的缘故，这些部分就会被剪切掉。

桌面窗口

桌面窗口由视窗管理器自动创建，总是覆盖整个显示区域。它始终是一个最底层的窗口。如果没有定义其它窗口，它就是默认（活动）窗口。所有窗口都是桌面窗口的继承窗口。

句柄

当一个新的窗口被创建，WM 会给它分配一个唯一的标识符，称为句柄。句柄将用于对特定窗口更进一步操作的执行。

隐藏/显示窗口

一个隐藏的窗口是不可见的，尽管它仍然存在（有句柄）。当创建一个窗口时，如果没有创建指定的标识的话，默认状态是隐藏。使一个窗口可见则可以将其显示；使其不可见则可以隐藏它。

透明

带有透明部分的窗口包括有窗口静止时不被重绘的区域。这些区域的操作就仿佛是下面的窗口可以透过它们显示出来。在这种情况下，下面的窗口在这个透明窗口之前重绘就显得很重要了。WM 能自动处理正确的重绘顺序。

有效/无效

一个有效的窗口是一个完全更新了的窗口，它不需要重绘。

一个无效的窗口不再对所有的更新有反应，因此需要完全或部分重绘。当改变影响一个特定的窗口时，WM 标记该窗口无效。下一次窗口重绘（手动或通过回调函数）后，它将有效。

Z-序，底层/顶层

尽管一个窗口是在一个只有 X 和 Y 坐标构成的二维的屏幕上显示，WM 也管理被认为是 Z-序，或者深度的座标——一个虚拟的三维的坐标，决定窗口从背景到前景放置。因此窗口可以在另一窗口的上面或下面显示。

将一个窗口设置到底层将会把它放在所有同胞窗口（如果存在）的下面；设置为顶层将会把它放在所有同胞窗口的上面。创建一个窗口时，如果没有指定创建标识符，默认情况下

它会被设置在顶层。

12.2 WM API 函数

下表列出了与 μ C/GUI 视窗管理器有关的函数，在各自的类型中按字母顺序进行排列。函数的详细描述在本章稍后列出。

函数	说明
基本函数	
WM_CreateWindow()	创建一个窗口。
WM_CreateWindowAsChild()	创建一个子窗口。
WM_DeleteWindow()	删除一个窗口。
WM_Exec()	通过执行回调函数（所有工作）重绘有效窗口。
WM_Exec1()	通过执行一个回调函数（仅一个工作）重绘有效窗口。
WM_GetClientRect()	返回活动窗口的。
WM_GetDialogItem()	返回一个对话框项目（控件）的窗口句柄。
WM_GetOrgX()	返回活动窗口的原点 X 坐标。
WM_GetOrgY()	返回活动窗口的原点 Y 坐标。
WM_GetWindowOrgX()	返回一个窗口的原点 X 坐标。
WM_GetWindowOrgY()	返回一个窗口的原点 Y 坐标。
WM_GetWindowRect()	返回活动窗口的屏幕坐标。
WM_GetWindowSizeX()	返回一个窗口的水平尺寸（宽度）。
WM_GetWindowSizeY()	返回一个窗口的垂直尺寸（高度）。
WM_HideWindow()	使一个窗口不可见。
WM_InvalidateArea()	使显示屏的某些部分无效。
WM_InvalidateRect()	使一个窗口部分无效。
WM_InvalidateWindow()	使一个窗口无效。
WM_MoveTo()	设置一个窗口的坐标。
WM_MoveWindow()	移动一个窗口到另一个位置。
WM_Paint()	立即绘制或重绘一个窗口。
WM_ResizeWindow()	改变一个窗口尺寸。
WM_SelectWindow()	设置用于绘图操作的活动窗口。
WM_ShowWindow()	使一个窗口可见。
高级特性	
WM_Activate()	激活视窗管理器。
WM_BringToBottom()	在其同胞窗口之后放置一个窗口。
WM_BringToTop()	在其同胞窗口之前放置一个窗口。
WM_ClrHasTrans()	清除 has 透明标识。
WM_Deactivate()	解除视窗管理器。
WM_DefaultProc()	处理信息的默认函数。

WM_GetActiveWindow()	返回活动窗口的句柄。
WM_GetDesktopWindow()	返回桌面窗口的句柄。
WM_GetFirstChild()	返回窗口的第一个子窗口的句柄。
WM_GetNextSibling()	返回窗口的下一相同胞窗口的句柄。
WM_GetHasTrans()	返回 has 透明标志的当前值。
WM_GetParent()	返回窗口的父窗口的句柄。
WM_Init()	初始化视窗管理器。不再需要由 GUI_Init() 来完成。
WM_IsWindow()	判断一个指定的句柄是否一个有效句柄。
WM_SendMessage()	向一个窗口发送信息。
WM_SetDesktopColor()	设置桌面窗口颜色。
WM_SetCallback()	为一个窗口设置回调函数。
WM_SetCreateFlags()	当创建一个新窗口时设置一个默认标识符。
WM_SetHasTrans()	设置 has 透明标志。
WM_SetUserClipRect()	临时减小剪切区域。
WM_ValidateRect()	使一个窗口的部分有效。
WM_ValidateWindow()	使一个窗口有效。
存储设备支持（可选）	
WM_DisableMemdev()	禁止用于重绘一个窗口的存储设备的使用。
WM_EnableMemdev()	启用用于重绘一个窗口的存储设备的使用。

12.3 视窗管理器的回调机制

WM 可以在有或没有回调函数的情况下使用。在大多数情况下，使用回调函数更为可取。

回调机制后面的哲学

μ C/GUI 为窗口和窗口对象（控件）提供的回调机制实质是一个事件驱动系统。正如在大多数视窗系统中一样，原则是控制流程不只是从用户程序到图形系统（用户程序调用图形系统函数来更新窗口），而且可以从用户程序到图形系统，同时也从图形系统回到用户程序，意思是图形系统也可以调用用户程序提供的回调函数来达到更新窗口的目的。这种机制——常常表现好莱坞法则的特点（“不要打电话给我们，我们会打电话给你们！”）——主要是视窗管理器为了启动窗口重绘的需要。与传统程序比较有差异，但它使对视窗管理器的无效逻辑开发成为可能。

不使用回调函数

你不一定非要用回调函数不可，但这样做，WM 在重绘窗口管理时会降低效率。也可以混合使用，例如，一些窗口使用回调而另一些却不使用。然而，如果一个窗口不使用回调机制，

你的应用程序必须负责更新窗口内容。

警告：当没有使用回调机制时，屏幕更新的管理就成了你的责任。

12.4 使用回调函数

为了使用一个回调函数创建窗口，你必需要有一个回调函数。函数的名称将与创建窗口时对应的回调函数指针参数名称相一致（即 `WM_CreateWindow()` 中的 `cb` 参数）。所有的回调函数必须具有以下函数原型：

函数原型

```
void callback(WM_MESSAGE* pMsg);
```

参 数	含 意
<code>pMsg</code>	消息的指针。

回调函数的执行行为依赖于它收到的消息类型。上面的函数原型通常带有一个开关声明，用于定义了对于不同的使用一个或更多的事件声明的消息所采用的不同的处理方式（典型的至少有对 `WM_PAINT()` 的处理）。

范例

创建一个回调函数自动重绘一个窗口：

```
void WinHandler(WM_MESSAGE* pMsg)
{
    switch (pMsg->MsgId)
    {
        case WM_PAINT:
            GUI_SetBkColor(0xFF00);
            GUI_Clear();
            GUI_DispStringAt("Hello world", 0, 0);
            break;
    }
}
```

WM_MESSAGE 元素

MsgId	消息的类型（参照下表）
HWin	目标窗口。
hWinSrc	源窗口。
Data.p	数据指针。
Data.v	数据数值。

MsgId 元素使用的消息类型

WM_PAINT	重绘窗口（因为内容至少部分无效）。
WM_CREATE	一个窗口创建后即发送。
WM_DELETE	告诉窗口释放它的数据结构（如果有的话），然后它将会被删除。
WM_SIZE	当一个窗口的大小改变后发送到它。
WM_MOVE	当一个窗口移动后发送到它。
WM_SHOW	当一个窗口收到显示命令后发送到它。
WM_HIDE	当一个窗口收到隐藏命令后发送到它。
WM_TOUCH	触摸屏消息。

应用程序可以为它自己的用途定义附加消息。为了保证它们使用的消息 ID 不会与 μ C/GUI 使用的消息 ID 同名，用户定义的消息的编号以 WM_USER 为开始。你应该像下面所展示的一样定义你自己的消息：

```
#define MY_MESSAGE_AAA WM_USER+0
#define MY_MESSAGE_BBB WM_USER+1
```

等等

背景窗口重绘及回调

在初始化视窗管理器期间，会创建一个包括整个 LCD 区域的窗口作为背景窗口（或称桌面窗口）。该窗口的句柄是 WM_HBKWIN。WM 不会自动重绘背景窗口的区域，因为没有默认的背景颜色。这意味着如果你进一步创建新一层窗口，然后删除它，被删除的窗口仍然是可见的。需要指定 WM_SetBkWindowColor() 函数设置重绘背景窗口的颜色。

你也可以设置一个回调函数处理这个问题。如果一个窗口被创建，然后象前面一样被删除，回调函数将触发 WM 去识别背景窗口已不再有效，并自动进行重绘。想了解关于创建和使用回调函数的信息，参考本章末的范例。

12.5 基本函数

WM_CreateWindow()

描述

在一个指定位置创建一个指定尺寸的窗口

函数原型

```
WM_HWIN WM_CreateWindow (    int x0, int y0,
                             int width, int height,
                             U8 Style,
                             WM_CALLBACK* cb,
                             int NumExtraBytes);
```

参 数	含 意
x0	左上角 X 轴坐标。
y0	左上角 Y 轴坐标。
width	窗口的 X 轴尺寸。
height	窗口的 Y 轴尺寸。
Style	窗口创建标识，如下一表格所示。
cb	回调函数的指针，如果没有使用回调函数则为 NULL。
NumExtraBytes	分配的额外字节数，通常为 0。

参数 Style 允许数值（或数值组合）

WM_CF_HASTRANS	Has 透明标志。必须由窗口定义哪些客户区不能完全填充。
WM_CF_HIDE	创建窗口后将它隐藏（默认）。
WM_CF_SHOW	创建窗口后显示它。
WM_CF_FGND	创建窗口后把它放到前面。（默认）。
WM_CF_BGND	创建窗口后把它放到后面。
WM_CF_STAYONTOP	确定窗口继续停留在所有创建的不带这个标志的同胞窗口上面。
WM_CF_MEMDEV	重绘时自动使用一个存储设备。这可能避免闪烁，同时在大多数情况下会提高输出速度，因为剪切简化了。注意为了能够使用这个标志，要求用到存储设备软件包（并且要在配置中启用）。如果存储设备没有启用，该标志被忽略。

返回数值

所创建窗口的句柄。

附加信息

几个创建标志可以通过“OR”操作进行组合。使用负的座标系。

范例

用回调函数创建一个窗口：

```
hWin2 = WM_CreateWindow(100, 10 ,180, 100, WM_CF_SHOW, &WinHandler, 0);
```

用回调函数创建一个窗口：

```
hWin2 = WM_CreateWindow(100, 10 ,180, 100,WM_CF_SHOW, NULL, 0);
```

WM_CreateWindowAsChild()

描述

以子窗口的形式创建一个窗口。

函数原型

```
WM_HWIN WM_CreateWindowAsChild( int x0, int y0,  
                                int width, int height,  
                                WM_HWIN hWinParent,  
                                U8 Style,  
                                WM_CALLBACK* cb,  
                                int NumExtraBytes);
```

参数	含义
x0	相对于父窗口的左上角 X 轴坐标。
y0	相对于父窗口的左上角 Y 轴坐标。
width	窗口的 X 轴尺寸。如果为 0, 为父窗口客户区 X 轴尺寸。
height	窗口的 Y 轴尺寸。如果为 0, 为父窗口客户区 Y 轴尺寸。
hWinParent	父窗口的句柄。
Style	窗口创建标志（参阅 WM_CreateWindow ()）。
cb	回调函数的指针，没有使用回调函数的话，为 NULL。
NumExtraBytes	额外分配的字节数量，通常为 0。

返回数值

子窗口的句柄。

附加信息

如果 `hWinParent` 参数设为 0，背景窗口当作父窗口使用。默认情况下，一个子窗口会旋转在它的父客串和所有原先的同胞窗口的上面，所以，如果它们的 Z-序没有改变的话，这个“最年轻”的窗口将总会在最顶部。

同胞窗口的 Z-序可以改变，尽管不论它们的次序如何，它们总是保持在它们父窗口的上面。

WM_DeleteWindow()

描述

删除一个指定的窗口。

函数原型

```
void WM_DeleteWindow(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口句柄。

附加信息

在窗口被删除之前，它收到一个 `WM_DELETE` 消息。该消息典型用于删除任何它使用的对象（控件）并释放分配给窗口的动态内存。

如果指定的窗口有子窗口，在窗口本身被删除之前，这些子窗口自动被删除。

WM_Exec()

描述

通过执行回调函数（所有工作）重绘无效窗口。

函数原型

```
int WM_Exec(void);
```

返回数值

0: 如果没有工作执行; 1: 如果执行一项工作。

附加信息

该函数将自动重复调用 `WM_Exec1()`，直到它完成所有的工作——本质上是直到返回一个 0 数值。

推荐改为调用 `GUI_Exec()`。

通常该函数不需要由用户应用程序调用。它由 `GUI_Delay()` 函数自动调用。如果你在使用多任务系统，我们推荐使用一个单独的任务执行这个函数，如下所示：

```
void ExecIdleTask(void)
{
    while(1)
    {
        WM_Exec();
    }
}
```

WM_Exec1()

描述

通过执行一个回调函数（只有一项工作）重绘一个无效窗口。

函数原型

```
int WM_Exec1(void) ;
```

返回数值

0: 如果没有工作执行; 1: 如果执行一项工作。

附加信息

该函数可能反复调用直到返回 0，这意思是所有的工作已经完成。

推荐改为调用 `GUI_Exec1()`。

该函数由 `WM_Exec()` 函数自动调用。

WM_GetClientRect()

描述

返回活动窗口的客户区的座标。

函数原型

```
void WM_GetClientRect(GUI_RECT* pRect);
```

参数	含义
<code>pRect</code>	一个 <code>GUI_RECT</code> 的指针。

`WM_GetDialogItem()`

描述

返回一个对话框项目（控件）的窗口句柄。

函数原型

```
WM_HWIN WM_GetDialogItem(WM_HWIN hDialog, int Id);
```

参数	含义
<code>hDialog</code>	对话框的句柄。
<code>Id</code>	控件的窗口 ID。

返回数值

控件窗口的句柄。

附加信息

该函数总是在创建对话框时使用。在使用它之前，对话框使用的窗口 ID 必须转化成它的句柄。

WM_GetOrgX(), WM_GetOrgY()

描述

(分别) 返回一个活动窗口的客户区的原点的 X 轴或 Y 轴坐标。

函数原型

```
int WM_GetOrgX(void); int WM_GetOrgY(void);
```

返回数值

以像素为单位的客户区原点的 X 轴或 Y 轴坐标。

WM_GetWindowOrgX(), WM_GetWindowOrgY()

描述

(分别) 返回以像素为单位的指定窗口的客户区的原点的 X 轴或 Y 轴坐标。

函数原型

```
int WM_GetWindowOrgX(WM_HWIN hWin); int WM_GetWindowOrgY(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

返回数值

以像素为单位的客户区的原点的 X 轴或 Y 轴坐标。

WM_GetWindowRect()

描述

返回活动窗口的座标。

函数原型

```
void WM_GetWindowRect(GUI_RECT* pRect);
```

参数	含义
<code>pRect</code>	一个 GUI_RECT 结构的指针。

WM_GetWindowSizeX(), WM_GetWindowSizeY()

描述

(分别) 返回指定窗口的 X 轴或 Y 轴的尺寸。

函数原型

```
int WM_GetWindowSizeX(WM_HWIN hWin); int WM_GetWindowSizeY(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

返回数值

以像素为单位的窗口 X 轴或 Y 轴的尺寸。

在水平方向定义为 $x_1 - x_0 + 1$ ，垂直方向为 $y_1 - y_0 + 1$ ，这里 x_0 , x_1 , y_0 , y_1 是分别是窗口的最左边，最右边，最顶部，最底部坐标。

WM_HideWindow()

描述

使一个指定窗口不可见。

函数原型

```
void WM_HideWindow(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

附加信息

调用该函数后，窗口并不会立即显现“不可见”效果。其它窗口的无效区域（那些位于窗口后面的应该隐藏的区域）在执行 WM_Exec () 函数时会被重绘。如果你需要立即隐藏一个窗口的话，你应当调用 WM_Paint () 函数重绘其它窗口。

WM_InvalidateArea()

描述

使显示屏的指定矩形区域无效。

函数原型

```
void WM_InvalidateArea(GUI_RECT* pRect);
```

参数	含义
pRect	一个 GUI_RECT 结构的指针。

附加信息

调用该函数将告诉 WM 指定的区域不要更新。

该函数可以用于使任何窗口或重叠可相交的部分窗口无效。

WM_InvalidateRect()

描述

使一个窗口的指定矩形区域无效。

函数原型

```
void WM_InvalidateRect(WM_HWIN hWin, GUI_RECT* pRect);
```

参数	含义
hWin	窗口的句柄。
pRect	一个 GUI_RECT 结构的指针。

附加信息

调用该函数将告诉 WM 指定的区域不要更新。

接下来调用 WM_Paint () 进行窗口重绘，该区域同样也能够被重绘。

WM_InvalidateWindow()

描述

使一个指定的窗口无效。

函数原型

```
void WM_InvalidateWindow(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

附加信息

调用该函数告诉 WM 指定的窗口不更新。

WM_MoveTo()

描述

将一个指定的窗口移到某个位置。

函数原型

```
void WM_MoveTo(WM_HWIN hWin, int dx, int dy);
```

参数	含义
<code>hWin</code>	窗口的句柄。
<code>x</code>	新的 X 轴坐标。
<code>y</code>	新的 Y 轴坐标。

WM_MoveWindow()

描述

把一个指定的窗口移动一段距离。

函数原型

```
void WM_MoveWindow(WM_HWIN hWin, int dx, int dy);
```

参数	含义
hWin	窗口的句柄。
dx	移动的水平距离。
dy	移动的垂直距离。

WM_Paint()

描述

立即绘制或重绘一个指定窗口。

函数原型

```
void WM_Paint(WM_HWIN hWin);
```

参数	含义
hWin	窗口的句柄。

附加信息

窗口重绘，反映它最后一次绘制以来所有的更新。

WM_ResizeWindow()

描述

改变一个指定窗口的尺寸。

函数原型

```
void WM_ResizeWindow(WM_HWIN hWin, int XSize, int YSize);
```

参数	含义
hWin	窗口的句柄。
Xsize	窗口水平尺寸要修改的值。
YSize	窗口垂直尺寸要修改的值。

WM_SelectWindow()

描述

设置一个活动窗口用于绘制操作。

函数原型

```
WM_HWIN WM_SelectWindow(WM_HWIN hWin);_
```

参数	含义
<code>hWin</code>	窗口的句柄。

返回数值

选择的窗口。

范例

将活动窗口的句柄设为 `hWin2`，设置背景颜色，然后清除窗口：

```
WM_SelectWindow(hWin2);
GUI_SetBkColor(0xFF00);
GUI_Clear();
```

WM_ShowWindow()

描述

使一个指定窗口可见。

函数原型

```
void WM_ShowWindow(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

附加信息

该调用以后，窗口不会立即可见。它在执行 `WM_Exec()` 更新。如果你需要立即显示（绘

制) 这个窗口, 你应当调用 WM_Paint () 函数。

12.6 高级函数

WM_Activate()

描述

激活视窗管理器。

函数原型

```
void WM_Activate (void) ;
```

附加信息

初始化后, 默认情况下 WM 是激活的。该函数只有原先调用 WM_Deactivate () 函数, 才需要调用。

WM_BringToBottom()

描述

将一个指定窗口放到它的同胞窗口下面。

函数原型

```
void WM_BringToBottom(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

附加信息

该窗口会放在所有其它同胞窗口的下面, 但是位置保持在它的父窗口前面。

WM_BringToTop()

描述

将一个指定窗口放到它的同胞窗口上面。

函数原型

```
void WM_BringToTop(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

附加信息

该窗口会放在所有其它同胞窗口和父窗口的上面。

WM_ClrHasTrans()

描述

清除 has 透明标志（设为 0）。

函数原型

```
void WM_ClrHasTrans(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

附加信息

当该标志被设置时，它告诉视窗管理器一个窗口包含有不要重绘的部分，该部分因此而透明。然后 WM 知道背景重绘需要优先于窗口重绘，这是为了保证透明部分能正确地恢复。

该标志被 WM_ClrHasTrans () 清除后，在窗口重绘之前，WM 将不会自动重绘背景。

WM_Deactivate()

描述

使视窗管理器失效。

函数原型

```
void WM_Deactivate(void);
```

附加信息

该函数调用后，剪切区设为全部 LCD 区域，WM 将不执行窗口回调函数。WM_DefaultProc()

描述

WM_DefaultProc()

默认消息处理器。

函数原型

```
void WM_DefaultProc(WM_MESSAGE* pMsg)
```

参数	含义
pMsg	消息的指针。

附加信息

在下面的范例中，使用该函数处理未加工过的消息：

```
static WM_RESULT cbBackgroundWin(WM_MESSAGE* pMsg)
{
    switch (pMsg->MsgId)
    {
        case WM_PAINT:
            GUI_Clear();
        default:
            WM_DefaultProc(pMsg);
    }
}
```

WM_GetActiveWindow()

描述

返回用于绘图操作的活动窗口的句柄。

函数原型

```
WM_HWIN WM_GetActiveWindow(void);
```

返回值

活动窗口的句柄

WM_GetDesktopWindow()

描述

返回桌面窗口的句柄。

函数原型

```
WM_HWIN WM_GetDesktopWindow(void);
```

返回数值

桌面窗口的句柄。

附加信息

桌面窗口总是最底层的窗口，任何进一步创建的窗口都是它的子窗口。

WM_GetFirstChild()

描述

返回指定窗口的第一个子窗口的句柄。

函数原型

```
void WM_GetFirstChild(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

返回数值

窗口的第一个子窗口的句柄；如果没有子窗口，则为 0。

附加信息

窗口的第一个子窗口是基于特定的父窗口创建的第一个子窗口。如果窗口的 Z-序列没有改变，它将直接放在指定的父窗口上面。

WM_GetNextSibling()

描述

返回指定窗口的下一个同胞窗口的句柄。

函数原型

```
void WM_GetNextSibling(WM_HWIN hWin);
```

参数	含义
hWin	窗口的句柄。

返回数值

窗口的下一个同胞窗口的句柄，如果不存在，则为 0。

附加信息

窗口的下一个同胞窗口是基于同一个父窗口创建的下一个子窗口。如果窗口的 Z-序列没有改变，它将直接放在指定的窗口上面。

WM_GetHasTrans()

描述

返回 has 透明标志当前的值。

函数原型

```
U8 WM_GetHasTrans(WM_HWIN hWin);
```

参数	含义
hWin	窗口的句柄。

返回数值

0: 非透明; 1: 窗口有透明部分。

附加信息

当该标志被设置时，它告诉视窗管理器一个窗口包含有不要重绘的部分，该部分因此而透明。然后 WM 知道背景重绘需要优先于窗口重绘，这是为了保证透明部分能正确地恢复。

WM_GetParent()

描述

返回指定窗口的父窗口的句柄。

函数原型

```
void WM_GetParent(WM_HWIN hWin);
```

参数	含义
hWin	窗口的句柄。

返回数值

窗口父窗口的句柄，如果不存在则为 0。

附加信息

没有父窗口存在的唯一情况是桌面窗口的句柄用作参数。

WM_Init()

描述

初始化视窗管理器。

函数原型

```
void WM_Init (void) ;
```

附加信息

不再需要通过用户程序调用该函数，它通过 GUI_Init () 调用。

WM_IsWindow()

描述

确定一个指定句柄是否一个有效的窗口句柄。

函数原型

```
void WM_IsWindow(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

返回数值

0：句柄不是一个有效的窗口句柄男；1：句柄是一个有效的窗口句柄。

WM_SendMessage()

描述

向一个指定窗口发送一个消息。

函数原型

```
void WM_SendMessage(WM_HWIN hWin, WM_MESSAGE* pMsg)
```

参数	含义
<code>hWin</code>	窗口的句柄。
<code>pMsg</code>	消息的指针。

WM_SetDesktopColor()

描述

设置桌面窗口的颜色。

函数原型

```
GUI_COLOR WM_SetDesktopColor(GUI_COLOR Color);
```

返回数值

原先选择的桌面窗口颜色。

附加信息

桌面窗口的默认设置是不需重绘它自己本身。如果该函数没有调用，桌面窗口根本就不会重绘；因此其它窗口会保持可见，甚至它们已经被删除。

一旦这个函数指定一种颜色，桌面窗口将重绘自身。为了恢复默认值，调用这个函数并指定 GUI_INVALID_COLOR。

WM_SetCallback()

描述

设置为视窗管理器执行的回调函数。

函数原型

```
WM_CALLBACK* WM_SetCallback (WM_HWIN hWin, WM_CALLBACK* cb)
```

参数	含义
hWin	窗口的句柄。
cb	回调函数的指针。

返回数值

原先回调函数的指针。

WM_SetCreateFlags()

描述

创建一个新的窗口时设置用作默认值的标志。

函数原型

返回数值

该参数原先的值。

附加信息

这里指定的标志是二进制数 0Red，和 WM_CreateWindow() 及 WM_CreateWindowAsChild() 函数指定的标志一致。标志 WM_CF_MEMDEV 常常用于在所有窗口上启用存储设备。

范例

```
WM_SetCreateFlags(WM_CF_MEMDEV);    /* 在所有窗口上自动使用存储设备 */
```

WM_SetHasTrans()

描述

设置 has 透明标志（将其设为 1）。

函数原型

```
void WM_SetHasTrans(WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

附加信息

当设置时，该标志告诉视窗管理器一个窗口包括没有重绘的部分，因而以透明模式显示。然后 WM 知道背景的重绘要先于窗口的重绘，这了为了保证透明部分能正确的重建。

WM_SetUserClipRect()

描述

临时缩小当前窗口的剪切区为一个指定的矩形。

函数原型

```
const GUI_RECT* WM_SetUserClipRect(const GUI_RECT* pRect);
```

参数	含义
<code>pRect</code>	一个定义剪切区域的 GUI_RECT 结构的指针。

返回数值

原先剪切矩形的指针。

附加信息

可以传递一个空指针以恢复默认调协。在回调函数使用时，剪切矩形会被 WM 复位。

给出的矩形必须与当前窗口有关。你不能将剪切矩形面积放大超过当前窗口矩形。

你的应用中必须保证指定的矩形保留它的值直到它不再被使用。也就是，直到一个不同的剪切区被指定或一个空指针被传递。这意思是，如果剪切矩形保持活动直到返回以后，当作参数传递的矩形结构不应是一个自动变量（通常位于堆栈上）。既然这样，应该使用一个静态变量。

范例

这个范例来自进度指示器的绘制函数。进度指示器必须在进度条上写上文字，该处文字颜色必须与进度条左右两部分不同。这意思是一个数字的一半可能是一种颜色，而另一半是另一种颜色。最好的处理办法是在绘制如下所示的进度条时临时减小剪切区：

```
/* 绘制进度条的左边部分 */
```

```
r.x0=0; r.x1=x1-1; r.y0=0; r.y1 = GUI_YMAX;  
WM_SetUserClipRect(&r);  
GUI_SetBkColor(pThis->ColorBar[0]);  
GUI_SetColor(pThis->ColorText[0]);  
GUI_Clear();  
GUI_GotoXY(xText,yText);  
GUI_DispDecMin(pThis->v);  
GUI_DispChar('%');
```

```
/* 绘制进度条的右边部分 */
```

```
r.x0=r.x1; r.x1=GUI_XMAX;
```

```
WM_SetUserClipRect(&r);
GUI_SetBkColor(pThis->ColorBar[1]);
GUI_SetColor(pThis->ColorText[1]);
GUI_Clear();
GUI_GotoXY(xText, yText);
GUI_DispDecMin(pThis->v);
GUI_DispChar(' %');
```

进度条外观的屏幕截图



WM_ValidateRect()

描述

使一个指定的窗口矩形区域有效。

函数原型

```
void WM_ValidateRect (WM_HWIN hWin, GUI_RECT* pRect);
```

参数	含义
hWin	窗口的句柄。
pRect	一个 GUI_RECT 结构的指针。

附加信息

调用该函数会告诉 WM 指定区域被更新。通常该函数在内部调用，不需由用户应用程序调用。

WM_ValidateWindow()

描述

使一个指定的窗口有效。

函数原型


```
void WM_ValidateWindow (WM_HWIN hWin);
```

参数	含义
<code>hWin</code>	窗口的句柄。

附加信息

调用该函数会告诉 WM 指定窗口被更新。通常该函数在内部调用，不需由用户应用程序调用。

12.7 存储设备支持（可选）

当一个存储设备用于一个窗口的重绘，所有绘制操作自动发送到一个存储设备上下文并在内存中执行。所有绘制操作执行之后的唯一结果是窗口在 LCD 上重绘立即反映所有的更新。使用存储设备的优点是避免了任何闪烁现象（通常发生在绘制操作执行时屏幕不断更新的时候）。

要了解更多关于如何操作存储设备的内容，请参阅第 10 章“存储设备”。

WM_DisableMemdev()

描述

禁止用于重绘一个窗口的存储设备的使用。

函数原型

```
void WM_EnableMemdev (WM_HWIN hWin)
```

参数	含义
<code>hWin</code>	窗口的句柄。

WM_EnableMemdev()

描述

启用用于重绘一个窗口的存储设备的使用。

函数原型

```
void WM_EnableMemdev (WM_HWIN hWin)
```

参数	含义
<code>hWin</code>	窗口的句柄。

12.8 范例

下面的范例展示了使用或不使用回调函数进行重绘之间的区别。它也展示了如何设置你自己的回调函数。范例文件是随 μ C/GUI 一起发布的范例当中的 `WM_Redraw.c`：

```
/*-----
文件：      WM_Redraw.c
目的：      展示视窗管理器的重绘机制
-----*/

#include "GUI.H"

/*****
*                      背景窗口的回调函数                      *
*****/

static void cbBackgroundWin(WM_MESSAGE* pMsg)
{
    switch (pMsg->MsgId)
    {
        case WM_PAINT:
            GUI_Clear();
        default:
            WM_DefaultProc(pMsg);
    }
}

/*****
*                      前景窗口的回调函数                      *
*****/

static void cbForegroundWin(WM_MESSAGE* pMsg)
{

```

```

switch (pMsg->MsgId)
{
    case WM_PAINT:
        GUI_SetBkColor(GUI_GREEN);
        GUI_Clear();
        GUI_DispString("Foreground window");
        break;
    default:
        WM_DefaultProc(pMsg);
}
}

/*****
*                                     *
*                                $\mu$ C/GUI 重绘机制展示                               *
*                                     *
*****/

static void DemoRedraw(void)
{
    GUI_HWIN hWnd;
    while(1)
    {
        /* 创建前景窗口 */
        hWnd = WM_CreateWindow( 10, 10, 100, 100,
                                WM_CF_SHOW,
                                cbForegroundWin, 0);    /* 显示前景窗口 */

        GUI_Delay(1000);
        /* 删除前景窗口 */
        WM_DeleteWindow(hWnd);
        GUI_DispStringAt("Background of window has not been redrawn", 10, 10);
        /* 等待一会，背景不会重绘 */
        GUI_Delay(1000);
        GUI_Clear();
        /* 设置背景窗口的回调函数 */
        WM_SetCallback(WM_HBKWIN, cbBackgroundWin);
        /* 创建前景窗口 */
        hWnd = WM_CreateWindow( 10, 10, 100, 100,
                                WM_CF_SHOW,

```

```
        cbForegroundWin, 0);    /* 显示前景窗口 */

    GUI_Delay(1000);

    /* 删除前景窗口 */
    WM_DeleteWindow(hWnd);
    /* 等待一会，背景不会重绘 */
    GUI_Delay(1000);
    /* 删除背景窗口的回调函数 */
    WM_SetCallback(WM_HBKWIN, 0);
}

}

/*****
*                               主函数                               *
*****/

void main (void)
{
    GUI_Init ();
    DemoRedraw();
}
```

第13章 窗口对象（控件）

控件是具有对象性质的窗口，在视窗世界中它们被称为控件，构造用户接口的元素。它们能自动对某些事件起反应；例如，当一个按钮被按下时，它可能以不同的状态显示。控件需要建立，具有能在它们存在的任何时间修改的特性，在它们不再需要时被删除。与窗口类似，一个控件通过它的建立函数返回的句柄而引用。

控件需要使用视窗管理器。一旦一个控件被建立，它被处理成与其它窗口一样；WM 保证它在需要的时候能正确的显示（及重绘）。当编写一个应用或一个用户接口时，控件并不需要，但是它们能使编程更容易。

13.1 一些基础知识

有效的控件

下面的 μ C/GUI 控件在当前是有效的：

名 称	说 明
BUTTON	可以按下的按钮。在按钮上可以显示文本或位图。
CHECKBOX	复选框，提供多个选项。
EDIT	单行文本编辑框，提示用户输入数字或文本。
FRAMEWIN	框架窗口，建立典型的 GUI 外形。
LISTBOX	列表框，当被选中的项目会高亮显示。
PROGBAR	进度条，用于观察。
RADIOBUTTON	单选按钮，可以选择。同一时间只有一个按钮被选中。
SCROLLBAR	滚动条，可以是水平或垂直的。
SLIDER	滑动条，用于改变数值。
TEXT	文本控件，典型应用在对话框中。

理解重绘机制

一个控件根据它的特性绘制自己。这一工作通过调用 WM 的 API 函数 WM_Exec() 来完成。如果在程序中没有调用 WM_Exec()，就必须调用 WM_Paint() 函数来绘制控件。在多任务环境中的 μ C/GUI，一个后台任务通常用于调用 WM_Exec() 并更新控件（及其它所有带有回调函数的窗口）。这样就不必手工调用 WM_Paint()；然而，手工调用仍然是合法的，如果你想保证控件能立即被重绘的话，这样做也是有意义的。

当一个控件的属性被改变时，控件的窗口（或者它的一部分）被标记为无效，但是它不会立即重绘。因此，这部分代码运行非常快。重绘在后面的时间通过 WM 完成，或通过为控件调用 WM_Paint() 函数（或者 WM_Exec()，直到所有的窗口都被重绘）来强制执行。

如何使用控件

假设我们要显示一个进度条。需要写以下的代码：

```
PROGBAR_Handle hProgBar;  
GUI_DispStringAt("Progress bar", 100, 20);  
hProgBar = PROGBAR_Create(100, 40, 100, 20, WM_CF_SHOW);
```

第一行为控件的处理保留内存。最后一行实际建立控件。如果晚一些时候或在一个单独

任务中 WM_Exec() 被调用的话，控件会被视窗管理器自动绘出。



对于每一种控件来说，成员函数是有效的，通过它们允许修改控件的外观。一旦控件被建立，它的特性可以通过调用一个它的成员函数来改变。这些函数以控件的句柄作为它们的第一个参数。为了建立一个进度条，上面显示 45%，并将其颜色从默认值（深灰/浅灰）改为绿/红，要使用到以下部分代码：

```
PROGBAR_SetBarColor(hProgBar, 0, GUI_GREEN);  
PROGBAR_SetBarColor(hProgBar, 1, GUI_RED);  
PROGBAR_SetValue(hProgBar, 45);
```



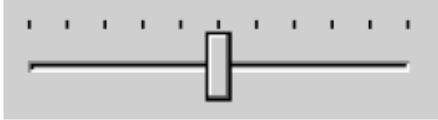
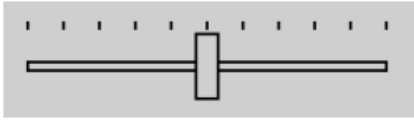
所有的控件也都有一个或多个配置宏，用于定义不同的默认设置，诸如字体和使用的颜色等。在本章后面每个控件各自部分，对应的有效的配置选项都会列出来。

用于控件的动态存储器

在嵌入式应用当中，通常来说，使用动态存储器确实不是非常合适，因为存储残片效果的缘故。有很多不同的方式可以用来这种情况，但是它们都工作在一个受限制的方式里，随时内存区域都可能被应用程序中的一个指针引用。因为这个原因， μ C/GUI 使用一个不同的方法：所有物体（以及所有在运行时存储的数据）被存入一个句柄引用的内存区域当中。这会使已分配好的内存区域在运行时重新分配成为可能。因而避免了使用指针时出现的长时间分配的问题。所有控件因此通过句柄引用。

3D 支持

一些控件能够以 3D 或非 3D 效果显示。默认情况下是启用 3D 支持，但是也可能通过将配置宏<WIDGET>_USE_3D 设置为 0 而禁止它。不管它是否使用三维效果，一个控件的功能完全一样的；唯一的不同是它的外观。下面是一个滑块控件的范例：

启用 3D 效果（默认）	禁止 3D 效果
	

13.2 通用控件 API 函数

API 参考：用于控件的 WM 函数

既然控件本质上是一个窗口，那么任何一种视窗管理器 API 函数都适用于它们。控件的句柄作为 `hWin` 参数，控件象其它窗口一样处理。最普遍用于控件的 WM 函数在下表列出：

函数	说明
<code>WM_DeleteWindow()</code>	删除一个窗口。
<code>WM_DisableMemdev()</code>	禁止存储设备的重绘功能。
<code>WM_EnableMemdev()</code>	启用存储设备的重绘功能。
<code>WM_InvalidateWindow()</code>	使一个窗口无效。
<code>WM_Paint()</code>	立即绘制或重绘一个窗口。

对于全部与 WM 相关的函数目录，请参阅第 12 章：视窗管理器。

API 参考：所有控件共有的函数

下表按字母顺序列出了与控件相关的函数。这些函数适用于所有的控件。在此列出是为了避免重复。函数的详细描述在后面给出。每个控件附加的成员函数在后面部分可以找到。

函数	说明
<code><WIDGET>_CreateIndirect()</code>	在对话框中自动创建时使用。
<code>WM_EnableWindow()</code>	将控件状态设置为启用（默认）。
<code>WM_DisableWindow()</code>	将控件状态设置为禁止。

<WIDGET>_CreateIndirect()

描述

建立一个用于对话框中的控件。

函数原型


```
<WIDGET>_Handle <WIDGET>_CreateIndirect(    const GUI_WIDGET_CREATE_INFO*
                                              pCreateInfo, WM_HWIN hParent,
                                              int x0, int y0,
                                              WM_CALLBACK* cb);
```

参 数	含 意
<code>pCreateInfo</code>	指向一个 GUI_WIDGET_CREATE_INFO 结构（下面提到）的指针。
<code>hParent</code>	父窗口的句柄。
<code>x0</code>	控件最左边的像素（桌面坐标）。

附加信息

通过使用适当的前缀，任何控件都可能被间接的建立。

例如：`BUTTON_CreateIndirect()` 间接建立一个按钮控件，`CHECKBOX_CreateIndirect()` 间接建立一个复选框控件，等等。

如果一个控件包括在一个对话框当中，它只需要间接建立。否则，它可以通过使用 `<WIDGET>_Create()` 函数直接建立。更多关于对话框的信息，请参阅第 14 章：对话框。

资源表

对话框资源表中的 GUI_WIDGET_CREATE_INFO 结构定义如下：

```
typedef struct
{
    const char* pName;           // 文本（不是所有的控件都使用）
    I16 Id;                      // 控件的窗口 ID
    I16 x0, y0, xSize, ySize;    // 控件的大小和位置
    I16 Flags;                   // 控件专用标识（或 0）
    I32 Para;                    // 控件专用参数（或 0）
} GUI_WIDGET_CREATE_INFO;

GUI_WIDGET_CREATE_FUNC* pfCreateIndirect;    // 创建函数
```

控件标识和参数是可选的，这非常依赖于控件的类型。每个控件的标识和参数（如果存在）会在本章后面的适当部分列出。

WM_EnableWindow()

描述

将一个指定控件状态设置为激活（启用）。

函数原型

```
void WM_EnableWindow(WM_Handle hObj);
```

参 数	含 意
hObj	控件的句柄。

附加信息

对于任何控件来说，这是默认设置。

WM_DisableWindow()

描述

将一个指定的控件状态设置为非活动（无用）。

函数原型

```
void WM_DisableWindow(WM_Handle hObj);
```

参 数	含 意
hObj	控件的句柄。

附加信息

一个被禁止的控件以灰色显示，不会接收用户的输入。但其实际外观可以改变（根据 控件/配置 的设置等等）

13.3 BUTTON：按钮控件

一般情况下，按钮控件作为触摸屏主要的用户接口部件使用。按钮可以显示文本，如下面所示，或者一幅位图。



所有与按钮相关的函数仅次于文件 `BUTTON*.c`，`BUTTON.h` 当中。所有的标识是前缀“`BUTTON`”。

配置选项

类型	宏	默认值	说明
N	<code>BUTTON_3D_MOVE_X</code>	1	按下状态文本/位图在水平方向偏移的像素数。
N	<code>BUTTON_3D_MOVE_Y</code>	1	按下状态文本/位图在垂直方向偏移的像素数。
N	<code>BUTTON_BKCOLOR0_DEFAULT</code>	0xAAAAAA	非按下状态的背景颜色。
N	<code>BUTTON_BKCOLOR1_DEFAULT</code>	<code>GUI_WHITE</code>	按下状态的背景颜色。
S	<code>BUTTON_FONT_DEFAULT</code>	<code>&GUI_Font13_1</code>	按钮文本的字体。
N	<code>BUTTON_TEXTCOLOR0_DEFAULT</code>	<code>GUI_BLACK</code>	非按下状态的文本颜色。
N	<code>BUTTON_TEXTCOLOR1_DEFAULT</code>	<code>GUI_BLACK</code>	按下状态的文本颜色。
B	<code>BUTTON_USE_3D</code>	1	启用3D效果支持。

默认状态下按钮按下时背景色是白色，是特意这样做的，因为这会使按钮按下时在任何显示屏上显示得更明显些。如果你想让按钮的背景色无论是按下还是不按下的状态下都是一样的，需将 `BUTTON_BKCOLOR1_DEFAULT` 改为 `BUTTON_BKCOLOR0_DEFAULT`。

BUTTON API 函数

下表按字母顺序列出了 μ C/GUI 与按钮有关的函数，详细的描述在本章后面部分绘出。

函数	说明
<code>BUTTON_Create()</code>	建立按钮。
<code>BUTTON_CreateAsChild()</code>	将按钮作为一个子窗口建立。
<code>BUTTON_CreateIndirect()</code>	从资源表项目中建立按钮。
<code>BUTTON_SetBitmap()</code>	设置按钮显示时使用的位图。
<code>BUTTON_SetBitmapEx()</code>	设置按钮显示时使用的位图。
<code>BUTTON_SetBkColor()</code>	设置按钮的背景颜色。
<code>BUTTON_SetFont()</code>	选择文本字体。
<code>BUTTON_SetState()</code>	设置按钮状态（由触摸屏模块自动处理）。
<code>BUTTON_SetStreamedBitmap()</code>	设置按钮显示时使用的位图。
<code>BUTTON_SetText()</code>	设置文本。
<code>BUTTON_SetTextColor()</code>	设置文本的颜色。

BUTTON_Create()

描述

在一个指定位置，以指定的大小建立一个 BUTTON 控件。

函数原型

```
BUTTON_Handle BUTTON_Create(    int x0, int y0,
                                int xsize, int ysize,
                                int ID,
                                int Flags);
```

参 数	含 意
x0	按钮最左边的像素（在桌面座标中）。
y0	按钮最顶部的像素（在桌面座标中）。
xsize	按钮的水平尺寸（以像素为单位）。
ysize	按钮的垂直尺寸（以像素为单位）。
ID	按钮按下时返回的 ID。
Flags	窗口建立标识。具有代表性的，WM_CF_SHOW 是为了使控件立即可见（请参阅第 12 章“视窗管理器”中 WM_CreateWindow() 的参数值列表）。

返回数值

所建立的按钮控件的句柄，如果函数执行失败，则返回 0。

BUTTON_CreateAsChild()

描述

以子窗口的形式建立一个按钮。

函数原型

```
BUTTON_Handle BUTTON_CreateAsChild( int x0, int y0,
                                     int xsize, int ysize,
                                     WM_HWIN hParent,
                                     int ID,
```

```
int Flags);
```

参 数	含 意
<code>x0</code>	按钮最左边的像素（在桌面坐标中）。
<code>y0</code>	按钮最顶部的像素（在桌面坐标中）。
<code>xsize</code>	按钮的水平尺寸（以像素为单位）。
<code>ysize</code>	按钮的垂直尺寸（以像素为单位）。
<code>hParent</code>	父窗口的句柄。如果为 0，新的按钮窗口作为桌面（顶层窗口）的子窗口。
<code>ID</code>	按钮按下时返回的 ID。
<code>Flags</code>	窗口建立标识（参阅 <code>BUTTON_Create()</code> ）。

返回数值

所建立的按钮控件的句柄，如果函数执行失败，则返回 0。

BUTTON_CreateIndirect()

函数原型的说明在本章开始部分。The elements `Flags` and `Para` of the resource passed as parameter are not used.

BUTTON_SetBitmap()

描述

显示一个指定按钮时使用的位图。

函数原型

```
void BUTTON_SetBitmap(  BUTTON_Handle hObj,
                        int Index,
                        const GUI_BITMAP* pBitmap);
```

参 数	含 意
<code>hObj</code>	按钮的句柄。
<code>Index</code>	位图的索引（参阅下表）。
<code>pBitmap</code>	位图结构的指针。

参数 `Index` 的允许值

0	设置按钮未按下时使用的位图。
---	----------------

1	设置按钮按下时使用的位图。
---	---------------

BUTTON_SetBitmapEx()

描述

显示一个指定按钮时使用的位图。

函数原型

```
void BUTTON_SetBitmapEx(    BUTTON_Handle hObj,
                           int Index,
                           const GUI_BITMAP* pBitmap,
                           int x, int y) ;
```

参 数	含 意
hObj	按钮的句柄。
Index	位图的索引（参阅 BUTTON_SetBitmap() ）。
pBitmap	位图结构的指针。
x	位图相对按钮的 X 轴坐标。

BUTTON_SetBkColor()

描述

设置按钮的背景颜色。

函数原型

```
void BUTTON_SetBkColor(BUTTON_Handle hObj, int Index, GUI_COLOR Color);
```

参 数	含 意
hObj	按钮的句柄。
Index	颜色的索引（参阅下表）。
Color	设置的背景颜色。

参数 [Index](#) 的允许值

0	设置按钮未按下时使用的颜色；
1	设置按钮按下时使用的颜色。

BUTTON_SetFont()

描述

设置按钮字体。

函数原型

```
void BUTTON_SetFont(BUTTON_Handle hObj, const GUI_FONT* pFont);
```

参 数	含 意
hObj	按钮的句柄。
pFont	字体的指针。

附加信息

如果没有选择字体，将使用 BUTTON_FONT_DEF 字体。

BUTTON_SetState()

描述

设置一个指定按钮物体的状态。

函数原型

```
void BUTTON_SetState(BUTTON_Handle hObj, int State)
```

参 数	含 意
hObj	按钮的句柄。
State	按钮的状态（参阅下表）。

参数 [State](#) 允许的值

BUTTON_STATE_PRESSED	按钮被按下。
BUTTON_STATE_INACTIVE	按钮处于非活动状态。

附加信息

该函数用于触摸屏模块。通常你不必调用这个函数。

BUTTON_SetStreamedBitmap()

描述

显示一个指定按钮对象时设置使用的位图数据流。

函数原型

```
void BUTTON_SetStreamedBitmap( BUTTON_Handle hObj,
                               int Index,
                               const GUI_BITMAP_STREAM* pBitmap);
```

参 数	含 意
hObj	按钮的句柄。
Index	位图的索引（参阅 <code>BUTTON_SetBitmap()</code> ）。
pBitmap	一个位图数据流的指针。

附加信息

为了能使用这个函数，你必须在 `LCDConf.h` 中包括下面一行：

```
#define BUTTON_SUPPORT_STREAMED_BITMAP 1
```

想了解关于 `streamed` 位图的详细情况，请参阅第 6 章的 `GUI_DrawStreamedBitmap()` 函数。

BUTTON_SetText()

描述

设置在按钮上显示的文本。

函数原型

```
void BUTTON_SetText(BUTTON_Handle hObj, const char* s);
```

参 数	含 意
hObj	按钮的句柄。
s	显示的文本。

BUTTON_SetTextColor()

描述

设置按钮文本的颜色。

函数原型

```
void BUTTON_SetTextColor(BUTTON_Handle hObj, int Index, GUI_COLOR Color);
```

参 数	含 意
<code>hObj</code>	按钮的句柄。
<code>Index</code>	文本颜色的索引（参阅 <code>BUTTON_SetBkColor()</code> ）。
<code>Color</code>	设置的文本颜色。

范例

使用 BUTTON 控件

下面的范例展示如何使用两幅位图创建一个简单的按钮。本范例是 μ C/GUI 提供的范例 `WIDGET_SimpleButton.c`：

```
/*-----
文件:      WIDGET_SimpleButton.c
目的:      展示一个按钮控件使用的例子
-----*/

#include "gui.h"
#include "button.h"

/*****
*                      按钮控件使用的展示                      *
*****/

static void DemoButton(void)
{
    BUTTON_Handle hButton;
    int Key = 0;
    GUI_Init() ;
```

```

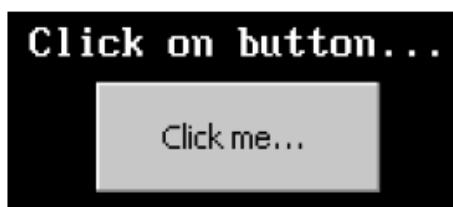
GUI_SetFont(&GUI_Font8x16);
GUI_DispStringHCenterAt("Click on button...", 160, 0);
/* 建立按钮 */
hButton = BUTTON_Create(110, 20, 100, 40, GUI_ID_OK, WM_CF_SHOW);
/* 设置按钮文本 */
BUTTON_SetText(hButton, "Click me...");
Key = GUI_WaitKey();
/* 删除按钮对象 */
BUTTON_Delete(hButton);
}

/*****
*
*                               主函数
*
*****/

void main(void)
{
    GUI_Init();
    DemoButton();
}

```

上面范例执行结果的截图



BUTTON 控件的高级应用

下面范例展示如何创建一个显示一幅电话图片的按钮。当按钮按下时，图片变成话筒提起的样子。本范例是μC/GUI 提供的范例 WIDGET_PhoneButton.c:

```

/*-----
文件:      WIDGET_PhoneButton.c
目的:      展示一个按钮控件使用的例子
-----*/

```



```
static const GUI_BITMAP bm_1bpp_0 = {32, 31, 4, 1, acPhone0, &Palette};
static const GUI_BITMAP bm_1bpp_1 = {32, 31, 4, 1, acPhone1, &Palette};

/*****
*
*                               按钮控件使用的展示
*
*****/
```

```

static void DemoButton(void)
{
    BUTTON_Handle hButton;
    int Stat = 0;
    GUI_Init();
    GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringHCenterAt("Click on phone button...", 160, 0);

    /* 创建按钮 */
    hButton = BUTTON_Create(142, 20, 36, 40, GUI_ID_OK, WM_CF_SHOW);

    /* 修改按钮属性 */
    BUTTON_SetBkColor(hButton, 1, GUI_RED);
    BUTTON_SetBitmapEx(hButton, 0, &bm_1bpp_0, 2, 4);
    BUTTON_SetBitmapEx(hButton, 1, &bm_1bpp_1, 2, 4);

    /* 循环，直到按钮按下 */
    while(GUI_GetKey() != GUI_ID_OK)
    {
        if(Stat ^= 1)
        {
            BUTTON_SetState(hButton, BUTTON_STATE_HASFOCUS |
                            BUTTON_STATE_INACTIVE);
        }
        else
        {
            BUTTON_SetState(hButton, BUTTON_STATE_HASFOCUS |
                            BUTTON_STATE_PRESSED);
        }
        GUI_Delay(500);
    }

    /* 删除按钮对象 */
    BUTTON_Delete(hButton);
}

/*****
*                                     主函数                                     *
*****/

void main(void)

```

```

{
    GUI_Init();
    DemoButton();
}

```

上面范例执行结果的截图



13.4 CHECKBOX：复选框控件

最常见的多选项的选择的控件中的一个复选框。一个复选框可由用户决定是选中还是未选中，在同一时间可以有多个复选框选中。如果复选框被禁止，它会以灰色显示。如下表所看到的一样，下表图示了四种可能的显示状态：

	选中	未选中
允许	<input checked="" type="checkbox"/>	<input type="checkbox"/>
禁止	<input checked="" type="checkbox"/>	<input type="checkbox"/>

所有与复选框相关的函数位于文件 CHECKBOX*.c, CHECKBOX.h 当中。所有标识符是前缀 CHECKBOX。

配置选项

类型	宏	默认值	说明
N	CHECKBOX_BKCOLOR0_DEFAULT	0x808080	禁止状态的背景颜色。
N	CHECKBOX_BKCOLOR1_DEFAULT	GUI_WHITE	允许状态的背景颜色。
N	CHECKBOX_FGCOLOR0_DEFAULT	0x101010	禁止状态的前景颜色。
N	CHECKBOX_FGCOLOR1_DEFAULT	GUI_BLACK	允许状态的前景颜色。
S	CHECKBOX_FONT_DEFAULT	&GUI_Font13_1	复选标志使用的字体。
B	CHECKBOX_USE_3D	1	3D 效果支持启用。

复选框 API 函数

下表按字母顺序列出了 μ C/GUI 与复选框相关的函数。函数详细的描述在下面给出。

函 数	说 明
CHECKBOX_Check()	将复选框设置为选中状态。
CHECKBOX_Create()	创建复选框。
CHECKBOX_CreateIndirect()	从资源表项目中创建复选框。
CHECKBOX_IsChecked()	返回复选框当前的状态（选中或未选中）。
CHECKBOX_Uncheck()	将复选框设置为未选中状态（默认）。

CHECKBOX_Check()

描述

将一个复选框设置为选中状态。

函数原型

```
void CHECKBOX_Check(CHECKBOX_Handle hObj);
```

参 数	含 意
hObj	复选框的句柄。

CHECKBOX_Create()

描述

在一个指定位置，以指定的大小建立一个复选框的控件。

函数原型

```
CHECKBOX_Handle CHECKBOX_Create(    int x0, int y0,
                                     int xsize, int ysize,
                                     WM_HWIN hParent,
                                     int ID,
                                     int Flags);
```

参 数	含 意
x0	复选框最左边的像素（在桌面坐标）。

y0	复选框最顶端的像素（在桌面坐标）。
xsize	复选框水平方向大小（以像素为单位）。
ysize	复选框垂直方向水平大小（以像素为单位）。
hParent	父窗口的句柄。
ID	返回的 ID。
Flags	窗口创建标识。具有代表性的是 WM_CF_SHOW，表示控件立即可见（请参阅第 12 章“视窗管理器”中的 WM_CreateWindow() 函数列出的可用的参数值）。

返回数值

建立的复选框的句柄，如果函数执行失败，则返回 0。

CHECKBOX_CreateIndirect()

函数原型的说明在本章开始部分。未使用当作参数传递的资源表的元素 Flags 和 Para。

CHECKBOX_IsChecked()

描述

返回一个指定的复选框控件当前的状态（选中或未选中）。

函数原型

```
int CHECKBOX_IsChecked(CHECKBOX_Handle hObj);
```

参 数	含 意
hObj	复选框的句柄。

返回数值

0：未选中；1：选中

CHECKBOX_Uncheck()

描述

设置一个指定的复选框状态为未选中。

函数原型

```
void CHECKBOX_Uncheck(CHECKBOX_Handle hObj);
```



参 数	含 意
<code>hObj</code>	复选框的句柄。

附加信息

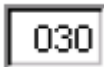

这是复选框的默认设置。

13.5 EDIT：文本编辑框控件

编辑区通常用作文本输入主要的用户接口：

空白编辑区	用户输入的编辑区
	

你也可能用编辑区输入二进制，十进制，或十六进制数值。一个十进制模式的编辑区可能与下表显示的相似。类似与复选框，编辑区如果被禁止的话，它会以灰色显示：

用户输入的编辑区（十进制）	禁止的编辑区
	

所有与编辑框相关的函数位于文件 `EDIT*.c`，`EDIT.h` 当中。所有标识符都是前缀 `EDIT`。

配置选项

类型	宏	默认值	说明
S	<code>EDIT_ALIGN_DEFAULT</code>	<code>GUI_TA_RIGHT</code> <code>GUI_TA_VCENTER</code>	编辑区域文本对齐方式
N	<code>EDIT_BKCOLOR0_DEFAULT</code>	<code>0xc0c0c0</code>	禁止状态的背景颜色。
N	<code>EDIT_BKCOLOR1_DEFAULT</code>	<code>GUI_WHITE</code>	允许状态的背景颜色。
N	<code>EDIT_BORDER_DEFAULT</code>	<code>1</code>	边框线宽（以像素为单位）
S	<code>EDIT_FONT_DEFAULT</code>	<code>&GUI_Font13_1</code>	编辑区域文本的字体。
N	<code>EDIT_TEXTCOLOR0_DEFAULT</code>	<code>GUI_BLACK</code>	禁止状态文本颜色。
N	<code>EDIT_TEXTCOLOR1_DEFAULT</code>	<code>GUI_BLACK</code>	允许状态文本颜色。
N	<code>EDIT_XOFF</code>	<code>2</code>	文本相对编辑区左边框的偏移量

可用的对齐标志：

水平对齐：GUI_TA_LEFT，GUI_TA_RIGHT，GUI_TA_HCENTER。

垂直对齐：GUI_TA_TOP，GUI_TA_BOTTOM，GUI_TA_VCENTER。

编辑框 API 函数

下表按字母的顺序列出了 μ C/GUI 编辑框相关的函数。函数的详细描述在稍后的部分给出。

函 数	说 明
EDIT_AddKey()	键盘输入函数。
EDIT_Create()	创建编辑区。
EDIT_CreateIndirect()	从资源表项目中创建编辑区。
EDIT_GetDefaultFont()	返回默认字体。
EDIT_GetText()	获得用户输入。
EDIT_GetValue()	返回当前数值。
EDIT_SetBinMode()	启用二进制编辑模式。
EDIT_SetBkColor()	设置编辑区的背景颜色。
EDIT_SetDecMode()	启用十进制编辑模式。
EDIT_SetDefaultFont()	设置用于编辑区的默认字体。
EDIT_SetDefaultTextAlign()	设置编辑区默认的文本对齐方式。
EDIT_SetFont()	给文本选择字体。
EDIT_SetHexMode()	启用十六进制模式。
EDIT_SetMaxLen()	设置编辑区的最大字符数量。
EDIT_SetText()	设置文本。
EDIT_SetTextAlign()	设置编辑区文本的对齐方式。
EDIT_SetTextColor()	设置文本的颜色。
EDIT_SetValue()	设置当前数值。
GUI_EditBin()	在当前光标位置编辑一个二进制数值。
GUI_EditDec()	在当前光标位置编辑一个十进制数值。
GUI_EditHex()	在当前光标位置编辑一个十六进制数值。
GUI_EditString()	在当前光标位置编辑一个字符串。

EDIT_AddKey()

描述

向一个指定编辑区输入内容。

函数原型

```
void EDIT_AddKey(EDIT_Handle hObj, int Key);
```

参 数	含 意
<code>hObj</code>	编辑区的句柄。
<code>Key</code>	输入的字符。

附加信息

向文本编辑框控件的用户输入区加入指定的字符。如果需要擦除最后一次显示的字符，你必须调用 `GUI_ID_BACKSPACE`。如果输入字符的数量已经到达最大值，其它字符将不会被加入。

EDIT_Create()

描述

在一个指定位置，以一个指定的大小创建一个文本编辑框控件。

函数原型

```
EDIT_Handle EDIT_Create(int x0, int y0, int xsize, int ysize, int ID, int MaxLen,
int Flags);
```

参 数	含 意
<code>x0</code>	编辑区最左边像素（桌面座标）。
<code>y0</code>	编辑区最顶部像素（桌面座标）。
<code>xsize</code>	编辑框水平方向尺寸（以像素为单位）。
<code>ysize</code>	编辑框垂直方向尺寸（以像素为单位）。
<code>ID</code>	返回的 ID。
<code>MaxLen</code>	字符的最大数量。
<code>Flags</code>	窗口创建标识。具有代表性的是 <code>WM_CF_SHOW</code> ，表示控件立即可见（请参阅第 12 章“视窗管理器”中的 <code>WM_CreateWindow()</code> 函数列出的可用的参数值）。

返回数值

建立的文本编辑框控件的句柄，如果函数执行失败，则返回 0。

EDIT_CreateIndirect()

原型的说明在本章开始部分。未使用资源表中以参数形式传递的元素 Flags，但是编辑区允许的最大字符数量指定为元素 Para。

EDIT_GetDefaultFont()

描述

设置用于文本编辑框控件的默认字体。

函数原型

```
const GUI_FONT* EDIT_GetDefaultFont(void);
```

返回数值

用于文本编辑框控件的默认字体。

EDIT_GetText()

描述

获得一个指定的编辑区的用户输入内容。

函数原型

```
void EDIT_GetText(EDIT_Handle hObj, char* sDest, int MaxLen);
```

参 数	含 意
hObj	编辑区的句柄。
sDest	缓冲区的指针。
MaxLen	缓冲区的大小。

EDIT_GetValue()

描述

返回编辑区当前的数值。当前数值只有在编辑区是二进制、十进制或十六制模式的情况

下才有用。

函数原型

```
I32 EDIT_GetValue(EDIT_Handle hObj);
```

参 数	含 意
<code>hObj</code>	编辑区的句柄。

返回数值

当前数值。

EDIT_SetBinMode()

描述

启用编辑区的二进制编辑模式。给出的数值可以在给出的范围内修改。

函数原型

```
void EDIT_SetBinMode(EDIT_Handle hObj, U32 Value, U32 Min, U32 Max);
```

参 数	含 意
<code>hObj</code>	编辑区的句柄。
<code>Value</code>	修改的数值。
<code>Min</code>	最小数值。
<code>Max</code>	最大数值。

EDIT_SetBkColor()

描述

设置编辑区颜色。

函数原型

```
void EDIT_SetBkColor(EDIT_Handle hObj, int Index, GUI_COLOR Color);
```

参 数	含 意
<code>hObj</code>	编辑区的句柄。

Index	必须设为 0，为将来的使用而保留。
Color	设置的顏色。

EDIT_SetDecMode()**描述**

启用编辑区的十进制编辑模式。给出的数值可以在给出的范围内修改。

函数原型

```
void EDIT_SetDecMode(EDIT_Handle hEdit, I32 Value, I32 Min, I32 Max, int Shift,
U8 Flags);
```

参 数	含 意
hObj	编辑区的句柄。
Value	修改的数值。
Min	最小值。
Max	最大值。
Shift	如果大于 0，它指定小数点的位置。
Flags	参阅下表。

参数 [Flags](#) 允许的数值（以“OR”组合）

0（默认）	在正常模式编辑，当数值为负数时符号才显示。
GUI_EDIT_SIGNED	显示“+”和“-”符号。

EDIT_SetDefaultFont()**描述**

设置用于编辑区的默认字体。

函数原型

```
void EDIT_SetDefaultFont(const GUI_FONT* pFont);
```

参 数	含 意
pFont	置为默认值的字体的指针。

EDIT_SetDefaultTextAlign()

描述

设置编辑区默认的文本对齐方式。

函数原型

```
void EDIT_SetDefaultTextAlign(int Align);
```

参 数	含 意
Align	默认的文本对齐方式（参阅 GUI_SetTextAlign() ）。

EDIT_SetFont()

描述

设置编辑区字体。

函数原型

```
void EDIT_SetFont(EDIT_Handle hObj, const GUI_FONT* pfont);
```

参 数	含 意
hObj	编辑区的句柄。
pFont	字体的指针。

EDIT_SetHexMode()

描述

启用编辑区的十六进制编辑模式。给出的数值可以在给出的范围内修改。

函数原型

```
void EDIT_SetHexMode(EDIT_Handle hObj, U32 Value, U32 Min, U32 Max);
```

参 数	含 意
hObj	编辑区的句柄。
Value	修改的数值。
Min	最小值。

Max	最大值。
-----	------

EDIT_SetMaxLen()

描述

设置绘出的编辑区能编辑的字符的最大数量。

函数原型

```
void EDIT_SetMaxLen(EDIT_Handle hObj, int MaxLen);
```

参 数	含 意
hObj	编辑区的句柄。
MaxLen	最大的字符数量。

EDIT_SetText()

描述

设置在编辑区中显示的文本。

函数原型

```
void EDIT_SetText(EDIT_Handle hObj, const char* s)
```

参 数	含 意
hObj	编辑区的句柄。
s	显示的文本。

EDIT_SetTextAlign()

描述

设置编辑区的文本对齐方式。

函数原型

```
void EDIT_SetTextAlign(EDIT_Handle hObj, int Align);
```

参 数	含 意
hObj	编辑区的句柄。

Align

默认的文本对齐方式（参阅 GUI_SetTextAlign()）。

EDIT_SetTextColor()

描述

设置编辑区文本颜色。

函数原型

```
void EDIT_SetBkColor(EDIT_Handle hObj, int Index, GUI_COLOR Color);
```

参 数	含 意
hObj	编辑区的句柄。
Index	必须设为 0，保留为将来使用。
Color	设置的颜色。

EDIT_SetValue()

描述

设置编辑区当前的数值。只有在设置了二进制，十进制或十六进制编辑模式的情况下才
有用。

函数原型

```
void EDIT_SetValue(EDIT_Handle hObj, I32 Value);
```

参 数	含 意
hObj	编辑区的句柄。
Value	新的数值。

GUI_EditBin()

描述

在当前光标位置编辑一个二进制数。

函数原型

```
U32 GUI_EditBin(U32 Value, U32 Min, U32 Max, int Len, int xsize);
```

参 数	含 意
Value	修改的数值。
Min	最小值。
Max	最大值。
Len	编辑的数字的数量。
xsize	编辑区 X 轴的尺寸（以像素为单位）。

返回数值

如果<ENTER>键按下，返回新的数值。如果<ESC>键按下，返回旧的数值。

附加信息

在<ENTER> 或 <ESC>键按下后，函数返回。给出文本的内容只有在<ENTER>键按下后才修改。

GUI_EditDec()

描述

在当前光标位置编辑一个十进制数。

函数原型

```
U32 GUI_EditDec (    I32 Value,
                    I32 Min, I32 Max,
                    int Len, int xsize, int Shift,
                    U8 Flags);
```

参 数	含 意
Value	修改的数值。
Min	最小值。
Max	最大值。
Len	编辑的数字的数量。
xsize	编辑区 X 轴的尺寸（以像素为单位）。
Shift	如果>0，它指定十进制数小数点的位置。

返回数值

如果<ENTER>键按下，返回新的数值。如果<ESC>键按下，返回旧的数值。

附加信息

在<ENTER> 或 <ESC>键按下后，函数返回。给出文本的内容只有在<ENTER>键按下后才修改。

GUI_EditHex()

描述

在当前光标位置编辑一个十六进制数。

函数原型

```
U32 GUI_EditHex(U32 Value, U32 Min, U32 Max, int Len, int xsize);
```

参 数	含 意
Value	修改的数值。
Min	最小值。
Max	最大值。
Len	编辑的数字的数量。
xsize	编辑区 X 轴的尺寸（以像素为单位）。

返回数值

如果<ENTER>键按下，返回新的数值。如果<ESC>键按下，返回旧的数值。

附加信息

在<ENTER> 或 <ESC>键按下后，函数返回。给出文本的内容只有在<ENTER>键按下后才修改。

GUI_EditString()

描述

在当前光标位置编辑一个字符串。

函数原型

```
void GUI_EditString(char * pString, int Len, int xsize);
```

参 数	含 意
<code>pString</code>	要编辑字符串的指针。
<code>Len</code>	字符的最大数量。
<code>xsize</code>	编辑区 X 轴的尺寸（以像素为单位）。

附加信息

在<ENTER> 或 <ESC>键按下后，函数返回。给出文本的内容只有在<ENTER>键按下后才修改。

范例

下面范例展示了编辑框控件的使用。范例文件是随μC/GUI 一起发布的范例当中的：

```

/*-----
文件:      WIDGET_Edit.c
目的:      展示文本编辑框控件使用的例子
-----*/

#include "gui.h" #include "edit.h"

/*****
*                  编辑一个字符串直到 ESC 键或 ENTER 键按下                  *
*****/

static int Edit(void)
{
    int Key;
    EDIT_Handle hEdit;
    char aBuffer[28] ;
    GUI_SetFont (&GUI_Font8x16);
    GUI_DispStringHCenterAt("Use keyboard to modify string...", 160, 0);
    /* 创建编辑框控件 */
    hEdit = EDIT_Create( 50, 20, 219, 25, ' ', sizeof(aBuffer), 0 );
    /* 修改编辑框控件 */
    EDIT_SetText(hEdit, "Press <ENTER> when done...");
    EDIT_SetFont(hEdit, &GUI_Font8x16);
    EDIT_SetTextColor(hEdit, 0, GUI_RED);
    /* 操作键盘直到 ESC 或 ENTER 键被按下 */

```

```

do
{
    Key = GUI_WaitKey();
    switch(Key)
    {
        case GUI_ID_ESCAPE:
        case GUI_ID_CANCEL:
            break;
        default:
            EDIT_AddKey(hEdit, Key);
    }
} while((Key != GUI_ID_ESCAPE) &&(Key != GUI_ID_ENTER) &&(Key != 0)); /* 从
编辑框控件取出结果 */

if(Key == GUI_ID_ENTER)
    EDIT_GetText(hEdit, aBuffer, sizeof(aBuffer));
else
    aBuffer[0] = 0;
EDIT_Delete(hEdit);
GUI_DispStringHCenterAt(aBuffer, 160, 50);
return Key;
}

/*****
*                                     *
*                                     *
*****/

void main(void)
{
    GUI_Init();
    Edit( ) ;
    while(1)
        GUI_Delay(100);
}

```

上面范例执行结果的屏幕截图



13.6 FRAMEWIN：框架窗口控件

框架窗口给你的应用程序一个 PC 的应用程序——窗口外形。它们由一个环绕的框架、一个标题栏和一人用户区组成。标题栏的颜色改变展示窗口是活动的还是非活动的，如下所示：

所有与框架窗口相关的函数位于文件 FRAMEWIN*.c 和 FRAMEWIN.h 当中。所有的标识符是前缀 FRAMEWIN。

配置选项

类型	宏	默认值	说明
N	FRAMEWIN_BARCOLOR_ACTIVE_DEFAULT	0xff0000	活动状态标题栏颜色。
N	FRAMEWIN_BARCOLOR_INACTIVE_DEFAULT	0x404040	非活动状态标题栏颜色。
N	FRAMEWIN_BORDER_DEFAULT	3	外部边界宽度（以像素为单位）。
N	FRAMEWIN_CLIENTCOLOR_DEFAULT	0xc0c0c0	窗口客户区颜色。
S	FRAMEWIN_DEFAULT_FONT	&GUI_Font8_1	标题栏文字使用的字体。
N	FRAMEWIN_FRAMECOLOR_DEFAULT	0xaaaaaa	框架颜色。
N	FRAMEWIN_IBORDER_DEFAULT	1	内部边界宽度（以像素为单位）。
B	FRAMEWIN_USE_3D	1	

框架窗口API函数

下表按字母顺序列出了μC/GUI 有效的与框架窗口相关的函数。函数的详细描述在后面部分给出。

函数	说明
FRAMEWIN_Create()	创建框架窗口。
FRAMEWIN_CreateAsChild()	创建一个作为一个子窗口的框架窗口控件。
FRAMEWIN_CreateIndirect()	从资源表条目中创建一个框架窗口。
FRAMEWIN_GetDefaultBorderSize()	返回默认边框尺寸。
FRAMEWIN_GetDefaultCaptionSize()	返回标题栏的默认尺寸。
FRAMEWIN_GetDefaultFont()	返回用于框架窗口标题的默认字体。
FRAMEWIN_SetActive()	设置框架窗口的状态。
FRAMEWIN_SetBarColor()	设置标题栏的背景颜色。

FRAMEWIN_SetClientColor()	设置客户区的背景。
FRAMEWIN_SetDefaultBarColor()	设置标题栏默认颜色。
FRAMEWIN_SetDefaultBorderSize()	设置默认边框尺寸。
FRAMEWIN_SetDefaultCaptionSize()	设置标题栏的默认高度。
FRAMEWIN_SetDefaultFont()	设置标题栏的默认字体。
FRAMEWIN_SetFont()	为标题文本选择字体。
FRAMEWIN_SetText()	设置标题文本。
FRAMEWIN_SetTextAlign()	设置标题文本的对齐方式。
FRAMEWIN_SetTextColor()	设置标题文本的颜色。
FRAMEWIN_SetTextPos()	设置标题文本的位置。

FRAMEWIN_Create()

描述

在一个指定位置以指定的尺寸创建一个框架窗口控件。

函数原型

```
FRAMEWIN_Handle FRAMEWIN_Create(    const char* pTitle,
                                     WM_CALLBACK* cb,
                                     int Flags,
                                     int x0, int y0,
                                     int xsize, int ysize);
```

参 数	含 意
pTitle	标题栏中显示的标题。
cb	为将来的应用保留。
Flags	窗口创建标识。具有代表性的是 WM_CF_SHOW，表示控件立即可见（请参阅第 12 章“视窗管理器”中的 WM_CreateWindow() 函数列出的可用的参数值）。
x0	框架窗口的 X 轴坐标。
y0	框架窗口的 Y 轴坐标。
xsize	框架窗口的 X 轴尺寸。
ysize	框架窗口的 Y 轴尺寸。

返回数值

所创建的框架窗口控件的句柄；如果函数执行失败则返回 0。

FRAMEWIN_CreateAsChild()

描述

创建一个作为一个子窗口的框架窗口控件。

函数原型

```
FRAMEWIN_Handle FRAMEWIN_CreateAsChild (    int x0, int y0,
                                             int xsize, int ysize,
                                             WM_HWIN hParent,
                                             const char* pText,
                                             WM_CALLBACK* cb,
                                             int Flags);
```

参 数	含 意
<code>x0</code>	框架窗口的 X 轴坐标。
<code>y0</code>	框架窗口的 Y 轴坐标。
<code>xsize</code>	框架窗口的 X 轴尺寸。
<code>ysize</code>	框架窗口的 Y 轴尺寸。
<code>hParent</code>	父窗口的句柄。
<code>pTitle</code>	标题栏中显示的文本。
<code>cb</code>	为将来的应用保留。

返回数值

所创建的框架窗口控件的句柄；如果函数执行失败则返回 0。

FRAMEWIN_CreateIndirect()

函数原型的说明在本章开始部分。下面的标志当作作为参数传递的资源的标志元素使用。

允许的间接创建标志

`FRAMEWIN_CF_MOVEABLE` 使框架窗口可移动（默认情况是固定的）。在资源表中未使用 Para 元素。

FRAMEWIN_GetDefaultBorderSize()

描述

返回框架窗口边框的默认尺寸。

函数原型

```
int FRAMEWIN_GetDefaultBorderSize(void);
```

返回数值

一个框架窗口边框的默认尺寸。

FRAMEWIN_GetDefaultCaptionSize()

描述

返回框架窗口标题栏的默认高度。

函数原型

```
int FRAMEWIN_GetDefaultCaptionSize(void);
```

返回数值

标题栏默认的高度。

FRAMEWIN_GetDefaultFont()

描述

返回用于框架窗口标题的默认字体。

函数原型

```
const GUI_FONT* FRAMEWIN_GetDefaultFont(void);
```

返回数值

用于框架窗口标题的默认字体。

FRAMEWIN_SetActive()

描述

设置框架窗口的状态。标题栏的颜色根据状态改变。

函数原型

```
void FRAMEWIN_SetActive(FRAMEWIN_Handle hObj, int State);
```

参 数	含 意
hObj	框架窗口的句柄。
State	框架窗口的状态（参阅下表）。

参数 [State](#) 允许的数值

0	框架窗口是非活动的。
1	框架窗口是活动的。

FRAMEWIN_SetBarColor()

描述

设置标题栏的背景颜色。

函数原型

```
void FRAMEWIN_SetBarColor(FRAMEWIN_Handle hObj, int Index, GUI_COLOR Color);
```

参 数	含 意
hObj	框架窗口的句柄。
Index	必须设为 0，为将来的使用而保留。
Color	作为标题栏背景颜色使用的颜色。

FRAMEWIN_SetClientColor()

描述

设置客户区的颜色。

函数原型

```
void FRAMEWIN_SetClientColor(FRAMEWIN_Handle hObj, GUI_COLOR Color);
```

参 数	含 意
hObj	框架窗口的句柄。
Color	设置的颜色。

FRAMEWIN_SetDefaultBarColor()

描述

设置标题栏的颜色。

函数原型

```
void FRAMEWIN_SetDefaultBarColor(int Index, GUI_COLOR Color);
```

参 数	含 意
Index	颜色索引（参阅下表）。
Color	设置的颜色。

参数 [Index](#) 允许的数值

0	设置框架窗口处于非活动状态时，使用的颜色。
1	设置框架窗口处于活动状态时，使用的颜色。

FRAMEWIN_SetDefaultBorderSize()

描述

设置框架窗口边框默认尺寸。

函数原型

```
void FRAMEWIN_SetDefaultBorderSize(int BorderSize);
```

参 数	含 意
BorderSize	设置的尺寸。

FRAMEWIN_SetDefaultCaptionSize()**描述**

设置标题栏 Y 轴尺寸。

函数原型

```
void FRAMEWIN_SetDefaultCaptionSize(int CaptionSize);
```

参 数	含 意
<code>CaptionSize</code>	设置的尺寸。

FRAMEWIN_SetDefaultFont()**描述**

设置用于显示标题的默认字体。

函数原型

```
void FRAMEWIN_SetDefaultFont(const GUI_FONT* pFont);
```

参 数	含 意
<code>pFont</code>	用作默认字体的字体的指针。

FRAMEWIN_SetFont()**描述**

设置标题字体。

函数原型

```
void FRAMEWIN_SetFont(FRAMEWIN_Handle hObj, const GUI_FONT* pfont);
```

参 数	含 意
<code>hObj</code>	框架窗口的句柄。
<code>pFont</code>	字体的指针。

FRAMEWIN_SetText()

描述

设置标题文本。

函数原型

```
void FRAMEWIN_SetText(FRAMEWIN_Handle hObj, const char* s);
```

参 数	含 意
hObj	框架窗口的句柄。
s	作为标题显示的文字。

FRAMEWIN_SetTextAlign()

描述

设置标题文本的对齐方式。

函数原型

```
void FRAMEWIN_SetTextAlign(FRAMEWIN_Handle hObj, int Align);
```

参 数	含 意
hObj	框架窗口的句柄。
Align	标题栏对齐属性（参阅下表）。

参数 Align 允许的值

GUI_TA_HCENTER	标题居中（默认）。
GUI_TA_LEFT	标题左对齐。
GUI_TA_RIGHT	标题右对齐。

附加信息

如果该函数没有调用，默认行为是显示居中的文本。

FRAMEWIN_SetTextColor()**描述**

设置标题文本的颜色。

函数原型

```
void FRAMEWIN_SetTextColor(FRAMEWIN_Handle hObj, GUI_COLOR Color);
```

参 数	含 意
<code>hObj</code>	框架窗口的句柄。
<code>Color</code>	作为标题文字颜色使用的颜色。

FRAMEWIN_SetTextPos()**描述**

设置标题文本的位置。

函数原型

```
void FRAMEWIN_SetTextPos(FRAMEWIN_Handle hObj, int XOff, int YOff);
```

参 数	含 意
<code>hObj</code>	框架窗口的句柄。
<code>XOff</code>	标题文字的 X 轴坐标。
<code>YOff</code>	标题文字的 Y 轴坐标。

范例

下面的范例展示如何使用框架控件。首选创建控件，修改几个它的属性，然后创建一个客户窗口。你不能使用框架控件本身显示任何东西，必须是始终要创建一个客户窗口。该范例文件是 `samples` 目录下的 `WIDGET_FrameWin.c`：

```

/*-----
文件：      WIDGET_FrameWin.c
目的：      展示一个框架窗口控件使用的例子
-----*/

#include "gui.H"
```

```

#include "framewin.h"
#include <stddef.h>

/*****
*                               客户窗口的回调函数                               *
*****/

static WM_RESULT CallbackChild(WM_MESSAGE * pMsg)
{
    WM_HWIN hWin =(FRAMEWIN_Handle) (pMsg->hWin);
    switch(pMsg->MsgId)
    {
        case WM_PAINT:
            /* Handle the paint message */
            GUI_SetBkColor(GUI_BLUE);
            GUI_SetColor(GUI_WHITE);
            GUI_SetFont(&GUI_FontComic24B_ASCII);
            GUI_SetTextAlign(GUI_TA_HCENTER | GUI_TA_VCENTER);
            GUI_Clear();
            GUI_DispStringHCenterAt(    "Child window",
                                      WM_GetWindowSizeX(hWin) / 2,
                                      WM_GetWindowSizeY(hWin) / 2);
            break;
    }
}

/*****
*                               创建框架及子窗口                               *
*****/

static void DemoFramewin(void)
{
    /* 创建框架窗口 */
    FRAMEWIN_Handle hFrame = FRAMEWIN_Create(    "Frame window",
                                                NULL, WM_CF_SHOW,
                                                10, 10, 150, 60);

    FRAMEWIN_SetFont(hFrame, &GUI_Font16B_ASCII);
    FRAMEWIN_SetTextColor(hFrame, GUI_RED);

```

```

FRAMEWIN_SetBarColor(hFrame, 0, GUI_GREEN);
FRAMEWIN_SetTextAlign(hFrame, GUI_TA_HCENTER);
/* 创建客户窗口 */
WM_CreateWindowAsChild(0, 0, 0, 0, hFrame, WM_CF_SHOW, CallbackChild, 0);
}

/*****
*                               *
*****/

void main(void)
{
    GUI_Init();
    DemoFramewin();
    while(1)
        GUI_Delay(100);
}

```

上面范例执行结果的屏幕截图



13.7 LISTBOX: 列表框控件

列表框用于在一个列表中选择一个元素。一个列表框可以以一个无边框的窗口的形式创建，如下所示，或者作为一个框架窗口控件的子窗口（参阅本部分末的附加屏幕截图）。当列表框的项目被选中，它们会以高亮显示。注意所选择项目的背景颜色依赖于列表框窗口是否有输入焦点。

带焦点的列表框	不带焦点的列表框

所有与列表框相关的函数位于文件 LISTBOX*.c, LISTBOX.h 当中。所有的标识符是前缀名 LISTBOX。

配置选项

类型	宏	默认值	说明
N	LISTBOX_BKCOLOR0_DEFAULT	GUI_WHITE	非选中状态的背景颜色。
N	LISTBOX_BKCOLOR1_DEFAULT	GUI_GRAY	没有焦点的选中状态的背景颜色。
N	LISTBOX_BKCOLOR2_DEFAULT	GUI_WHITE	有焦点的选中状态的背景颜色。
S	LISTBOX_FONT_DEFAULT	&GUI_Font13_1	使用的字体。
N	LISTBOX_TEXTCOLOR0_DEFAU	GUI_BLACK	非选中状态的文本颜色。
N	LISTBOX_TEXTCOLOR1_DEFAU	GUI_BLACK	没有焦点的选中状态的文本颜色。
N	LISTBOX_TEXTCOLOR2_DEFAU	GUI_BLACK	有焦点的选中状态的文本颜色。
B	LISTBOX_USE_3D	1	启用3D支持。

列表框API函数

下表根据字母的顺序列出了μC/GUI 与列表框有关的函数。函数详细的描述随后给出。

函数	说明
LISTBOX_Create()	创建列表框。
LISTBOX_CreateAsChild()	以子窗口的形式追寻列表框。
LISTBOX_CreateIndirect()	从资源表条目追寻列表框。
LISTBOX_DecSel()	减小选择范围。
LISTBOX_GetDefaultFont()	返回列表框的默认字体。
LISTBOX_GetSel()	返回所选择行的数目。
LISTBOX_IncSel()	增加选择范围。
LISTBOX_SetBackColor()	设置背景颜色。
LISTBOX_SetDefaultFont()	修改列表框控件的默认字体。
LISTBOX_SetFont()	选择字体。
LISTBOX_SetSel()	设置选择的行。
LISTBOX_SetTextColor()	设置前景颜色。

LISTBOX_Create()

描述

在指定位置，以指定的尺寸创建一个列表框控件。

函数原型

```
LISTBOX_Handle LISTBOX_Create( const GUI_ConstString* ppText,
                               int x0, int y0,
                               int xSize, int ySize,
                               int Flags);
```

参 数	含 意
<code>ppText</code>	包含有要显示的元素的一批字符串指针的指针。
<code>x0</code>	列表框 X 轴坐标。
<code>y0</code>	列表框 Y 轴坐标。
<code>xSize</code>	列表框 X 轴尺寸。
<code>ySize</code>	列表框 Y 轴尺寸。
<code>Flags</code>	窗口创建标志。典型的是 <code>WM_CF_SHOW</code> ，它使控件能立即可见（请参考第 12 章“视窗管理器”中的 <code>WM_CreateWindow()</code> ，其中有一列有效的参数值。）

返回数值

所创建的列表框控件的句柄，如果失败则为 0。

附加信息

如果参数 `ySize` 大于控件内容绘制所需要的间隔，则 Y 轴尺寸将被减小到要求的数值。对于参数 `xSize` 也是同一个道理。

LISTBOX_CreateAsChild()

描述

以一个子窗口的形式创建列表框控件。

函数原型

```
LISTBOX_Handle LISTBOX_CreateAsChild ( const GUI_ConstString* ppText,
                                       HBWIN hWinParent,
                                       int x0, int y0,
                                       int xSize, int ySize,
                                       int Flags);
```

参 数	含 意
<code>ppText</code>	包含有要显示的元素的一批字符串指针的指针。
<code>hWinParent</code>	父窗口句柄。
<code>x0</code>	列表框相对于父窗口的 X 轴坐标。
<code>y0</code>	列表框相对于父窗口的 Y 轴坐标。
<code>xSize</code>	列表框 X 轴尺寸。
<code>ySize</code>	列表框 Y 轴尺寸。
<code>Flags</code>	窗口创建标志（参阅 <code>LISTBOX_Create()</code> ）。

返回数值

所创建列表框控件的句柄，如果函数执行失败，则返回 0。

附加信息

如果参数 `ySize` 大于控件内容绘制所需要的间隔，则 Y 轴尺寸将被减小到要求的数值。如果 `ySize = 0`，控件的 Y 轴尺寸将被设置为来自父窗口的客户区的 Y 轴尺寸。对于参数 `Xsize` 也是同一个道理。

LISTBOX_CreateIndirect()

函数原型的说明在本章开始部分。资源表作为参数传递的元素 `Flags` 和 `Para` 未使用。

LISTBOX_DecSel()

描述

减小列表框选择范围（把指定列表框的选择条向上移动一个元素）。

函数原型 s

```
void LISTBOX_DecSel (LISTBOX_Handle hObj);
```

参 数	含 意
<code>hObj</code>	列表框的句柄。

LISTBOX_GetDefaultFont()

描述

返回用于创建列表框控件的默认字体。

函数原型

```
const GUI_FONT* LISTBOX_GetDefaultFont(void);
```

返回数值

默认字体的指针。

LISTBOX_GetSel()

描述

返回在一个指定列表框中当前选择的单元的数量。

函数原型

```
int LISTBOX_GetSel(LISTBOX_Handle hObj);
```

参 数	含 意
hObj	列表框的句柄。

返回数值

当前选择的单元的数量。

LISTBOX_IncSel()

描述

增加列表框选择范围（把指定列表框的选择条向下移动一元素）。

函数原型

```
void LISTBOX_IncSel(LISTBOX_Handle hObj);
```

参 数	含 意
hObj	列表框的句柄。

LISTBOX_SetBackColor()

描述

设置列表框背景颜色。

函数原型

```
void LISTBOX_SetBackColor(LISTBOX_Handle hObj, int Index, GUI_COLOR Color);
```

参 数	含 意
hObj	列表框的句柄。
Index	背景颜色的索引（参阅下表）。
Color	设置的颜色。

参数 [Index](#) 允许的值

0	为未选择的单元设置颜色。
1	为选择的单元设置颜色。

LISTBOX_SetDefaultFont()

描述

设置用于列表框控件的默认字体。

函数原型

```
void LISTBOX_SetDefaultFont(const GUI_FONT* pFont)
```

参 数	含 意
pFont	字体的指针。

LISTBOX_SetFont()

描述

设置列表框字体。

函数原型

```
void LISTBOX_SetFont(LISTBOX_Handle hObj, const GUI_FONT* pfont);
```

参 数	含 意
hObj	列表框的句柄。
pFont	字体的指针。

LISTBOX_SetSel()

描述

设置指定列表框选择的单元。

函数原型

```
void LISTBOX_SetSel(LISTBOX_Handle hObj, int Sel);
```

参 数	含 意
hObj	列表框的句柄。
Sel	选择的单元。

LISTBOX_SetTextColor()

描述

设置列表框文本颜色。

函数原型

```
void LISTBOX_SetTextColor(LISTBOX_Handle hObj, int Index, GUI_COLOR Color);
```

参 数	含 意
hObj	列表框的句柄。
Index	文本颜色的索引（参考 LISTBOX_SetBackColor()）。
Color	设置的颜色。

范例

使用列表框控件

下面的范例展示了列表框控件的简单使用。其源代码文件是范例 WIDGET_SimpleListBox.c:

```

/*-----
文件:      WIDGET_SimpleListBox.c
目的:      展示一个列表框控件使用的例子
-----*/

#include "gui.H" #include "listbox.h" #include <stddef.h>
const GUI_ConstString ListBox[] =
{
    "English", "Deutsch", "Français", "Japanese", "Italiano", NULL
};
#define countof(Array) (sizeof(Array) / sizeof(Array[0]))

/******
*                               列表框使用的示范                               *
******/

void DemoListBox(void)
{
    int i;
    int Entries = countof(ListBox) - 1;
    LISTBOX_Handle hListBox;
    int ySize = GUI_GetFontSizeYOf(&GUI_Font13B_1) * Entries;
    /* 建立列表框 */
    hListBox = LISTBOX_Create(ListBox, 10, 10, 60, ySize, WM_CF_SHOW);
    /* 改变列表框当前的选择 */
    for(i = 0; i < Entries-1; i++)
    {
        GUI_Delay(500);
        LISTBOX_IncSel(hListBox);
        WM_Exec();
    }
    for(i = 0; i < Entries-1; i++)
    {
        GUI_Delay(500);
        LISTBOX_DecSel(hListBox);
        WM_Exec();
    }
}

```

```

/* 删除列表框控件 */
LISTBOX_Delete(hListBox);
GUI_Clear();
}

/*****
*                               *
*               列表框控件示范               *
*****/

static void DemoListbox(void)
{
    GUI_SetBkColor(GUI_BLUE);
    GUI_Clear();
    while(1)
    {
        DemoListBox();
    }
}

/*****
*                               *
*               主函数                               *
*****/

void main(void)
{
    GUI_Init();
    DemoListbox();
}

```

上面范例执行结果的屏幕截图



使用一个带边框或无边框的列表框控件

下面的范例展示了列表框控件在有框架窗口和没有框架窗口的情况下的使用。其源代码文件是 WIDGET_ListBox.c:

```

/*-----
文件:      WIDGET_ListBox.c
目的:      展示一个列表框控件使用的例子
-----*/

#include "gui.H"
#include "framewin.h" #include "listbox.h" #include <stddef.h>
const GUI_ConstString ListBox[] =
{
    "English", "Deutsch", "Français", "Japanese", "Italiano", NULL
};
#define countof(Array) (sizeof(Array) / sizeof(Array[0] ) )

/*-----
*                                     单独使用列表框控件                                     *
*****
*/

void DemoListBox(void)
{
    int i;
    int Entries = countof(ListBox) - 1;
    LISTBOX_Handle hListBox;
    int ySize = GUI_GetFontSizeYOf(&GUI_Font13B_1) * Entries;
    /* 创建列表框 */
    hListBox = LISTBOX_Create(ListBox, 10, 10, 120, ySize, WM_CF_SHOW);
    /* 修改列表框属性 */
    LISTBOX_SetFont(hListBox, &GUI_Font13B_1);
    LISTBOX_SetBackColor(hListBox, 0, GUI_BLUE);
    LISTBOX_SetBackColor(hListBox, 1, GUI_LIGHTBLUE);
    LISTBOX_SetTextColor(hListBox, 0, GUI_WHITE);
    LISTBOX_SetTextColor(hListBox, 1, GUI_BLACK);
    /* 改变列表框当前的选择 */
    for(i = 0; i < Entries - 1; i++)
    {
        GUI_Delay(500);
    }
}

```

```

        LISTBOX_IncSel(hListBox);
    }
    for(i = 0; i < Entries - 1; i++)
    {
        GUI_Delay(500);
        LISTBOX_DecSel(hListBox);
    }
    GUI_Delay(500);
    /* 删除列表框控件 */
    LISTBOX_Delete(hListBox);
    GUI_Clear();
}

/*****
*                把列表框控件当作一个框架窗口控件的子窗口使用                *
*****/

void DemoListBoxAsChild(void)
{
    int i;
    int Entries = countof(ListBox) - 1;
    FRAMEWIN_Handle hFrame;
    LISTBOX_Handle hListBox;
    int ySize = GUI_GetFontSizeYOf(&GUI_Font13B_1)
                * Entries + GUI_GetFontSizeYOf(&GUI_Font16B_ASCII) + 7;
    /* 创建框架窗口 */
    hFrame = FRAMEWIN_Create("List box", NULL, WM_CF_SHOW, 10, 10, 120, ySize);
    /* 修改框架窗口属性 */
    FRAMEWIN_SetFont(hFrame, &GUI_Font16B_ASCII);
    FRAMEWIN_SetTextColor(hFrame, GUI_RED);
    FRAMEWIN_SetBarColor(hFrame, 0, GUI_GREEN);
    FRAMEWIN_SetTextAlign(hFrame, GUI_TA_HCENTER);
    /* 创建列表框 */
    hListBox = LISTBOX_CreateAsChild(ListBox, hFrame, 0, 0, 0, 0, WM_CF_SHOW);
    /* 修改列表框属性 */
    LISTBOX_SetBackColor(hListBox, 0, GUI_BLUE);
    LISTBOX_SetBackColor(hListBox, 1, GUI_LIGHTBLUE);

```

```

LISTBOX_SetTextColor(hListBox, 0, GUI_WHITE);
LISTBOX_SetTextColor(hListBox, 1, GUI_BLACK);
LISTBOX_SetFont(hListBox, &GUI_Font13B_1);
/* 改变列表框当前的选择 */
for(i = 0; i < Entries - 1; i++)
{
    GUI_Delay(500);
    LISTBOX_IncSel(hListBox);
}
for(i = 0; i < Entries - 1; i++)
{
    GUI_Delay(500);
    LISTBOX_DecSel(hListBox);
} GUI_Delay(500);
/* 删除列表框控件 */
LISTBOX_Delete(hListBox);
/* 删除框架窗口控件 */
FRAMEWIN_Delete(hFrame);
GUI_Clear();
}

/*****
*                               *
*                               *
*****/

static void DemoListbox(void)
{
    GUI_SetBkColor(GUI_GRAY);
    GUI_Clear();
    while(1)
    {
        DemoListBox();
        DemoListBoxAsChild();
    }
}

/*****

```

```

*                               主函数                               *
*****/

void main(void)
{
    GUI_Init();
    DemoListbox();
}

```

上面范例执行结果的屏幕截图



13.8 PROGBAR: 进度条控件

进度条通常在可视化应用中使用。例如，一个容器的填充容积指示器或一个油压指示器。例子的屏幕截图可以在本章开始部分和本部分末。所有与进度条相关的函数在文件 PROGBAR*.c, PROGBAR.h 当中。所有的标识符是前缀 PROGBAR。

配置选项

类型	宏	默认值	说明
S	PROGBAR_DEFAULT_FONT	GUI_DEFAULT_FONT	使用的字体。
N	PROGBAR_DEFAULT_BARCOLOR0	0x555555 （深灰）	左侧条棒的颜色。
N	PROGBAR_DEFAULT_BARCOLOR1	0xAAAAAA （浅灰）	右侧条棒的颜色。
N	PROGBAR_DEFAULT_TEXTCOLOR0	0xFFFFFFFF	左侧条棒文字的颜色。
N	PROGBAR_DEFAULT_TEXTCOLOR1	0x000000	右侧条棒文字的颜色。

进度条API 函数

下表按字母的顺序列出了μC/GUI 与进度条有关的函数。函数详细的描述在后面给出。

函数	说明
PROGBAR_Create()	创建进度条。
PROGBAR_CreateIndirect()	从资源表条目创建进度条。
PROGBAR_SetBarColor()	设置进度条的颜色。
PROGBAR_SetFont()	选择文字使用的字体。
PROGBAR_SetMinMax()	设置进度条的最小和最大数值。
PROGBAR_SetText()	设置（可选择）条棒图形的文字。bargraph.
PROGBAR_SetTextAlign()	设置对齐方式（默认为居中）。
PROGBAR_SetTextColor()	设置文字的颜色。
PROGBAR_SetTextPos()	设置坐标（默认为 0,0）。
PROGBAR_SetValue()	设置条棒图形的数值（如果没有文字则显示百分比）。

PROGBAR_Create()

描述

在一个指定位置，以指定的尺寸追寻一个进度条。

函数原型

```
PROGBAR_Handle PROGBAR_Create(int x0, int y0, int xsize, int ysize, int Flags);
```

参 数	含 意
x0	进度条最左边像素（在桌面座标）。
y0	进度条最顶部的像素（在桌面座标）。
xsize	进度条水平方向的尺寸（以像素为单位）。
ysize	进度条垂直方向的尺寸（以像素为单位）。
Flags	窗口创建标志。典型的是 WM_CF_SHOW，它使控件能立即可见（请参考第 12 章“视窗管理器”中的 WM_CreateWindow()，其中有一列有效的参数值。）

返回数值

所创建的进度条控件的句柄；如果函数执行失败则为 0。

PROGBAR_CreateIndirect()

函数原型的说明在本章开始部分。作为参数传递的资源表的元素 Flags 和 Para 未使用到。

PROGBAR_SetBarColor()

描述

设置进度条的颜色。

函数原型

```
void PROGBAR_SetBarColor(PROGBAR_Handle hObj, int Index, GUI_COLOR Color);
```

参 数	含 意
hObj	进度条的句柄。
Index	参阅下表。其它数值是不允许的。
Color	设置的颜色（24 位 RGB 数值）。

参数 [Index](#) 允许的值

0	进度条左侧部分。
1	进度条右侧部分。

PROGBAR_SetFont()

描述

选择进度条当中显示文本的字体。

函数原型

```
void PROGBAR_SetFont(PROGBAR_Handle hObj, const GUI_FONT* pFont);
```

参 数	含 意
hObj	进度条的句柄。
pFont	字体的指针。

附加信息

如果该函数没有被调用，将使用进度条默认字体（GUI 默认字体）。不过，进度默认字体可以在 GUIConf.h 文件中修改。

如下所示，简单地用“#define”定义默认字体：

```
#define PROGBAR_DEFAULT_FONT &GUI_Font13_ASCII
```

PROGBAR_SetMinMax()

描述

设置用于进度条的最小数值和最大数值。

函数原型

```
void PROGBAR_SetMinMax(PROGBAR_Handle hObj, int Min, int Max);
```

参 数	含 意
hObj	进度条的句柄。
Min	最小值，范围：-16383 < Min ≤ 16383。
Max	最大值，范围：-16383 < Max ≤ 16383。

附加信息

如果该函数未被调用，将使用默认数值 [Min](#) = 0, [Max](#) = 100。

PROGBAR_SetText()

描述

设置进度条当中显示的文本。

函数原型

```
void PROGBAR_SetText(PROGBAR_Handle hObj, const char* s);
```

参 数	含 意
hObj	进度条的句柄。
s	显示的文本。允许一个空指针；在这种情况下显示一个百分数。

附加信息

如果该函数未被调用，一个百分比数值会作为默认值显示。如果你不打算显示任何文本，你应当设置一个空字符串。

PROGBAR_SetTextAlign()

描述

设置文本对齐方式。

函数原型

```
void PROGBAR_SetTextAlign(PROGBAR_Handle hObj, int Align);
```

参 数	含 意
hObj	进度条的句柄。
Align	文本的水平对齐属性（参阅下表）。

参数 [Align](#) 允许的值

GUI_TA_HCENTER	标题对中（默认）。
GUI_TA_LEFT	在左侧显示标题。
GUI_TA_RIGHT	在右侧显示标题。

附加信息

如果该函数未被调用，默认的行为是文本对中显示。

PROGBAR_SetTextPos()

描述

以像素为单位设置文本的坐标。

函数原型

```
void PROGBAR_SetTextPos(PROGBAR_Handle hObj, int XOff, int YOff);
```

参 数	含 意
hObj	进度条的句柄。
XOff	水平方向文本移动的像素数。该值为正值则文本向右侧移动。
YOff	垂直方向文本移动的像素数。该值为正值则文本向下移动。

附加信息

该数值在控件内移动文本到指定像素数处。通常，默认值（0, 0）应该足够了。

PROGBAR_SetValue()

描述

设置进度条的数值。

函数原型

```
void PROGBAR_SetValue(PROGBAR_Handle hObj, int v);
```

参 数	含 意
<code>hObj</code>	进度条的句柄。
<code>v</code>	设置的数值。

附加信息

进度条指示由最大/最小值的关系计算得来。如果自动显示一个百分数，该百分数同样可以由给出的最大/最小值算出，如下所示：

$$p = 100\% * (v - \text{Min}) / (\text{Max} - \text{Min})$$

控件创建后的默认数值为 0。

范例

使用进度条控件

下面的简单范例展示进度条控件的使用。其源代码文件是范例中的 WIDGET_SimpleProgbar.c:

```
/*-----  
文件:      WIDGET_SimpleProgbar.c  
目的:      展示一个进度条控件使用的例子  
-----*/  
  
#include "gui.h"  
#include "progbar.h"
```

```

/*****
*
*                               展示进度条的使用
*
*****/

static void DemoProgBar(void)
{
    int i;
    PROGBAR_Handle ahProgBar;
    GUI_Clear();
    GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringAt("Progress bar", 100, 80);

    /* 创建进度条 */
    ahProgBar = PROGBAR_Create(100, 100, 100, 20, WM_CF_SHOW);
    GUI_Delay(500);

    /* 修改进度条 */
    for(i = 0; i <= 100; i++)
    {
        PROGBAR_SetValue(ahProgBar, i);
        GUI_Delay(10);
    }
    GUI_Delay(500);

    /* 删除进度条 */
    PROGBAR_Delete(ahProgBar);
}

/*****
*
*                               主函数
*
*****/

void main(void)
{
    GUI_Init();
    while(1)
    {
        DemoProgBar();
    }
}

```

上面范例执行结果的屏幕截图



进度条控件的高级应用

这个更高级的范例创建一个容器的填充级别指示器。其源代码文件是范例中的 WIDGET_Progbar.c:

```
/*-----
文件:      WIDGET_Progbar.c
目的:      进度条控件使用的简单演示
-----*/

#include "gui.h"
#include "progbars.h"

/******
*                      展示进度条的使用                      *
******/

static void DemoProgBar(void)
{
    int i;
    PROGBAR_Handle ahProgBar[2] ;
    GUI_Clear();
    GUI_SetColor(GUI_WHITE);
    GUI_SetFont(&GUI_Font8x16);
    GUI_DispStringAt("Progress bar", 100, 80);
    /* 创建全身 */
    ahProgBar[0] = PROGBAR_Create(100, 100, 100, 20, WM_CF_SHOW);
    ahProgBar[1] = PROGBAR_Create( 80, 150, 140, 10, WM_CF_SHOW);
    /* 使用存储设备（可选，为了看起来更好一些）*/
    PROGBAR_EnableMemdev(ahProgBar[0]);
    PROGBAR_EnableMemdev(ahProgBar[1]);
}
```

```
GUI_Delay(1000);
PROGBAR_SetMinMax(ahProgBar[1], 0, 500);
while(1)
{
    PROGBAR_SetFont(ahProgBar[0], &GUI_Font8x16);
    #if(LCD_BITSPERPIXEL <= 4)
        PROGBAR_SetBarColor(ahProgBar[0], 0, GUI_DARKGRAY);
        PROGBAR_SetBarColor(ahProgBar[0], 1, GUI_LIGHTGRAY);
    #else
        PROGBAR_SetBarColor(ahProgBar[0], 0, GUI_GREEN);
        PROGBAR_SetBarColor(ahProgBar[0], 1, GUI_RED);
    #endif
    for(i=0; i<=100; i++)
    {
        PROGBAR_SetValue(ahProgBar[0], i);
        PROGBAR_SetValue(ahProgBar[1], i);
        GUI_Delay(5);
    }
    PROGBAR_SetText(ahProgBar[0], "Tank empty");
    for(; i>=0; i--)
    {
        PROGBAR_SetValue(ahProgBar[0], i);
        PROGBAR_SetValue(ahProgBar[1], 200-i);
        GUI_Delay(5);
    }
    PROGBAR_SetText(ahProgBar[0], "Any text ...");
    PROGBAR_SetTextAlign(ahProgBar[0], GUI_TA_LEFT);
    for(; i<=100; i++)
    {
        PROGBAR_SetValue(ahProgBar[0], i);
        PROGBAR_SetValue(ahProgBar[1], 200+i);
        GUI_Delay(5);
    }
    PROGBAR_SetTextAlign(ahProgBar[0], GUI_TA_RIGHT);
    for(; i>=0; i--)
    {
        PROGBAR_SetValue(ahProgBar[0], i);
```

```

        PROGBAR_SetValue(ahProgBar[1], 400-i); GUI_Delay(5);
    }
    PROGBAR_SetFont(ahProgBar[0], &GUI_FontComic18B_1);
    PROGBAR_SetText(ahProgBar[0], "Any font ...");
    for(; i<=100; i++)
    {
        PROGBAR_SetValue(ahProgBar[0], i);
        PROGBAR_SetValue(ahProgBar[1], 400+i);
        GUI_Delay(5);
    }
    GUI_Delay(1000);
}

/*****
*                               *
******/

void main(void)
{
    GUI_Init() ;
    DemoProgBar();
}

```

上面范例执行结果的屏幕截图



13.9 RADIO: 单选按钮控件

单选按钮，类似于复选框，用于选项的选择。当一个单选按钮开启或被选中后，在它上

面会显示一个点。与复选框不同的是在同一时间，用户只能选择一个单选按钮。当一个按钮被选择时，控件中其它的按钮被关闭，如下图所示。一个单选按钮控件可以包括几个按钮，它们总是垂直排列的。



与单选框相关的函数位于文件 RADIO*.c，RADIO.h 中。所有的标识符是前缀 RADIO。

配置选项

类型	宏	默认值	说明
	RADIO_BKCOLOR0_DEFAULT	0xc0c0c0	非活动状态按钮的背景颜色。
	RADIO_BKCOLOR1_DEFAULT	GUI_WHITE	活动状态按钮的背景颜色。

单选按钮控件 API 函数

下表按字母顺序列出了 μ C/GUI 与单选按钮有关的函数。函数详细的描述在稍后给出。

函数	说明
RADIO_Create()	建立一组单选按钮。
RADIO_CreateIndirect()	从资源表条目创建一组单选按钮。
RADIO_Dec()	将按钮选择序号减 1。
RADIO_GetValue()	返回当前选择的按钮。
RADIO_Inc()	将按钮选择序号加 1。
RADIO_SetValue()	设置选择的按钮。

RADIO_Create()

描述

在一个指定位置，以指定的大小创建一个单选按钮控件。

函数原型

```
RADIO_Handle RADIO_Create ( int x0, int y0,
                             int xsize, int ysize,
                             WM_HWIN hParent,
```

```
int Id, int Flags, unsigned Para);
```

参 数	含 意
x0	单选按钮最左侧像素（在桌面坐标）。
y0	单选按钮最顶部像素（在桌面坐标）。
xsize	单选按钮水平尺寸（以像素为单位）。
ysize	单选按钮垂直尺寸（以像素为单位）。
hParent	父窗口句柄。
Id	返回的 ID 号。
Flags	窗口创建标志。典型的是 WM_CF_SHOW 用于使控件立即可见（请参阅第 12 章“视窗管理器”中的 WM_CreateWindow() 所列出的参数值）。
Para	按钮在组当中的数值。

返回数值

所创建的单选按钮控件的句柄；如果函数执行失败则为 0。

RADIO_CreateIndirect()

函数原型的说明在本章的开始部分。资源表作为参数传递的元素 Flags 未使用，但是所使用的单选按钮号码指定为 Para 元素。

RADIO_Dec()

描述

把选择数值减 1。

函数原型

```
void RADIO_Dec(RADIO_Handle hObj);
```

参 数	含 意
hObj	单选按钮控件的的句柄。

附加信息

请注意，按钮的编号总是从顶部以 0 开始；因此，选择按钮的序号减 1 实际上将选择的按钮往上移一个。

RADIO_GetValue()

描述

返回当前选择的按钮。

函数原型

```
void RADIO_GetValue(RADIO_Handle hObj);
```

参 数	含 意
hObj	单选按钮控件的的句柄。

返回数值

当前选择按钮的值。

RADIO_Inc()

描述

选择数值加 1。

函数原型

```
void RADIO_Inc(RADIO_Handle hObj);
```

参 数	含 意
hObj	单选按钮控件的的句柄。

附加信息

请注意，按钮的编号总是从顶部以 0 开始；因此，选择按钮的序号加 1 实际上将选择的按钮往下移一个。

RADIO_SetValue()

描述

设置当前选择的按钮。

函数原型

```
void RADIO_SetValue(RADIO_Handle hObj, int v);
```

参 数	含 意
<code>hObj</code>	单选按钮控件的的句柄。
<code>v</code>	设置的数值。

附加信息

单选按钮控件最顶部的单选按钮的序号总是 0，往下一个按钮总是 1，再往下一个是 2，等等。

13.10 SCROLLBAR：滚动条控件

滚动条用于滚动一个列表框或任何没有完全适合它们框架的窗口。它们可以以水平或垂直方式创建。



典型的，一个滚动条依附在一个存在的窗口，如下面展示列表框例子：



所有与滚动条相关的函数位于文件 `SCROLLBAR*.c`，`SCROLLBAR.h` 当中。所有的标识符是前缀名 `SCROLLBAR`。

配置选项

类型	宏	默认值	说明
N	<code>SCROLLBAR_BKCOLOR0_DEFA</code>	<code>0x808080</code>	背景（thumb 区域）颜色。
N	<code>SCROLLBAR_BKCOLOR1_DEFA</code>	<code>GUI_BLACK</code>	滚动条（thumb）颜色。
N	<code>SCROLLBAR_COLOR0_DEFAULT</code>	<code>0xc0c0c0</code>	箭头按钮的颜色。
B	<code>SCROLLBAR_USE_3D</code>	1	启用 3D 支持。

滚动条 API

下表列出了 μ C/GUI 与滚动条相关函数的，函数详细的描述在稍后给出。

函数	说明
SCROLLBAR_AddValue()	滚动条的数值增加或减少一个指定的值。
SCROLLBAR_Create()	创建滚动条。
SCROLLBAR_CreateAttached()	创建一个吸附到一个窗口的滚动条。
SCROLLBAR_CreateIndirect()	从资源表条目中创建滚动条。
SCROLLBAR_Dec()	滚动条数值减 1。
SCROLLBAR_GetValue()	返回当前条目的数值。
SCROLLBAR_Inc()	滚动条数值加 1。
SCROLLBAR_SetNumItems()	设置滚动条目的值。
SCROLLBAR_SetPageSize()	设置页尺寸（按条目的数量）。
SCROLLBAR_SetValue()	设置滚动条当前数值。
SCROLLBAR_SetWidth()	设置滚动条的宽度。

SCROLLBAR_AddValue()

定义

滚动条的数值增加或减少一个指定的值。

函数原型

```
void SCROLLBAR_AddValue(SCROLLBAR_Handle hObj, int Add);
```

参 数	含 意
hObj	滚动条控件的句柄。
Add	在一个时间条目增加或减少的数值。

附加信息

滚动条的数值不能超过函数 [SCROLLBAR_SetNumItems\(\)](#) 当中设置的值。例如，如果窗口包括 200 个条目，当前滚动条数值是 195，滚动条增加 3 个条目，它将移动到数值 198。然而，增加 10 个条目最多也只能移动到数值 200，即该特定窗口的最大数值。

SCROLLBAR_Create()**描述**

在指定位置，以指定大小创建一个滚动条控件。

函数原型

```
SCROLLBAR_Handle SCROLLBAR_Create ( int x0, int y0,
                                     int xsize, int ysize,
                                     WM_HWIN hParent, int Id,
                                     int WinFlags, int SpecialFlags);
```

参 数	含 意
x0	滚动条最左侧像素（在桌面座标）。
y0	滚动条最顶部像素（在桌面座标）。
xsize	滚动条水平尺寸（以像素为单位）。
ysize	滚动条垂直尺寸（以像素为单位）。
hParent	父窗口句柄。
Id	返回的 ID 号。
WinFlags	窗口创建标志。典型的是 WM_CF_SHOW 用于使控件立即可见（请参阅第 12 章“视窗管理器”中的 WM_CreateWindow() 所列出的参数值）。
SpecialFlags	指定的创建标志（参阅 SCROLLBAR_CreateIndirect() 下的间接创建标志）。

返回数值

所创建的滚动条的句柄；如果函数执行失败，则为 0。

SCROLLBAR_CreateAttached()**描述**

依附一个已存在的窗口创建一个滚动条。

函数原型

```
SCROLLBAR_Handle SCROLLBAR_CreateAttached(WM_HWIN hParent, int SpecialFlags);
```

参 数	含 意
hParent	父窗口的句柄。

SpecialFlags	特定的创建标志(参阅在 SCROLLBAR_CreateIndirect() 下的间接创建标志)。
--------------	---

返回数值

所创建滚动条控件的句柄；如果函数执行失败则为 0。

附加信息

一个依附的滚动条实质上是一个把它自己放置在父窗口上并因而起作用的子窗口。垂直依附滚动条将自动放置在父窗口的右侧；水平滚动条放置在底部。

范例

创建一个带有滚动条的列表框：

```
LISTBOX_Handle hListBox;
hListBox = LISTBOX_Create(ListBox, 50, 50, 100, 100, WM_CF_SHOW);
SCROLLBAR_CreateAttached(hListBox, SCROLLBAR_CF_VERTICAL);
```

上面范例执行结果的屏幕截图

左图展示的是创建后显示的列表框样子；右图展示依附有垂直滚动条的列表框。



SCROLLBAR_CreateIndirect()

函数原型的说明在本章开始部分。下面的标志当作作为参数传递的资源表的元素 Flags 使用。

允许的间接创建标志（以”OR”组合）

SCROLLBAR_CF_VERTICAL	创建一个垂直滚动条（默认值是水平）。
SCROLLBAR_CF_FOCUSSABLE	给滚动条输入焦点。

在资源表中 Para 元素未使用。

SCROLLBAR_Dec()

描述

滚动条当前数值减 1。

函数原型

```
void SCROLLBAR_Dec(SCROLLBAR_Handle hObj);
```

参 数	含 意
hObj	滚动条控件的句柄。

附加信息

一个“条目”的定义是特定的应用，尽管在大部分情况下它等于一行。条目编号顺序从上到下或从左到右，以数值 0 开始。

SCROLLBAR_GetValue()

描述

返回当前条目的数值。

函数原型

```
int SCROLLBAR_GetValue(SCROLLBAR_Handle hObj);
```

参 数	含 意
hObj	滚动条控件的句柄。

返回数值

当前条目的数值。

SCROLLBAR_Inc()

描述

滚动条当前数值加 1。

函数原型

```
void SCROLLBAR_Inc(SCROLLBAR_Handle hObj);
```

参 数	含 意
<code>hObj</code>	滚动条控件的句柄。

附加信息

一个“条目”的定义是特定的应用，尽管在大部分情况下它等于一行。条目编号顺序从上到下或从左到右，以数值 0 开始。

SCROLLBAR_SetNumItems()

描述

设置滚动的条目的数量。

函数原型

```
void SCROLLBAR_SetNumItems(SCROLLBAR_Handle hObj, int NumItems);
```

参 数	含 意
<code>hObj</code>	滚动条控件的句柄。
<code>NumItems</code>	设置条目的数量。

附加信息

一个“条目”的定义是特定的应用，尽管在大部分情况下它等于一行。

指定的条目数是最大值；滚动条不能超过该值。

SCROLLBAR_SetPageSize()

描述

设置页尺寸。

函数原型

```
void SCROLLBAR_SetPageSize(SCROLLBAR_Handle hObj, int PageSize);
```

参 数	含 意
<code>hObj</code>	滚动条控件的句柄。
<code>PageSize</code>	页尺寸（条目的数量）。

附加信息

页尺寸由一页条目的数量指定。如果用户需要向上或向下翻页，可以使用键盘或通过在滚动条区域点击鼠标，窗口将会按一页指定的条目数量向上或向下滚动。

SCROLLBAR_SetValue()

描述

设置滚动条当前的值。

函数原型

```
void SCROLLBAR_SetValue(SCROLLBAR_Handle hObj, int v) ;
```

参 数	含 意
<code>hObj</code>	滚动条控件的句柄。
<code>v</code>	设置的数值。

SCROLLBAR_SetWidth()

描述

设置滚动条的宽度。

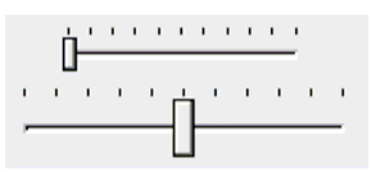
函数原型

```
void SCROLLBAR_SetWidth(SCROLLBAR_Handle hObj, int Width);
```

参 数	含 意
<code>hObj</code>	滚动条控件的句柄。
<code>Width</code>	设置的宽度。

13.11 SLIDER：滑动条控件

滑动条控件通常通过使用一个滑动条来改变数值。



所有与滑动条有关的函数位于文件 SLIDER*.c, SLIDER.h 中。所有的标识符是前缀 SLIDER。

配置选项

类型	宏	默认值	说明
N	SLIDER_BKCOLOR0_DEFAULT	0xc0c0c0	背景颜色。
N	SLIDER_COLOR0_DEFAULT	0xc0c0c0	滑动条（滑块 thumb）颜色。
B	SLIDER_USE_3D	1	启用 3D 支持。

SLIDER API 函数

下表依字母顺序列出了 μ C/GUI 与滑动条有关的函数。函数详细的描述在下面给出。

函 数	说 明
<code>SLIDER_Create()</code>	创建滑动条。
<code>SLIDER_CreateIndirect()</code>	从资源表条目创建滑动条。
<code>SLIDER_Dec()</code>	滑动条数值减 1。
<code>SLIDER_GetValue()</code>	返回滑动条当前数值。
<code>SLIDER_Inc()</code>	滑动条数值增 1。
<code>SLIDER_SetRange()</code>	设置滑动条数值范围。
<code>SLIDER_SetValue()</code>	设置滑动条当前数值。
<code>SLIDER_SetWidth()</code>	设置滑动条的宽度。

SLIDER_Create()

描述

在指定位置，以指定尺寸创建一个滑动条控件。

函数原型

```
SLIDER_Handle SLIDER_Create (   int x0, int y0,
                                int xsize, int ysize,
                                WM_HWIN hParent,
                                int Id,
                                int WinFlags,
                                int SpecialFlags);
```

参 数	含 意
x0	滑动条最左侧像素（在桌面坐标）。
y0	滑动条最顶部像素（在桌面坐标）。
xsize	滑动条水平尺寸（以像素为单位）。
ysize	滑动条垂直尺寸（以像素为单位）。
hParent	父窗口的句柄。
Id	返回的 ID 号。
WinFlags	窗口创建标志。典型的是 WM_CF_SHOW 用于使控件立即可见（请参阅第 12 章“视窗管理器”中的 WM_CreateWindow() 所列出的参数值）。
SpecialFlags	特定的创建标志（参阅 SLIDER_CreateIndirect() 下的间接创建标志）。

返回数值

创建的滑动条的控件；如果函数执行失败，则为 0。

SLIDER_CreateIndirect()

函数原型的说明在本章开始部分。下面的标志可以作为当作参数传递的资源表元素 Flags 使用：

允许的间接标志

SLIDER_CF_VERTICAL 创建一个垂直滑动条（默认值是水平）。

在资源表中 Para 元素未使用。

SLIDER_Dec()

描述

滑动条当前条目数减 1。

函数原型

```
void SLIDER_Dec(SLIDER_Handle hObj);
```

参 数	含 意
hObj	滑动条控件的句柄。

SLIDER_GetValue()

描述

返回滑动条当前的数值。

函数原型

```
int SLIDER_GetValue(SLIDER_Handle hObj);
```

参 数	含 意
hObj	滑动条控件的句柄。

返回数值

滑动条当前的数值。

SLIDER_Inc()

描述

滑动条当前条目数加 1。

函数原型

```
void SLIDER_Inc(SLIDER_Handle hObj);
```

参 数	含 意
<code>hObj</code>	滑动条控件的句柄。

SLIDER_SetRange()

描述

设置滑动条的范围。

函数原型

```
void SLIDER_SetRange(SLIDER_Handle hObj, int Min, int Max);
```

参 数	含 意
<code>hObj</code>	滑动条控件的句柄。
<code>Min</code>	最小值
<code>Max</code>	最大值

附加信息

建立一个滑动条后，其默认值为 0~100。

SLIDER_SetValue()

描述

设置滑动条当前的数值。

函数原型

```
void SLIDER_SetValue(SLIDER_Handle hObj, int v);
```

参 数	含 意
<code>hObj</code>	滑动条控件的句柄。
<code>v</code>	设置的数值。

SLIDER_SetWidth()

描述

设置滑动条的宽度。

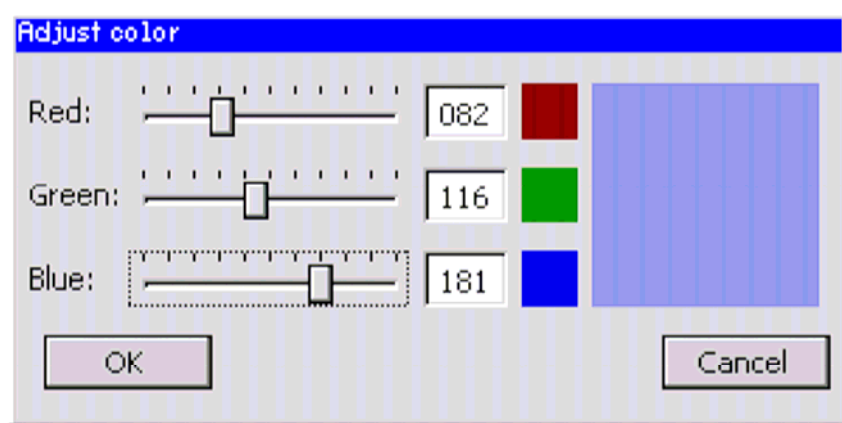
函数原型

```
void SLIDER_SetWidth(SLIDER_Handle hObj, int Width);
```

参 数	含 意
<code>hObj</code>	滑动条控件的句柄。
<code>Width</code>	设置的宽度。

范例

下面范例的源代码是随 μ C/GUI 一同提供的范例中的 `DIALOG_SliderColor.c` 文件：

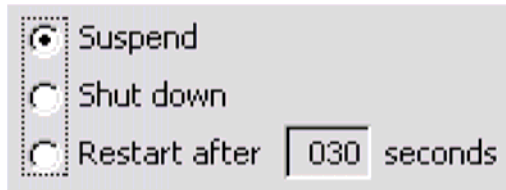


13.12 TEXT：文本控件

文本控件典型用于显示一个对话框的文本区域，如下面所示的消息框：



当然，文本区域也可以用作其它控件的标签，如下所示：



所有与文本有关的函数位于文件 `TEXT*.c`，`TEXT.h` 当中。所有的标识符是前缀 `TEXT`。

配置选项

类型	宏	默认值	说明
S	<code>TEXT_FONT_DEFAULT</code>	<code>&GUI_Font13_1</code>	使用的字体。

文本 API 函数

下表依字母顺序列出了 μ C/GUI 与文本有关的函数。函数详细的描述在下面给出。

函数	说明
<code>TEXT_Create()</code>	创建文本。
<code>TEXT_CreateIndirect()</code>	从资源表条目中创建文本。
<code>TEXT_GetDefaultFont()</code>	返回用于文本的默认字体。
<code>TEXT_SetDefaultFont()</code>	设置用于文本的默认字体。
<code>TEXT_SetFont()</code>	设置用于一个指定的文本控件的字体。
<code>TEXT_SetText()</code>	设置一个指定的文本控件的文本。
<code>TEXT_SetTextAlign()</code>	设置一个指定的文本控件的文本对齐方式。
<code>TEXT_SetTextPos()</code>	设置在一个指定的文本控件当中文本的位置。

`TEXT_Create()`

描述

在一个指定位置，指定大小创建一个文本控件。

函数原型

```
TEXT_Handle TEXT_Create ( int x0, int y0,
                          int xsize, int ysize,
                          int Id, int Flags,
                          const char* s, int Align);
```

参 数	含 意
-----	-----

<code>x0</code>	滑动条最左侧像素（在桌面坐标）。
<code>y0</code>	滑动条最顶部像素（在桌面坐标）。
<code>xsize</code>	滑动条水平尺寸（以像素为单位）。
<code>ysize</code>	滑动条垂直尺寸（以像素为单位）。
<code>Id</code>	返回的 ID 号。
<code>Flags</code>	窗口创建标志。典型的是 <code>WM_CF_SHOW</code> 用于使控件立即可见（请参阅第 12 章“视窗管理器”中的 <code>WM_CreateWindow()</code> 所列出的参数值）。
<code>s</code>	显示文本的指针。
<code>Align</code>	文本的对齐属性（参阅 <code>TEXT_CreateIndirect()</code> 下的间接创建标志）。

返回数值

创建的文件控件的句柄；如果函数执行失败则为 0。

TEXT_CreateIndirect()

函数原型的说明在本章开始部分。下面的标志可以用作资源表中当作参数传递的 `Flags` 元素：

允许的间接创建标志（以“OR”组合）

<code>TEXT_CF_LEFT</code>	水平对齐：左
<code>TEXT_CF_RIGHT</code>	水平对齐：右
<code>TEXT_CF_HCENTER</code>	水平对齐：中
<code>TEXT_CF_TOP</code>	垂直对齐：顶
<code>TEXT_CF_BOTTOM</code>	垂直对齐：底
<code>TEXT_CF_VCENTER</code>	垂直对齐：中

资源表中 `Para` 元素未使用。

TEXT_GetDefaultFont()

描述

返回用于文本控件的默认字体。

函数原型

```
const GUI_FONT* TEXT_GetDefaultFont(void);
```

返回数值

用于文本控件的默认字体的指针。

TEXT_SetDefaultFont()

描述

设置用于文本控件的默认字体。

函数原型

```
void TEXT_SetDefaultFont(const GUI_FONT* pFont) ;
```

参 数	含 意
pFont	设置为默认值的字体的指针。

TEXT_SetFont()

描述

设置用于指定文本控件的字体。

函数原型

```
void TEXT_SetFont(TEXT_Handle hObj, const GUI_FONT* pFont);
```

参 数	含 意
hObj	文本控件的句柄。
PFont	使用的字体的指针。

TEXT_SetText()

描述

设置用于指定文本控件的文本。

函数原型

```
void TEXT_SetText(TEXT_Handle hObj, const char* s);
```

参 数	含 意
-----	-----

<code>hObj</code>	文本控件的句柄。
<code>s</code>	显示的文本。

TEXT_SetTextAlign()

描述

设置指定文本控件的文本对齐方式。

函数原型

```
void TEXT_SetTextAlign(TEXT_Handle hObj, int Align);
```

参 数	含 意
<code>hObj</code>	文本控件的句柄。
<code>s</code>	文本对齐方式（参阅 TEXT_Create()）。

TEXT_SetTextPos()

描述

设置一个指定文本控件中的文本位置。

函数原型

```
void TEXT_SetTextPos(TEXT_Handle hObj, int XOff, int YOff);
```

参 数	含 意
<code>hObj</code>	文本控件的句柄。
<code>XOff</code>	文本相对于控件原点 X 轴的偏移距离。
<code>YOff</code>	文本相对于控件原点 Y 轴的偏移距离。

第14章 对话框

控件可以创建和独立使用，因为它们自己天生就是窗口。可是，使用对话框常常是很需要的，它是一种包含一个或多个控件的窗口。

一个对话框通常是一个窗口，它的出现要求使用者输入信息。它可能包括多个控件，通过对这些不同的控件的选择向使用者发出请求信息，或者它采用一个提供简单信息（例如提醒使消息框用者注意或警告）和一个“OK”按钮的形式。

14.1 对话框基础

1. 输入焦点

视察管理器能记住一个窗口或窗口物体最终被选择是通过用户使用触摸屏，鼠标，键盘或者其它的什么。一个窗口收到键盘输入消息了，我们说它有了输入焦点。

记住输入焦点的主要原因是为了确定键盘命令发送到哪去了。具有输入焦点的窗口会接收由键盘产生的事件。

2. 模块化和非模块化对话框

对话框分为模块化和非模块化两种。一个模块化对话框会阻塞执行的线程。默认情况下，它有输入焦点，必须在线程继续执行前被用户关闭。另一方面，一个非模块化对话框不会阻塞调用的线程——在它显示的时候，它允许任务继续运行。

请注意在其它的一些环境中，模块化和非模块化对话框分别被称为模态和非模态。

3. 对话框消息

一个对话框就是一个窗口，它接收消息，就象系统中其它所有窗口做的一样。大多数消息被对话框自动处理了，其它传给了在建立对话框上指定的回调函数（也为对话框程序所知）。

有两种类型的附加消息被送到对话框处理：WM_INIT_DIALOG 和 WM_NOTIFY_PARENT。在一个对话框显示前，WM_INIT_DIALOG 消息会立即被发送到对话框处理，而对话框程序对它典型的用法是用这个消息初始化控件，及实现其它影响对话框显示的初始化任务。WM_NOTIFY_PARENT 消息是通过对话框的子窗口发送到对话框的，通知父窗口一些为了保证同步的事件。

14.2 建立对话框

建立对话框需要做两件基本的事情：一个资源表，定义包括的控件，和一个对话框程序，定义控件的初始值，与它们的行为一样。一旦两个条件都存在，你只需进行单个函数调用（GUI_CreateDialogBox() 或 GUI_ExecDialogBox()）实际建立对话框。

1. 资源表

对话框可以基于阻塞（使用 GUI_ExecDialogBox()）或非阻塞（使用

GUI_CreateDialogBox()) 方式建立。

必须首先定义一个资源表，以指定包括在对话框中的所有控件。下面的范例展示如何建立一个资源表。这个特定的范例是手工建立的，尽管它也能通过一个 GUI-builder 做到。

```
static const GUI_WIDGET_CREATE_INFO_aDialogCreate[] =
{
    { FRAMEWIN_CreateIndirect, "Dialog", 0, 10, 10, 180, 230, 0, 0 },
    { BUTTON_CreateIndirect, "OK", GUI_ID_OK, 100, 5, 60, 20, 0, 0 },
    { BUTTON_CreateIndirect, "Cancel", GUI_ID_CANCEL, 100, 30, 60, 20, 0, 0 },
    { TEXT_CreateIndirect, "LText", 0, 10, 55, 48, 15, 0, GUI_TA_LEFT },
    { TEXT_CreateIndirect, "RText", 0, 10, 80, 48, 15, 0, GUI_TA_RIGHT },
    { EDIT_CreateIndirect, NULL, GUI_ID_EDIT0, 60, 55, 100, 15, 0, 50 },
    { EDIT_CreateIndirect, NULL, GUI_ID_EDIT1, 60, 80, 100, 15, 0, 50 },
    { TEXT_CreateIndirect, "Hex", 0, 10, 100, 48, 15, 0, GUI_TA_RIGHT },
    { EDIT_CreateIndirect, NULL, GUI_ID_EDIT2, 60, 100, 100, 15, 0, 6 },
    { TEXT_CreateIndirect, "Bin", 0, 10, 120, 48, 15, 0, GUI_TA_RIGHT },
    { EDIT_CreateIndirect, NULL, GUI_ID_EDIT3, 60, 120, 100, 15, 0, 0 },
    { LISTBOX_CreateIndirect, NULL, GUI_ID_LISTBOX0, 10, 20, 48, 40, 0, 0 },
    { CHECKBOX_CreateIndirect, NULL, GUI_ID_CHECK0, 10, 5, 0, 0, 0, 0 },
    { CHECKBOX_CreateIndirect, NULL, GUI_ID_CHECK1, 30, 5, 0, 0, 0, 0 },
    { SLIDER_CreateIndirect, NULL, GUI_ID_SLIDER0, 60, 140, 100, 20, 0, 0 },
    { SLIDER_CreateIndirect, NULL, GUI_ID_SLIDER1, 10, 170, 150, 30, 0, 0 }
};
```

2. 对话框程序

```

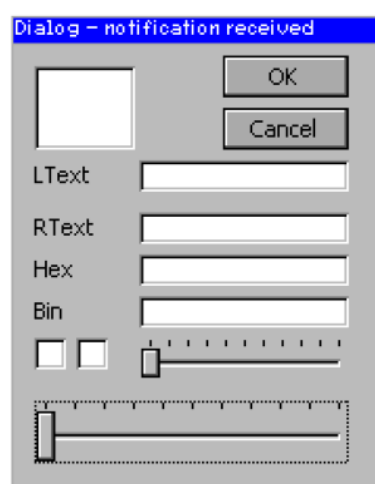
/*****
*                                     *
*****/

static void_cbCallback(WM_MESSAGE * pMsg)
{
    switch(pMsg->MsgId)
    {
        default: WM_DefaultProc(pMsg);
    }
}
```

对于这个范例，随着下面一行代码的执行，对话框会显示出来。

```
GUI_ExecDialogBox( _aDialogCreate,
                  GUI_COUNTOF(_aDialogCreate),
                  &_cbCallback,
                  0, 0, 0);
```

结果的对话框看起来如下图一样，或者与之相似（实际外观依赖于你的配置和默认设置）：



建立对话框后，所有包括在资源表中的控件将可见，尽管看起来与上面的屏幕截图差不多，这样控件是以“空”的形式出现。这是因为对话框程序依旧没有包括初始化单个元素的代码。控件的初始化数值，由它们引起的行为，以及它们之间的交互作用都需要在对话框程序中定义。

3. 初始化对话框

最典型的下一步是使用它们各自的初始化数值对控件进行初始化。在对话框程序中，这是对 WM_INIT_DIALOG 消息做出反应时的通常做法。下面的程序说明这个事情：

```
/******
 *
 *                               对话框程序
 *
 *****/

static void _cbCallback(WM_MESSAGE * pMsg)
{
    int NCode, Id;
    WM_HWIN hEdit0, hEdit1, hEdit2, hEdit3, hListBox;
```

```
WM_HWIN hWin = pMsg->hWin;
switch(pMsg->MsgId)
{
    case WM_INIT_DIALOG:
        /* 获得所有控件的窗口句柄 */
        hEdit0 = WM_GetDialogItem(hWin, GUI_ID_EDIT0);
        hEdit1 = WM_GetDialogItem(hWin, GUI_ID_EDIT1);
        hEdit2 = WM_GetDialogItem(hWin, GUI_ID_EDIT2);
        hEdit3 = WM_GetDialogItem(hWin, GUI_ID_EDIT3);
        hListBox = WM_GetDialogItem(hWin, GUI_ID_LISTBOX0);
        /* 初始化所有控件 */
        EDIT_SetText(hEdit0, "EDIT widget 0");
        EDIT_SetText(hEdit1, "EDIT widget 1");
        EDIT_SetTextAlign(hEdit1, GUI_TA_LEFT);
        EDIT_SetHexMode(hEdit2, 0x1234, 0, 0xffff);
        EDIT_SetBinMode(hEdit3, 0x1234, 0, 0xffff);
        LISTBOX_SetText(hListBox, _apListBox);
        WM_DisableWindow(WM_GetDialogItem(hWin, GUI_ID_CHECK1));
        CHECKBOX_Check(WM_GetDialogItem(hWin, GUI_ID_CHECK0));
        CHECKBOX_Check(WM_GetDialogItem(hWin, GUI_ID_CHECK1));
        SLIDER_SetWidth(WM_GetDialogItem(hWin, GUI_ID_SLIDER0), 5);
        SLIDER_SetValue(WM_GetDialogItem(hWin, GUI_ID_SLIDER1), 50);
        break;
    default:
        WM_DefaultProc(pMsg);
}
}
```

初始化后的对话框现在看起来象下面的一样了，所有的控件都有了初始数值：



4. 定义对话框行为

对话框一旦被初始化，剩下的所有要向对话框程序添加的代码将定义控件的行为，使其可充分的操作。继续同一个范例，最终的对话框程序如下所示：

```

/*****
*                                     *
*                                     * 对话框程序 *
*                                     *
*****/

static void_cbCallback(WM_MESSAGE * pMsg)
{
    int NCode, Id;
    WM_HWIN hEdit0, hEdit1, hEdit2, hEdit3, hListBox;
    WM_HWIN hWin = pMsg->hWin;
    switch(pMsg->MsgId)
    {
        case WM_INIT_DIALOG:
            /* 获得所有控件的窗口句柄 */
            hEdit0 = WM_GetDialogItem(hWin, GUI_ID_EDIT0);
            hEdit1 = WM_GetDialogItem(hWin, GUI_ID_EDIT1);
            hEdit2 = WM_GetDialogItem(hWin, GUI_ID_EDIT2);
            hEdit3 = WM_GetDialogItem(hWin, GUI_ID_EDIT3);
            hListBox = WM_GetDialogItem(hWin, GUI_ID_LISTBOX0);

            /* 初始化所有控件 */
            EDIT_SetText(hEdit0, "EDIT widget 0");
            EDIT_SetText(hEdit1, "EDIT widget 1");
            EDIT_SetTextAlign(hEdit1, GUI_TA_LEFT);
    }
}

```

```

EDIT_SetHexMode(hEdit2, 0x1234, 0, 0xffff);
EDIT_SetBinMode(hEdit3, 0x1234, 0, 0xffff);
LISTBOX_SetText(hListBox, _apListBox);
WM_DisableWindow(WM_GetDialogItem(hWin, GUI_ID_CHECK1));
CHECKBOX_Check(WM_GetDialogItem(hWin, GUI_ID_CHECK0));
CHECKBOX_Check(WM_GetDialogItem(hWin, GUI_ID_CHECK1));
SLIDER_SetWidth(WM_GetDialogItem(hWin, GUI_ID_SLIDER0), 5);
SLIDER_SetValue(WM_GetDialogItem(hWin, GUI_ID_SLIDER1), 0);
break;
case WM_KEY:
    switch(((WM_KEY_INFO*) (pMsg->Data.p))>Key)
    {
        case GUI_ID_ESCAPE:
            GUI_EndDialog(hWin, 1); break;
        case GUI_ID_ENTER:
            GUI_EndDialog(hWin, 0); break;
    }
    break;
case WM_NOTIFY_PARENT:
    Id = WM_GetId(pMsg->hWinSrc);          /* 控件的 Id */
    NCode = pMsg->Data.v;                  /* 通知代码 */
    switch(NCode)
    {
        case WM_NOTIFICATION_RELEASED:    /* 如果释放则起作用 */
            if(Id == GUI_ID_OK)
            {
                /* “OK” 按钮 */
                GUI_EndDialog(hWin, 0);
            }
            if(Id == GUI_ID_CANCEL)
            {
                /* “Cancel” 按钮 */
                GUI_EndDialog(hWin, 1);
            }
            break;
        case WM_NOTIFICATION_SEL_CHANGED: /* 改变选择 */
            FRAMEWIN_SetText(hWin, "Dialog - sel changed");
    }

```

```
        break;
    default:
        FRAMEWIN_SetText( hWin,
                           "Dialog-notification received");
    }
    break;
default:
    WM_DefaultProc(pMsg);
}
}
```

想了解更详细的资料, 请参考随 μ C/GUI 一起发布的范例中的 Dialog.c 这个范例的全部代码。

14.3 对话框API 参考函数

下表列出了与对话框相关的函数, 在各自的类型中按字母顺序进行排列。函数的详细描述后面列出。

函 数	说 明
	对话框
GUI_CreateDialogBox	建立一个非阻塞式的对话框。
GUI_ExecDialogBox	建立一个阻塞式的对话框。
GUI_EndDialog	结束一个对话框。
	消息框
GUI_MessageBox	建立一个消息框。

14.4 对话框

GUI_CreateDialogBox

描述

建立一个非阻塞式的对话框。

函数原型

```
WM_HWIN GUI_CreateDialogBox (  const GUI_WIDGET_CREATE_INFO* paWidget,
                               int NumWidgets,
```



```
WM_CALLBACK* cb,  
WM_HWIN hParent,  
int x0, int y0);
```

参 数	含 意
paWidget	定义包含在对话框中所有控件的资源表的指针。
NumWidgets	包含在对话框中所有控件的数量。
cb	一个具体应用的回调函数的指针（对话框程序）。
hParent	父窗口的句柄（0 表示没有父窗口）。
x0	对话框相对于父窗口的 X 轴坐标。
y0	对话框相对于父窗口的 Y 轴坐标。

GUI_ExecDialogBox

描述

建立一个阻塞式的对话框。

函数原型

```
int GUI_ExecDialogBox ( const GUI_WIDGET_CREATE_INFO* paWidget,  
                        int NumWidgets,  
                        WM_CALLBACK* cb,  
                        WM_HWIN hParent,  
                        int x0, int y0);
```

参 数	含 意
paWidget	定义包含在对话框中所有控件的资源表的指针。
NumWidgets	包含在对话框中所有控件的数量。
cb	一个具体应用的回调函数的指针（对话框程序）。
hParent	父窗口的句柄（0 表示没有父窗口）。
x0	对话框相对于父窗口的 X 轴坐标。
y0	对话框相对于父窗口的 Y 轴坐标。

GUI_EndDialog

描述

结束（关闭）一个对话框。

函数原型

```
void GUI_EndDialog(WM_HWIN hDialog, int r);
```

参 数	含 意
<code>hDialog</code>	对话框的句柄。

返回数值

从建立对话框的函数（只是与 `GUI_ExecDialogBox` 相关）指定返回调用任务的数值。对于非阻塞方式的对话框，没有等待的应用任务，则返回值被忽略。

14.5 消息框

消息框实际上是一种对话框，只不过是它的默认属性被指定了。它只需要一行代码就能建立。一条消息显示在一个带标题栏的窗框内，同时带有一个“OK”按钮，要关闭窗口必需按下它。

GUI_MessageBox

描述

建立及显示一个消息框。

函数原型

```
void GUI_MessageBox(const char* sMessage, const char* sCaption, int Flags);
```

参 数	含 意
<code>sMessage</code>	显示的消息。
<code>sCaption</code>	窗口框标题栏的标题内容。
<code>Flags</code>	为以后的扩展而保留，数值无关紧要。

附加信息

消息框的默认特性可以通过修改 `GUIConf.h` 文件中相关内容而改变。下表显示了所有可用的配置宏：

类型	宏	默认值	说明
N	<code>MESSAGEBOX_BORDER</code>	4	消息框元素和客户窗框元素之间的距离。
N	<code>MESSAGEBOX_XSIZEOK</code>	50	“OK”按钮的 X 轴尺寸。

N	MESSAGEBOX_YSIZEOK	20	“OK” 按钮的 Y 轴尺寸。
S	MESSAGEBOX_BKCOLOR	GUI_WHITE	客户窗口的背景颜色。

消息框显示的内容多于一行是可以的。下面的范例展示如何做到这一点。

范例

下面的例子展示了一个简单消息框的用法。

```

/*-----
文件:      DIALOG_MessageBox. c
目的:      展示 GUI_MessageBox 的例子
-----*/

#include "GUI.h"

/*****
*                               主函数                               *
*****/

void main(void)
{
    GUI_Init();
    WM_SetBkWindowColor(GUI_RED);
    /* 建立消息框，并等待至其关闭 */
    GUI_MessageBox("This text is shown\nin a message box",
                   "Caption/Title",
                   GUI_MESSAGEBOX_CF_MOVEABLE);

    GUI_Delay(500);
    /* 等待一小段时间…… */
    GUI_MessageBox("New message !",
                   "Caption/Title",
                   GUI_MESSAGEBOX_CF_MOVEABLE);
}

```

上面范例程序执行结果的屏幕截图



第15章 抗锯齿

直线近似由一系列位于显示器坐标的像素组成，因此它们会显示锯齿，除了那些接近于水平或垂直线的直线。这种锯齿现象被称为图形失真。

抗锯齿是平滑的直线或曲线。它减少了锯齿现象，不完全是水平或垂直方向的任何直线的阶梯现象。`µC/GUI` 支持不同的抗锯齿质量，抗锯齿字体和高分辨率坐标。

抗锯齿的支持是一个独立的软件项目，它不包括在`µC/GUI` 基本的软件包中。抗锯齿软件位于子目录 `GUI\AntiAlias` 下。

15.1 介绍

抗锯齿通过“混合”前景色和背景色来达到平滑曲线和斜线的效果。前景色和背景色之间用到的阴影的数量越多，抗锯齿效果越佳（计算时间相应的也越长）。

抗锯齿的品质

抗锯齿的品质由函数 `GUI_AA_SetFactor` 设定。这在本章的后面部分介绍。对于在抗锯齿和相应结果之间关系的概念，看一看所绘出的图片。



第一条直线在没有抗锯齿的情况下（抗锯齿系数为 1）绘出，第二条直线使用抗锯齿系数 2 绘出。这意思是从前景到背景的阴影数值为 $2 \times 2 = 4$ 。下一条直线使用抗锯齿系数 3 绘出，因此阴影数值为 $3 \times 3 = 9$ ，以下类推。抗锯齿系数取 4 对于大多数应用来说应该是足够了，再增大抗锯齿系数对于最终结果的影响并不显著，只会增加计算时间。

抗锯齿字体

支持两种类型的抗锯齿字体，低质量（2bpp）和高质量（4bpp）。当要使用它们的时候，程序需要显示这些字体会自动连接。下表显示没有使用抗锯齿和使用两种抗锯齿字体分别绘一个字符“C”的效果：

字体类型	黑色在白色上	白色在黑色上
标准 (无抗锯齿) 1 bpp 2 级阴影		



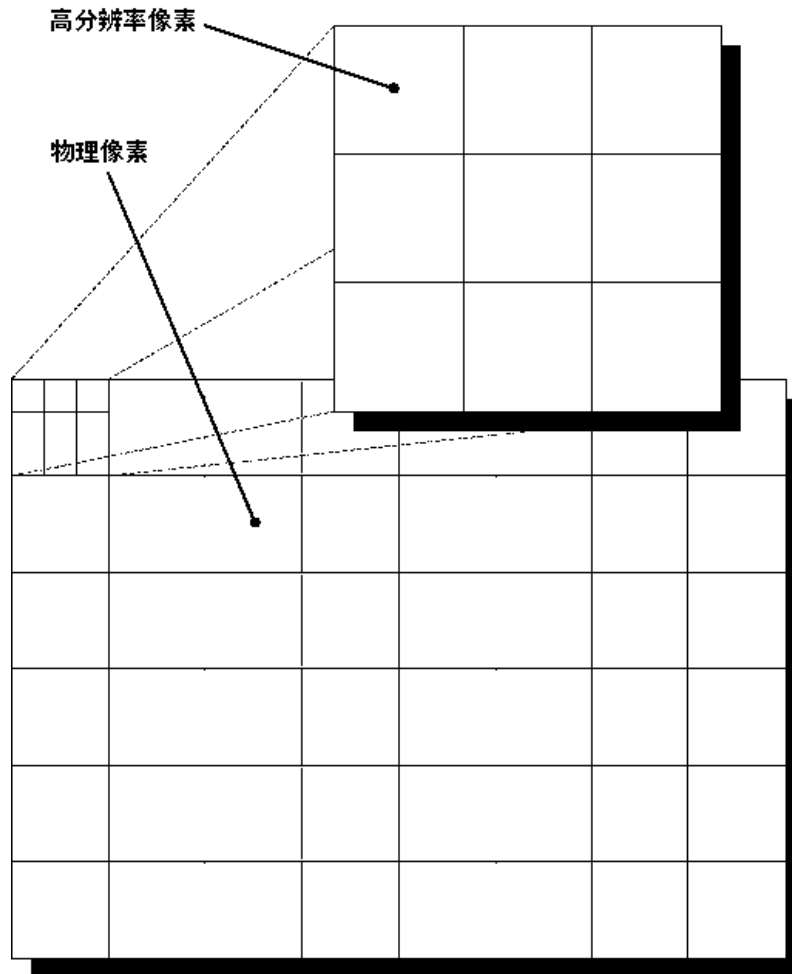
抗锯齿字体可以由 μ C/GUI 字体转换器建立。通常使用抗锯齿字体的目的是改善文字的外观。虽然使用高质量抗锯齿的效果比低质量的抗锯齿更具视觉上的舒适性，但是相应的，它也会占用更多的计算时间和消耗更多的内存。与非抗锯齿（1bpp）字体相比，低质量（2bpp）字体消耗的内存是其两倍，而高质量（4bpp）字体消耗的内存是其 4 倍。

高分辨率坐标

当使用抗锯齿进行一个项目绘制时，使用的坐标与正常（非抗锯齿）的绘图函数的一样的，这是默认模式。在函数参数中你不必考虑抗锯齿系数。例如，从（50, 100）到（100, 50）绘一条抗锯齿直线，程序应当这样写：

```
GUI_AA_DrawLine(50, 100, 100, 50);
```

μ C/GUI 的高分辨率特性让你使用由抗锯齿系数和显示屏尺寸所决定的虚拟区域。高分辨率坐标必须由函数 GUI_AA_EnableHiRes 启动，使用 GUI_AA_DisableHiRes 函数进行显示。这两个函数在本章的后面部分说明。使用高分辨率坐标的优点是对象不仅可以放置在显示屏的物理坐标中，也可以放在这些物理坐标“中间”。下面展示了一个抗锯齿系数是 3 的高分辨率像素虚拟区域。



为了使用抗锯齿系数为 3 的高分辨率模式从像素 (50, 100) 到 (100, 50) 绘一条直线，程序应当这样写：

```
GUI_AA_DrawLine(150, 300, 300, 150);
```

使用高分辨率特性的范例程序请参考本章最后的范例。

15.2 抗锯齿 API 函数

下表列出了 μ C/GUI 的抗锯齿软件包的有效函数，所有函数在各自的类型中按字母顺序进行排列。函数的详细描述在后面列出。

控制函数	说 明
GUI_AA_DisableHiRes()	禁止高分辨率座标。
GUI_AA_EnableHiRes()	启用高分辨率座标。
GUI_AA_GetFactor()	返回当前的抗锯齿系数。
GUI_AA_SetFactor()	设置当前的抗锯齿系数。

绘图函数	说 明
GUI_AA_DrawArc()	绘一段抗锯齿圆弧。
GUI_AA_DrawLine()	绘一根抗锯齿直线。
GUI_AA_DrawPolyOutline()	绘一个抗锯齿多边形的轮廓。
GUI_AA_FillCircle()	绘一个抗锯齿的圆。
GUI_AA_FillPolygon()	绘一个填充和抗锯齿的多边形。

15.3 控制函数

GUI_AA_DisableHiRes()

描述

禁止高分辨率座标。

函数原型

```
void GUI_AA_DisableHiRes(void);
```

附加信息

默认情况下，高分辨率座标被禁止。

GUI_AA_EnableHiRes()

描述

启用高分辨率座标。

函数原型

```
void GUI_AA_EnableHiRes(void);
```

GUI_AA_GetFactor()

描述

返回当前的抗锯齿品质系数。

函数原型

```
int GUI_AA_GetFactor(void);
```

返回数值

当前的抗锯齿系数。

GUI_AA_SetFactor()

描述

设置当前的抗锯齿品质系数。

函数原型

```
void GUI_AA_SetFactor(int Factor);
```

参 数	含 意
<code>Factor</code>	新的抗锯齿系数

附加信息

尽管设置参数 `Factor` 为 1 是允许的，这将会禁止抗锯齿，导致使用一种标准字体。我们还是推荐使用抗锯齿，其品质系数范围是 2~4。默认系数是 3。

15.4 绘图函数

GUI_AA_DrawArc()

描述

在当前窗口的指定位置，使用当前画笔大小和画笔形状显示一段抗锯齿的圆弧。

函数原型

```
void GUI_AA_DrawArc(int x0, int y0, int rx, int ry, int a0, int a1);
```

参 数	含 意
<code>x0</code>	中心的水平坐标
<code>y0</code>	中心的垂直坐标

<code>rx</code>	水平半径
-----------------	------

限制

现在 `ry` 参数无效，取而代之的是 `rx` 参数。

附加信息

如果工作在高分辨率模式，坐标和半径必须在高分辨率座标内否则，必须以像素值指定。

GUI_AA_DrawLine()

描述

在当前窗口的指定位置，使用当前画笔大小及画笔形状显示一条抗锯齿的直线。

函数原型

```
void GUI_AA_DrawLine(int x0, int y0, int x1, int y1);
```

参 数	含 意
<code>x0</code>	起始 X 轴坐标。
<code>y0</code>	起始 Y 轴坐标。

附加信息

如果工作在高分辨率模式，座标必须在高分辨率座标里面。否则，必须以像素值指定。

GUI_AA_DrawPolyOutline()

描述

在当前窗口的指定位置指定线宽，显示一个由一系列点组成的多边形的轮廓线。

函数原型

```
void GUI_AA_DrawPolyOutline (    const GUI_POINT* pPoint,  
                                int NumPoints,  
                                int Thickness,  
                                int x, int y)
```

参 数	含 意
PPoint	显示的多边形的指针。
NumPoints	点的序列中指定点的数量。
Thickness	轮廓的线宽。

附加信息

绘出的折线自动连接起点和终点而形成闭合。起始点不必重复指定为终点。如果工作在高分辨率模式，座标必须在高分辨率座标里面。否则，必须以像素值指定。

范例

```
#define countof(Array)      (sizeof(Array) / sizeof(Array[0]))
static GUI_POINT aPoints[] =
{
    {0, 0}, {15, 30}, {0, 20}, {-15, 30}
}

void Sample (void)
{
    GUI_AA_DrawPolyOutline(aPoints, countof(aPoints), 3, 150, 40);
}
```

上面范例程序执行结果的屏幕截图



GUI_AA_FillCircle()

描述

在当前窗口的指定位置显示一个填充和抗锯齿的圆。

函数原型

```
void GUI_AA_FillCircle(int x0, int y0, int r);
```

参 数	含 意
<code>x0</code>	圆的中心水平坐标（在客户窗口以像素为单位）
<code>y0</code>	圆的中心垂直坐标（在客户窗口以像素为单位）
<code>r</code>	圆的半径（直径的一半）

附加信息

如果工作在高分辨率模式，座标必须在高分辨率座标里面。否则，必须以像素值指定。

GUI_AA_FillPolygon()

描述

在当前窗口指定位置，能过一个点的序列填充一个抗锯齿多边形。

函数原型

```
void GUI_AA_FillPolygon(const GUI_POINT* pPoint, int NumPoints, int x, int y)
```

参 数	含 意
<code>PPoint</code>	显示的多边形的指针。
<code>NumPoints</code>	点的序列中指定点的数量。
<code>x</code>	原点的 X 轴坐标。
<code>y</code>	原点的 Y 轴坐标。

附加信息

绘出的折线自动连接起点和终点而形成闭合。起始点不必重复指定为终点。

如果工作在高分辨率模式，坐标必须在高分辨率座标中。否则，它们必须以像素为单位指定。

15.5 范例

不同的抗锯齿系数

下面的范例使用非抗锯齿方式和抗锯齿方式绘斜线。源代码文件是 `\Misc\AntialiasedLines.c`。

```
/*-----
```

文件: AntialiasedLines.c

目的: 使用不同的抗锯齿质量显示一条直线。

```
-----*/

#include "GUI.H"

/*****
 *
 *          使用不同的抗锯齿质量显示一条直线
 *
 *****/

static void DemoAntialiasing(void)
{
    int i, x1, x2;
    int y = 2;
    /* 设置绘图特性 */
    GUI_SetColor(GUI_BLACK);
    GUI_SetBkColor(GUI_WHITE);
    GUI_SetPenShape(GUI_PS_FLAT);
    GUI_Clear();
    x1 = 10;
    x2 = 90;
    /* 绘一条没有抗锯齿的直线 */
    GUI_DispStringHCenterAt("\nNormal", (x1 + x2) / 2, 10);
    for (i = 1; i < 12; i++)
    {
        GUI_SetPenSize(i);
        GUI_DrawLine(x1, 40 + i * 15, x2, 40 + i * 15 + y);
    }
    x1 = 110;
    x2 = 190;
    /* 绘一条抗锯齿品质因数为 2 的直线 */
    GUI_AA_SetFactor(2);
    GUI_DispStringHCenterAt ( "Antialiased\n\nusing factor 2",
                              (x1 + x2) / 2, 10);
    for (i = 1; i < 12; i++)
    {
        GUI_SetPenSize(i);
```

```

        GUI_AA_DrawLine(x1, 40 + i * 15, x2, 40 + i * 15 + y);
    }
    x1 = 210;
    x2 = 290;
    /* 绘一条抗锯齿品质因数为 6 的直线 */
    GUI_AA_SetFactor(6);
    GUI_DispStringHCenterAt ( "Antialiased\n\nusing factor 6",
                              (x1 + x2) / 2, 10);

    for (i = 1; i < 12; i++)
    {
        GUI_SetPenSize(i);
        GUI_AA_DrawLine(x1, 40 + i * 15, x2, 40 + i * 15 + y);
    }
}

/*****
*                                     *
*                               主函数                               *
*                                     *
*****/

void main(void)
{
    GUI_Init();
    DemoAntialiasing();
    while(1)
        GUI_Delay(100);
}

```

上面范例执行结果的屏幕截图

正常

使用抗锯齿系数 2

使用抗锯齿系数 6



在高分辨率座标中放置的直线

这个范例展示在高分辨率座标中放置一条抗锯齿直线。源代码文件是：
Sample\Misc\HiResPixel.c.

```

/*-----
文件:      HiResPixel.c
目的:      高分辨率像素的示范
-----*/

#include "GUI.H"

/*-----
*                      显示一条在高分辨率像素中的直线                      *
*-----*/

void ShowHiResPixel(void)
{
    int i, Factor = 5;
    GUI_SetBkColor(GUI_WHITE);
    GUI_SetColor(GUI_BLACK);
    GUI_Clear();
    GUI_SetPenSize(2);
    GUI_SetPenShape(GUI_PS_FLAT);
    GUI_AA_EnableHiRes();          /* 启动高分辨率 */

```



```

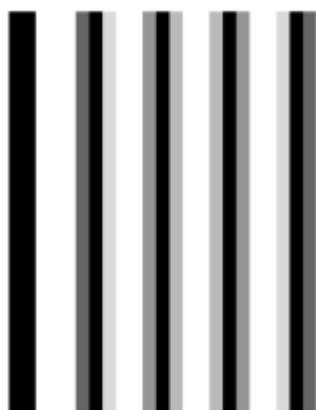
GUI_AA_SetFactor(Factor);          /* 设置品质系数 */
/* 使用虚拟分辨率绘直线 */
for (i = 0; i < Factor; i++)
{
    int x = (i + 1) * 5 * Factor + i - 1;
    GUI_AA_DrawLine(x, 50, x, 199);
}
}

/*****
*                                     *
*                               主函数                               *
*****/

void main (void)
{
    GUI_Init () ;
    ShowHiResPixel();
    while(1);
}

```

上面范例执行结果的屏幕截图放大后的效果



使用高分辨率抗锯齿绘运动的指针

这个范例通过绘一个旋转的指针（每步旋转 0.1 度）展示高分辨率抗锯齿的应用。本例没有屏幕截图，因为高分辨率抗锯齿效果只有在指针运动时才可见。没有使用高分辨率的情况下，指针的运动是以短暂的“跳动”形式显示。而在高分辨率下，不会出现跳动。

范例源代码文件是：\Misc\HiResAntialiasing.c。

```

/*-----
文件：      HiResAntialiasing.c
目的：      高分辨率抗锯齿展示
-----*/

#include "GUI.H"

/*****
*                               数据                               *
*****/

#define countof (Obj)    (sizeof (Obj) / sizeof (Obj[0] ) )
static const GUI_POINT aPointer[] =
{
    {0,3}, {85,1}, {90,0}, {85,-1}, {0,-3}
};
static GUI_POINT aPointerHiRes[countof(aPointer)];
typedef struct
{
    GUI_AUTODEV_INFO AutoInfo;
    GUI_POINT aPoints[countof(aPointer)];
    int Factor;
} PARAM;

/*****
*                               绘图函数                               *
*****/

static void DrawHiRes(void * p)
{
    PARAM * pParam = (PARAM *)p;
    if (pParam->AutoInfo.DrawFixed)
    {
        GUI_ClearRect(0, 0, 99, 99);
    }
    GUI_AA_FillPolygon( pParam->aPoints,

```

```

        countof(aPointer),
        5 * pParam->Factor,
        95 * pParam->Factor);
}

static void Draw(void * p)
{
    PARAM * pParam = (PARAM *)p;
    if (pParam->AutoInfo.DrawFixed)
    {
        GUI_ClearRect(100, 0, 199, 99);
    }
    GUI_AA_FillPolygon(pParam->aPoints, countof(aPointer), 105, 95);
}

/*****
*                               *
*      通过绘旋转的指针来展示高分辨率      *
*                               *
*****/

static void ShowHiresAntialiasing(void)
{
    int i;
    GUI_AUTODEV aAuto[2] ;
    PARAM Param;
    Param.Factor = 3;
    GUI_DispStringHCenterAt("Using\nhigh\nresolution\nmode", 50, 120);
    GUI_DispStringHCenterAt("Not using\nhigh\nresolution\nmode", 150, 120);
    /* 建立 GUI_AUTODEV 物体 */
    for (i = 0; i < countof(aAuto); i++)
    {
        GUI_MEMDEV_CreateAuto(&aAuto[i]);
    }
    /* 计算指针的高分辨率坐标 */
    for (i = 0; i < countof(aPointer); i++)
    {
        aPointerHiRes[i].x = aPointer[i].x * Param.Factor;
        aPointerHiRes[i].y = aPointer[i].y * Param.Factor;
    }
}

```

```

    }
    GUI_AA_SetFactor(Param.Factor);          /* 设置抗锯齿系数 */
    while(1)
    {
        for (i = 0; i < 1800; i++)
        {
            float Angle = (i >= 900) ? 1800 - i : i;
            Angle *= 3.1415926f / 1800;

            /* 在高分辨率中绘指针 */
            GUI_AA_EnableHiRes();
            GUI_RotatePolygon ( Param.aPoints,
                                aPointerHiRes,
                                countof(aPointer),
                                Angle);
            GUI_MEMDEV_DrawAuto(&aAuto[0],
                                &Param.AutoInfo,
                                DrawHiRes,
                                &Param);

            /* 在非高分辨率中绘指针 */
            GUI_AA_DisableHiRes();
            GUI_RotatePolygon ( Param.aPoints,
                                aPointer,
                                countof(aPointer),
                                Angle);
            GUI_MEMDEV_DrawAuto(&aAuto[1],
                                &Param.AutoInfo,
                                Draw,
                                &Param);

            #ifdef WIN32
                GUI_Delay(2);
            #endif
        }
    }
}

/* *****
*                                     主函数                                     *
* *****

```

*****/

```
void main (void)
{
    GUI_Init();
    ShowHiresAntialiasing();
}
```

第16章 Unicode

Unicode 编码标准是一个 16 位字符的编码规范。所有全世界的有效字符都在一个 16 位的字符集中（适用于全世界范围内使用）。Unicode 标准由 Unicode 协会定义。

μC/GUI 能够以 Unicode 编码显示单个字符或字符串，尽管最普遍是使用混合字符串，即在一个 ASCII 字符串当中有许多 Unicode 序列。

16.1 显示 Unicode 字符

Unicode 字符

μ C/GUI 使用的字符输出函数 (GUI_DisPChar) 总是处理 16 位无符号数 (U16)，具有显示一个由 Unicode 定义的字符的基本的能力。它仅仅需要一种包括有你想显示字符的字体。

显示 Unicode 字符串

μ C/GUI 中用于字符串输出的函数通常是 GUI_DisPString。既然 GUI_DisPString 使用 8 位字符，你不能直接将一个 Unicode 字符串传递给它，因为 Unicode 字符串使用 16 位字符。有两个选项可利用：

使用 GUI_DisPString_UC，它接受 16 位 Unicode 字符串，或者通过嵌入式 Unicode 将一个 8 位字符串转换为 2 字节的字符。这样就可以支持任何编译器支持的标准 C 字符串函数的使用。

显示混合 Unicode 和 ASCII 的代码

这是 Unicode 显示的最值得推荐的方法。你不必使用专门的函数做这项工作。只需要两个宏定义一个字符串中 Unicode 序列的起始点和结束点。

宏	说明
<code>GUI_UC_START</code>	标志一个 Unicode 序列的开始。
<code>GUI_UC_END</code>	标志一个 Unicode 序列的结束。

16.2 Unicode 和双字节转换

为什么使用双字节结构？

首先，使用 Unicode 双字节变量不是必需的。不过，这样做可以减少文字消息的内存消耗，处理字符串更容易。如果你的编译器不能直接处理 16 位字符串的话，这样做的效果确实显著。

如何构造 Unicode 字符串

通常的想法是：当能够使用 Unicode 时，能够与已有的软件保持 100% 的兼容性。

这意思是说，那些仍旧是字节（8 位）阵列的字符串和那些“标准”的 ASCII 或西欧字符串可以写成规则的字符串。

Unicode 部分起点由 GUI_UC_START 说明，可以在字符串中的任何位置。

GUI_UC_END 定义 Unicode 部分的终点。按惯例，“0” 字符结束字符串；如果最后的字符是 Unicode 编码，GUI_UC_END 可以被忽略。

范例

```
GUI_SetBkColor(GUI_RED);
GUI_SetColor(GUI_WHITE);
GUI_Clear();
GUI_SetFont(&GUI_Font16_1HK);
GUI_DispStringHCenterAt ( "English mixed ... "
                           GUI_UC_START    /* 切换到 UNICODE (双字节) */
                           "\x30\x40"      /* 日文 */
                           "\x30\x45"      /* 日文 */
                           "\x30\x52"      /* 日文 */
                           "\x30\x51"      /* 日文 */
                           GUI_UC_END      /* 返回单字节字符 */
                           " ..with Japanese", 160, 50);
```

输出结果：



双字节结构如何工作？

所有字符，不存在 which do not have a code where 无论是 16 位字符代码的高位字节还是低位字节为 0 的情况的，either the high- or low-byte of the 16-bit character code is 0 will 将保持它们的 16 位字符编码。高字节为 0 的字符移到 0xe000~0xe0ff 区域。对于低字节代码为 0 的字符，高字节当作低字节使用，新的高字节设为 0xe1。结果的代码保存为 2 字节，高字节在前。

16.3 范例

下面的范例定义一种包括 6 个字符：“A”，“B”，“C”及 Unicode 字符 0x3060，0x3061 和 0x3062 的小字体。然后向显示屏写一个混合字符串。源代码文件是 Sample\Misc\Unicode.c。

```

/*-----
文件:      Unicode.c
目的:      展示 μC/GUI 的 Unicode 性能的例子
-----*/

#include "GUI.H"

/*-----
*                  定义包括 Unicode 字符的字体
*-----*/

/* unicode 区开始 <基本拉丁文> */

static const unsigned char acFontUC13_0041[13] = { /* 代码 0041 */
    _____,
    _____,
    __X_____,
    __X_____,
    __X_X____,
    __X_X____,
    __X_X____,
    _XXXXX_,
    _X____X_,
    _X____X_,
    XXX_XXX_,
    _____,
    _____};

static const unsigned char acFontUC13_0042[13] = { /* 代码 0042 */
    _____,
    _____,
    XXXXX____,
    _X____X_,
    _X____X_,
    _X____X_,
    _XXXX____,
    _X____X_,

```

```

_X__X__,
_X__X__,
XXXXX__,
_____,
_____};

```

```

static const unsigned char acFontUC13_0043[13] = { /* 代码 0043 */
_____,
_____,
__XX_X__,
_X__XX__,
X__X__,
X_____,
X_____,
X_____,
X_____,
_X__X__,
__XXX__,
_____,
_____};

```

```

/* unicode 区开始 <平假名> */

```

```

static const unsigned char acFontUC13_3060[26] = { /* 代码 3060 */
__XX__,X_X_____,
__X__,_X_X_____,
X__XXX_,_____,
__XXXX__,_____,
__X__XX,XXX_____,
__X__,X_____,
__X__X,_____,
__X__,_____,
__X__X__,_____,
__X__X__,_____,
__X__XX,XXX_____,
_____,_____,
_____,_____};

```

```

static const unsigned char acFontUC13_3061[26] = { /* 代码 3061 */
__XX__,_____,
__X__,_____,
X__XXXX,XX_____,
__XXXX__,_____,
__X__,_____,

```

```

__X_XXX,X_____,
__XXX__,_X_____,
__X_____,_X_____,
_____,_X_____,
_____,_X_____,
__XXXXX,X_____,
_____,_____,
_____,_____};

```

```
static const unsigned char acFontUC13_3062[26] = { /* 代码 3062 */
```

```

__XX__,X_X_____,
__X_____,_X_X_____,
X__XXXX,X_____,
__XXXX__,_____,
__X_____,_____,
__X_XXX,X_____,
__XXX__,_X_____,
__X_____,_X_____,
_____,_X_____,
_____,_X_____,
__XXXXX,X_____,
_____,_____,
_____,_____};

```

```
static const GUI_CHARINFO GUI_FontUC13_CharInfo[6] =
```

```

{
    {7, 7, 1, (void *)&acFontUC13_0041 },      /* 代码 0041 */
    {7, 7, 1, (void *)&acFontUC13_0042 },      /* 代码 0042 */
    {7, 7, 1, (void *)&acFontUC13_0043 },      /* 代码 0043 */
    {14, 14, 2, (void *)&acFontUC13_3060},      /* 代码 3060 */
    {14, 14, 2, (void *)&acFontUC13_3061},      /* 代码 3061 */
    {14, 14, 2, (void *)&acFontUC13_3062}      /* 代码 3062 */
};

```

```
static const GUI_FONT_PROP GUI_FontUC13_Prop2 =
```

```

{
    0x3060 ,      /* 第一个字符 */
    0x3062 ,      /* 最后一个字符 */
    &GUI_FontUC13_CharInfo[3], /* 第一个字符的地址 */
    (void*)0      /* 下一个 GUI_FONT_PROP 的指针 */
};

```

```

static const GUI_FONT_PROP GUI_FontUC13_Prop1 =

{
    0x0041,                /* 第一个字符 */
    0x0043,                /* 最后一个字符 */
    &GUI_FontUC13_CharInfo[0], /* 第一个字符的地址 */
    (void *)&GUI_FontUC13_Prop2 /* 下一个 GUI_FONT_PROP 的指针*/
};

static const GUI_FONT GUI_FontUC13 =

{
    GUI_FONTTYPE_PROP,    /* 字体类型 */
    13,                   /* 字体的调试*/
    13,                   /* 字体 Y 轴的间距 */
    1,                    /* X 轴的放大倍数 */
    1,                    /* Y 轴的放大倍数 */
    (void *)&GUI_FontUC13_Prop1
};

/*****
*                      定义包括 ASCII 和 UNICODE 字符的字符串                      *
*****/

static const char sUC_ASCII [] =

{
    "ABC"GUI_UC_START"\x30\x60\x30\x61\x30\x62"GUI_UC_END"\x0"
};

/*****
*                      展示 UNICODE 字符的输出                      *
*****/

static void DemoUNICODE(void)
{
    GUI_SetFont(&GUI_Font13HB_1);
    GUI_DispStringHCenterAt("μC/GUI-sample: UNICODE characters", 160, 0);
    /* 设置 ShiftJIS 字体 */
    GUI_SetFont(&GUI_FontUC13);

```

```
/* 显示字符串 * /  
    GUI_DispStringHCenterAt(sUC_ASCII, 160, 40);  
}  
  
/*****  
*                                     *  
*****/  
  
void main (void)  
{  
    GUI_Init () ;  
    DemoUNICODE();  
    while(1)  
        GUI_Delay(100);  
}
```

上面范例执行结果的屏幕截图

μC/GUI sample: UNICODE characters

ABCだぢぢ

第17章 Shift-JIS支持

最通用的日文编码方法是 Shift-JIS。Shift-JIS 编码使 8 位字符得到丰富的使用，第一个字节的值用来区别单字节和多字节字符。

你不需要调用特殊函数来显示一个 Shift-JIS 字符串。最主要的必要条件是要有一种包含有用于显示的 Shift-JIS 字符的字体。

17.1 创建 Shift-JIS 字体

字体转换器能够从任何的 Windows 字体产生一种 Shift-JIS 字体用于 μ C/GUI。当使用一种 Shift-JIS 字体时，用于显示 Shift-JIS 字符的函数会自动地与库链接。

关于如何创建 Shift-JIS 字体的详细信息，请联系 Micrium 公司（info@micrium.com）。一份单独的字体转换器的技术资料描述了所有你需要在你的 μ C/GUI 工程中实现 Shift-JIS 的有效方法。

17.2 例子

下面的例子定义了一种包括 6 个字符的小字体：“A”，“B”，“C”和 Shift-JIS 字符 0x8350（片假名字母 KE），0x8351（片假名字母 GE）和 0x8352（片假名字母 KO）。然后在显示屏上绘出一个复合字符串。该例子的源代码是 Sample\Misc\ShiftJIS.c。

```

/*-----
文件:      ShiftJIS.c
目的:      展示 $\mu$ C/GUI 的 ShiftJIS 性能
-----*/

#include "GUI.H"

/*-----
*****
*                               ShiftJIS 字体的定义                               *
*****
-----*/

/* 拉丁文大写字母 A */
static const unsigned char acFontSJIS13_0041[13] = { /* 编码 0041 */
    _____,
    _____,
    __X_____,
    __X_____,
    __X_X_____,
    __X_X_____,
    __X_X_____,
    __XXXXX_____,
    __X__X_____,
    __X__X_____,
    XXX_XXX_,
    _____,
    _____};

```

```
/* 拉丁文大写字母 B */
```

```
static const unsigned char acFontSJIS13_0042[13] = { /* 编码 0042 */
    _____,
    _____,
    XXXXXX____,
    _X__X__,
    _X__X__,
    _X__X__,
    _XXXX____,
    _X__X__,
    _X__X__,
    _X__X__,
    XXXXXX____,
    _____,
    _____};
```

```
/* 拉丁文大写字母 C */
```

```
static const unsigned char acFontSJIS13_0043[13] = { /* 编码 0043 */
    _____,
    _____,
    __XX_X__,
    _X__XX__,
    X____X__,
    X_____,
    X_____,
    X_____,
    X_____,
    _X__X__,
    __XXX____,
    _____,
    _____};
```

```
/* 片假名字母 KE */
```

```
static const unsigned char acFontSJIS13_8350[26] = { /* 编码 8350 */
    __XX____, _____,
    __X____, _____,
    __X____, _____,
    _____, XXXXX____,
    __X__X__, _____,
    _X____X__, _____,
    X____X__, _____,
    _____X__, _____,
```



```

        _X_, _____,
        _X_, _____,
        _X_, _____,
        _XX_, _____,
        _____, _____};

```

/* 片假名字母 GE */

```

static const unsigned char acFontSJIS13_8351[26] = { /* 编码 8351 */
    _XX_, X_X_,
    _X_, _X_X_,
    _X_, _____,
    _XXXXXX, XXX_,
    _X_ X_, _____,
    _X_ X_, _____,
    X_ X_, _____,
    _X_, _____,
    _X_, _____,
    _X_, _____,
    _X_, _____,
    _XX_, _____,
    _____, _____};

```

/* 片假名字母 KO */

```

static const unsigned char acFontSJIS13_8352[26] = { /* 编码 8352 */
    _____, _____,
    _____, _____,
    _XXXXXX, XX_,
    _____, _X_,
    _____, _X_,
    _____, _X_,
    _____, _X_,
    _____, _X_,
    _____, _X_,
    _XXXXXX, XXX_,
    _____, _____,
    _____, _____,
    _____, _____};

```

```

static const GUI_CHARINFO GUI_FontSJIS13_CharInfo[6] = {
    {7, 7, 1, (void *)&acFontSJIS13_0041} /* 编码 0041 */
    , {7, 7, 1, (void *)&acFontSJIS13_0042} /* 编码 0042 */
    , {7, 7, 1, (void *)&acFontSJIS13_0043} /* 编码 0043 */
};

```

```

    , { 14, 14, 2, (void *)&acFontSJIS13_8350}      /* 编码 8350 */
    , { 14, 14, 2, (void *)&acFontSJIS13_8351}      /* 编码 8351 */
    , { 14, 14, 2, (void *)&acFontSJIS13_8352}      /* 编码 8352 */
};

static const GUI_FONT_PROP GUI_FontSJIS13_Prop2 = {
    0x8350                      /* 第一个字符 */
    , 0x8352                    /* 最后一个字符 */
    , &GUI_FontSJIS13_CharInfo[3] /* 第一个字符的地址 */
    , (void *)0                 /* 指向下一个 GUI_FONT_PROP 的指针 */
};

static const GUI_FONT_PROP GUI_FontSJIS13_Prop1 = {
    0x0041                      /* 第一个字符 */
    , 0x0043                    /* 最后一个字符 */
    , &GUI_FontSJIS13_CharInfo[ 0] /* 第一个字符的地址 */
    , (void *)&GUI_FontSJIS13_Prop2 /* 指向下一个 GUI_FONT_PROP 的指针 */
};

static const GUI_FONT GUI_FontSJIS13 = {
    GUI_FONTTYPE_PROP_SJIS      /* 字体类型 */
    , 13                        /* 字体高度 */
    , 13                        /* 字体 Y 轴方向的间隔 */
    , 1                          /* X 轴方向放大系数 */
    , 1                          /* Y 轴方向放大系数 */
    , (void *)&GUI_FontSJIS13_Prop1
};

/*****
 *                               包含 ASCII 字符和 ShiftJIS 字符的字符串的定义                               *
 *****/

static const char aSJIS[] = {
    "ABC\x83\x50\x83\x51\x83\x52\x0"
};

/*****
 *                               展示 ShiftJIS 字符的输出                               *
 *****/

void DemoShiftJIS(void)

```

```

{
    GUI_SetFont (&GUI_Font13HB_1);
    GUI_DispStringHCenterAt ("μC/GUI-sample: ShiftJIS characters", 160, 0);
    /* 设置 ShiftJIS 字体 */
    GUI_SetFont (&GUI_FontSJIS13);
    /* 显示字符串 */
    GUI_DispStringHCenterAt (aSJIS, 160, 40);
}

/*****
*                                     主函数                                     *
*****/

void main(void)
{
    GUI_Init();
    DemoShiftJIS();
    while(1)
        GUI_Delay(100);
}

```

范例程序运行结果的屏幕截图

μC/GUI sample: UNICODE characters

ABCだぢぢ

第18章 输入设备

μC/GUI 提供触摸屏，鼠标，和键盘支持。基本μC/GUI 程序包包括一个用于模拟触摸屏驱动程序和一个 PS2 鼠标驱动程序，不过别的种类的触摸板和鼠标装置在适当的驱动程序下也可以使用。任何类型的键盘驱动程序都适合μC/GUI。

用于输入设备的软件位于子目录 GUI\Core 中。

18.1 指针光标输入设备

指针光标输入设备包括鼠标和触摸屏。它们共用一组通用的指针光标输入设备（PID）函数使得鼠标和触摸屏能同时起作用。该函数一般由视窗管理器自动地调用，如先前所描述的那样，起刷新显示屏的作用。如果视窗管理器未使用，你的应用程序要负责调用 PID 函数。

数据结构

GUI_PID_STATE 类型的结构通过程序使用当前值填入的参数 pState 所引用。该结构如下所述定义：

```
typedef struct {
    int x, y;
    unsigned char Pressed;
}GUI_PID_STATE;
```

指针光标输入设备API函数

下表按字母顺序列出指针光标输入设备函数。函数的详细说明在稍后给出。

函 数	说 明
GUI_PID_GetState()	返回 PID 的当前状态。
GUI_PID_StoreState()	存储 PID 的当前状态。

GUI_PID_GetState()

描述

返回指针光标输入设备的当前状态。

函数原型

```
void GUI_PID_GetState(const GUI_PID_STATE *pState);
```

参数	含意
pState	指向一个 GUI_PID_STATE 类型的结构的指针。

返回值

如果输入设备当前被按下为 1；如果未按下为 0。

GUI_PID_StoreState()

描述

存储指针光标输入设备的当前状态。

函数原型

```
int GUI_PID_StoreState(GUI_PID_STATE *pState);
```

参数	含意
<code>pState</code>	指向一个 GUI_PID_STATE 类型的结构的指针。

18.1.1 鼠标输入驱动程序

鼠标支持由两个“层”组成：一个通用层和一个鼠标驱动程序层。通用程序参考那些始终存在的函数，不管你使用什么样的鼠标驱动程序。另一方面，该有效的鼠标驱动程序，会根据需要调用适当的通用函数，这些函数只能够用于 μ C/GUI 提供的 PS2 鼠标驱动程序。如果你自己写驱动程序，在程序中你要负责调用这些通用函数。

通用鼠标函数会依次调用对应的 PID 函数。

通用鼠标API

下表按字母顺序列出了通用鼠标函数。这些函数可以用于任何类型的鼠标驱动程序。函数的详细说明在稍后给出。

函 数	说 明
<code>GUI_MOUSE_GetState()</code>	返回鼠标的当前状态。
<code>GUI_MOUSE_StoreState()</code>	存储鼠标的当前状态。

GUI_MOUSE_GetState()

描述

返回鼠标的当前状态。

函数原型

```
int GUI_MOUSE_GetState(GUI_PID_STATE *pState);
```

参数	含意
<code>pState</code>	指向一个 GUI_PID_STATE 类型的结构的指针。

返回值

如果鼠标当前被按下为 1；如果未按下为 0。

附加信息

该函数会调用 GUI_PID_GetState ()。

GUI_MOUSE_StoreState()

描述

存储鼠标的当前状态。

函数原型

```
void GUI_MOUSE_StoreState(const GUI_PID_STATE *pState) ;
```

参数	含意
<code>pState</code>	指向一个 GUI_PID_STATE 类型的结构的指针。

附加信息

该函数会调用 GUI_PID_StoreState ()。

PS2鼠标驱动程序API

下表按字母顺序列出了有效的鼠标驱动函数。这些函数仅仅在你使用包括在μC/GUI 中的 PS2 鼠标驱动程序时才应用。

函 数	说 明
<code>GUI_MOUSE_DRIVER_PS2_Init()</code>	初始化鼠标驱动程序。
<code>GUI_MOUSE_DRIVER_PS2_OnRx()</code>	从接收中断程序调用。

GUI_MOUSE_DRIVER_PS2_Init()

描述

初始化鼠标驱动程序。

函数原型

```
void GUI_MOUSE_DRIVER_PS2_Init(void);
```

GUI_MOUSE_DRIVER_PS2_OnRx()

描述

必须从接收中断程序调用。

函数原型

```
void GUI_MOUSE_DRIVER_PS2_OnRx(unsigned char Data);
```

参数	含意
Data	通过中断服务程序（ISR）接收到的数据字节数。

附加信息

该 PS2 鼠标驱动程序是一种串行驱动程序，意思是它每次接受一个字节。

你需要保证这些函数从你的接收中断程序调用，每次接收一个字节（字符）。

18.1.2 触摸屏输入驱动程序和配置

触摸屏支持也是由一个通用层和一个驱动程序层组成。通用函数用于任何类型的驱动程序（模拟，数字，等等.）。该有效的模拟触摸屏驱动程序，会根据需要调用适当的通用函数，这些函数只能够用于 μ C/GUI 提供的模拟触摸屏驱动程序。就象鼠标支持一样，如果你自己写驱动程序，在程序中你要负责调用这些通用函数。

通用触摸屏函数会调用对应的 PID 函数。

驱动程序层同时包括一个可能需要修改的配置模块。

通用触摸屏API

下表按字母顺序列出了通用触摸屏函数。 这些函数可以用于任何类型的触摸屏驱动程序。 函数的详细说明在稍后给出。

函 数	说 明
GUI_TOUCH_GetState()	返回触摸屏的当前状态。
GUI_TOUCH_StoreState()	存储触摸屏的当前状态。

GUI_TOUCH_GetState()

描述

返回触摸屏的当前状态。

函数原型

```
int GUI_TOUCH_GetState(GUI_PID_STATE *pState);
```

参数	含意
pState	指向一个 GUI_PID_STATE 类型的结构的指针。

返回值

如果触摸屏当前被按下为 1；如果未按下为 0。

GUI_TOUCH_StoreState()

描述

存储触摸屏的当前状态。

函数原型

```
void GUI_TOUCH_StoreState(int x int y);
```

参数	含意
x	X 轴坐标。
y	Y 轴坐标。

附加信息

该函数会调用 GUI_PID_StoreState()。

用于模拟触摸屏驱动程序API

μC/GUI 触摸屏驱动程序处理模拟输入（来自一个 8 位或更好的 A/D 转换器），对触摸屏进行去抖动和校准处理。

该触摸屏驱动程序通过使用函数 GUI_TOUCH_Exec() 连续地监视和刷新触摸板，该函数在它辨认出一个动作已经执行或者情况有所变化时，调用适当的通用触摸屏 API 函数。

下表按字母顺序列出了有效的模拟触摸屏驱动程序函数。 这些函数仅在你使用包括在 μC/GUI 中的驱动程序时才应用。

函 数	说 明
GUI_TOUCH_Calibrate()	更改刻度。
GUI_TOUCH_Exec()	激活 X 轴和 Y 轴的测量;需要大约每秒 100 次的调用。
GUI_TOUCH_SetDefaultCalibration()	恢复默认刻度。

TOUCH_X 函数

如果你使用 μC/GUI 提供的驱动程序，下列四个与硬件相关函数需要加到你工程当中，而在轮询触摸板时，它们通过 GUI_TOUCH_Exec() 函数调用。 一个建议的位置是在文件 GUI_X.C 中。 这些函数在下表列出：

函 数	说 明
TOUCH_X_ActivateX()	准备 Y 轴的测量。
TOUCH_X_ActivateY()	准备 X 轴的测量。
TOUCH_X_MeasureX()	返回 A/D 转换器 X 轴的结果。
TOUCH_X_MeasureY()	返回 A/D 转换器 Y 轴的结果。

GUI_TOUCH_Calibrate()

描述

在运行时更改刻度。

函数原型

```
int GUI_TOUCH_Calibrate(int Coord, int Log0, int Log1, int Phys0, int Phys1);
```

参数	含意
Coord	用于 X 轴是 0，用于 Y 轴是 1。
Log0	以像素为单位逻辑值 0。
Log1	以像素为单位逻辑值 1。
Phys0	在 Log0 时 A/D 转换器的值。
Phys1	在 Log1 时 A/D 转换器的值。

附加信息

该函数把要校正的轴，两个用于该轴的以像素为单位的逻辑值和两个对应 A/D 转换器的物理值作为参数。

GUI_TOUCH_Exec()

描述

通过调用该 TOUCH_X 函数对触摸屏进行轮询，以激活 X 和 Y 轴的测量。你必须保证这些函数大约每秒钟被调用 100 次。

函数原型

```
void GUI_TOUCH_Exec(void);
```

附加信息

如果你在使用一个实时操作系统，确定这些函数被调用的最轻松的方式是创建一个单独的任务。当没有使用一个多任务系统时，你可以使用一个中断服务程序来做这项工作。

GUI_TOUCH_SetDefaultCalibration()

描述

将刻度复位为配置文件中设置的默认值。

函数原型

```
void GUI_TOUCH_SetDefaultCalibration(void);
```

附加信息

如果在配置文件中没有设置数值，该刻度将恢复到原始的默认值。

TOUCH_X_ActivateX(), TOUCH_X_ActivateY()

描述

从 GUI_TOUCH_Exec() 调用这些函数以激活 X 和 Y 轴的测量。 TOUCH_X_ActivateX () 接通 X 轴的测量电压； TOUCH_X_ActivateY() 接通 Y 轴的电压。接通 X 轴电压意思是 X 轴的值能够测量了，反之亦然。

函数原型

```
void TOUCH_X_ActivateX(void); void TOUCH_X_ActivateY(void);
```

TOUCH_X_MeasureX(), TOUCH_X_MeasureY()

描述

从 GUI_TOUCH_Exec () 调用这些函数以返回来自 A/D 转换器的 X 和 Y 轴的测定值。

函数原型

```
int TOUCH_X_MeasureX(void); int TOUCH_X_MeasureY(void);
```

配置触摸屏模块

在你的配置文件夹中需要有一单独的配置文件，命名为 GUITouchConf.h。下表展示用于 μ C/GUI 提供的模拟触摸屏驱动程序的所有配置宏：

类型	宏	默认	说明
B	GUI_TOUCH_SWAP_XY	0	设为 1 则 X 轴和 Y 轴相互交换。
B	GUI_TOUCH_MIRROR_X	0	X 轴镜像。
B	GUI_TOUCH_MIRROR_Y	0	Y 轴镜像。
N	GUI_TOUCH_AD_LEFT	30	由 A/D 转换器返回的最小值。
N	GUI_TOUCH_AD_RIGHT	220	由 A/D 转换器返回的最大值。
N	GUI_TOUCH_AD_TOP	30	由 A/D 转换器返回的最小值。
N	GUI_TOUCH_AD_BOTTOM	220	由 A/D 转换器返回的最大值。
N	GUI_TOUCH_XSIZE	LCD_XSIZE	被触摸屏覆盖的水平区域。

N	GUI_TOUCH_YSIZE	LCD_YSIZE	被触摸屏覆盖的垂直区域。
---	-----------------	-----------	--------------

18.2 键盘输入

一个键盘输入设备使用 ASCII 字符编码，为了能够区别不同的字符。例如，只有一个“A”键在键盘上，但是一个大写的“A”和一个小写的“a”的 ASCII 编码是不一样的（分别是 0x41 和 0x61）。

μC/GUI预定义字符代码

μC/GUI 也能定义字符代码，用于其它的“虚拟”键盘操作。这些代码在下表列出，它们在 GUI.h 的一个标识符表中定义。因此，一个在μC/GUI 中字符代码能够可以是 ASCII 字符值的任何扩展值或任一个下列的μC/GUI 预定义值。

预定义的虚拟键代码	描述
GUI_KEY_BACKSPACE	退格键。
GUI_KEY_TAB	TAB 键。
GUI_KEY_ENTER	回车/返回键。
GUI_KEY_LEFT	左箭头键。
GUI_KEY_UP	向上箭头键。
GUI_KEY_RIGHT	右箭头键。
GUI_KEY_DOWN	向下箭头键。
GUI_KEY_HOME	Home 键（移到当前行的开头）。
GUI_KEY_END	End 键（移到当前行的末端）。
GUI_KEY_SHIFT	换档键。
GUI_KEY_CONTROL	控制键。
GUI_KEY_ESCAPE	换码键。
GUI_KEY_INSERT	插入键。
GUI_KEY_DELETE	删除键。

18.2.1 驱动程序层 API

键盘驱动程序层操作键盘信息函数。当具体的键（或者键组合）已经按下或释放时，程序会通知视窗管理器。

下表按字母顺序列出了驱动程序层键盘处理函数。函数的详细描述在稍后给出。

函 数	说 明
GUI_StoreKeyMsg()	在一个指定键中存储一个状态消息。
GUI_SendKeyMsg()	向一个指定的按键发送一个状态消息。

GUI_StoreKeyMsg()

描述

在一个指定键中存储一个状态消息。

函数原型

```
void GUI_StoreKeyMsg(int Key, int Pressed);
```

参数	含意
Key	可以是任何可扩展的 ASCII 字符（在 0x20 和 0xFF 之间）或者任何预定义的 μ C/GUI 信息码。
Pressed	键的状态，参考下面说明。

参数 [pressed](#) 允许的值

- 1: 按下状态。
- 0: 释放（未按下）状态

GUI_SendKeyMsg()

描述

向一个指定的按键发送一个状态消息。

函数原型

```
void GUI_SendKeyMsg(int Key, int Pressed);
```

参数	含意
Key	可以是任何可扩展的 ASCII 字符（在 0x20 和 0xFF 之间）或者任何预定义的 μ C/GUI 信息码。
Pressed	键的状态（参 GUI_StoreKeyMsg() ）

18.2.2 应用层 API

下表按字母顺序列出了应用层键盘处理函数。 函数的详细说明在稍后给出。

函 数	说 明
GUI_ClearKeyBuffer()	清除键缓冲区。

<code>GUI_GetKey()</code>	返回键缓冲区的内容。
<code>GUI_StoreKey()</code>	在缓冲区中存储一个键。
<code>GUI_WaitKey()</code>	等待一个键被按下。

GUI_ClearKeyBuffer()

描述

清除键缓冲区。

函数原型

```
void GUI_ClearKeyBuffer(void);
```

GUI_GetKey()

描述

返回键缓冲区的当前内容。

函数原型

```
int GUI_GetKey(void);
```

返回值

在键缓冲区中的字符代码；如果没有键在缓冲区中为 0。

GUI_StoreKey()

描述

在缓冲区中存储一个键。

函数原型

```
void GUI_StoreKey(int Key);
```

参数	含意
<code>Key</code>	可以是任何可扩展的 ASCII 字符（在 0x20 和 0xFF 之间）或者任何预定义的 μ C/GUI 信息码。

附加信息

通常该函数通过驱动程序调用而不是通过应用程序本身调用。

GUI_WaitKey()

描述

等待一个键被按下。

函数原型

```
int GUI_WaitKey(void);
```

附加信息

该应用是“锁定”的，意思是它在一个按键被按下之前是不会返回的。

第19章 与时间相关的函数

一些控件，以及我们的示范代码，需要与时间相关的函数。 μ C/GUI 的另外一部分图形库不需要一个时间轴。

示范代码大量使用函数 `GUI_Delay()`，该函数产生一定时期的时延。一个时间单位称为一个节拍。

定时和执行API

下表按字母顺序列出了有效的定时-运行相关的函数。 函数详细说明在后面给出。

函 数	说 明
GUI_Delay()	延时一个指定时段。
GUI_Exec()	运行回调函数（所有的工作）。
GUI_Exec1()	运行一个回调函数（仅一项工作）。
GUI_GetTime()	返回当前的系统时间。

GUI_Delay()

描述

延时一个指定时段。

函数原型

```
void GUI_Delay(int Period);
```

参 数	说 明
Period	以节拍为单位的时间，直到函数将返回为止。

附加信息

该时间单位（节拍）通常是毫秒（取决于 GUI_X_ 函数）。 GUI_Delay() 为了这个给出的时间，仅仅执行空函数。 如果使用了视窗管理器，延迟时间用于更新无效窗口（通过函数 WM_Exec() 的执行）。

该函数会调用 GUI_X_Delay()。

GUI_Exec()

描述

执行回调函数（一般是重绘窗口）。

函数原型

```
int GUI_Exec(void);
```

返回值

如果没有任务执行为 0。 如果有一个工作执行为 1。

附加信息

该函数会自动地重复地调用 GUI_Exec1() 直到它完成全部工作——本质上是直到返回一个 0 值。

通常该函数不需要被用户应用程序所调用。 它自动地被 GUI_Delay() 所调用。

GUI_Exec1()

描述

执行一个回调函数（仅一项工作——一般是重绘窗口）。

函数原型

```
int GUI_Exec1(void) ;
```

返回值

如果没有任务执行为 0。 如果有一个工作执行为 1。

附加信息

该函数可以重复地调用直到返回 0，其时意味着所有工作都已经完成。

该函数自动地被 GUI_Delay() 所调用。

GUI_GetTime()

描述

返回当前的系统时间。

函数原型

```
int GUI_GetTime(void) ;
```

返回值

当前的系统时间（以节拍为单位）。

附加信息

该函数会被 GUI_X_GetTime() 所调用。

第20章 底层配置

在目标系统上使用 μ C/GUI 之前，需要为你的应用配置软件。配置指的是配置（头）文件的修改，这些文件通常位于（子）目录“Config”中。我们尽可能使配置保持简单些，但有一些配置宏（在文件 LCDConf.h 中）是需要的，这是为了使系统能正确工作。包括：

LCD 宏，定义显示屏的尺寸及可选择功能（例如镜像等等）。

LCD 控制宏，定义如何访问你使用的控制器。

20.1 可用的配置宏

下表列出了可供 μ C/GUI 的底层配置使用的宏：

类型	宏	默认值	说明
通用（必需的）配置			
S	LCD_CONTROLLER	---	选择 LCD 控制器。
N	LCD_BITSPERPIXEL	---	指定每像素的位。
S	LCD_FIXEDPALETTE	---	定义颜色查询表（则必须定义 LCD_PHYSCOLORS）。
N	LCD_XSIZE	---	定义 LCD 的水平分辨率。
N	LCD_YSIZE	---	定义 LCD 的垂直分辨率。
初始化控制器			
F	LCD_INIT_控制器()	---	LCD 控制器初始化顺序。不适用于所有控制器。
显示方向			
B	LCD_MIRROR_X	0	激活 X 轴镜像。
B	LCD_MIRROR_Y	0	激活 Y 轴镜像。
B	LCD_SWAP_XY	0	行转换成列，列转换成行（XY 轴交换）。
N	LCD_VXSIZE	LCD_XSIZE	虚拟显示的水平分辨率。不适用于所有驱动器。
N	LCD_VYSIZE	LCD_YSIZE	虚拟显示的垂直分辨率。不适用于所有驱动器。
N	LCD_XORG<n>	0	LCD controller <n>：最左边（最小的）X 坐标。
N	LCD_YORG<n>	0	LCD controller <n>：最顶端（最小的）Y 坐标。
颜色配置			
N	LCD_MAX_LOG_COLORS	256	在一幅位图中驱动器能支持的逻辑颜色的最大数量。
A	LCD_PHYSCOLORS	---	定义颜色查询表的内容。只有在 LCD_FIXEDPALETTE 被设置为 0 时才需要。
B	LCD_PHYSCOLORS_IN_RAM	0	只有定义了物理颜色的情况下才相应有效。把物理颜色放入 RAM 中，使它们在运行时可以修改。
B	LCD_REVERSE	0	激活在编译时的反转显示。
B	LCD_SWAP_RB	0	激活红、蓝基色的交换。
LCD 的放大			
N	LCD_XMAG<n>	1	LCD 的水平方向放大系数。
N	LCD_YMAG<n>	1	LCD 的垂直方向放大系数。
简单的总线接口配置			
F	LCD_READ_A0(Result)	---	址址线为低电平，从 LCD 控制器读一个字节。
F	LCD_READ_A1(Result)	---	址址线为高电平，从 LCD 控制器读一个字节。
F	LCD_WRITE_A0(Byte)	---	址址线为低电平，向 LCD 控制器写一个字节。
F	LCD_WRITE_A1(Byte)	---	址址线为高电平，向 LCD 控制器写一个字节。
F	LCD_WRITE_MEM	---	址址线为高电平，向 LCD 控制器写入多个字节。
完整总线接口配置			
F	LCD_READ_MEM(Index)	---	读取控制器图像存储器的内容。
F	LCD_READ_REG(Index)	---	读取控制器一个配置寄存器的内容。
F	LCD_WRITE_MEM(Index, Data)	---	向控制器图像存储器（显示数据 RAM）写入内容。

F	LCD_WRITE_REG(Index, Data)	---	向控制器配置寄存器写入内容。
S	LCD_BUSWIDTH	16	选择 LCD 控制器/CPU 的接口的总线宽度 (8/16)。
F	LCD_ENABLE_REG_ACCESS	---	切换 M/R 信号进行寄存器访问。并不适用于所有的控制器。
F	LCD_ENABLE_MEM_ACCESS	---	切换 M/R 信号进行存储器访问。并不适用于所有的控制器。
B	LCD_SWAP_BYTE_ORDER	0	在使用一个 16 位总线接口时，激活在 CPU 和 LCD 控制器之间的端模式的反转（高低字节交换）。
LCD 控制器配置：公共极（行）/段（列）连线			
N	LCD_XORG<n>	0	LCD controller <n>: 最左边（最小的）X 坐标
N	LCD_YORG<n>	0	LCD controller <n>: 最顶端（最小的）Y 坐标
N	LCD_FIRSTSEG<n>	0	LCD controller <n>: 使用的第 1 段（列）连线。
N	LCD_LASTSEG<n>	LCD_XSIZE-1	LCD controller <n>: 使用的最后 1 段（列）连线。
N	LCD_FIRSTCOM<n>	0	LCD controller <n>: 使用的第 1 个公共极（行）连线。
N	LCD_LASTCOM<n>	LCD_YSIZE-1	LCD controller <n>: 使用的最后 1 个公共极（行）连线。
COM/SEG（公共极/段） 查询表			
A	LCD_LUT_COM	---	控制器 COM 查询表。
A	LCD_LUT_SEG	---	控制器 SEG 查询表。
杂项			
N	LCD_NUM_CONTROLLERS	1	使用的 LCD 控制器的数量。
B	LCD_CACHE	1	停用以禁止显示数据高速缓存使用，使驱动器的速度慢下来。并不适用于所有的控制器。
B	LCD_USE_BITBLT	1	停用以禁止 BitBLT 引擎，如果设为 1，驱动器将使用所有有效的硬件加速度。
B	LCD_SUPPORT_CACHECONTROL	0	激活以启用驱动器 API 函数 LCD_L0_ControlCache() 的高速缓存控制功能。并不适用于所有的控制器。
N	LCD_TIMERINIT0	---	使用 CPU 作为控制器时，为显示 pane 0 而用于 ISR 的计时值。
N	LCDTIMERINIT1	---	使用 CPU 作为控制器时，为显示 pane 1 而用于 ISR 的计时值。
F	LCD_ON	---	打开 LCD 的函数替换宏。
F	LCD_OFF	---	切断 LCD 的函数替换宏。

如何配置 LCD。

我们推荐使用下面的操作步骤：

1. 对有相似配置的配置文件进行拷贝，我们要对拷贝的文件进行编辑。在目录 Sample\LCDConf\ xxx 中有几个特定 LCD 控制器的配置范例，其中“xxx”是你的 LCD 驱动

器。

2. 通过定义简单总线或完全总线宏来配置总线接口。
3. 定义你的 LCD 尺寸 (LCD_XSIZE, LCD_YSIZE)。
4. 选择你的系统使用的控制器, 同时选择合适的 bpp 和调色板模式 (LCD_CONTROLLER, LCD_BITSPERPIXEL, LCD_FIXEDPALETTE)。
5. 如果需要, 配置所使用的 公共极 (行) / 段 (列) 连线。lines。
6. 测试系统。
7. 如果有需要, 进行 X/Y 轴反转 (LCD_REVERSE); 返回第 6 步。
8. 如果有需要, 进行 X/Y 轴镜象 (LCD_MIRROR_X, LCD_MIRROR_Y); 返回第 6 步。
9. 检查所有其它配置开关。
10. 删除配置中未使用的部分。

20.2 通用（必需的）配置

LCD_CONTROLLER

描述

定义使用的 LCD 控制器。

类型

选择开关。

附加信息

使用的 LCD 控制器通过适当的数字指定。请参阅第 22 章：“LCD 驱动”以获得更多可用选项的信息。

范例

指定一个 Epson SED1565 控制器：

```
#define LCD_controller 1565      /* 选择 SED1565 LCD 控制器 */
```

LCD_BITSPERPIXEL

描述

指定每像素的位的数量。

类型

数值。

LCD_FIXEDPALETTE

描述

指定固定调色板模式。

类型

选择开关。

附加信息

将数值设置为 0 表示使用一个颜色查询表而不是一个固定设色板模式。这样 LCD_PHYSCOLORS 宏必须定义。

LCD_XSIZE; LCD_YSIZE

描述

（分别）定义所用显示屏水平和垂直的分辨率。

类型

数值。

附加信息

数值是逻辑尺寸；X 轴方向指定了所有 LCD 驱动函数的 X 轴方向。通常 X 轴尺寸等于段(列)的数量。

20.3 初始化控制器

LCD_INIT_CONTROLLER()

描述

初始化 LCD 控制器。

类型

函数替换。

附加信息

该宏必须被用户定义以初始化一些控制器。它在驱动函数 LCD_L0_Init() 和 LCD_L0_Reinit() 之间执行。请参考你的控制器的资料手册以获得如何初始化你的硬件设备的更多信息。

范例

下面的范例测试一个 Epson SED1565 控制器，它使用一个内部电源调整器。

```
#define LCD_INIT_CONTROLLER()
    \LCD_WRITE_A0(0xe2);    /* 内部复位 */
    \LCD_WRITE_A0(0xae);    /* 显示开/关: 关 */
    \LCD_WRITE_A0(0xac);    /* Power save 开始: static indicator off */
    \LCD_WRITE_A0(0xa2);    /* LCD 偏置选择: 1/9 */
    \LCD_WRITE_A0(0xa0);    /* ADC 选择: 正常 */
    \LCD_WRITE_A0(0xc0);    /* 公共模式: 正常 */
    \LCD_WRITE_A0(0x27);    /* 5V 电压调节器: 中 */
    \LCD_WRITE_A0(0x81);    /* 进入电子音量模式 */
    \LCD_WRITE_A0(0x13);    /* 电子音量: 中 */
    \LCD_WRITE_A0(0xad);    /* Power save 结束: static indicator on */
    \LCD_WRITE_A0(0x03);    /* static indicator 寄存器设置: 开 (常开) */
    \LCD_WRITE_A0(0x2F);    /* 电源控制设置: 升压器, 调节器及跟随器关闭 */
    \LCD_WRITE_A0(0x40);    /* 显示开始行 */
    \LCD_WRITE_A0(0xB0);    /* 显示地址 0 页 */
    \LCD_WRITE_A0(0x10);    /* 显示列地址 MSB */
```

```
\LCD_WRITE_A0(0x00);    /* 显示列地址 LSB */  
\LCD_WRITE_A0(0xaf);    /* 显示 开/关: 开 */  
\LCD_WRITE_A0(0xe3);    /* 空命令
```

20.4 显示方向

LCD_MIRROR_X

描述

反转显示屏的 X 方向（水平）。

类型

二进制开关

0: 禁止, X 轴方向未镜像（默认）; 1: 激活, X 轴方向镜像。

附加信息

如果激活: $X \rightarrow \text{LCD_XSIZE}-1-X$ 。

该宏与 LCD_MIRROR_Y 和 LCD_SWAP_XY 结合, 能用于对显示屏任何方向的支持。在改变这个配置开关之前, 确认 LCD_SWAP_XY 设为你的应用所需要的值。

LCD_MIRROR_Y

描述

反转显示屏的 Y 方向（垂直）。

类型

二进制开关

0: 禁止, Y 轴方向未镜像（默认）; 1: 激活, Y 轴方向镜像。

附加信息

如果激活：Y \rightarrow LCD_YSIZE-1-Y。

该宏与 LCD_MIRROR_X 和 LCD_SWAP_XY 结合，能用于对显示屏任何方向的支持。在改变这个配置开关之前，确认 LCD_SWAP_XY 设为你的应用所需要的值。

LCD_SWAP_XY

描述

交换显示屏水平和垂直的方向。

类型

二进制开关

0：禁止，X-Y 未交换（默认）；1：激活，X-Y 交换。

附加信息

如果设为 0（没有交换），SEG 连线作为列，COM 连线作为行。如果激活：X \rightarrow Y。

修改这个开关时，你也必须交换显示屏分辨率 X-Y 设置(使用 LCD_XSIZE 和 LCD_YSIZE)。

LCD_VXSIZE; LCD_VYSIZE

描述

定义虚拟显示的水平和垂直分辨率。

类型

数值。

附加信息

数值是逻辑尺寸；X 方向指定所有 LCD 驱动函数使用的 X 方向。

如果没有使用虚拟显示器，这些数值应该与 LCD_XSIZE，LCD_YSIZE（这些默认设置）一样。

虚拟显示器特性要求硬件支持，同时并不是对所有驱动器都适用。

LCD_XORG<n>; LCD_YORG<n>

描述

（分别的）定义由配置驱动器控制的显示器的水平和垂直的原点。

类型

数值。

附加信息

在一个单显示屏系统，两个宏通常都设为 0（默认数值）。

20.5 颜色配置

LCD_MAX_LOG_COLORS

描述

定义在一幅位图中驱动器支持的颜色的最大数量。

类型

数值（默认值为 256）。

附加信息

如果你使用 4 级灰度 LCD，通常将该值设为 4 足够了。然而，在这种情况下，记住不要试图使用超过 4 种颜色来显示位图。

LCD_PHYSCOLORS

描述

定义颜色查询表的内容，如果要用到的话。

类型

别名。

附加信息

该宏只要求 LCD_FIXEDPALETTE 设为 0。参考颜色选择以获得更多信息。

LCD_PHYSCOLORS_IN_RAM**描述**

如果启用的话，把物理颜色的内容放入 RAM 中。

类型

二进制开关

0：停用（默认值）；1：激活。

LCD_REVERSE**描述**

在编译时翻转显示屏。

类型

二进制开关

0：停用，不翻转（默认值）；1：激活，翻转。

LCD_SWAP_RB**描述**

交换红蓝两种藕色。

类型

二进制开关

0: 停用, 不交换 (默认值); 1: 激活, 交换。

20.6 LCD 的放大

为了能正确地显示图片, 一些硬件要求 LCD 能放大。这样, 必须通过软件对这些有放大需要的硬件进行补偿。这可以通过激活一个层 (在驱动层之上), 自动处理显示屏的放大工作而做到。

LCD_XMAG

描述

指定 LCD 水平放大系数。

类型

数值 (默认为 1)。

附加信息

因数为 1 表示没有放大。

LCD_YMAG

描述

指定 LCD 垂直放大系数。

类型

数值 (默认是 1)

附加信息

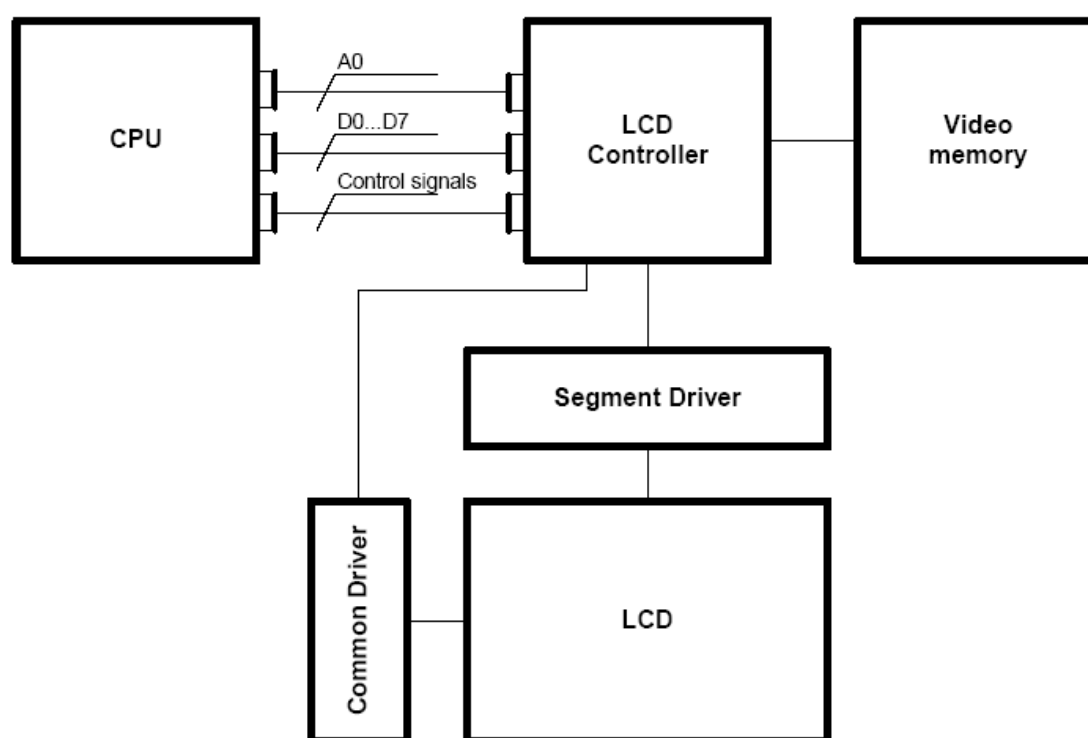
因数为 1 表示没有放大。

20.7 简单总线接口配置

LCD 控制器有两种基本类型的总线接口：完全和简单的总线接口。

大多数用于小一些的显示屏（通常最大为 240×128 或 320×240 ）的 LCD 控制器使用一个简单总线接口与 CPU 连接。对于一个简单总线，只有一个地址位（通常是 A0）连接到 LCD 控制器。一些这样的控制器非常慢，所以硬件设计者可以考虑将其连接到 I/O 脚而不是地址总线。

带有简单总线的 LCD 控制器的方框图



8 个数据位，一个地址位和 2 或 3 根控制线连接 CPU 和 LCD 控制器。4 个宏告诉 LCD 驱动器如何访问每个使用的控制器。如果 LCD 控制器直接连接到 CPU 的地址总线，配置就简单了，通常每个宏只占用一根连线。如果 LCD 控制器连到 I/O 脚，必须要模拟总线接口，每个宏要占用大约 5~10 根连线（或者调用一个模拟总线接口的函数）。

下面的宏仅用于简单总线接口的 LCD 控制器。

LCD_READ_A0

描述

在 A0 (C/D) 引脚为低电平时，从 LCD 控制器读取一个字节。

类型

函数替换。

原型

```
#define LCD_READ_A0(Result)
```

参数	含义
<code>Result</code>	读取的结果。这不是一个指针，而是一个变量（读取的数值保存在里面）的占位符。

LCD_READ_A1

描述

在 A0 (C/D) 引脚为低电平时，从 LCD 控制器读取一个字节。

类型

函数替换。

原型

```
#define LCD_READ_A1(Result)
```

参数	含义
<code>Result</code>	读取的结果。这不是一个指针，而是一个变量（读取的数值保存在里面）的占位符。

LCD_WRITE_A0

描述

在 A0 (C/D) 引脚为低电平时，向 LCD 控制器写一个字节。

类型

函数替换。

原型

```
#define LCD_WRITE_A0(Byte)
```

参数	含义
Byte	写入的字节。

LCD_WRITE_A1

描述

在 A0 (C/D) 引脚为高电平时，向 LCD 控制器写一个字节。

类型

函数替换。

原型

```
#define LCD_WRITE_A1(Byte)
```

参 数	含 意
Byte	写入的字节。

LCD_WRITEM_A1

描述

在 A0 (C/D) 引脚为高电平时，向 LCD 控制器写入几个字节。

类型

函数替换。

原型

```
#define LCD_WRITEM_A1(paBytes, NumberOfBytes)
```

参 数	含 意
paBytes	第一个字节数据指针的占位符。
NumberOfBytes	要写入的数据的字节数。

实际总线接口范例

下面的范例展示如何通过一个实际的总线接口访问：

```
void WriteM_A1(char *paBytes, int NummerOfBytes)
{
    int i;
    for(i = 0; i < NummerOfBytes; i++)
    {
        (*(volatile char *) 0xc0001) = *(paBytes + i) ;
    }
}

#define LCD_READ_A1(Result)      Result =(*(volatile char *)0xc0000)
#define LCD_READ_A0(Result)      Result =(*(volatile char *)0xc0001)
#define LCD_WRITE_A1(Byte)       (*(volatile char *)0xc0000) = Byte
#define LCD_WRITE_A0(Byte)       (*(volatile char *)0xc0001) = Byte
#define LCD_WRITEM_A1(paBytes, NummerOfBytes)
                                WriteM_A1( paBytes, NummerOfBytes)
```

连接到I/O 脚的简单函数

在目录 Sample\LCD_X 下可以找到的几个范例：

- 6800 接口的端口函数。
- 8080 接口的端口函数。
- 一个串行接口的简单端口函数。
- 一个简单I²C总线接口的端口函数。

这些范例可以直接使用。你需要做的是定义在每个范例顶部的端口访问宏，并将它们映射到你的 LCDConf.h 中，与下面展示的样式类似：

```
void LCD_X_Write00(char c);
void LCD_X_Write01(char c);
char LCD_X_Read00(void);
char LCD_X_Read01(void);
#define LCD_WRITE_A1(Byte)      LCD_X_Write01(Byte)
#define LCD_WRITE_A0(Byte)      LCD_X_Write00(Byte)
#define LCD_READ_A1(Result)     Result = LCD_X_Read01()
```

```
#define LCD_READ_A0(Result)    Result = LCD_X_Read00()
```

注意不是所有的 LCD 控制器用同样的方法处理 A0 或 C/D 位。例如，一个 Toshiba 控制器要求在存取数据时，该位为低电平，而 Epson SED1565 要求它为高电平。

多LCD 控制器的硬件访问

如果 LCD 使用的 LCD 控制器数量超过一个，你必须根据你的硬件要求为它们分别单独定义存取宏。每个 LCD 控制器需要 4 个宏。附加控制器的宏常常与第一个控制器的宏非常相似。对于总线的直接连接的情况，通常只是地址不同的。而使用 I/O 引脚连接时，除了片选信号以外，其它存取时序是一样的。

当使用的 LCD 控制器数量超过一个时，控制器的宏要增加一条下划线和控制器的索引作为后缀。例如：

控制器 #0: LCD_READ_A0_0，控制器 #1: LCD_READ_A0_1 等等。

注意第 1 个控制器被当作是控制器 #0，这样第 2 个控制器定义为#1，等等。附加 LCD 控制器的宏在下表列出。

第 2 个 LCD 控制器

类型	宏	说明
F	LCD_READ_A0_1(Result)	LCD 控制器 1: A0 = 0 时读一个字节。
F	LCD_READ_A1_1(Result)	LCD 控制器 1: A0 = 1 时读一个字节。
F	LCD_WRITE_A0_1(Byte)	LCD 控制器 1: A0 = 0 时写一个字节。
F	LCD_WRITE_A1_1(Byte)	LCD 控制器 1: A0 = 1 时写读一个字节。

第 3 个 LCD 控制器

类型	宏	说明
F	LCD_READ_A0_2(Result)	LCD 控制器 2: A0 = 0 时读一个字节。
F	LCD_READ_A1_2(Result)	LCD 控制器 2: A0 = 1 时读一个字节。
F	LCD_WRITE_A0_2(Byte)	LCD 控制器 2: A0 = 0 时写一个字节。
F	LCD_WRITE_A1_2(Byte)	LCD 控制器 2: A0 = 1 时写读一个字节。

第 4 个 LCD 控制器

类型	宏	说明
F	LCD_READ_A0_3(Result)	LCD 控制器 3: A0 = 0 时读一个字节。

F	LCD_READ_A1_3(Result)	LCD 控制器 3: A0 = 1 时读一个字节。
F	LCD_WRITE_A0_3(Byte)	LCD 控制器 3: A0 = 0 时写一个字节。
F	LCD_WRITE_A1_3(Byte)	LCD 控制器 3: A0 = 1 时写读一个字节。

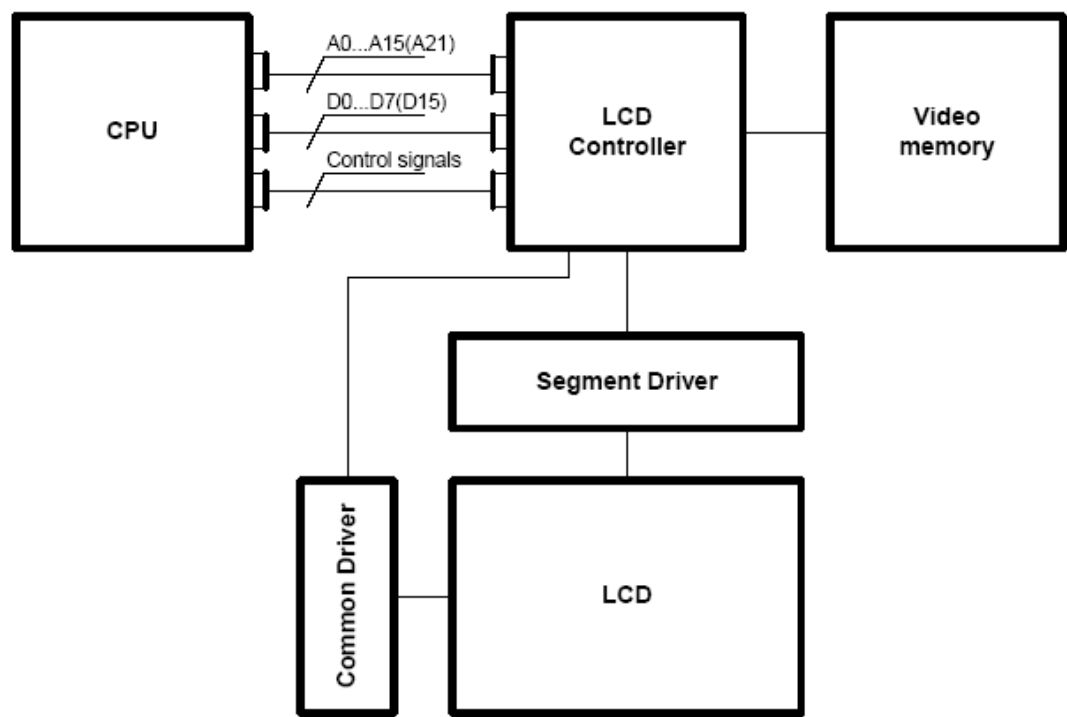
20.8 完全总线接口配置

一些 LCD 控制器（特别是那些用于更高分辨率显示的显示屏的控制器）要求一个完全地址总线，意思是说它们至少需要连接 14 个地址位。在一个完全地址总线配置中，CPU 直接访问图像存储器；完全地址总线连接到 LCD 控制器。

配置一个完全地址总线接口时唯一要知道的是地址范围（产生 LCD 控制器的片选信号）以及是使用 8 位还是 16 位存取（LCD 控制的地址总线宽度）。换句话说，你需要知道下面这些内容：

- 图像存储器存取的基地址；
- 寄存器访问的基地址；
- 相邻的图像存储器位置之间的距离（通常是 1/2/4 字节）；
- 相邻的寄存器位置之间的距离（通常是 1/2/4 字节）；
- 图像存储器的存取类型（8/16/32 位）。
- 寄存器的存取类型（8/16/32 位）

完全总线接口的LCD 控制器的典型框图



配置范例

该范例假设以下条件成立：

图像存储器基地址	0x80000
寄存器基地址	0xc0000
视频 RAM 存取	16 位
寄存器存取	16 位
相邻的图像存储器位置之间的距离	2 字节
相邻的寄存器位置之间的距离	2 字节

```
#define LCD_READ_REG(Index)          *((U16* ) (0xc0000+(Off<<1)))
#define LCD_WRITE_REG(Index, data)   *((U16*) (0xc0000+(Off<<1)))=data
#define LCD_READ_MEM(Index)         *((U16* ) (0x80000+(Off<<1)))
#define LCD_WRITE_MEM(Index, data)   *((U16*) (0x80000+(Off<<1)))=data
```

下面的宏仅用于带有完全总线接口的 LCD 控制器。

LCD_READ_MEM

描述

从 LCD 控制器的图象存储器读取数据。

类型

函数替换。

原型

```
#define LCD_READ_MEM(Index)
```

参 数	含 意
Index	控制器图像存储器的索引。

附加信息

该宏定义如何读取 LCD 控制器的图像存储器。

为了能正确配置这个开关，你需要知道图像存储器的基地址，间隔及是否允许 8/16/或 32 位存取。你也应知道适合你的编译器的正确的语法，因为这种硬件存取不是基于 ANSI C 定义的，因此会因为不同的编译器而不一样。

LCD_READ_REG

描述

从 LCD 控制器的寄存器读取数据。

类型

函数替换。

原型

```
#define LCD_READ_REG(Index)
```

参 数	含 意
Index	读取的寄存器的索引。

附加信息

该宏定义如何读取 LCD 控制器的寄存器。通常寄存器是内存映射的。在这种情况下，该宏一般可以单独写为一行。

为了能正确配置这个开关，你需要知道寄存器映射到的地址，间隔及是否允许 8/16/或 32 位存取。你也应知道适合你的编译器的正确的语法，因为这种硬件存取不是基于 ANSI C 定义的，因此会因为不同的编译器而不一样。不过，下面的语法可以在它们当中大部分正常工作。

范例

如果寄存器映射至起始地址为 0xc0000 的内存区，间隔为 2 以及使用 16 位存取方式；对于大部分编译器，定义看起来应该像下面一样：

```
#define LCD_READ_REG(Index)      *((U16*) (0xc0000+(Off<<1)))
```

LCD_WRITE_MEM

描述

向 LCD 控制器的图像存储器写数据。

类型

函数替换。

原型

```
LCD_WRITE_MEM(Index, Data)
```

参 数	含 意
Index	控制器的图像存储器的索引。

附加信息

该宏定义如何向 LCD 控制器的图像存储器写数据。

为了能正确配置这个开关，你需要知道图像存储器的基地址，间隔和是否允许 8/16 或者

32 位存取，还有适合你的编译器的正确语法。

对于 8 位存取，数值 1 表示 1 个字节。对于 16 位存取，数值 1 表示 1 个字。

LCD_WRITE_REG

描述

向 LCD 控制器指定的寄存器写入数据。

类型

函数替换。

原型

LCD_WRITE_REG(Index, Data)

参 数	含 意
Index	写入的寄存器的索引。

附加信息

该宏定义如何写 LCD 控制器的寄存器。如果寄存器是内存映射的，该宏一般可以单独写作一行。为了能正确配置这个开关，你需要知道寄存器映射到的地址，间隔和是否允许 8/16 或者 32 位存取，还有适合你的编译器的正确语法。

对于 8 位存取，数值 1 表示 1 个字节。对于 16 位存取，数值 1 表示 1 个字。

范例

如果寄存器映射至起始地址为 0xc0000 的内存区，间隔为 4 以及使用 8 位存取方式；对于大部分编译器来说，定义看起来应该像下面一样：

```
#define LCD_WRITE_REG(Index, Data)
    *((U8volatile *) (0xc0000+(Off<<2)))=data
```

LCD_BUSWIDTH

描述

定义 LCD 控制器/CPU 接口（外部显示屏访问）的总线宽度。

类型

选择开关。

8: 8 位宽度 VRAM; 16: 16 位宽度 VRAM （默认）

附加信息

因为这些完全依赖于你的硬件，你将不得不替换这些宏。Epson SED1352 在存储器和寄存器之间是有区别的；存储器是 LCD 控制器的图像存储器，寄存器是 15 个配置寄存器。该宏定义如何存取（读/写）VRAM 和寄存器。

LCD_ENABLE_REG_ACCESS

描述

启用寄存器访问并将 M/R 信号设置为高。

类型

函数替换。

原型

```
#define LCD_ENABLE_REG_ACCESS() MR = 1
```

附加信息

只用于 Epson SED1356 和 SED1386 控制器。

使用该宏之后，LCD_ENABLE_MEM_ACCESS 也必须被定义，为了在寄存器访问后切换回存储器访问。

LCD_ENABLE_MEM_ACCESS

描述

切换 M/R 信号到存储器访问。它在寄存器访问函数后执行，并将 M/R 信号设置为低。

类型

函数替换。

原型

```
#define LCD_ENABLE_MEM_ACCESS() MR = 0
```

附加信息

只用于 Epson SED1356 及 SED1386 控制器。

LCD_SWAP_BYTE_ORDER**描述**

使用 16 位地址总线时，在 CPU 和 LCD 控制器中反转端模式（交换高低字节）。

类型

二进制开关

0: 停用，端模式没有交换（默认）；1: 激活，端模式交换。

20.9 LCD 控制器配置：公共极（行）/段（列）连线

对于大部分 LCD，公共极（COM）和段（SEG）连线的设置是易于理解的，既不需要特定的设置，也不需要配置宏。

这一部分说明 LCD 控制器如何与你的显示屏进行物理的连接。方向无关紧要，它只是假设 COM 和 SEG 连线是连续的。如果 SEG 或 COM 的方向反了，使用 LCD_MIRROR_X/LCD_MIRROR_Y 设置它们为你的应用所需要的方向。如果使用非连续的 COM/SEG 连线地，你必须修改驱动器（插入一个翻译表）或者最好找到硬件（LCD 模块）设计者，告诉他/她要重新设计。

LCD_XORG<n>; LCD_YORG<n>**描述**

通过配置驱动器，（分别）定义显示屏水平和垂直方向的原点。

类型

数值。

附加信息

在一个单显示屏系统，两个宏通常都设为 0（默认值）。

LCD_FIRSTSEG<n>

描述

Controller <n>: 使用的第 1 段（列）连线。

类型

数值。

LCD_LASTSEG<n>

描述

Controller <n>: 使用的最后 1 段（列）连线。

类型

数值。

LCD_FIRSTCOM<n>

描述

Controller <n>: 使用的第1个公共极（行）连线。

类型

数值。

LCD_LASTCOM<n>

描述

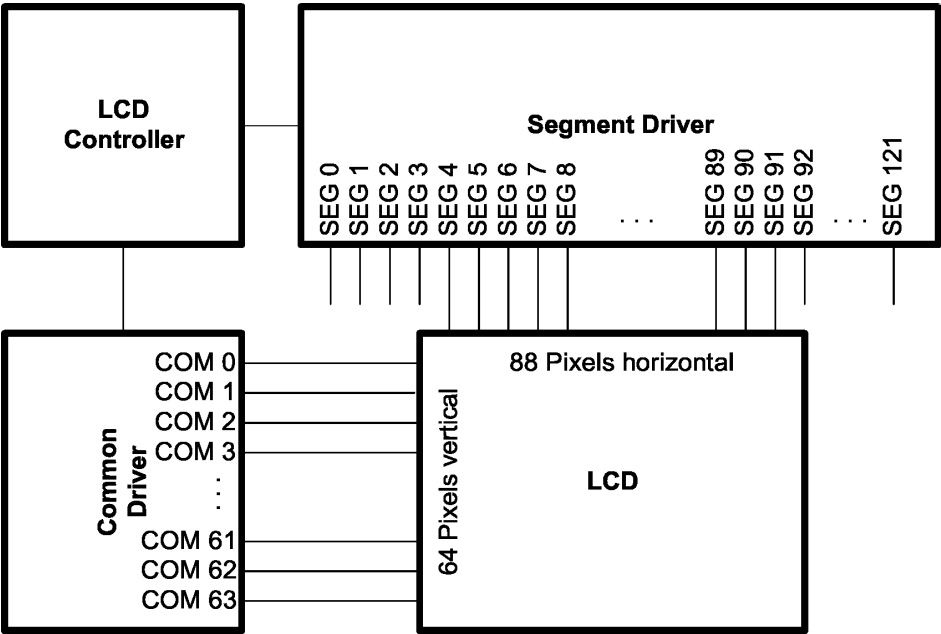
Controller <n>: 使用的最后 1 个公共极（行）连线。

类型

数值。

单LCD 控制器配置

下面的框图展示由单个 LCD 控制器控制的单个 LCD。使用另外的 COM 和 SEG 驱动器。所有公共极驱动器的输出口（COM0~COM63）都用上，但是只有部分段驱动器的输出口（SEG4~SEG91）被用到。注意：为了简明起见，视频 RAM 没有在框图内给出。



上面范例的配置

```
#define LCD_FIRSTSEG0 (4) /* Contr. 0: 使用的第 1 条 seg 连线 */
#define LCD_LASTSEG0 (91) /* Contr. 0: 使用的最后 1 条 seg 连线 */
#define LCD_FIRSTCOM0 (0) /* Contr. 0: 使用的第 1 条 com 连线 */
#define LCD_LASTCOM0 (63) /* Contr. 0: 使用的最后 1 条 com 连线 */
```

同时也请注意，如果 COM 或 SEG 连线被镜像，甚至 LCD 侧转（90° 旋转，X、Y 轴互换），上面的配置也是同样的。如果 COM/SEG 驱动器集成到 LCD 控制器当中，如一些为小 LCD 所设

计的控制器方案一样，也同样适用。这种控制器的一个典型的例子是 Epson SED15XX 系列。

配置附加 LCD 控制器

μ C/GUI 提供了一个 LCD 可以使用多个 LCD 控制器（最多为 4 个）的可能。配置开关与第一个控制器(控制器 0)是一样的，除了索引是 1, 2 或 3，而不是 0。

第 2 个 LCD 控制器

类型	宏	说明
N	LCD_FIRSTSEG1	LCD 控制器 1: 使用的第 1 段连线。
N	LCD_LASTSEG1	LCD 控制器 1: 使用的最后 1 段连线。
N	LCD_FIRSTCOM1	LCD 控制器 1: 使用的第 1 个公共极连线。
N	LCD_LASTCOM1	LCD 控制器 1: 使用的最后 1 个公共极连线。

第 3 个 LCD 控制器

类型	宏	说明
N	LCD_FIRSTSEG2	LCD 控制器 2: 使用的第 1 段连线。
N	LCD_LASTSEG2	LCD 控制器 2: 使用的最后 1 段连线。
N	LCD_FIRSTCOM2	LCD 控制器 2: 使用的第 1 个公共极连线。
N	LCD_LASTCOM2	LCD 控制器 2: 使用的最后 1 个公共极连线。

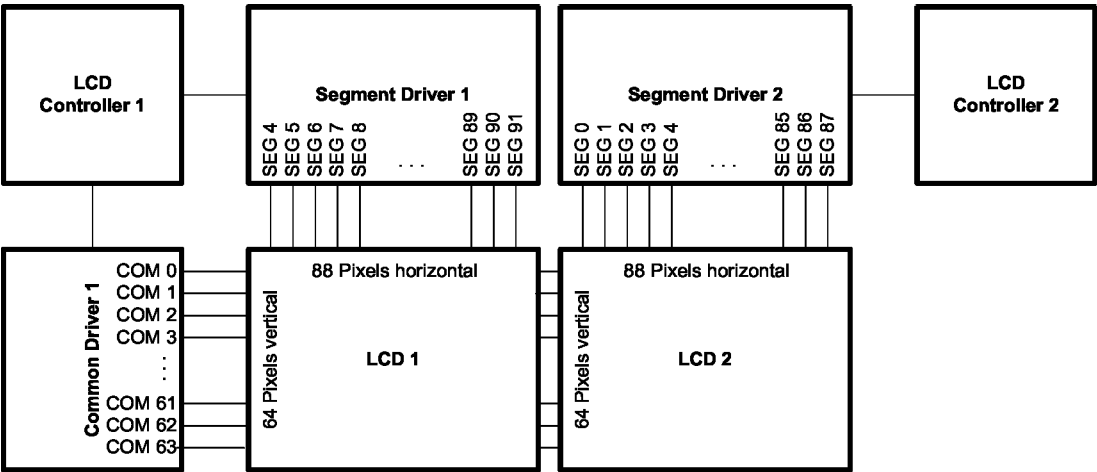
第 4 个 LCD 控制器

类型	宏	说明
N	LCD_FIRSTSEG3	LCD 控制器 3: 使用的第 1 段连线。
N	LCD_LASTSEG3	LCD 控制器 3: 使用的最后 1 段连线。
N	LCD_FIRSTCOM3	LCD 控制器 3: 使用的第 1 个公共极连线。
N	LCD_LASTCOM3	LCD 控制器 3: 使用的最后 1 个公共极连线。

当使用 LCD 控制器数量超过一个时，记住要确定定义了所使用的控制器的数量（参考下一节的宏 LCD_NUM_CONTROLLERS）。

范例

下图展示了一个使用两个 LCD 控制器的硬件配置。COM 连线由公共极驱动器驱动连接到控制器 1，并且直接连接到第 2 个 LCD。LCD 1 连接到段驱动器 1，使用 SEG 连线 4 到 91。LCD 2 通过段驱动器 2 的 SEG 0 到 SEG 87 驱动。



上面范例的配置

```
#define LCD_FIRSTSEG0 (4) /* Contr.0: 使用的第 1 条 seg 连线。 */
#define LCD_LASTSEG0 (91) /* Contr.0: 使用的最后 1 条 seg 连线。 */
#define LCD_FIRSTCOM0 (0) /* Contr.0: 使用的第 1 条 com 连线。 */
#define LCD_LASTCOM0 (63) /* Contr.0: 使用的最后 1 条 com 连线。 */
#define LCD_XORG0 (0) /* Contr.0: 最左边（最小）X 坐标 */
#define LCD_YORG0 (0) /* Contr.0: 最顶端（最小）Y 坐标 */
#define LCD_FIRSTSEG1 (0) /* Contr.1: 使用的第 1 条 seg 连线 */
#define LCD_LASTSEG1 (87) /* Contr.1: 使用的最后 1 条 seg 连线。 */
#define LCD_FIRSTCOM1 (0) /* Contr.1: 使用的第 1 条 com 连线。 */
#define LCD_LASTCOM1 (63) /* Contr.1: 使用的最后 1 条 com 连线。 */
#define LCD_XORG1 (88) /* Contr.1: 最左边（最小）X 坐标 */
#define LCD_YORG1 (0) /* Contr.1: 最顶端（最小）Y 坐标 */
```

20.10 COM/SEG 查询表

当使用“chip on glass”技术时，有时非常难保证控制器的 COM 和 SEG 输出是以线性方式连接到情况显示屏。在这种情况下，要求有一个 COM/SEG 查询表，告诉驱动器关于 COM/SAEG 连线是如何连接。

LCD_LUT_COM

描述

为控制器定义一个 COM 查询表。

类型

别名。

范例

我们假设你的显示屏只包含 10 条 COM 连线，它们的连接序列是 0, 1, 2, 6, 5, 4, 3, 7, 8, 9。为了配置 LCD 驱动器，使 COM 连线能以正确的序列访问，下面的宏应该加入你的 LCDConf.h 文件当中：

```
#define LCD_LUT_COM 0, 1, 2, 6, 5, 4, 3, 7, 8, 9
```

如果你需要修改段序列，你应该以同样的形式使用 LCD_LUT_SEG 宏。

LCD_LUT_SEG**描述**

为控制器定义一个 SEG 查询表。

类型

别名。

20.11 杂项**LCD_NUM_CONTROLLERS****描述**

定义使用的 LCD 控制器数量。

类型

数值（默认为 1）。

LCD_CACHE**描述**

控制 CPU 内存中视频内存的高速缓存。

类型

二进制开关

0：禁止，没有使用显示数据高速缓存；1：启用，使用显示数据高速缓存（默认）。

附加信息

该开关并不是所有 LCD 驱动器都支持。

如果访问图像存储器的速度太慢的话，推荐使用一个显示屏高速缓存（可以对存取速度进行加速），这是大一些的显示屏和使用简单总线接口（特别是使用端口访问或串行接口）时的通常做法。禁止高速缓存会使用驱动器的速度慢下来。

LCD_USE_BITBLT

描述

控制硬件加速度的使用。

类型

二进制开关。

0：禁止，未使用 BitBLT 引擎；1：启用，使用 BitBLT 引擎（默认）。

附加信息

禁止 BitBLT 引擎将通知驱动器不使用有效的硬件加速。

LCD_SUPPORT_CACHECONTROL

描述

开关支持驱动器函数 LCD_L0_ControlCache()。

类型

二进制开关

0: 禁止, LCD_L0_ControlCache() 不能使用 (默认); 1: 启用, LCD_L0_ControlCache() 可以使用。

附加信息

API 函数 LCD_L0_ControlCache() 允许高速缓存的锁定, 解锁或清除。要了解更多的信息, 参阅第 23 章 “LCD 驱动 API 函数”。请注意, 该特性只有一些使用简单总线接口的 LCD 控制器才有。用尽可能短的时间访问控制器这很重要, 这样可以获得最大的速度。对于其它控制器, 该开关无效。

LCD_TIMERINIT0

描述

用于一个显示一个像素的 pane 0 的中断服务函数的定时值。

类型

数值。

附加信息

该宏只有在没有使用 LCD 控制器时才有作用, 因为它是一项 CPU 通过一个中断服务程序来更新显示的工作。

LCD_TIMERINIT1

描述

用于一个显示一个像素的 pane 1 的中断服务函数的定时值。

类型

数值。

附加信息

该宏只有在没有使用 LCD 控制器时才有作用, 因为它是一项 CPU 通过一个中断服务程序

来更新显示的工作。

LCD_ON

描述

切换 LCD 到开状态。

类型

函数替换。

LCD_OFF

描述

切换 LCD 到关状态。

类型

函数替换。

第21章 高层次配置

高层次配置相当简单。在一开始，你通常可以使用已有的配置文件（例如，那些在仿真器当中使用的）。只有在需要对系统进行调整，或减小内存消耗时，配置文件 GUIConf.h 才需要修改。该文件通常位于你的工程根目录的“Config”子目录下。使用 GUIConf.h 进行任意的高层次配置。

在你的硬件上使用 μ C/GUI 要做的第二件事情是修改依赖于硬件的函数，位于文件 Sample\GUI_X\GUI_X.c 中。

21.1 有效的GUI 配置宏

下表列出了用于 μ C/GUI 的高层次配置的有效宏。

类型	宏	默认值	说明
N	GUI_ALLOC_SIZE	1000	定义可选的动态存储器的尺寸（有效的字节数）。动态存储器只有窗口和存储设备才需要。该存储器只能在 GUIAlloc 模块连接时才使用。而这种连接也只是在有动态存储器需求时，才发生。
S	GUI_DEBUG_LEVEL		定义哪个消息要通过 GUI_X_Log() 传递。你应该在本章后面有关 GUI_X_Log() 的内容中找到详细的描述。
N	GUI_DEFAULT_BKCOLOR		定义默认背景颜色。
N	GUI_DEFAULT_COLOR		定义默认前景颜色。
S	GUI_DEFAULT_FONT	&GUI_Font6x8	定义在执行 GUI_Init() 后，哪一种字体作为默认字体。如果你没有使用默认字体，那么改变为不同的默认值是很有意义。一旦默认字体被代码引用，它会因此总是处于连接状态。
N	GUI_MAXTASK	4	当多任务支持被启用时，定义调用 μ C/GUI 访问显示屏的的最大任务的数量（请参阅第 11 章“执行模型：单任务/多任务”）。
B	GUI_OS	0	多个任务调用 μ C/GUI 时激活启用多任务的支持（请参阅第 11 章“执行模型：单任务/多任务”）。
B	GUI_SUPPORT_MEMDEV	1	启用可选的存储器设备支持。不使用存储器设备会节省大约 40 字节用于函数指针的 RAM，同时轻微地增加运行速度。
B	GUI_SUPPORT_TOUCH	0	启用可选的触摸屏支持。
B	GUI_SUPPORT_UNICODE	1	启用含有 8 位字符串的 Unicode 字符支持。请注意：Unicode 字符始终显示，而字符代码始终被当作 16 位处理。
B	GUI_WINSUPPORT	0	启用可选的视窗管理器支持。

如何配置GUI

我们推荐使用下面的操作步骤

1. 将原始配置文件备份。
2. 检查所有的配置开关。
3. 删除配置中不使用的部分。

配置范例

以下是一个简略的 GUI 配置文件（ConfigSample\GUIConf.h）：

```

/*****
*
*                      期望功能的配置
*
*****/

#define GUI_WINSUPPORT      (1)    /* 如果为真（1），使用视窗管理器。*/
#define GUI_SUPPORT_TOUCH  (1)    /* 使用触摸屏 */

/*****
*
*                      动态存储器的配置
*
*****/

动态存储器用于存储器设备和视窗管理器。
如果你不使用这些特性，没有必要使用动态存储器，它可以完全关闭。（这部分可以删除）
*/

#ifndef GUI_ALLOC_SIZE
#define GUI_ALLOC_SIZE  5000    /* 动态存储器的尺寸 */
#endif

/*****
*
*                      字体的配置
*
*****/

如果你建立了附加字体（通常使用μC/GUI-FontConvert 来做到），为了能使用它们，这些
字体需要按外来字体定义。这里是做这项工作的好地方。*/

#define GUI_DEFAULT_FONT  &GUI_Font6x8    /* 该字体用作默认字体 */

```

21.2 GUI_X 函数参考

当在你的硬件上使用 μ C/GUI 的时候，在你的工程当中必须有几个依赖于硬件的函数。使用仿真器时，库已经包含了它们。可以找到一个范例文件，Sample\GUI_X\GUI_X.c。下表在各自的分类中按字母顺序列出了有效的依赖于硬件的函数。这些函数的详细描述在本章稍后部分给出。

函 数	说 明
初始化函数	
GUI_X_Init()	从 GUI_Init() 调用，能用于初始化硬件。
时间函数	
GUI_X_Delay()	在一个指定的时间段后返回。
GUI_X_ExecIdle()	只从视窗管理器的非阻塞函数调用。
GUI_X_GetTime()	返回当前系统时间，以毫秒为单位。
内核接口函数	
GUI_X_InitOS()	初始化内核接口模块（建立一个资源旗语/互斥）。
GUI_X_GetTaskId()	返回一个当前任务/线程唯一的 32 位标识符。
GUI_X_Lock()	锁上 GUI（阻塞资源旗语/互斥）。
GUI_X_Unlock()	解锁 GUI（解除资源旗语/互斥阻塞）。
GUI_X_Log()	返回调试信息。如果启用记录的话，这是必需的。

21.3 初始化函数

GUI_X_Init()

描述

从 GUI_Init() 调用，能用于初始化硬件。

函数原型

```
void GUI_X_Init (void) ;
```

21.4 时间函数

GUI_X_Delay()

描述

在一个指定的时间段（以毫秒为单位）后返回。

函数原型

```
void GUI_X_Delay(int Period)
```

参数	含义
Period	以毫秒为单位的周期。

GUI_X_ExecIdle()

描述

只从视窗管理器的非阻塞函数调用。

函数原型

```
void GUI_X_ExecIdle (void) ;
```

附加信息

当不再有要求处理的消息时调用。在这一点是，GUI 是最新的。

GUI_X_GetTime()

描述

用于函数 GUI_GetTime，返回当前系统时间（以毫秒为单位）。

函数原型

```
int GUI_X_GetTime (void)
```

返回值

以毫秒为单位的当前系统时间，整数类型。

21.5 内核接口函数

这些函数的详细描述请参阅第 11 章“执行模型：单任务/多任务”。

21.6 调试

GUI_X_Log()

描述

从 μ C/GUI 返回调试信息。

函数原型

```
void GUI_X_Log(const char * s);
```

参数	含义
s	要发送的字符串的指针。

附加信息

μ C/GUI 调用该函数进行错误消息和警告的传送，如果启用记录的话，它是必需的。GUI 调用这个函数依赖于配置宏 GUI_DEBUG_LEVEL。下表列出了 GUI_DEBUG_LEVEL 允许的值：

值	说明
GUI_DEBUG_LEVEL_NOCHECK	没有执行运行时间检测。
GUI_DEBUG_LEVEL_CHECK_PARA	执行参数检测以避免碰撞。
GUI_DEBUG_LEVEL_CHECK_ALL	执行参数检测和一致性检测。
GUI_DEBUG_LEVEL_LOG_ERRORS	记录错误。
GUI_DEBUG_LEVEL_LOG_WARNINGS	记录错误和警告。
GUI_DEBUG_LEVEL_LOG_ALL	记录错误、警告和消息。

第22章 LCD驱动程序

一个 LCD 驱动程序支持一个具体系列的 LCD 控制器，而所有的 LCD 都配备一个或多个这些控制器。驱动程序本质上是通用的，意思是它可以通过修改配置文件 LCDConf.h 来进行配置。这些文件包含所有可配置的选项用于驱动程序，以及多重的定义用于硬件如何访问及控制器如何与 LCD 连接。

本章提供 μ C/GUI的LCD控制器的概述。它针对每一个驱动程序说明以下内容：

- 哪一个 LCD 控制器能被访问，及支持的颜色深度和接口类型。
- 额外的 RAM 需求。
- 附加函数。
- 如何访问硬件。
- 指定配置开关。
- 特定的 LCD 控制器的特别需求。

22.1 支持的LCD控制器及各自的驱动程序

下表列出了驱动程序及那些控制器支持它们：

驱动程序	宏 LCD_CONTROLLER 的值	LCD 控制器	支持的位/像素 (bps)
LCD07X1	711	Samsung KS0711	2
	741	Samsung KS0741	
LCD13XX	1352	Epson SED1352, S1D13502	1, 2, 4, 8, 16
	1354	Epson SED1354, S1D13504	
	1356	Epson SED1356, S1D13506	
	1374	Epson SED1374, S1D13704	
	1375	Epson SED1375, S1D13705	
	1376	Epson SED1376, S1D13706	
	1386	Epson SED1386, S1D13806	
	1300	Epson S1D13A03, S1D13A04	
LCD159A	0x159A	Epson SED159A	8
LCD15E05	0x15E05	Epson S1D15E05	2
LCD15XX	713	Samsung KS0713	1
	1560	Epson SED1560	
	1565	Epson SED1565	
	1566	Epson SED1566	
	1567	Epson SED1567	
	1568	Epson SED1568	
	1569	Epson SED1569	
	1575	Epson SED1575	
LCD6642X	66420	Hitachi HD66420	2
	66421	Hitachi HD66421	
LCDMem	0	无控制器，写入 RAM (单色显示)	2
LCDMemC	0	无控制器，写入 RAM (彩色显示)	3, 6
LCDPage1bpp	8811	Philips PCF8810, PCF8811	1
LCDSLin	1330	Epson SED1330	1
	1335	Epson SED1335	
	6963	Toshiba T6963	

选择一个驱动程序

如第 20 章“底层配置”中所描述的那样，宏 LCD_CONTROLLER 定义所使用的 LCD 控制器。一个控制器由上表列出它的适当的值指定。

下面部分分别讨论每一个有效的驱动程序。

22.2 LCD07X1

支持的硬件

控制器

该驱动程序在下列控制器身上测试通过：

- Samsung KS0711
- Samsung KS0741

应该能假设它也能够任何与这两种控制器结构相类似的控制器上工作。

每像素的位

支持颜色深度为 2 bpp。

接口

芯片支持三种类型接口：

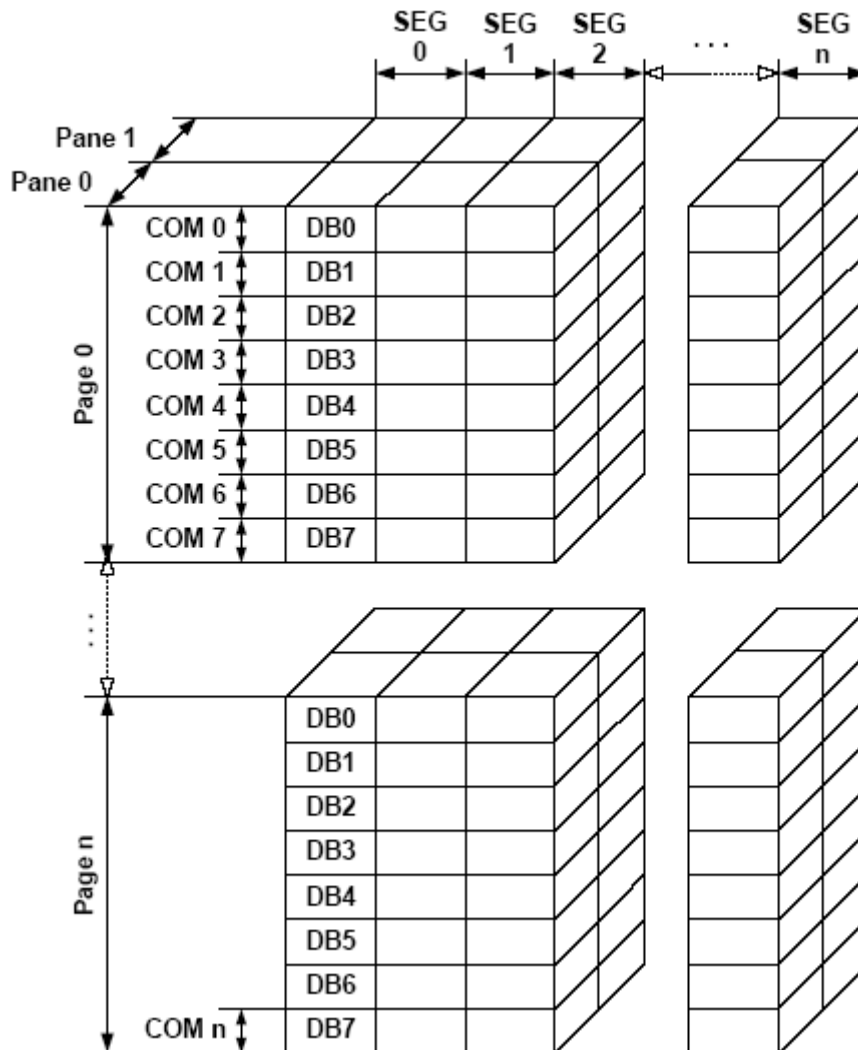
8 位并行（简单总线）接口

4 脚串行外围设备接口（SPI）

3 脚 SPI。

当前版本的驱动程序支持并行或 4 脚 SPI 模式。

显示屏数据 RAM 的结构



上图展示了显示存储器和 LCD 的 SEG 和 COM 引线之间的关系。对于每个像素，显示存储器被分成两个方框。每个像素较低的位存储在 pane0 中而较高的位存储在 pane 1 中。

驱动程序的额外 RAM 要求

这些 LCD 驱动程序可以使用或不使用一个显示数据高速缓存，包含一个 LCD 数据 RAM 的容量的完全拷贝。如果高速缓存未使用，则没有额外的 RAM 需求。

推荐使用这些驱动程序时，一起使用一个数据高速缓存，以获得更快的 LCD 访问速度。用于高速缓存的内存的数值可以由以下公式计算：

RAM 的大小（字节）= (LCD_YSIZE+7)/8*LCD_XSIZE*2

附加的驱动函数

LCD_L0_ControlCache

有关这个函数的信息，请参阅第 23 章“LCD 驱动程序 API”。

硬件配置

这个驱动程序使用一个如第 20 章“低层配置”所描述的简单总路线接口访问硬件。下表列出了必须为硬件访问所定义的宏：

并行模式

宏	说 明
LCD_INIT_CONTROLLER	初始化 LCD 控制器序列。
LCD_READ_A0	A 线（A-line）为低电平时从 LCD 控制器读一个字节。
LCD_READ_A1	A 线（A-line）为高电平时从 LCD 控制器读一个字节。
LCD_WRITE_A0	A 线（A-line）为低电平时向 LCD 控制器写入一个字节。
LCD_WRITE_A1	A 线（A-line）为高电平时向 LCD 控制器写入一个字节。

串行模式

宏	说 明
LCD_INIT_CONTROLLER	初始化 LCD 控制器序列。
LCD_WRITE_A0	A 线（A-line）为低电平时向 LCD 控制器写入一个字节。
LCD_WRITE_A1	A 线（A-line）为高电平时向 LCD 控制器写入一个字节。
LCD_WRITEM_A0	A 线（A-line）为低电平时向 LCD 控制器写入多个字节。
LCD_WRITEM_A1	A 线（A-line）为高电平时向 LCD 控制器写入多个字节。

附加的配置开关

无。

某些LCD控制器的特定要求

无。

22.3 LCD13XX

支持的硬件

控制器

该驱动程序通过了下列 LCD 控制器的测试：

- Epson SED1352, S1D13502
- Epson SED1354, S1D13504
- Epson SED1356, S1D13506
- Epson SED1374, S1D13704
- Epson SED1375, S1D13705
- Epson SED1376, S1D13706
- Epson SED1386, S1D13806
- Epson S1D13A03, S1D13A04

应该能假设它也能够任何与以上控制器结构相类似的控制器上工作。

每像素的位

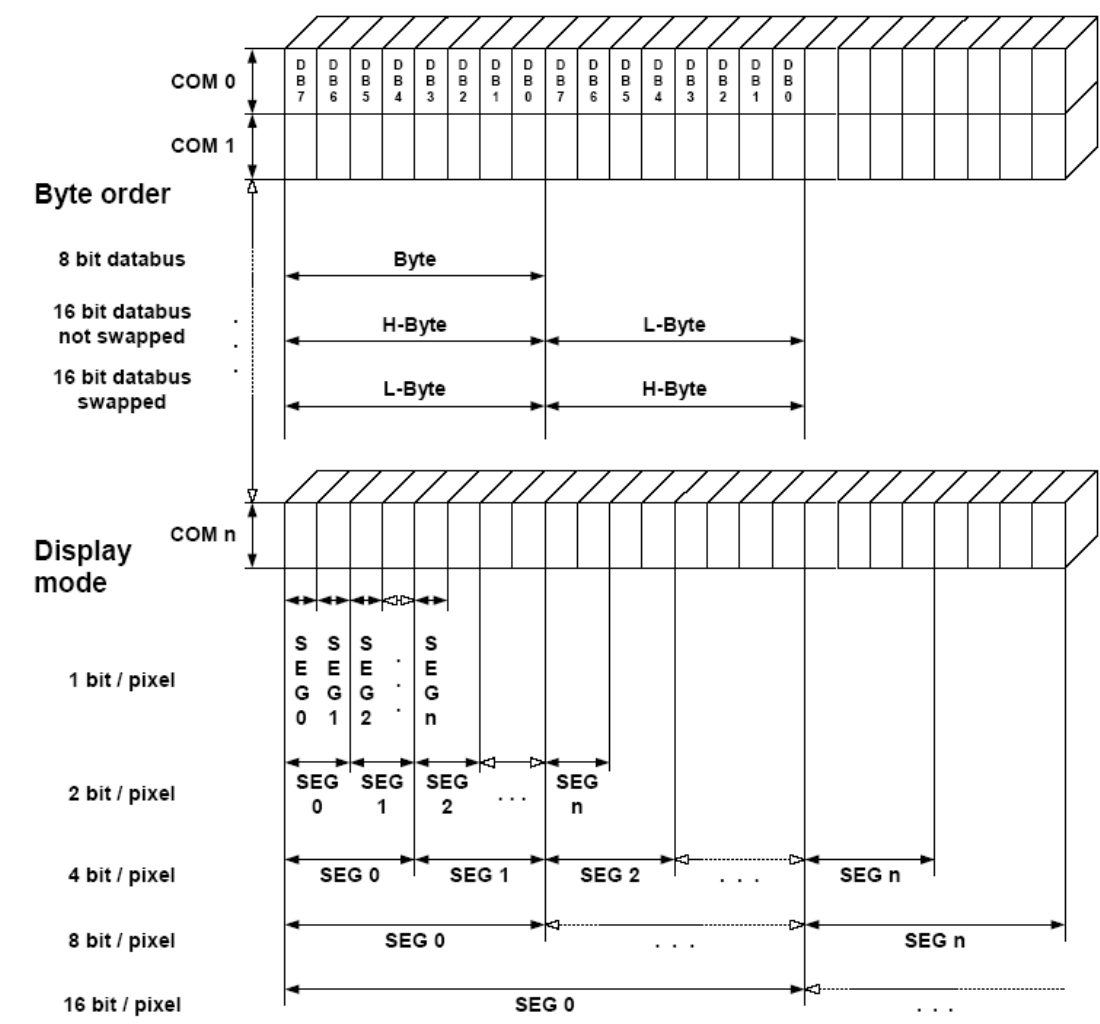
支持的颜色深度是 1, 2, 4, 8 和 16 bpp。

接口

由这个驱动程序支持的芯片，能够以 8/16 位并行（完全总线）模式进行连接。

驱动程序同时支持两种接口。请参阅该各自的 LCD 控制器手册以确定是否你的芯片能够以 8 位模式连接。

显示屏数据 RAM 的结构



上图展示了依据颜色深度，显示存储器和 LCD 的 SEG 和 COM 引线之间的关系。

驱动程序的附加 RAM 要求

无。

附加的驱动函数

无。

硬件配置

该驱动程序需要使用一个如第20章“低层配置”所描述的完全的总线接口来进行硬件访

问。下表列出了必须为硬件访问所定义的宏：

宏	说明
<code>LCD_INIT_CONTROLLER</code>	初始化 LCD 控制器序列。
<code>LCD_READ_MEM</code>	读控制器的图像存储器的内容。
<code>LCD_READ_REG</code>	读控制器的一个配置寄存器的内容。
<code>LCD_WRITE_MEM</code>	向控制器的图像存储器（显示数据随机 RAM）写入数据。
<code>LCD_WRITE_REG</code>	向控制器的一个配置寄存器写入数据。

附加的配置开关

下表展示了对于这个驱动程序有效的可选择配置开关：

宏	说明
<code>LCD_BUSWIDTH</code>	选择 LCD 控制器/ CPU 接口的总线宽度（8/16）。默认值是 16。
<code>LCD_ENABLE_MEM_ACCESS</code>	切换 M/R 信号到存储器访问。仅仅用于 SED1356 和 SED1386 LCD 控制器。
<code>LCD_ENABLE_REG_ACCESS</code>	切换 M/R 信号到寄存器访问。仅仅用于 SED1356 和 SED1386 LCD 控制器。
<code>LCD_SWAP_BYTE_ORDER</code>	当使用一个 16 位总线接口时，反转 CPU 和 LCD 控制器间的头端模式（高低字节交换）。
<code>LCD_USE_BITBLT</code>	如果设置为 0，禁止 BitBLT 引擎。如果设置为 1（缺省值），驱动程序将使用全部有效的硬件加速。
<code>LCD_ON</code>	LCD 切换到“开”的功能置换宏。
<code>LCD_OFF</code>	LCD 切换到“关”的功能置换宏。

某些LCD控制器SED1386或者S1 D13806的特殊要求

LCD_SWAP_RB

该配置开关 LCD_SWAP_RB（交换红和蓝色部分）必须通过向 LCDConf.h 插入下面一行而激活（设置为 1）：

```
#define LCD_SWAP_RB (1)          /* 必须设置 */
```

LCD_INIT_CONTROLLER

当写或者修改初始化宏时，要考虑下列问题：

初始化嵌入 SDRAM，寄存器 20 的第 7 位（SDRAM 初始化位）必须设置为 1（至少在复位

200μs 后)。

- 当 SDRAM 初始化位设置后，实际的初始化序列发生在第一个 SDRAM 刷新周期。该初始化序列需要大约 16 个 MCLK 来完成，并且在初始化在进行中的时候不能有存储器访问的操作。

更多信息，请参见 LCD 控制器技术资料。

LCD_READ_REG, LCD_WRITE_REG

为了使 BitBLT（位块传送）引擎工作，偏移量的数据类型必须是无符号长整数（unsigned long）。这要用配置宏 LCD_READ_REG 和 LCD_WRITE_REG 作如下所示进行设置：

```
#define LCD_READ_REG(Off) *((volatile U16*) (0x800000+(((U32) (Off)) <<1)))  
#define LCD_WRITE_REG(Off,Data) *((volatile U16*) (0x800000+(((U32) (Off)) <<1)))=Data
```

22.4 LCD159A

支持的硬件

控制器

该驱动程序已经通过下列 LCD 控制器的测试：

- Epson SED159A

应假定它也能工作在任何与些结构相类似的控制器上。

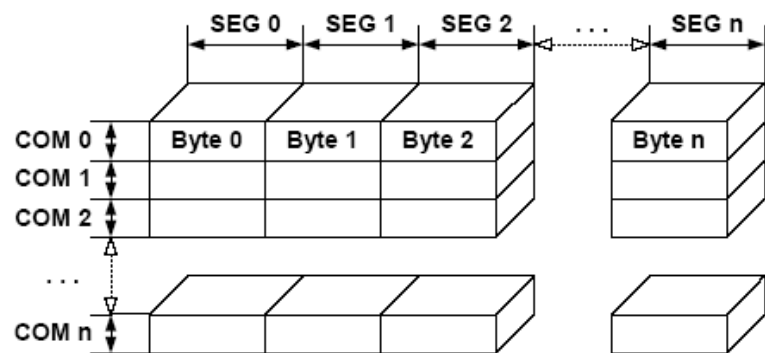
每像素的位

支持 8bpp 颜色深度。

接口

该驱动程序支持 8 位并行（简单总线）接口。

显示屏数据 RAM 的结构



上图展示了显示内存和 LCD 的 SEG 和 COM 线之间的关系。

额外的 RAM 要求

无。

附加的驱动函数

无。

硬件配置

这个驱动程序使用一个如第 20 章“低层配置”所描述的简单总路线接口访问硬件。下表列出了必须为硬件访问所定义的宏：

宏	说明
LCD_INIT_CONTROLLER	初始化 LCD 控制器序列。
LCD_READ_A0	A 线（A-line）为低电平时从 LCD 控制器读一个字节。
LCD_READ_A1	A 线（A-line）为高电平时从 LCD 控制器读一个字节。
LCD_WRITE_A0	A 线（A-line）为低电平时向 LCD 控制器写入一个字节。
LCD_WRITE_A1	A 线（A-line）为高电平时向 LCD 控制器写入一个字节。

附加的配置开关

无。

某些LCD控制器的特殊要求

无。

22.5 LCD15E05

支持的硬件

控制器

该驱动程序已经通过下列 LCD 控制器的测试：

- Epson S1D15E05

应假定它也能工作在任何与些结构相类似的控制器上。

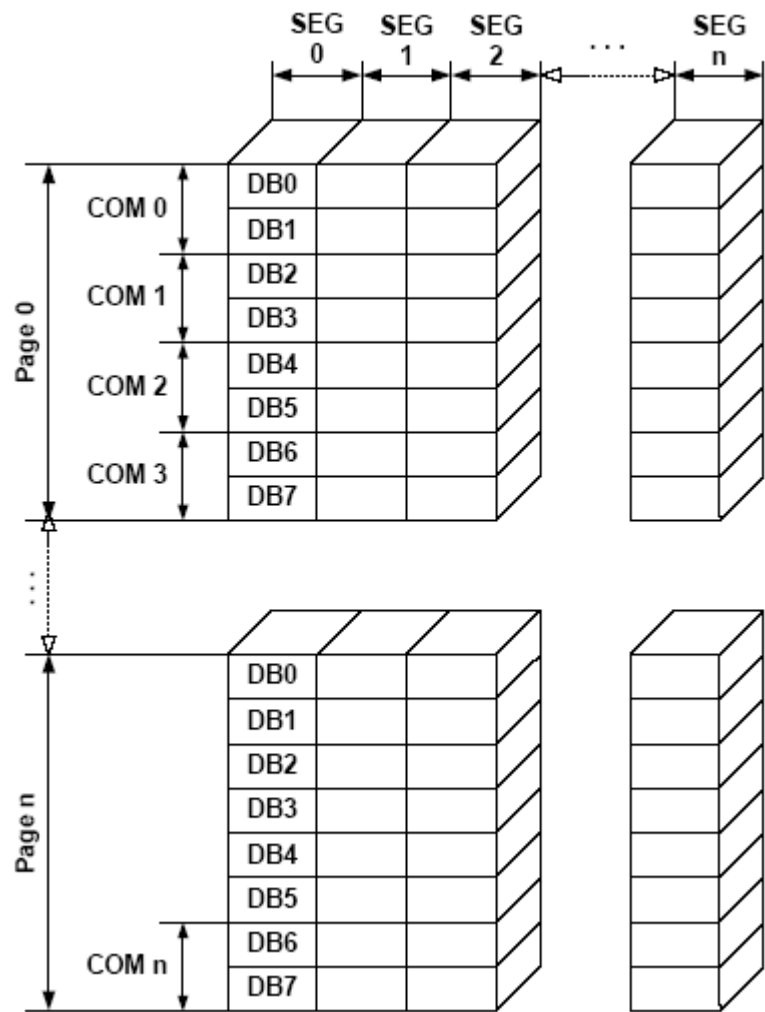
每像素的位

支持的颜色深度是 2bpp。

接口

同时支持 8 位并行（简单总线）和串联（SPI）接口。

显示屏数据 RAM 的结构



上图展示了显示存储器和 LCD 的 SEG 和 COM 引线之间的关系。

驱动程序额外的RAM需求

这些 LCD 驱动程序可以使用或不使用一个显示数据高速缓存，包含一个 LCD 数据 RAM 的容量的完全拷贝。 如果不使用高速缓存，则没有额外的 RAM 需求。

推荐使用这些驱动程序时，一起使用一个数据高速缓存，以获得更快的 LCD 访问速度。用于高速缓存的内存的数值可以由以下公式计算：

RAM 的大小（字节）= (LCD_YSIZE+7)/8* LCD_XSIZE

附加的驱动函数

无。

硬件配置

这个驱动程序使用一个如第 20 章“低层配置”所描述的简单总路线接口访问硬件。下表列出了必须为硬件访问所定义的宏：

宏	说 明
<code>LCD_INIT_CONTROLLER</code>	初始化 LCD 控制器序列。
<code>LCD_READ_A0</code>	A 线 (A-line) 为低电平时从 LCD 控制器读一个字节。
<code>LCD_READ_A1</code>	A 线 (A-line) 为高电平时从 LCD 控制器读一个字节。
<code>LCD_WRITE_A0</code>	A 线 (A-line) 为低电平时向 LCD 控制器写入一个字节。
<code>LCD_WRITE_A1</code>	A 线 (A-line) 为高电平时向 LCD 控制器写入一个字节。

附加的配置开关

下表展示了对于这个驱动程序有效的可选择配置开关：

宏	说 明
<code>LCD_CACHE</code>	当设置为 0 时，没有使用显示数据高速缓存，会降低驱动程序的速度。默认值是 1（高速缓存激活）。

某些LCD控制器的特殊要求

无。

22.6 LCD15XX

支持的硬件

控制器

该驱动程序已经通过下列 LCD 控制器的测试：

- Samsung KS0713
- Epson SED1560
- Epson SED1565

- Epson SED1566
- Epson SED1567
- Epson SED1568
- Epson SED1569
- Epson SED1575

应假定它也能工作在任何与些结构相类似的控制器上。

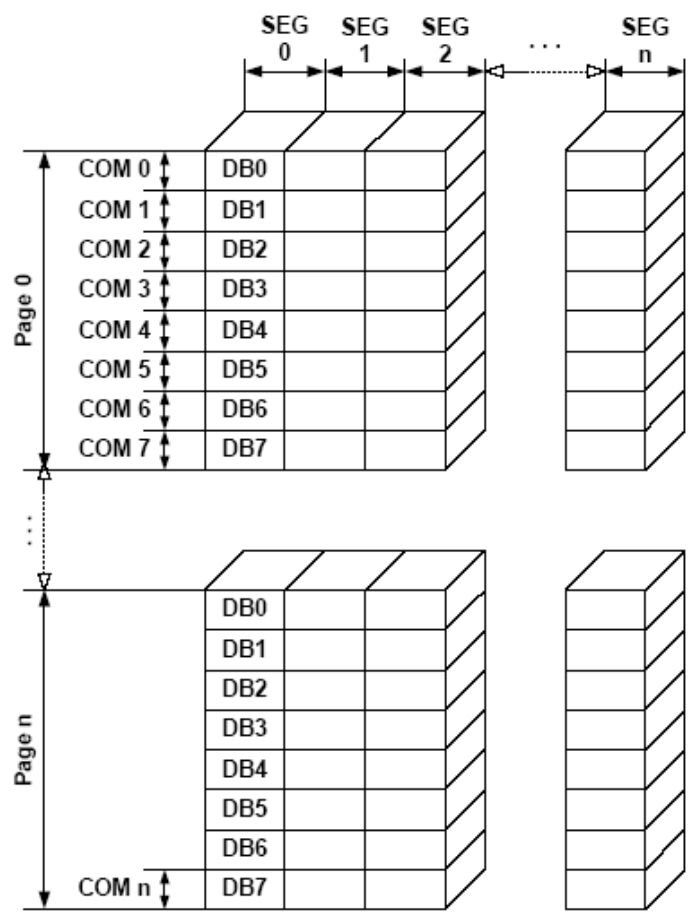
每像素的位

支持的颜色深度是 1bpp。

接口

同时支持 8 位并行（简单总线）和串联（SPI）接口。

显示屏数据 RAM 的结构



上图展示了显示存储器和 LCD 的 SEG 和 COM 引线之间的关系。

驱动程序额外的 RAM 需求

这些 LCD 驱动程序可以使用或不使用一个显示数据高速缓存，包含一个 LCD 数据 RAM 的容量的完全拷贝。如果不使用一个高速缓存，则没有额外的 RAM 需求。

推荐使用这些驱动程序时，一起使用一个数据高速缓存，以获得更快的 LCD 访问速度。用于高速缓存的内存的数值可以由以下公式计算：

$$\text{RAM 的大小（以字节为单位）} = (\text{LCD_YSIZE} + 7) / 8 * \text{LCD_XSIZE}$$

附加的驱动函数

LCD_L0_ControlCache

有关这个函数的信息，请参阅第 23 章“LCD 驱动程序 API”。

硬件配置

这个驱动程序使用一个如第 20 章“低层配置”所描述的简单总路线接口访问硬件。下表列出了必须为硬件访问所定义的宏：

宏	说 明
<code>LCD_INIT_CONTROLLER</code>	初始化 LCD 控制器序列。
<code>LCD_READ_A0</code>	A 线（A-line）为低电平时从 LCD 控制器读一个字节。
<code>LCD_READ_A1</code>	A 线（A-line）为高电平时从 LCD 控制器读一个字节。
<code>LCD_WRITE_A0</code>	A 线（A-line）为低电平时向 LCD 控制器写入一个字节。
<code>LCD_WRITE_A1</code>	A 线（A-line）为高电平时向 LCD 控制器写入一个字节。

附加的配置开关

下表展示了对于这个驱动程序有效的可选择配置开关：

宏	说 明
<code>LCD_CACHE</code>	当设置为 0 时，没有使用显示数据高速缓存，会降低驱动程序的速度。默认值是 1（高速缓存激活）。
<code>LCD_SUPPORT_CACHECONTROL</code>	当设置为 0 时，LCD_L0_ControlCache () 驱动程序 API 的高速缓存控制功能被禁止。

某些LCD控制器的特殊要求

无。

22.7 LCD6642X

支持的硬件

控制器

该驱动程序已经通过下列 LCD 控制器的测试：

- Hitachi HD66420
- Hitachi HD66421

应假定它也能工作在任何与些结构相类似的控制器上。

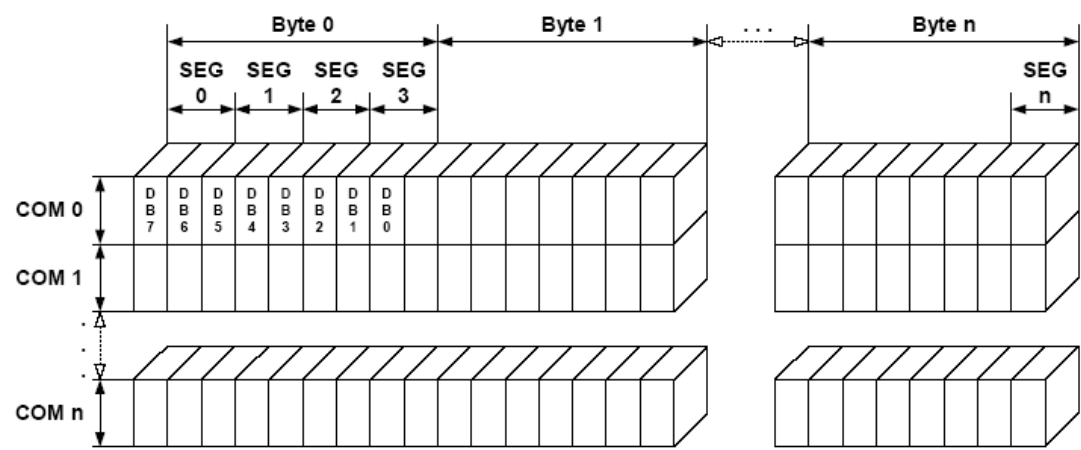
每像素的位

支持的颜色深度是 2bpp。

接口

该驱动程序支持 8 位并行（简单总线）接口。

显示屏数据 RAM 的结构



上图展示了显示存储器和 LCD 的 SEG 和 COM 引线之间的关系。

驱动程序额外的RAM需求

这些 LCD 驱动程序可以使用或不使用一个显示数据高速缓存，包含一个 LCD 数据 RAM 的容量的完全拷贝。如果不使用高速缓存，则没有额外的 RAM 需求。

可选的（而不是推荐），使用这些驱动程序时，同时使用一个数据高速缓存，以获得更快的LCD访问速度。用于高速缓存的内存的数值可以由以下公式计算：

$$\text{RAM的大小（字节）} = (\text{LCD_YSIZE} + 7) / 8 * \text{LCD_YSIZE} * 2$$

附加的驱动函数

无。

硬件配置

这个驱动程序使用一个如第20章“低层配置”所描述的简单总路线接口访问硬件。下表列出了必须为硬件访问所定义的宏：

宏	说 明
<code>LCD_INIT_CONTROLLER</code>	初始化 LCD 控制器序列。
<code>LCD_READ_A0</code>	A 线（A-line）为低电平时从 LCD 控制器读一个字节。
<code>LCD_READ_A1</code>	A 线（A-line）为高电平时从 LCD 控制器读一个字节。
<code>LCD_WRITE_A0</code>	A 线（A-line）为低电平时向 LCD 控制器写入一个字节。
<code>LCD_WRITE_A1</code>	A 线（A-line）为高电平时向 LCD 控制器写入一个字节。

附加的配置开关

下表展示了对于这个驱动程序有效的可选择配置开关：

宏	说 明
<code>LCD_CACHE</code>	当设置为 0 时，没有使用显示数据高速缓存，会降低驱动程序的速度。默认值是 1（高速缓存激活）。

某些 LCD 控制器的特殊要求

无。

22.8 LCDMem

使用CPU作为LCD控制器

在相对快速的 CPU 和小的（四分之一 VGA 或者更小）LCD 中，没有必要使用一个 LCD 控制器。微型控制器（CPU）能兼做 LCD 控制器的工作，在一中断服务程序中刷新屏幕。该 CPU 的内存被用作图像存储器。

该方法包括下列优点：

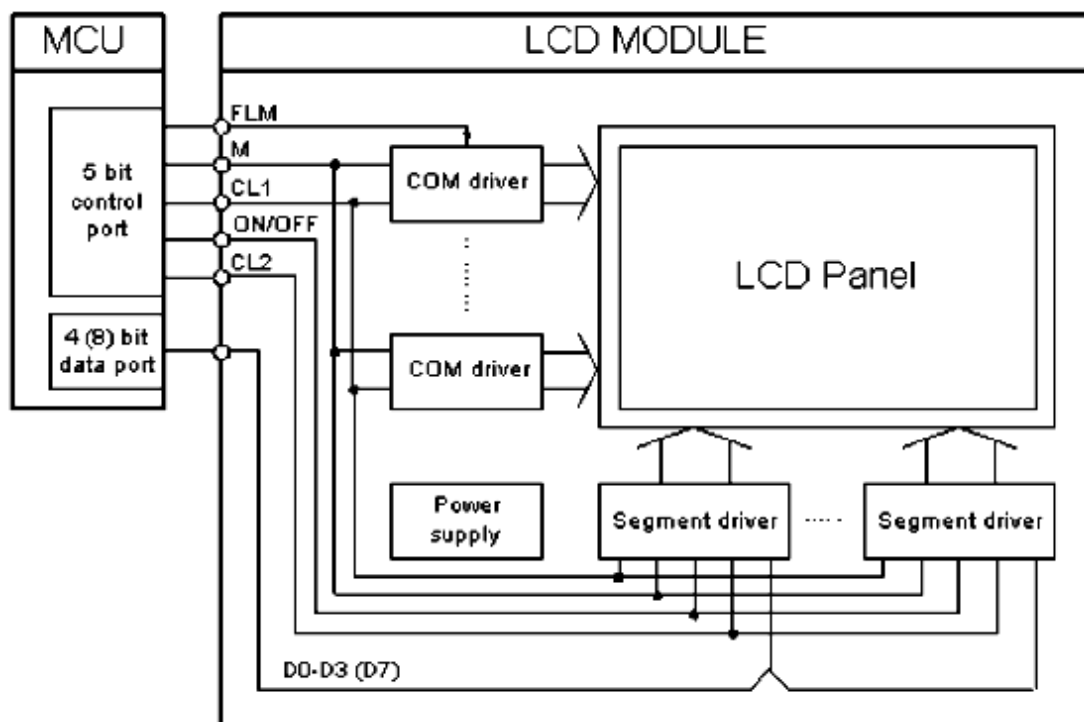
- 可以做到非常快速的屏幕更新。
- 除去了 LCD 控制器（和它的外部 RAM），减少了硬件成本。
- 简化了硬件设计。
- 可以显示 4 级灰阶。

缺点是会占用很大一部分 CPU 计算时间。根据不同的 CPU，这会占到 CPU 的开销的 20% 到几乎 100%之间；对于较慢的 CPU，它根本是极不合理的。

这类接口不需要一个特殊的 LCD 驱动程序，因为 uC/GUI 简单地将所有显示数据放入 LCD 高速缓存中。你自己必须写硬件相关部分软件，周期性地将数据从高速缓存的内存传递到你的 LCD。对于 M16C 和 M16C/80，传递图像到显示屏中的范例代码可以用“C”和最佳化的汇编程序实现。

如何连接 CPU 到 行/列 驱动程序

连接微型控制器到行/列驱动程序是相当容易的。需要 5 根控制线，以及 4 根或 8 根数据线（取决于列驱动程序是否能工作在 8 位模式中）。建议使用 8 位模式，它更有效率，节约 CPU 的计算时间。全部数据线应该在单个端口上，使用端口位 0..3 或者 0..7 以保证高效率的访问。该设置说明如下：



CPU 占用率

CPU 负荷取决于硬件和使用的控制器，以及显示屏的尺寸。例如：

Mitsubishi M16C62 控制器, 16MHz, 160×100 显示屏, 8 位接口, 80 Hz 刷新频率 = 大约 12% 的 CPU 占用率。

Mitsubishi M16C62 控制器, 16MHz, 240×128 显示屏, 8 位接口, 80 Hz 刷新频率 = 大约 22% 的 CPU 占用率。

支持的硬件

控制器

无。

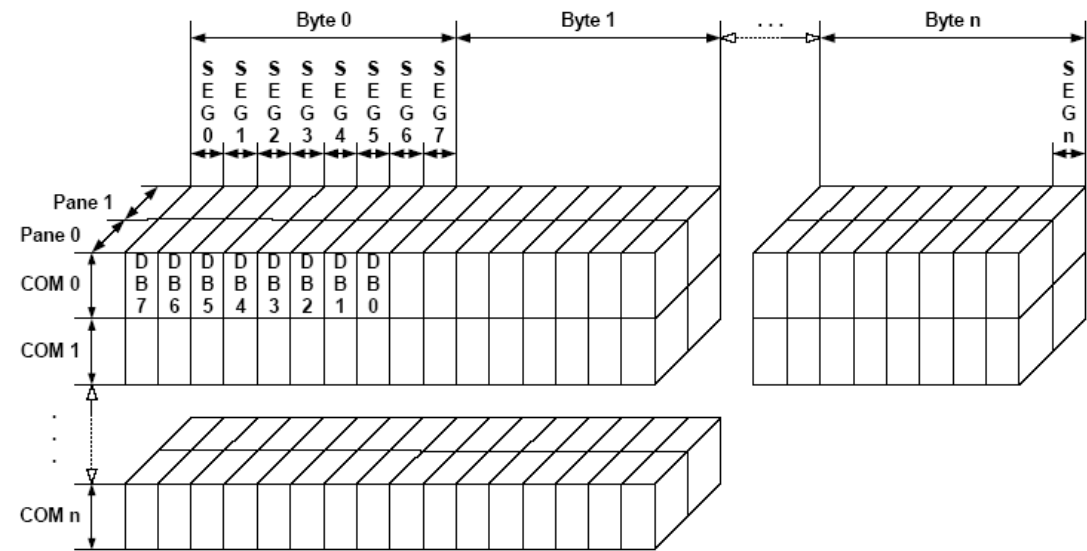
每像素的位

支持的颜色深度是 2bpp。

接口

该驱动程序支持 1/4/8 位的 CPU 连接 LCD 的接口。

显示屏数据 RAM 的结构



上图展示了显示存储器和 LCD 的 SEG 和 COM 引线之间的关系。对于每个像素，显示存储器被分成两个方框。每个像素较低的位存储在 pane0 中而较高的位存储在 pane 1 中。该方法的优点是显示数据的输出能够立即得到执行。

驱动程序的RAM需求

该驱动程序仅仅使用一个包含显示数据的存储区。所需显示存储器的大小可以计算如下：

$$\text{RAM 的大小 (字节)} = (\text{LCD_YSIZE} + 7) / 8 * \text{LCD_YSIZE} * 2$$

附加的驱动函数

无。

硬件配置

通常，该硬件接口是一个更新 LCD 的中断服务程序（ISR）。一个用“C”代码编写的输出程序随 uC/GUI 一道发布。该程序仅仅作为一个例子提供。为了最优化执行速度，它必须改编为汇编程序代码。

对于如何写输出程序的详细信息，请看一下随驱动程序一道提供的例子或与我们联系。

附加的配置开关

下表展示了对于这个驱动程序有效的可选择配置开关：

宏	说 明
<code>LCD_TIMERINIT0</code>	用于显示 pane 0 的 ISR 的时间值。
<code>LCD_TIMERINIT1</code>	用于显示 pane 1 的 ISR 的时间值。
<code>LCD_ON</code>	LCD 切换到“开”的功能置换宏。
<code>LCD_OFF</code>	LCD 切换到“关”的功能置换宏。

22.9 LCDMemC

该驱动程序，如 LCDMem，用于一个没有 LCD 控制器的系统的设计。差异是 LCDMemC 支持彩色显示器。有关使用 CPU 代替 LCD 控制器的更多信息，请参见该上述的 LCDMem 驱动程序部分。

支持的硬件

控制器

无。

每像素的位

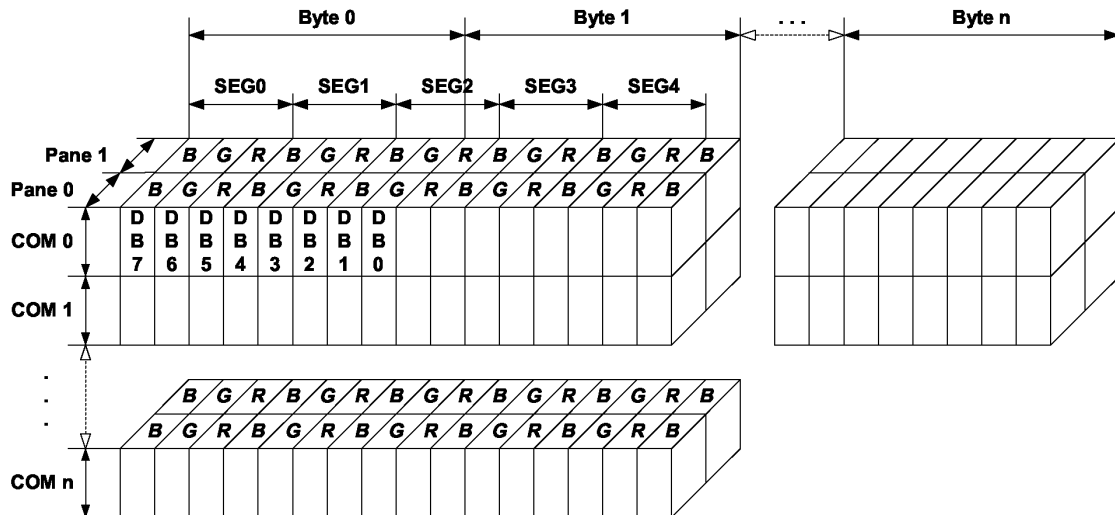
支持的颜色深度是 3 和 6bpp。

接口

该驱动程序支持 1/4/8 位的 CPU 连接 LCD 的接口。

显示屏数据 RAM 的结构

每像素 6 位，固定调色板 = 222



每像素 3 位，固定调色板 = 111

该驱动程序支持一个 3 或 6 bpp 存储区用于彩色显示器。上图展示了依据颜色深度，驱动程序和 LCD 的 SEG 和 COM 引线之间的相关性。

每像素 6 位，固定调色板模式 222

当使用 6 bpp 模式时，显示存储器被分成每像素 2 个方框。每个像素较低的位存储在 pane0 中而较高的位存储在 pane 1 中。该方法的优点是显示数据的输出能够立即得到执行。

每像素 3 位，固定调色板模式 111

当使用这个模式时，每个像素仅仅存在一个方框。

驱动程序的RAM需求

该驱动程序仅仅使用一个包含显示数据的存储区。所需显示存储器的大小可以计算如下：

每像素 6 位，固定调色板模式 222

RAM 的大小（字节）= (LCD_XSIZE+7)/8*3*2

每像素 3 位，固定调色板模式 111

RAM 的大小（字节）= (LCD_XSIZE+7)/8*3

附加的驱动函数

无。

硬件配置

通常，该硬件接口是一个更新LCD的中断服务程序（ISR）。一个用“C”代码编写的输出程序随uC/GUI一道发布。该程序仅仅作为一个例子提供。为了最优化执行速度，它必须改编为汇编程序代码。

对于如何写输出程序的详细信息，请看一下随驱动程序一道提供的例子或与我们联系。

附加的配置开关

下表展示了对于这个驱动程序有效的可选择配置开关：

宏	说 明
<code>LCD_TIMERINIT0</code>	用于显示 pane 0 的 ISR 的时间值。
<code>LCD_TIMERINIT1</code>	用于显示 pane 1 的 ISR 的时间值(仅仅用于 6 bpp 模式)。
<code>LCD_ON</code>	LCD 切换到“开”的功能置换宏。
<code>LCD_OFF</code>	LCD 切换到“关”的功能置换宏。

22.10 LCDPage1bpp

支持的硬件

控制器

该驱动程序已经通过下列 LCD 控制器的测试：

- Philips PCF8810
- Philips PCF8811

应假定它也能工作在任何与些结构相类似的控制器上。

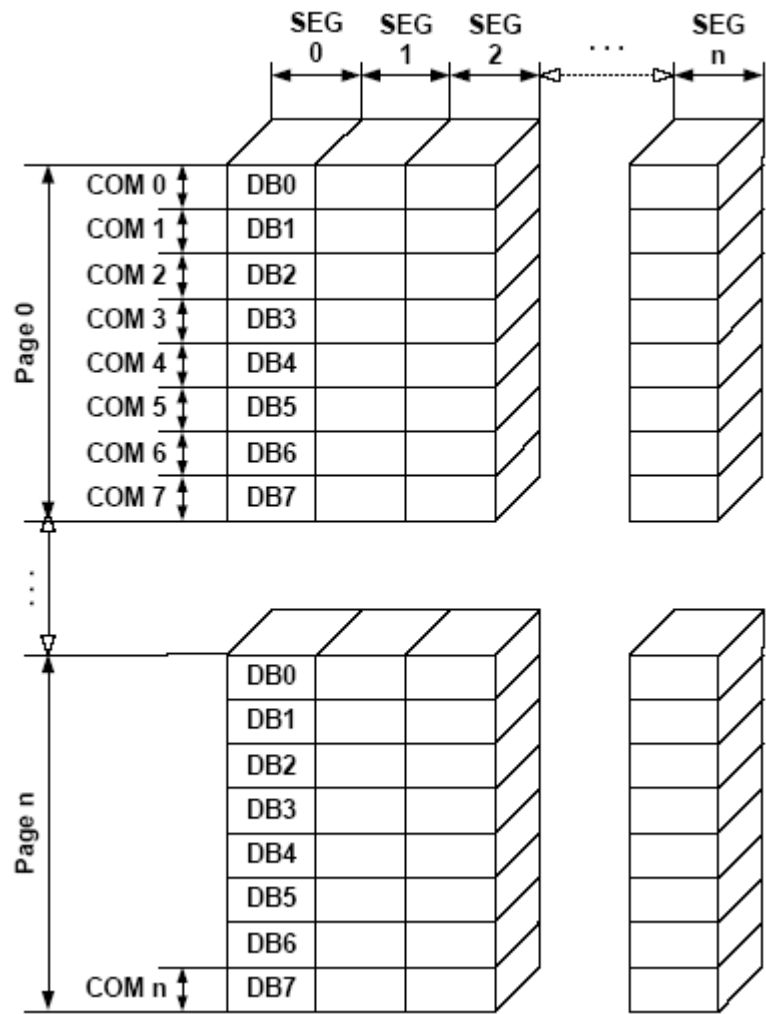
每像素的位

支持的颜色深度是 1bpp。

接口

同时支持 8 位并行（简单总线）和串联（SPI）接口。

显示屏数据 RAM 的结构



上图展示了显示存储器和 LCD 的 SEG 和 COM 引线之间的关系。

驱动程序额外的RAM需求

这些 LCD 驱动程序可以使用或不使用一个显示数据高速缓存，包含一个 LCD 数据 RAM 的容量的完全拷贝。 如果不使用高速缓存，则没有额外的 RAM 需求。

推荐使用这些驱动程序时，一起使用一个数据高速缓存，以获得更快的 LCD 访问速度。用于高速缓存的内存的数值可以由以下公式计算：

RAM 的大小（字节）=（ LCD_YSIZE+7)/8* LCD_XSIZE

附加的驱动函数

LCD_L0_ControlCache

有关这个函数的信息，请参阅第 23 章“LCD 驱动程序 API”。

硬件配置

这个驱动程序使用一个如第 20 章“低层配置”所描述的简单总路线接口访问硬件。 下表列出了必须为硬件访问所定义的宏：

宏	说 明
LCD_INIT_CONTROLLER	初始化 LCD 控制器序列。
LCD_READ_A0	A 线（A-line）为低电平时从 LCD 控制器读一个字节。
LCD_READ_A1	A 线（A-line）为高电平时从 LCD 控制器读一个字节。
LCD_WRITE_A0	A 线（A-line）为低电平时向 LCD 控制器写入一个字
LCD_WRITE_A1	A 线（A-line）为高电平时向 LCD 控制器写入一个字

附加的配置开关

下表展示了对于这个驱动程序有效的可选择配置开关：

宏	说 明
LCD_CACHE	当设置为 0 时，没有使用显示数据高速缓存，会降低驱动程序的速度。 默认值是 1（高速缓存激活）。
LCD_SUPPORT_CACHECONTROL	当设置为 0 时,LCD_L0_ControlCache () 驱动程序 API 的高速缓存控制功能被禁止。

某些LCD控制器的特殊要求

无。

22.11 LCDSLin

支持的硬件

控制器

该驱动程序已经通过下列LCD控制器的测试：

- Epson SED1330
- Epson SED1335
- Toshiba T6963

应假定它也能工作在任何与些结构相类似的控制器上。

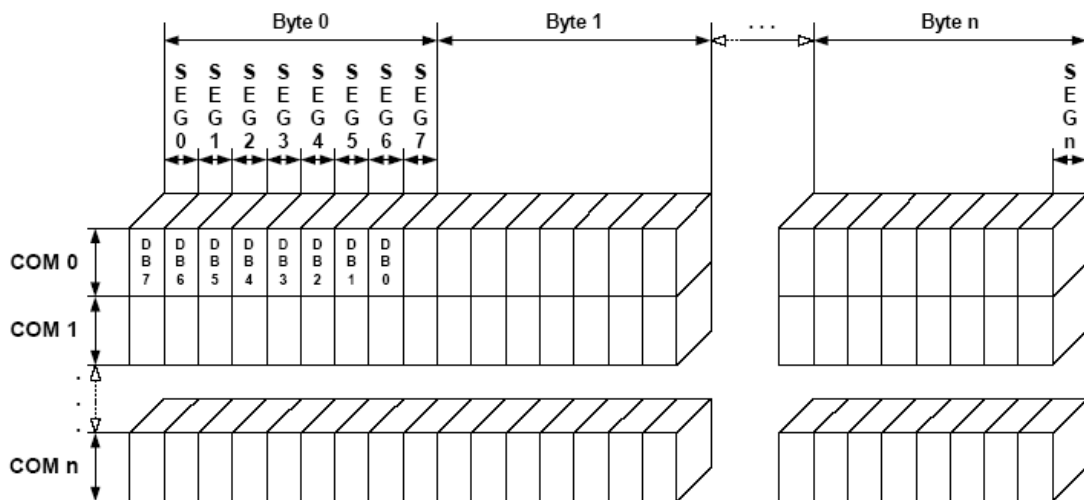
每像素的位

支持的颜色深度是 1bpp。

接口

该驱动程序支持 8 位并行（简单总线）接口。

显示屏数据 RAM 的结构



上图展示了显示存储器和 LCD 的 SEG 和 COM 引线之间的关系。

额外的RAM需求

这些 LCD 驱动程序可以使用或不使用一个显示数据高速缓存，包含一个 LCD 数据 RAM 的容量的完全拷贝。 如果没有使用调整缓存，则没有额外的 RAM 需要。

推荐使用这些驱动程序时，一起使用一个数据高速缓存，以获得更快的 LCD 访问速度。用

于高速缓存的内存的数值可以由以下公式计算：

$$\text{RAM 的大小 (字节)} = (\text{LCD_YSIZE} + 7) / 8 * \text{LCD_YSIZE}$$

附加的驱动函数

无。

硬件配置

这个驱动程序使用一个如第 20 章“低层配置”所描述的简单总路线接口访问硬件。下表列出了必须为硬件访问所定义的宏：

宏	说 明
<code>LCD_INIT_CONTROLLER</code>	初始化 LCD 控制器序列。
<code>LCD_READ_A0</code>	A 线 (A-line) 为低电平时从 LCD 控制器读一个字节。
<code>LCD_READ_A1</code>	A 线 (A-line) 为高电平时从 LCD 控制器读一个字节。
<code>LCD_WRITE_A0</code>	A 线 (A-line) 为低电平时向 LCD 控制器写入一个字节。
<code>LCD_WRITE_A1</code>	A 线 (A-line) 为高电平时向 LCD 控制器写入一个字节。

附加的配置开关

无。

某些 LCD 控制器的特殊要求

无。

第23章 LCD驱动API函数

μ C/GUI 要求一个驱动器用于硬件。本章说明一个 LCD 驱动器能为 μ C/GUI 做什么，以及它为 μ C/GUI 提供什么函数（应用程序接口，或 API）。

在大多数情况下，你大概不需要读这一章，因为大多数 μ C/GUI 的对于 LCD 层的函数调用将会由 GUI 层完成。实际上，我们推荐你仅仅在没有 GUI 等价物（例如，如果你希望直接修改 LCD 控制器的查询表）的情况下调用 LCD 函数。原因是 LCD 驱动函数是非线程保护，不象它们的 GUI 等价物。因此它们在多任务环境不应当被直接调用。

23.1 LCD驱动API

下表按字母顺序列出了 μ C/GUI 与 LCD 有关的函数。函数详细的描述在下面部分给出。

LCD_L0: 驱动器函数

函数	说明
初始化及显示控制组	
LCD_L0_Init ()	初始化显示屏。
LCD_L0_ReInit ()	不擦除内容而重新初始化 LCD。
LCD_L0_Off ()	关闭 LCD。
LCD_L0_On ()	开启 LCD。
绘制组	
LCD_L0_DrawBitmap ()	通用绘制位图函数。
LCD_L0_DrawHLine ()	绘制一条水平线。
LCD_L0_DrawPixel ()	以当前前景色绘制一个像素。
LCD_L0_DrawVLine ()	绘制一条垂直线。
LCD_L0_FillRect ()	直译一个矩形区域。
LCD_L0_SetPixelIndex ()	以指定颜色绘制一个像素。
LCD_L0_XorPixel ()	反转一个像素。
“Get” 组	
LCD_L0_GetPixelIndex ()	返回指定像素的颜色的索引。
“Set” 组	
LCD_L0_SetOrg ()	不再使用，保留给将来使用（必须在驱动器中存在）。
查询表组	
LCD_L0_SetLUTEntry ()	修改 LUT 的单个条目。
Misc. 组（可选）	
LCD_L0_ControlCache ()	锁/解锁/清除 LCD 高速缓存。

LCD: LCD 层函数

函数	说明
LCD_GetXSize ()	返回 LCD 的物理 X 轴尺寸（以像素为单位）。
LCD_GetYSize ()	返回 LCD 的物理 Y 轴尺寸（以像素为单位）。
LCD_GetVXSize ()	返回 LCD 的虚拟 X 轴尺寸（以像素为单位）。
LCD_GetVYSize ()	返回 LCD 的虚拟 Y 轴尺寸（以像素为单位）。
LCD_GetBitsPerPixel ()	返回每像素的位数。
LCD_GetNumColors ()	返回有效颜色的位数。
LCD_GetFixedPalette ()	返回固定调色板模式。

23.2 初始化及显示控制组

LCD_L0_Init()

描述

使用 LCDConf.h 中的配置设置初始化 LCD。如果使用上一 GUI 层，该函数被 GUI_Init () 自动调用，因此该函数不需要手工调用。

函数原型

```
void LCD_L0_Init (void);
```

LCD_L0_ReInit()

描述

使用配置设置再次初始化 LCD，不删除显示屏的内容。

函数原型

```
void LCD_L0_ReInit (LCD_INITINFO* pInitInfo) ;
```

LCD_L0_Off (), LCD_L0_On()

描述

分别打开或关闭显示。

函数原型

```
void LCD_L0_Off(void); void LCD_L0_On(void);
```

附加信息

函数的使用不影响图像存储器的内容或其它设置。因此你可以安全的关闭显示屏，然后重新打开而不需要更新内容。

23.3 绘制组

LCD_L0_DrawBitMap()

描述

绘制一个预先转换了的位图。

函数原型

```
LCD_L0_DrawBitMap ( int x0, int y0,  
                    int Xsize, int Ysize,  
                    int BitsPerPixel, int BytesPerLine,  
                    const U8* pData, int Diff,  
                    const LCD_PIXELINDEX* pTrans) ;
```

参 数	含 意
x0	要绘制的位图的左上角 X 轴坐标。
y0	要绘制的位图的左上角 Y 轴坐标。
Xsize	水平方向像素的数量。
Ysize	垂直方面像素的数量。
BitsPerPixel	每像素的位数。
BytesPerLine	图片每行的字节数。
pData	实际图片的指针，该数据定义了位图看起来象什么。
Diff	从左侧开始跳过的像素数。

LCD_L0_DrawHLine()

描述

在指定位置，使用当前前景颜色绘制一条一个像素宽的水平线。

函数原型

```
void LCD_L0_DrawHLine(int x0, int y, int x1);
```

参 数	含 意
x0	线段开始坐标
y	绘制线段的 Y 轴坐标。
x1	线段结束坐标。

附加信息

对于大多数 LCD 控制器，该函数执行速度非常快，因为可以立即放置多个像素，而不需要计算。很清楚，在绘制水平直线方面，该函数执行速度要比 DrawLine 函数快。

LCD_L0_DrawPixel()

描述

在指定位置，使用当前前景颜色绘制一个像素。

函数原型

```
void LCD_L0_DrawPixel(int x, int y);
```

参 数	含 意
x	绘制的像素的 X 轴坐标。
y	绘制的像素的 Y 轴坐标。

LCD_L0_DrawVLine()

描述

在指定位置，使用当前前景颜色绘制一条一个像素宽的垂直线。

函数原型

```
void LCD_L0_DrawVLine(int x, int y0, int y1);
```

参 数	含 意
x	绘制线段的 X 轴坐标。
y0	线段开始坐标。
y1	线段结束坐标。

附加信息

对于大多数 LCD 控制器，该函数执行速度非常快，因为可以立即放置多个像素，而不需要计算。很清楚，在绘制垂直直线方面，该函数执行速度要比 DrawLine 函数快。

LCD_L0_FillRect()

描述

在指定位置，使用当前前景颜色绘制一个填充的矩形。

函数原型

```
void LCD_L0_FillRect(int x0, int y0, int x1, int y1);
```

参 数	含 意
x0	左上角 X 轴坐标。
y0	左上角 Y 轴坐标。
x1	左下角 X 轴坐标。
y1	右下角 Y 轴坐标。

LCD_L0_SetPixelIndex()

描述

以指定的颜色绘制一个像素。

函数原型

```
void LCD_L0_SetPixelIndex(int x, int y, int ColorIndex);
```

参 数	含 意
x	绘制的像素的 X 轴坐标。
y	绘制的像素的 Y 轴坐标。
ColorIndex	使用的颜色。

LCD_L0_XorPixel()

描述

反转一个像素。

函数原型

```
void LCD_L0XorPixel(int x, int y) ;
```

参 数	含 意
x	反转的像素的 X 轴坐标。
y	反转的像素的 Y 轴坐标。

23.4 “Get” 组

LCD_L0_GetPixelFormat()

描述

返回指定像素的 RGB 颜色索引。

函数原型

```
int LCD_L0_GetPixelFormat(int x, int y);
```

参 数	含 意
x	像素的 X 轴坐标。
y	像素的 Y 轴坐标。

返回值

像素的索引。

附加信息

更多的信息请参阅第 9 章：颜色。

23.5 查询表（LUT）组

LCD_L0_SetLUTEntry()

描述

修改 LCD 控制器的 LUT 的单个条目。

函数原型

```
void LCD_L0_SetLUTEntry(U8 Pos, LCD_COLOR Color;
```

参 数	含 意
Pos	查询中的位置。应当小于颜色的数量，例如，对于 2bpp，为 0~3，4bpp 为 0~15，8bpp 为 0~255。
Color	24 位 RGB 值。最接近的可能的值将用于 LUT。如果一个颜色 LUT 被初始化，所有 3 个组成部分会被使用。在单色模式，使用绿色部分，但是依然推荐（为了更好的理解程序代码）将 3 种基色设置为同一个数值（例如 0x555555 或 0xa0a0a0）。

23.6 杂项组

LCD_L0_ControlCache()

描述

锁/解锁/清除 LCD 高速缓存。该函数可以用于将高速缓存设置为一个锁定状态，这样导致所有的在驱动器上的绘制操作转而在图像存储器缓存（在 CPU RAM 中）中实现，不过不会导致任何可见的输出。解锁或清除导致这样改变写入到显示屏当中。这可以帮助避免显示的闪烁，也可以加快绘制速度。它与有多少不同的绘制操作被执行无关；改变会立即写入显示屏。为了能够做到这样，LCD_SUPPORT_CACHECONTROL 必须在配置文件中启用。

函数原型

U8 LCD_ControlCache(U8 command);

参 数	含 意
Command	指定赋予高速缓存的命令。使用下表所示的符号值。

参数 Command 的允许值

LCD_CC_UNLOCK	设置默认模式：高速缓存是透明的。
LCD_CC_LOCK	锁定高速缓存，直到高速缓存被解锁或清除，才能执行写操作。
LCD_CC_FLUSH	清除高速缓存，将所有修改了的数据写入视频 RAM 中。

返回值

缓存状态的信息。忽略。

附加信息

当高速缓存被锁定，驱动器维持一个“hitlist”——一系列已经被修改并需要写入显示屏

的字节。该“hitlist”在图像存储器中每字节使用一位。

这是一个可选择的特性，并不支持所有的 LCD 驱动器。

范例

下面范例的代码在显示屏上执行交迭的绘制操作。为了加速显示屏的更新，避免闪烁，锁定高速缓存，直到这些操作执行完毕后才解锁。

```
LCD_ControlCache(LCD_CC_LOCK);
GUI_FillCircle(30, 30, 20);
GUI_SetDrawMode(GUI_DRAWMODE_XOR);
GUI_FillCircle(50, 30, 10);
GUI_SetTextMode(GUI_TEXTMODE_XOR);
GUI_DispStringAt("Hello world\n", 0, 0);
GUI_DrawHLine(16, 5, 25);
GUI_DrawHLine(18, 5, 25);
GUI_DispStringAt("XOR Text", 0, 20);
GUI_DispStringAt("XOR Text", 0, 60);
LCD_ControlCache(LCD_CC_UNLOCK);
```

LCD_GetXSize(), LCD_GetYSize()

描述

分别返回 LCD 的物理 X 轴或 Y 轴尺寸，以像素为单位。

函数原型 s

```
int LCD_GetXSize (void) int LCD_GetYSize (void)
```

返回值

显示屏的物理 X/Y 轴尺寸。

LCD_GetVXSize(), LCD_GetVYSize()

描述

分别返回 LCD 的虚拟 X 轴或 Y 轴尺寸，以像素为单位。在大多数情况下，虚拟尺寸等于

物理尺寸。

函数原型

```
int LCD_GetVXSize (void), int LCD_GetVYSize (void)
```

返回值

显示屏的虚拟 X/Y 尺寸。

LCD_GetBitsPerPixel()

描述

返回每像素位的数量。

函数原型

```
int LCD_GetBitsPerPixel(void);
```

返回值

每像素位的数量。

LCD_GetNumColors()

描述

返回 LCD 当前有效颜色的数量。

函数原型

```
int LCD_GetNumColors(void);
```

返回值

有效颜色的数量。

LCD_GetFixedPalette()

描述

返回固定调色板的模式。

函数原型

```
int LCD_GetFixedPalette(void);
```

返回值

固定调色板模式。关于固定调色板的更多信息，请参阅第 9 章：“颜色”。

第24章 性能和资源占用

高性能与低资源占用的结合总是一个主要的设计考虑。 $\mu\text{C}/\text{GUI}$ 可以在 8/16/32 位 CPU 上运行。依靠使用的模块，甚至小于 64KB ROM 和 2KB RAM 的单片系统都可以为 $\mu\text{C}/\text{GUI}$ 所支持。该实际的性能和资源占用取决于许多因素（CPU，编译程序，存储模型，最优化，结构，LCD 控制器接口，等等）。本章包含标准检查程序和有关在可用于获得最多目标系统的充分估计量的典型系统中的资源占用。

24.1 性能基准

我们使用一个基准测试测量在有效的目标上的软件的速度。本标准检查程序无论如何也不能完成，但是它给出一个在不同的目标上的通用的操作的需要的持续时间的近似值。

结构和性能表（所有的值位于 $\mu\text{s}/\text{pixel}$ 中）

CPU	LCD 控制器	bpp	试验台 1 填充	试验台 2 小字体	试验台 3 大字体	试验台 4 1bpp 位图
M16C/60（16 位），16MHz	T6963	1				
M16C/60（16 位），16MHz	T6963	1				
M16C/80（16 位），16MHz	1375	1	0.26	9.26	4.43	7.38
M16C/80（16 位），20MHz	1375	4	0.46	5.60	2.29	2.94
M16C/80（16 位），20MHz	1375	8	0.63	5.45	2.30	3.26

CPU	LCD 控制器	bpp	试验台 5 2bpp 位图	试验台 6 4bpp 位图	试验台 7 8bpp 位图	试验台 8 DDP 位图
M16C/60（16 位），16MHz	T6963	1				
M16C/60（16 位），16MHz	T6963	1				
M16C/80（16 位），16MHz	1375	1	7.93	8.01	7.99	7.14
M16C/80（16 位），20MHz	1375	4	8.21	3.14	7.86	1.54
M16C/80（16 位），20MHz	1375	8	7.65	3.23	2.81	1.61

用于基准测试的的测试顺序的描述

试验台 1：填充

填充速度的试验台。一个 64×64 的区域，填充不同的颜色。

试验台 2：小字体

小字体输出速度的试验台。一个填充小字体文本的 60×64 像素的区域。

试验台 3：大字体

大字体输出速度的试验台。一个填充大字体的文本的 65×48 像素的区域。

试验台 4: 1bpp 位图

1bpp 位图速度的试验台。一个填充一幅 1bpp 位图的 58×8 像素的区域。

试验台 5: 2bpp 位图

2bpp 位图速度的试验台。一个填充一幅 2bpp 位图的 32×11 像素的区域。

试验台 6: 4bpp 位图

4bpp 位图速度的试验台。一个填充一幅 4bpp 位图的 32×11 像素的区域。

试验台 7: 8bpp 位图

8bpp 位图速度的试验台。一个填充一幅 8bpp 位图的 32×11 像素的区域。

试验台 8: 与 8 或者 16bpp 位图相关的设备

每像素 8 或者 16 位的位图速度试验台。一个填充一幅位图的 64×8 像素的区域。测试位图的颜色深度取决于其结构。对于结构 ≤ 8 bpp，使用一幅 8bpp 的位图；16bpp 结构使用一幅 16bpp 位图。

24.2 内存需求

下表会给你一个用于 $\mu\text{C}/\text{GUI}$ 的内存需求的概念。在这个表格中的值是近似值，它取决于 CPU，使用的 C 编译程序，存储模型和编译时间切换（即 $\mu\text{C}/\text{GUI}$ 的哪些部分被归入你的应用程序中）。然而，下列数据对于 32 位 CPU 来说，表示了一个不错的平均值，以 x86 和富士通 FR30 CPU 的规格为根据。至于 16 位 CPU，ROM 代码长度应该更小（显而易见，30% 在自然的存储模型中，即有 16 位指针）；ROM 用于数据（字体）的大小是同样的。

简略描述	开关	RAM (字节)	ROM (字节)
基本系统			
核心软件，没有字体和图库。		120	3900
颜色（调色板）管理			
颜色管理，支持 16 种颜色，没有高速缓存		16	336
颜色管理，支持 256 种颜色，没有高速缓存		256	336
颜色管理，支持 256 种颜色，1000 字节高速缓存		1256	580
字体			
F4×6 字体（仅仅 ASCII 码）		---	600
F6×8 字体（ISO8859-1）		---	1536

F8×8 字体 (ISO8859-1)		---	1536
F8×16 字体 (ISO8859-1)		---	3072
FD24×32 字体 (大的数字)		---	1374
F4×6 字体 (仅仅 ASCII 码)		---	600
图形库			
位图		---	u. i
简单的线段, 任何角度		---	u. i
有线型的线段		---	u. i
折线, 绘制		---	u. i
折线, 填充 (多边形)		---	u. i
圆形, 绘制		---	u. i
圆形, 填充		---	u. i
***** 总数		---	9500
储存设备 (可选择)			
核心软件		由区域的大小决定	大约 500
支持 1 位/像素驱动器		---	大约 1800
支持 2 位/像素驱动器		---	大约 2400
支持 4 位/像素驱动器		---	大约 2400
支持 8 位/像素驱动器		---	大约 1700
图库, 抗锯齿 (u. d.)			
简单的线段, 任何角度。			u. i
多边形			u. i
圆形			u. i
驱动器			
SED1565, 单个 LCD 控制器核		12	1600
SED1565, 另外支持调色板管理			大约 100
SED1565, 另外支持用于驱动器的 4 种颜色位图			大约 800
SED1565, 另外支持用于驱动器的 16 种颜色位图			大约 600
SED1565, 另外支持用于驱动器的 256 种颜色位图			大约 520
SED1565, 另外支持用于驱动器的比例位图			大约 260
SED1565, 另外支持用于驱动器的抗锯齿处理			大约 80
SED1352, 单个 LCD 控制器核			
SED1352, 双的个 LCD 控制器核			
视窗管理器			
核心软件			
附加窗口		大约 40	---