TI StellarisWare 图形库使用指南

[V1.0]

[作者: Richard Ma]

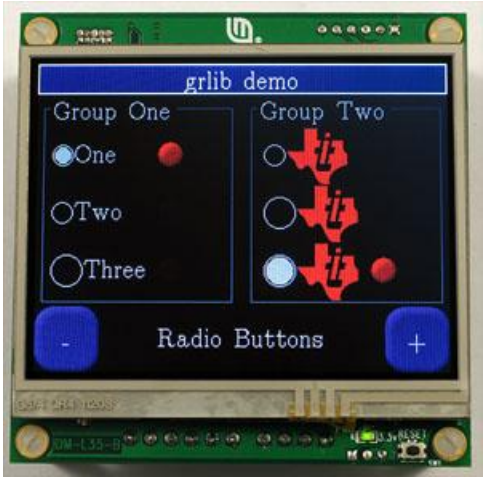
[Email: mxschina@gmail.com]

1.前言

TI Stellaris 系列 Cortex-M3/Cortex-M4F 系列 MCU 的方便强大，StellarisWare 软件库提供的快速软件开发解决方案功不可没。作为 TI StellarisWare 软件包的一部分，StellarisWare 图形库(Grlib)提供了一套比较完整的 MCU 图形显示方案，既可以进行基础的图形、文字绘制，也可以轻松实现 PC 机上常见的，基于消息的控件(Widget)。伴随着 Stellaris LM4F 系列的推出，StellarisWare 图形库也会进一步升级，支持汉字字库。

由于 Stellaris 图形库没有相关的中文文档，有时候会让大家觉得入门非常困难，所以写这篇文档，抛砖引玉，让对 Stellaris 感兴趣的朋友都能够通过阅读这篇文档，快速上手 StellarisWare 图形库，为 MCU 世界增加更多美丽的应用。本文所介绍的 StellarisWare 可以在 TI 的 Stellaris M3/M4F 系列上运行。

下面图就是用 Stellaris 图形库进行控件绘制的示例，Stellaris 最高可支持 24bit 颜色，精心设计过的界面可以非常有吸引力。本文以 TI 的 LM3S9B96 开发板为基础，示例程序均在开发板上演示、运行。



2.基础知识

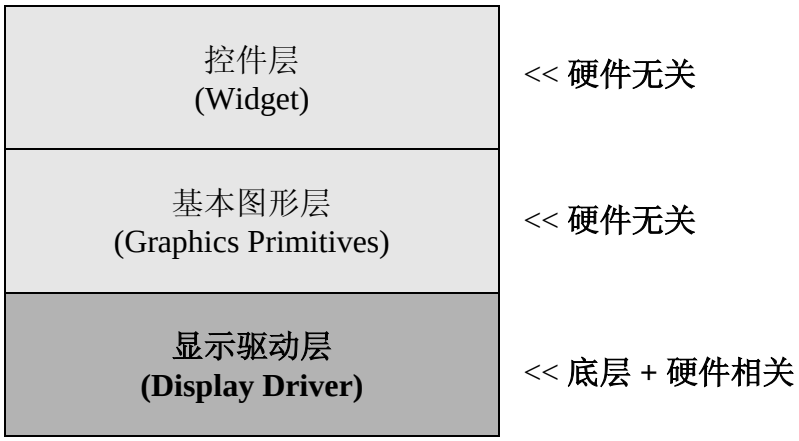
虽然 Stellaris 图形库应用起来比较简单，但理论知识还是需要的。下面谈谈 Stellaris 图形库的基本结构。

Stellaris 图形库分成了三层，分别是：

(1) 显示驱动层 (Display Driver Layer)

(2) 基本图形层 (Graphics Primitives Layer)

(3) 控件层 (Widget Layer)



下面分别介绍下各层的大概作用。

2.1.显示驱动层 (Display Driver Layer)

2.1.1. 基本功能

显示驱动层提供了和硬件通信的基本功能，这层直接和硬件直接通信。提供了两类驱动，分别是图形输出驱动、用户输入驱动。图形库应用中，用户输入不是必须的。

辛苦的分层带来了巨大好处，当需要把程序从一个硬件平台移植到另一个新的时候，如果显示内容不变，开发者们只需要重新实现驱动层，而上层的代码可以保持不变。

2.1.2. 图形输出驱动

图形输出驱动和显示屏控制器打交道，实现诸如在屏幕上画点之类的基本作用（毕竟再复杂的图像也是一个点一个点画出来的），参考 TI LM3S9B96 开发板的驱动，有如下的显示驱动程序：

kitronix320x240x16_ssd2119_8bit.c

它们就是开发板上 320x240 彩色 LCD 显示屏的驱动。打开驱动程序，能找到如下对象：

tDisplay g_sKitronix320x240x16_SSD2119;

对象中定义了显示相关的参数(如尺寸，屏幕的横竖等)，并实现了下面的函数：

Kitronix320x240x16_SSD2119PixelDraw (绘制点)
Kitronix320x240x16_SSD2119PixelDrawMultiple (绘制多个点)
Kitronix320x240x16_SSD2119LineDrawH (绘制水平线)
Kitronix320x240x16_SSD2119LineDrawV (绘制垂直线)
Kitronix320x240x16_SSD2119RectFill (填充方块)
Kitronix320x240x16_SSD2119ColorTranslate (颜色变换)
Kitronix320x240x16_SSD2119Flush (使绘图结果生效)

没错，它们实现了基本的绘图功能，在图形库更上层基本图形层中，这些函数将被调用，直接控制 LCD，在屏幕上显示点(Pixel)、线(Line)以及面(Rect)等。所以在移植的时候，这些函数需要充分调试，以保证它们能正确画出所需图形。

2.1.3. 用户输入驱动

响应用户输入事件用的硬件驱动（如触摸屏幕驱动），也算作显示的驱动的一部分，归在显示驱动层。在 LM3S9B96 开发板上，提供了触摸屏的驱动：

touch.c

里面的函数与 Stellaris 图形库直接相关，用户需要用到的主要是：

TouchScreenCallbackSet

在触摸功能初始化的时候，这个函数通过回调，将用户动作事件和 Stellaris 图形库的事件响应函数连接在一起。

当用户动作时，输入驱动可以调用 Stellaris 图形库的 WidgetPointerMessage 函数，传入动作的信息（如动作的 x、y 坐标，动作方式等）。图形库会处理这些信息，进行画面更新，响应用户的动作。

2.2.基本图形层 (Graphics Primitives Layer)

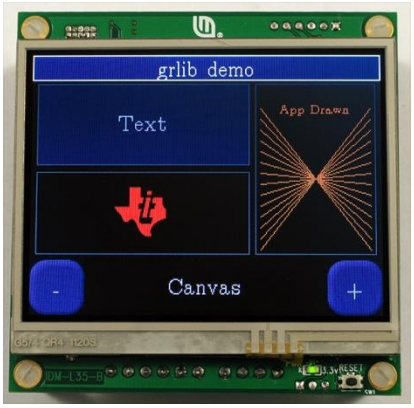
只能画点线面显然是不够的。Stellaris 图形库的基本图形层实现了形状、文字以及图片的绘制功能。如果只需要基本的图形显示功能，可以仅使用该层而不用控件层。

2.3.控件层 (Widget Layer)

对 PC 上的按钮、点选框等控件，想必各位应该相当熟悉了。控件层的作用就是实现这些类似的功能。Stellaris 图形库可以实现的控件有：

画布 (Canvas)
控件容器 (Container)
按钮(Push Button)
图形按钮(Image Button)
选择/多选框 (Checkbox)
单选框 (Radio Button)
列表框 (ListBox)
拖滑/进度条 (Slider)

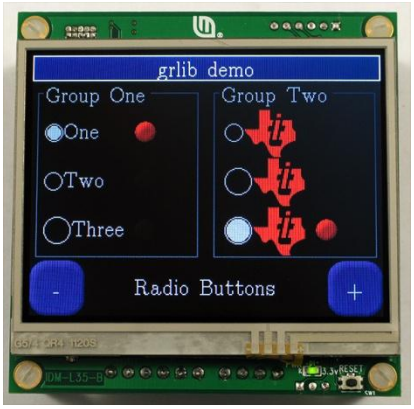
StellarisWare 图形库中，实现了这些控件的自动绘制、事件响应，使用户不需花费时间在重复繁琐的用户输入处理工作上，为应用带来方便。后文会具体介绍如何使用这些控件。



画布 (Canvas)

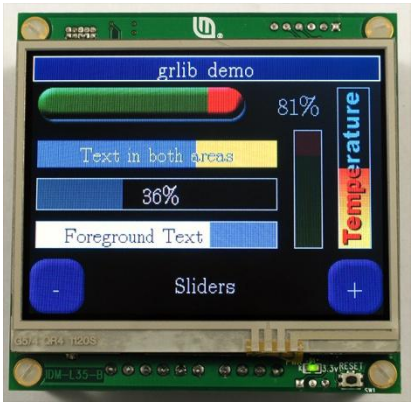
选择/多选框 (Checkbox)

控件容器 (Container)



单选框 (Radio Button)

按钮(Push Button)



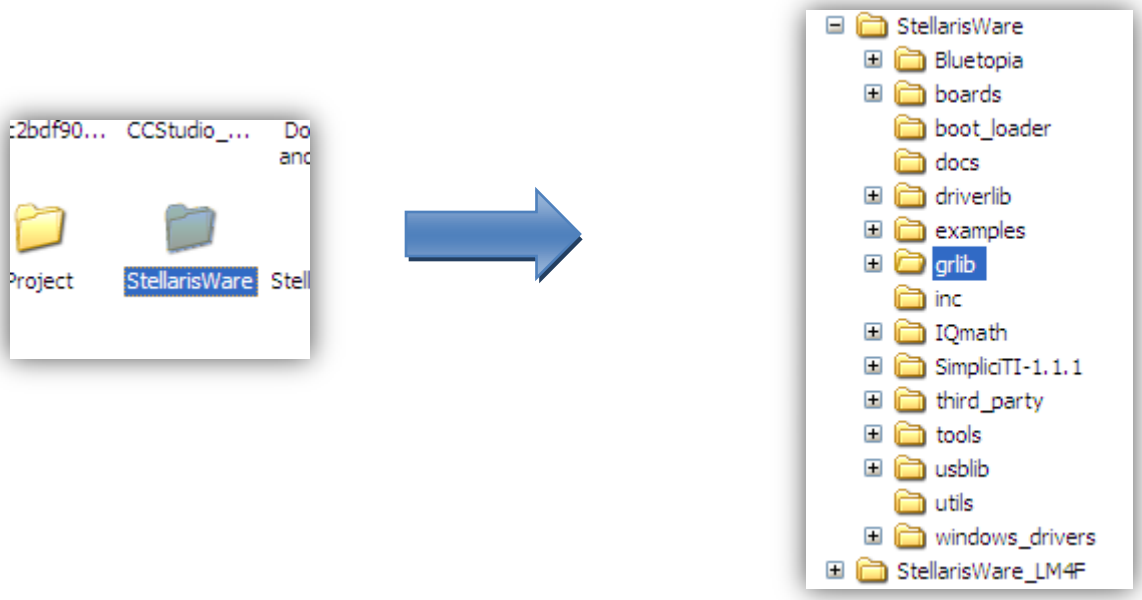
拖滑/进度条 (Slider)

3.StellarisWare 图形库基本使用

3.1.开发环境及 StellarisWare 安装

本文以 IAR Embedded Workbench v6.2 (ARM)为开发环境，以 TI Stellaris LMS9B96 开发板为基础，介绍 StellarisWare 图形库的用法。

要使用 StellarisWare 图形库，StellarisWare 软件包当然必不可少。在 TI 官网免费可以下载到这个软件包。



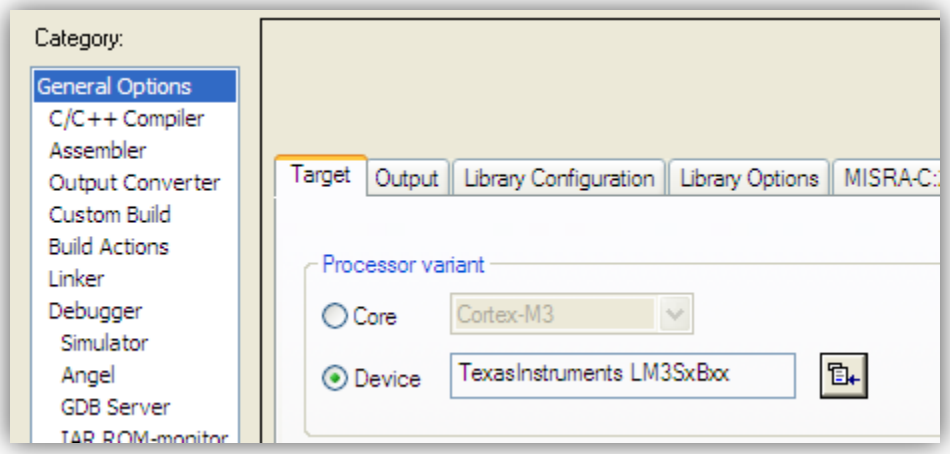
安装 StellarisWare 后，进入安装路径，在 grlib 目录下可以找到 StellarisWare 图形库相关内容。Grlib 根目录下包含了图形库的源代码，如 ccs、ewarm 这些目录则包含了针对不同 IDE 的库文件。

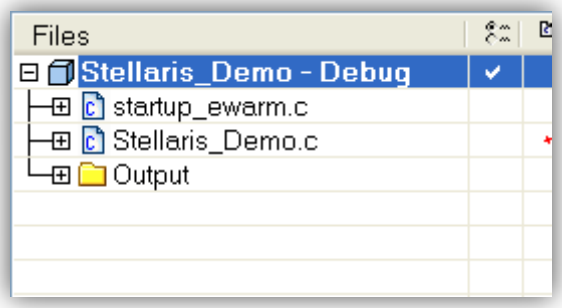
3.2.图形库添加与编译 (显示部分)

为了比较详细地说明用法，我们完全手动建立项目，实际项目时可以从 StellarisWare/boards/dk-lm3s9b96/hello 示例开始以节约时间。首先介绍的是显示部分的设置，触摸功能将放在下一章。

3.2.1. 建立新项目

在 IAR 中新建项目 Stellaris_Demo，设置项目 Device 为 TexasInstruments LM3SxBxx。在项目中加入 Stellaris_Demo.c，写一个空的 main 函数。同时也别忘记加入启动代码 startup_ewarm.c (该文件可以在 StellarisWare/boards/dk-lm3s9b96/hello 中的应用中找到，_ewarm 表示该启动代码为针对 IAR 的)。

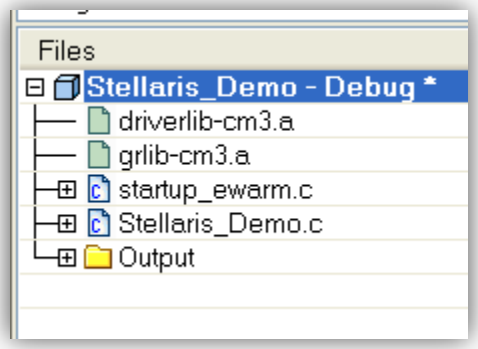




新项目中只有默认的中断函数处理程序(定义在 startup_ewarm.c 中)，尝试编译通过，进入下一步。

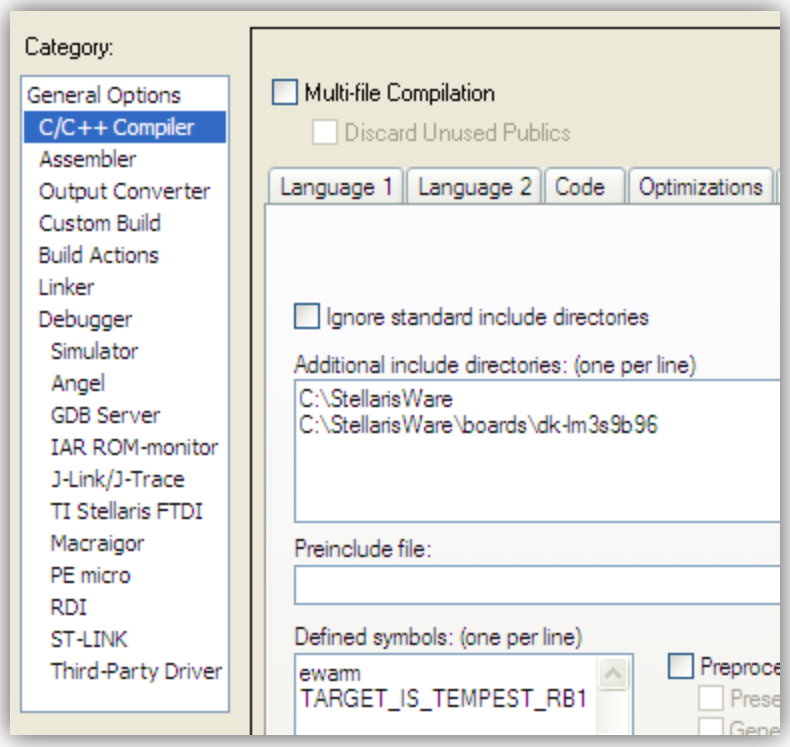
3.2.2. 添加图形库

首先在项目中添加 StellarisWare 驱动库和图形库。分别在 StellarisWare/driverlib 和 StellarisWare/grlib 下面，找为 IAR (ewarm)已编译好的库。因为是在 M3 平台上的，所以用-cm3 版本。-cm4f 是为 LM4F 系列准备的，如果是在 LM4F 平台使用，只需要连接 LM4F 的库，而代码不需要改动。



打开程序设置，在 C/C++ Compiler 中 Preprocessor 选项卡下 Additional include directories 添加 C:\StellarisWare (StellarisWare 的安装路径)。如果使用了 LM3S9B96 开发板，需要再添加 C:\StellarisWare\boards\dk-lm3s9b96，为后面使用其驱动程序做准备。

由于使用 IAR 作为开发环境，需要在 Defined symbols 中加入 ewarm (小写)定义。图中 TARGET_IS_TEMPEST_RB1 是驱动库可能用到的，用以区别器件版本，根据不同的器件版本应填入不同的参数。



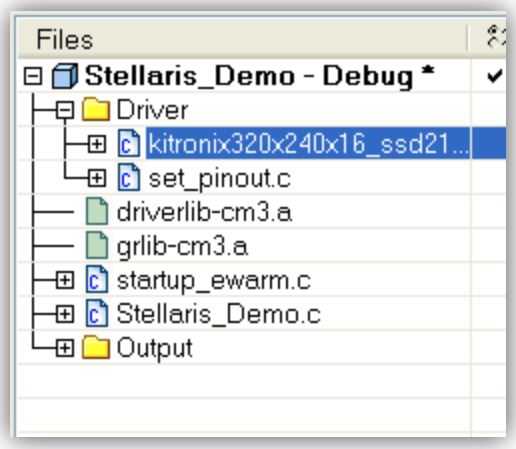
在 Stellaris_Demo.c 中加入驱动库和图形库的引用，如下所示：

```
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"

int main(void)
{
   // 设置系统时钟为 50MHz
   SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
                  SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);

   while(1);
}
```

除了图形库和驱动库，还需要有相应硬件的驱动程序。LM3S9B96 开发板的驱动程序可以在 StellarisWare/boards/drivers 找到源代码。将 set_pinout.c 以及 kitronix320x240x16_ssd2119_8bit.c 两个驱动加入项目。要说明的是，这里的 set_pinout.c 只是用于初始化 LM3S9B96 开发板的外围功能，可理解为显示驱动程序的一部分。



在 Stellaris_Demo.c 中再加入驱动的引用。

```
#include "drivers/kitronix320x240x16_ssd2119_8bit.h"
#include "drivers/set_pinout.h"
```

至此，StellarisWare 图形库以及相应驱动已经引入项目。接下来都是代码工作，需要对硬件、图形库进行初始化。

3.2.3. 驱动程序初始化

使用以下代码进行显示驱动的初始化，放在系统时钟配置部分之后：

```
// 初始化显示驱动
PinoutSet();
Kitronix320x240x16_SSD2119Init();
```

3.3.图形库添加与编译 (触摸部分)

如果需要触摸功能，则需添加触摸驱动程序及触摸事件处理。否则该节可以跳过。

在 3.2 节的基础上，在项目中添加触摸屏的驱动程序，若使用了 LM3S9B96 开发板，就是 touch.c 文件(在 StellarisWarae/boards/dk-lm3s9b96/drivers 目录下)。LM3S9B96 的触摸

屏驱动使用了 ADC Sequence 3 中断，所以还需要在 startup_ewarm.c 的中断向量中添加 TouchScreenIntHandler 作为中断处理函数。

由于 Touch 功能往往和控件功能相关，若需要图形库来处理控件事件，还需要在触摸事件发生时调用 StellarisWare 图形库的事件处理函数。这部分将在后文控件使用章节中介绍。

4.基本图形绘制

当完成 3.2 中驱动的设置后，就可以进行基本图形的绘制了。基本图形层的主要功能为绘制线、框、文字及位图文件。本章将分别介绍如何绘制这些对象。

4.1.绘图上下文 (tContext)

前文已经提到过，StellarisWare 图形库可以在多个设备上进行绘制，所以在进行基本绘图时要指定绘图的设备。在 StellarisWare 图形库中，绘图设备由绘图上下文 (tContext) 这个对象指定。各类绘图函数都需要提供这个对象 (如 GrRectFill)。

声明 tContext 后，需要用 GrContextInit 函数将其初始化，指明其所指的设备。显示设备对象为 tDisplay，在显示驱动中提供。在 LM3S9B96 开发板上，tDisplay 对象在 Kitronix320x240x16_SSD2119.c 中提供。代码如下所示：

```
// main 函数外声明驱动库中定义的 tDisplay 对象
extern const tDisplay g_sKitronix320x240x16_SSD2119;

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

// 声明绘图上下文
tContext sContext;

// 初始化 StellarisWare 图形库上下文
GrContextInit(&sContext, &g_sKitronix320x240x16_SSD2119);
```

4.2.颜色设置

通过以下两个函数可以设置绘图的颜色：

GrContextForegroundSet(pContext, ulValue) (设置前景色)

GrContextBackgroundSet(pContext, ulValue) (设置背景色)

其中一个是前景色，指的是绘制的图形或文字的颜色。另一个是背景色，但不是屏幕背景的颜色，而是绘制时可能会用到的颜色设置，如文字的底色等。

pContext 指的当然是前面声明的 tContext 对象的指针，而 ulValue 是颜色值。为 24bit 颜色。用 unsigned long 表示，高 8bit 无效。如 0x00FF0000 代表纯红色，0x0000FF00 代表纯绿色，0x000000FF 代表纯蓝色。StellarisWare 图形库中也定义了一些常用的颜色，如 ClrBlue，ClrYellow 等等。这些颜色 (及图片) 可以在图形库说明手册的附录中找到。或者也可以在 grlib.h 文件中找到这些定义。

4.3.绘制基本图形

下面的函数实现了圆形的绘制。其中 GrCircleDraw 画的是空心圆，GrCircleFill 画的是实心圆。圆的颜色就是 GrContextForegroundSet 设置的颜色。Radius 表示要绘制的圆形的半径。

```
void GrCircleDraw (const tContext *pContext, long lX, long lY, long lRadius)
void GrCircleFill (const tContext  *pContext, long lX, long lY, long lRadius)
```

GrRectDraw 实现了空心方框的绘制而 GrRectFill 绘制实心方框。

```
void GrRectDraw (const tContext  *pContext, const tRectangle *pRect)
void GrRectFill (const tContext *pContext, const tRectangle *pRect)
```

pRect 是用于描述方框大小和位置的，其定义如下：

```
typedef struct
{
    short sXMin;// 方框的最小 X 位置
    short sYMin;  // 方框的最小 Y 位置
    short sXMax;        // 方框的最大 X 位置
    short sYMax;        // 方框的最大 Y 位置
}tRectangle;
```

当绘制完成，调用 GrFlush(const tContext *pContext)来确保图形都正常显示在屏幕上。下面的示例代码完成了两个不同颜色圆形的绘制：

```
// 声明绘图上下文
tContext sContext;

// 初始化 StellarisWare 图形库上下文
GrContextInit(&sContext, &g_sKitronix320x240x16_SSD2119);

// ================================================

// 设置画笔为黄色
GrContextForegroundSet(&sContext, ClrYellow);
// 画一个实心圆
GrCircleFill(&sContext, 80, 120, 40);

// 设置画笔为白色
GrContextForegroundSet(&sContext, ClrBlue);
// 画一个空心圆
GrCircleDraw(&sContext, 240, 120, 40);

// 确保图形被绘制在屏幕上
GrFlush(&sContext);
```

线条绘制同理。使用下面函数：

GrLineDraw(const tContext *pContext, long lX1, long lY1, long lX2, long lY2)

4.4.绘制文字

文字的绘制需要先设置字体，其它方面则和图形绘制相类似，如设置颜色。这里背景色也会用到。当文字背景色不为透明时，就会填充文字底色。

设置字体的函数为

GrContextFontSet(tContext *pContext, const tFont *pFont)

其中 tFont 定义了字体，可以用于新建字体 (包括中文，可以查询网上其它资料)，这里不详述。StellarisWare 图形库已经内置了上百种字体。这些字体可以在手册附录中查到名称及示例。

常见的字体名如 g_sFontCm12，g_sFontCmss18i，g_sFontCmsc20b 等。结尾的 b、i 分别表示加粗和斜体，数字则是字体大小，Font 后是字体名称。例如 g_sFontCmss18i 则表示 18 像素的斜体 Cmss 字体。

绘制文字的函数主要有以下两种：

```
GrStringDraw(const tContext *pContext, const char *pcString,
        long lLength, long lX, long lY, unsigned long bOpaque);
GrStringDrawCentered(const tContext *pContext, const char *pcString,
        long lLength, long lX, long lY, unsigned long bOpaque);
```

参数中，pcString 是指向文本的指针，lLength 表示要显示的文字的长度，lX、lY 是位置。Opaque 表示是否绘制文字底色 (背景色)，当为 0 (false)时则不绘制，非零(true)时绘制底色。

Draw 和 DrawCentered 的区别是对齐方式。Draw 是文字最左侧坐标为 X，而 DrawCentered 是字符串的中间位置坐标为 X。下面的示例代码完成了文字绘制：

```
// 声明绘图上下文
tContext sContext;

// 初始化 StellarisWare 图形库上下文
GrContextInit(&sContext, &g_sKitronix320x240x16_SSD2119);

// ================================================

// 设置画笔为黄色
GrContextForegroundSet(&sContext, ClrYellow);
// 设置字体为 Cm，18 号，粗体
GrContextFontSet(&sContext, &g_sFontCm18b);
// 输出文字
GrStringDraw(&sContext, "Hello World!", 12, 80, 150, true);

// 确保图形被绘制在屏幕上
GrFlush(&sContext);
```

4.5.绘制图片

图片绘制要用到下面的函数：

GrImageDraw(const tContext *pContext, const unsigned char *pucImage,
long lX, long lY)

lX、lY 表示图片的位置。pucImage 是图片的数据。图片信息在 StellarisWare 图形库中有专门结构表示。

4.5.1. 图片表示方式

用结构体可以比较轻易地描述 StellarisWare 对图片结构的定义，如下所示：

```
typedef struct
{
    // 图片数据表示格式选择，可以有如下几种：
    // IMAGE_FMT_1BPP_UNCOMP, IMAGE_FMT_4BPP_UNCOMP,
    // IMAGE_FMT_8BPP_UNCOMP, IMAGE_FMT_1BPP_COMP
    // IMAGE_FMT_4BPP_COMP or IMAGE_FMT_8BPP_COMP.
    // UNCOMP 表示非压缩格式，COMP 表示压缩格式
    unsigned char ucFormat;

    // 图片的像素宽度
    unsigned short usWidth;

    // 图片的像素高度
    unsigned short usHeight;

    // 图片数据
    unsigned char pucData[];
}tImage;
```

图片数据格式就不做过多说明，具体细节可以查看 StellarisWare 图形库的用户手册。

有一点需要注意到，GrImageDraw 中图片指针的类型不是 tImage 而是 unsigned char。这样做的原因是出于对效率的考虑，因为用 tImage->pucData 这样的形式在 C 中的效率比较低。在实际使用中，图片文件均被定义为 unsigned char 的数组而非结构体。TI 提供了专门的工具从图片生成所需的数组。

4.5.2. 图片生成工具

在用 StellarisWare 图形库开发过程中，图片是用 C 语言中 unsigned char 数组表示的。

TI StellarisWare 图形库提供了一个将图片文件转换成 C 代码的软件。在 StellarisWare/tools 文件夹下，有一个叫 pnmtoc.exe 的程序 (pnmtoc 文件夹下是源代码)。使用 pnmtoc 可以将 pnm 格式的图片转换为 C 代码。pnm 不是常见的图片格式，但很多常见的绘图软件都可以直接生成 pnm 格式，如 Photoshop。同时，也有很多免费转换工具可以将 bmp、jpeg 等常见格式转换为 pnm 文件，如 NetPBM、GIMP 等。

当得到 pnm 格式图片后可以使用 pnmtoc 来转换，命令如下：

```
pnmtoc -c image.pnm > image.c
```

这里会将 image.pnm 文件转换输出为 image.c。-c 命令表示启用图片压缩。如果压缩后体积大于不压缩时，压缩会自动关闭。

通常生成如下形式的图片数组，g_pucImageLogo 可直接作为 GrImageDraw 中 pucImage 的参数。

```
const unsigned char g_pucImageLogo[] =
{
    IMAGE_FMT_8BPP_COMP,
    33, 0,
    16, 0,

    101,
    0x00, 0x00, 0x00,
    0x10, 0x10, 0x10,
    0x2a, 0x2a, 0x2a,
    …

    0xc7, 0x07, 0x07, 0x00, 0x0f, 0x11, 0x07, 0x07, 0x07, 0x87, 0x01, 0x1b,
    0x4d, 0x52, 0x1f, 0x07, 0x07, 0x04, 0x00, 0x0c, 0x37, 0x54, 0x55, 0x53,
    0x00, 0x2d, 0x2f, 0x33, 0x33, 0x33, 0x33, 0x34, 0x34, 0x10, 0x34, 0x33,
    …
};
```

5.控件使用

最常见的控件就是 Windows 系统中的按钮，当用户点击时，按钮的外观形状将发生变化，同时会有相应的动作执行。StellarisWare 图形库中控件的作用是类似的，就是将图形的绘制及用户的操作封装在一起，这样在程序设计时，开发人员可以不再费心处理每一个控件细节(如点击某个区域引起控件事件等)，而只用设置如按钮的名字、大小以及事件响应动作等属性。

要使用控件，需要以下几个步骤：

1) 配置显示及用户输入(如触摸)
2) 创建控件对象，设置控件对象的属性(结构、位置及外观等)
3) 加入控件列表将控件绘制出来

5.1.控件使用示例代码

首先用一个 Hello World 的例子进行一个示例，该例子在 LM3S9B96 上调试通过，会在 LCD 上显示一个“Show Welcome”的按钮，点击后会屏幕上会出现“Hello World”，同时按钮变为“Hide Welcome”；再次点击则 Hello World 会消失，按钮文字复原。这个例子主要为提供直观的印象，后文中会对图形库的使用做更详细的介绍。

```
#include "inc/hw_types.h"
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "driverlib/rom.h"
#include "grlib/grlib.h"
#include "grlib/widget.h"
#include "grlib/canvas.h"
#include "grlib/pushbutton.h"
#include "drivers/kitronix320x240x16_ssd2119_8bit.h"
#include "drivers/touch.h"
#include "drivers/set_pinout.h"
```

```
//*****************************************************
// 声明驱动库中定义的 tDisplay 显示设备对象。
//*****************************************************
extern const tDisplay g_sKitronix320x240x16_SSD2119;

//*****************************************************
// 提前声明要使用到的控件。（后文为建立控件树可能会相互引用）
//*****************************************************
extern tCanvasWidget g_sBackground;
extern tCanvasWidget g_sHeading
extern tCanvasWidget g_sHello;
extern tPushButtonWidget g_sPushBtn;

//*****************************************************
// 提前声明按钮按下时的处理函数。
//*****************************************************
void OnButtonPress(tWidget *pWidget);

//*****************************************************
// 创建并定义各控件。
// 包括 3 个画布控件（背景、标题、Hello World 显示），一个按钮控件。
//*****************************************************
// 作为屏幕标题的画布控件
Canvas(g_sHeading, &g_sBackground, 0, &g_sPushBtn,
       &g_sKitronix320x240x16_SSD2119, 0, 0, 320, 23,
       (CANVAS_STYLE_FILL | CANVAS_STYLE_OUTLINE |
       CANVAS_STYLE_TEXT),
       ClrDarkBlue, ClrWhite, ClrWhite, &g_sFontCm20,
       "hello-widget", 0, 0);

// 作为屏幕背景的画布控件
Canvas(g_sBackground, WIDGET_ROOT, 0, &g_sHeading,
       &g_sKitronix320x240x16_SSD2119, 0, 23, 320, (240 - 23),
       CANVAS_STYLE_FILL, ClrBlack, 0, 0, 0, 0, 0, 0);
```

```
// 用于点击，以显示“Hello World”的按钮控件
RectangularButton(g_sPushBtn, &g_sHeading, 0, 0,
        &g_sKitronix320x240x16_SSD2119, 60, 60, 200, 40,
        (PB_STYLE_OUTLINE | PB_STYLE_TEXT_OPAQUE |
        PB_STYLE_TEXT | PB_STYLE_FILL | PB_STYLE_RELEASE_NOTIFY),
        ClrDarkBlue, ClrBlue, ClrWhite, ClrWhite,
        &g_sFontCmss22b, "Show Welcome", 0, 0, 0, 0,
        OnButtonPress);

// 用于显示“Hello World”的画布控件
// 请注意这个控件并没有被加入控件树，因此是不可见的。只有在用户点击按钮后，
// 该控件被加入控件树，才会被显示。
Canvas(g_sHello, &g_sPushBtn, 0, 0,
        &g_sKitronix320x240x16_SSD2119, 0, 150, 320, 40,
        (CANVAS_STYLE_FILL | CANVAS_STYLE_TEXT),
        ClrBlack, 0, ClrWhite, &g_sFontCm40, "Hello World!", 0, 0);

//*****************************************************
// 全局变量，用于记录 Hello World 是否可见
//*****************************************************
tBoolean g_bHelloVisible = false;

//*****************************************************
// 当按钮按下时的事件处理函数。
// 如果 Hello World 为不可见，则将 g_sHello 控件加入控件树以显示；
// 如果 Hello World 为可见，则将 g_sHello 控件移除控件树以隐藏。
//*****************************************************
void OnButtonPress(tWidget *pWidget)
{
    // 翻转 Hello World 的可见状态
    g_bHelloVisible = !g_bHelloVisible;
    // 更新 Hello World 的显示
    if(g_bHelloVisible)
```

```
    {
        // 如果 Hello World 的新状态为可见，
        // 则将其加入控件列表（作为按钮控件的子控件）。
        WidgetAdd((tWidget *)&g_sPushBtn, (tWidget *)&g_sHello);

        // 更新按钮文字。
        PushButtonTextSet(&g_sPushBtn, "Hide Welcome");

        // 重画按钮控件及其下子控件。
        WidgetPaint((tWidget *)&g_sPushBtn);
    }
    else
    {
        // 如果 Hello World 的新状态为不可见，则将其移除出控件列表。
        WidgetRemove((tWidget *)&g_sHello);

        // 更新按钮文字。
        PushButtonTextSet(&g_sPushBtn, "Show Welcome");

        // 重画整个控件树以清除屏幕上的多余文字。
        WidgetPaint(WIDGET_ROOT);
    }
}


//*****************************************************
// 主函数
//*****************************************************
int main(void)
{
    // 设置系统工作在 50MHz。
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
                       SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ);
```

```
    // 为 9B96 开发板分配管脚。（主要为使能各 GPIO ）
    PinoutSet();

    // 全局允许中断。
    ROM_IntMasterEnable();



    // 初始化显示驱动程序。
    Kitronix320x240x16_SSD2119Init();

        // 初始化触摸驱动程序。
    TouchScreenInit();

        // 设置触摸驱动函数的回调函数为图形库的处理函数。
    TouchScreenCallbackSet(WidgetPointerMessage);

    // 将 g_sBackground 以下的控件加入控件树。
    WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sBackground);

    // 绘制控件树中的所有控件。
    WidgetPaint(WIDGET_ROOT);

        // 主循环。
        while(1)
    {
        // 处理所有控件事件。
        WidgetMessageQueueProcess();
    }
}
```

5.2.控件使用步骤说明

现在对控件创建使用的几个主要步骤做以下说明：

5.2.1. 配置显示及用户输入

首先需要初始化显示，提供如 4.1 节中所提的到的 tDisplay 对象。

```
// 声明驱动库中定义的 tDisplay 对象
extern const tDisplay g_sKitronix320x240x16_SSD2119;

//=============================================

// 初始化显示驱动程序
Kitronix320x240x16_SSD2119Init();
```

同时要为 StellarisWare 图形库提供用户输入接口，让 StellarisWare 图形库来处理用户的输入操作。

```
// 初始化触摸屏
TouchScreenInit();

// 关联触摸屏与 StellarisWare 图形库的用户输入处理函数
TouchScreenCallbackSet(WidgetPointerMessage);
```

触摸事件与图形库的响应采用回调的方式，当用户点击时，WidgetPointerMessage 函数会被调用并传入用户点击屏幕的坐标等参数。WidgetPointerMessage 函数为 StellarisWare 的库函数，接受三个参数：ulMessage、lX、lY。Message 表示事件的内容，如点击、拖动等；lX 和 lY 表示操作的坐标。如果已有现成的触摸驱动，这些参数用户不必太关心，驱动程序会处理好这些参数。

5.2.2. 创建控件及属性配置

可以发现，5.1 节中的例子使用了类似如下的代码创建控件，并配置了属性。乍一看，这些属性似乎非常的繁琐复杂，而且不同的控件所需要设置的属性也不相同，这些具体的属性可以在第 6 章中找到具体的定义，读者先有个大概的了解，知道如何使用即可，不必纠结于每个参数的具体选项。

```
Canvas(g_sHeading, &g_sBackground, 0, &g_sPushBtn,
    &g_sKitronix320x240x16_SSD2119, 0, 0, 320, 23,
    CANVAS_STYLE_FILL | CANVAS_STYLE_OUTLINE | CANVAS_STYLE_TEXT,
    ClrDarkBlue, ClrWhite, ClrWhite, &g_sFontCm20, "hello-widget", 0, 0);

RectangularButton(g_sPushBtn, &g_sHeading, 0, 0,
    &g_sKitronix320x240x16_SSD2119, 60, 60, 200, 40,
    (PB_STYLE_OUTLINE | PB_STYLE_TEXT_OPAQUE | PB_STYLE_TEXT |
        PB_STYLE_FILL | PB_STYLE_RELEASE_NOTIFY),
    ClrDarkBlue, ClrBlue, ClrWhite, ClrWhite,
    &g_sFontCmss22b, "Show Welcome", 0, 0, 0, 0, OnButtonPress);
```

整体来看，这些控件其实有一些共同点，下文先介绍下这些共同点，相信各位读者在理解了这些共同点后，能够通过阅读第 6 章，很快上手每个不同的控件。

1) 控件名

每个控件都有自己的唯一名字，用于标识自己，以及被引用。上面例子中的 g_sHeading 和 g_sPushBtn 都是控件的名字。不同类型的控件，控件名对应的实际类型都不一样，如 RectangularButton 的类型是 ButtonWidget、Canvas 的类型是 tCanvasWidget 等。5.1 节中代码开始部分的控件声明处可以作为示例。

2) 控件管理 – 控件树

通常来说，一个界面中会有很多控件对象。为了方便管理，StellarisWare 图形库将众多控件以树状的形式管理，这样做有很多好处：每个控件(树)都可以动态加入或者移出被渲染的控件列表树，以决定屏幕要显示的内容；同时当需要部分更新屏幕内容时，可以只从某个节点开始渲染，这样就只绘制其及子节点以节约资源。

图形库中有一个虚拟的 WIDGET_ROOT 控件，总是作为最顶层的控件，其它控件都作为它的子节点或者更下层的子节点。不同控件在树状列表中的地位没有区别。

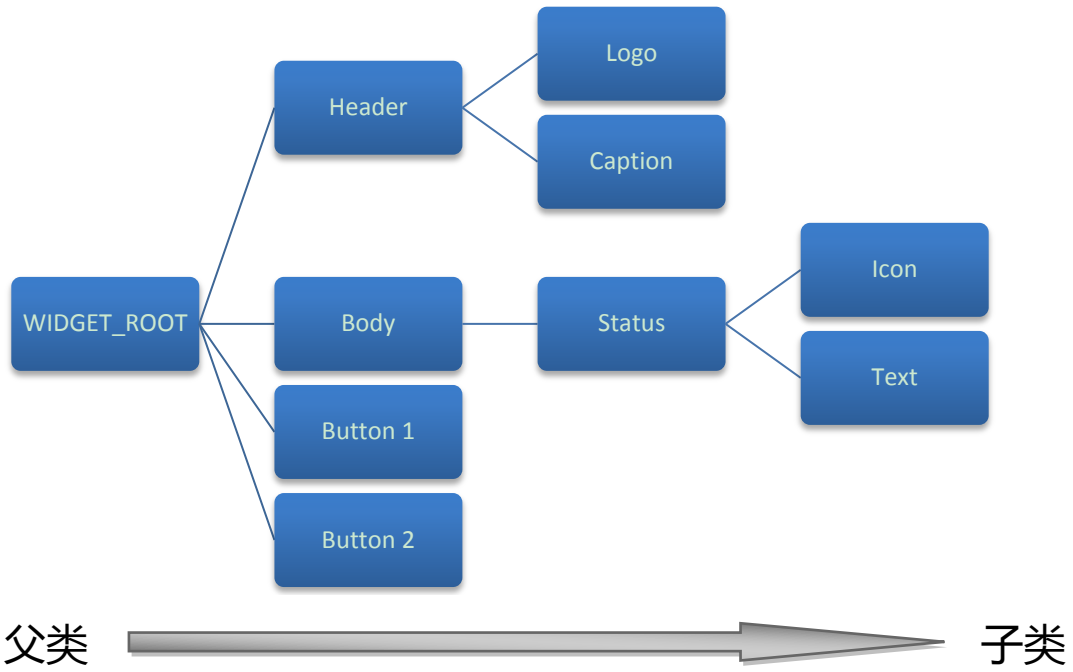
为了描述一个树状结构，每个控件都有三个属性，分别是：

父控件 (Parent)

下一个控件 (Next)

子控件 (Child)

这些属性各自指向了自己周围的节点。下图就是一个典型的控件树：



依照上图，各控件的三个属性可以按如下方式描述，当旁边的节点不存在时，用 0 表示：

	Parent	Child	Next
Header	WIDGET_ROOT	Logo	Body
- Logo	Header	Caption	0
- Caption	Header	0	0
Body	WIDGET_ROOT	Status	Button1
- Status	Body	Icon	0
- Icon	Status	0	Text
- Text	Status	0	0
Button1	WIDGET_ROOT	Button2	0
Button2	WIDGET_ROOT	0	0

3) 显示设备对象（pDisplay）

StellarisWare 图形库支持多屏幕显示，所以每一个控件也需要指定用显示设备对象(pDisplay)属性，用来指示要绘制控件的屏幕。这个属性需要传入在 4.1 节中介绍过的 tDisplay 对象，5.1 节例子中的 g_sKitronix320x240x16_SSD2119 就是由显示驱动提供的 tDisplay 对象。

4) 位置和尺寸

每一个控件都是在屏幕上可见的对象，所以都有位置和尺寸的属性。这个属性一般紧跟在 tContext 属性后面，lX、lY 表示横竖位置，Width 和 Height 分别表示控件宽度和高度。

5) 控件风格(Style)及其它属性

每个控件都有 style 属性，及一些与功能相关的其它属性，用来对其外观及功能等进行配置(如控件是否可见，滑动条控件的最大最小值等)。在配置好位置和尺寸之后，就需要设置这些属性。但由于每个控件在外观及作用上大相径庭，所以不同控件该属性的配置值意义并不一样，每个控件具体的风格属性可以在第 6 章中找到。

经常用到的属性也包括颜色、字体及图片，这些与第 4 章介绍的一致。

6) 事件响应

控件的作用就是用于响应用户的输入事件，不同的只是不同控件输入的事件不同。如按钮的输入事件是按钮按下，列表框的输入事件则是选择的项目变化。

除了容器控件(Container)以外，每种控件都有一种事件响应，作为属性的最后一项。如 5.1 节例子中的 OnButtonPress 属性，设置了由该函数处理按钮的点击事件。当不需要响应这些事件的时候，则设置其为 0。

5.2.3. 控件添加绘制及管理

5.2.2 节已经介绍了如何组织树形的控件列表，StellarisWare 图形库对控件的管理便是以这些树形结构为依据。

通过对树上节点的增删控制画面显示的内容。使用 WidgetAdd 和 WidgetRemove 来增删控件树。要注意的是这里增删的控件同时也包括了它们的子控件。用法如 5.1 节例子所示。

使用 WidgetPaint 函数可以绘制控件树。WidgetPaint 绘制的是某个节点及其全部子节点。所以使用 WidgetPaint(WIDGET_ROOT)可以绘制整个控件树。

完成一些列配置后，调用 WidgetMessageQueueProcess 函数来处理图形库的消息，实现用户响应。这个函数需要不断被调用，因为会有消息不断送达。通常最好如 5.1 节将该函数放入主循环中。如果想要将其放入中断则需仔细考虑，因为在处理消息时，可能会回调用户编写的事件处理函数。这些事件处理函数相当于在中断过程中运行，如果用户的事件处理函数过于复杂，可能会影响低实时性中断的执行。

WidgetMessageQueueProcess 会处理图形库的一系列消息，但在执行过程中，仍然允许新的消息到来，不会遗漏用户的操作。同时，用户也可以用 WidgetMessageQueueAdd 函数手工增加消息。消息的具体用法可以参考 TI 的 StellarisWare 图形库使用手册，这里不做具体说明。

6.各控件功能及属性

6.1.画布控件(Canvas)

画布控件就是一个简单的绘图表面，用户可以在上面进行图形、文字绘制，也支持显示图片。

数据类型:

tCanvasWidget

定义及参数:

```
#define Canvas(sName,           // 控件名
               pParent,         // 父控件
               pNext,           // 下一个控件
               pChild,          // 子控件
               pDisplay,        // 显示设备对象
               lX,              // 控件 X 位置
               lY,              // 控件 Y 位置
               lWidth,          // 控件宽度
               lHeight,         // 控件高度
               ulStyle,         // 控件风格
               ulFillColor,     // 填充颜色
```

```
ulOutlineColor,     // 边框颜色
ulTextColor,        // 文字颜色
pFont,              //文字字体
pcText,             // 显示的文字
pucImage,           // 显示的图片
pfnOnPaint)         // WIDGET_MSG_PAINT 事件处理函数
```

事件处理:

pfnOnPaint 当控件声明为 CANVAS_STYLE_APP_DRAWN 风格时，图形库会将控件的绘制交给该函数。如果不需要自己处理控件绘制，可直接填 0。

支持的控件风格:

```
CANVAS_STYLE_APP_DRAWN      // 使用用户提供的绘制函数
CANVAS_STYLE_FILL           // 填充画布
CANVAS_STYLE_OUTLINE        // 绘制边框
CANVAS_STYLE_IMG            // 显示图片(pucImage)
CANVAS_STYLE_TEXT           // 显示文字(pFont、pcText)
CANVAS_STYLE_TEXT_TOP       // 文字垂直顶部对齐
CANVAS_STYLE_TEXT_VCENTER   // 文字垂直中央对齐
CANVAS_STYLE_TEXT_BOTTOM    // 文字垂直底部对齐
```

CANVAS_STYLE_TEXT_LEFT // 文字水平左对齐

CANVAS_STYLE_TEXT_HCENTER // 文字水平中央对齐

CANVAS_STYLE_TEXT_RIGHT // 文字水平右对齐

CANVAS_STYLE_TEXT_OPAQUE // 绘制文字底色

常用函数(用于更改画布控件属性):

CanvasAppDrawnOff(pWidget)

CanvasAppDrawnOn(pWidget)

CanvasCallbackSet(pWidget, pfnOnPnt)

CanvasFillColorSet(pWidget, ulColor)

CanvasFillOff(pWidget)

CanvasFillOn(pWidget)

CanvasFontSet(pWidget, pFnt)

CanvasImageOff(pWidget)

CanvasImageOn(pWidget)

CanvasImageSet(pWidget, pImg)

CanvasOutlineColorSet(pWidget, ulColor)

CanvasOutlineOff(pWidget)

CanvasOutlineOn(pWidget)

CanvasTextAlignment(pWidget, ulAlign)

CanvasTextColorSet(pWidget, ulColor)

CanvasTextOff(pWidget)

CanvasTextOn(pWidget)

CanvasTextOpaqueOff(pWidget)

CanvasTextOpaqueOn(pWidget)

CanvasTextSet(pWidget, pcTxt)

6.2.选择/多选框控件(Checkbox)

提供一个可以选择或取消选择的方框。当选中时，框中会标注“X”；取消选择时为空。多个组合使用时，可以实现多选的功能。

数据类型:

tCheckBoxWidget

定义及参数:

```
#define CheckBox(sName,          // 控件名
                 pParent,        // 父控件
                 pNext,          // 下一个控件
                 pChild,         // 子控件
                 pDisplay,       // 显示设备对象
                 lX,             // 控件 X 位置
                 lY,             // 控件 Y 位置
                 lWidth,         // 控件宽度
                 lHeight,        // 控件高度
                 usStyle,        // 控件风格
                 usBoxSize,      // 方框大小(unsigned short)
                 ulFillColor,    // 填充颜色
                 ulOutlineColor, // 边框颜色
```

```
    ulTextColor,        // 文字颜色
    pFont,              // 文字字体
    pcText,             // 显示的文字
    pucImage,           // 显示的图片
    pfnOnChange)        // 选择结果改变事件的处理函数
```

事件处理:

pfnOnChange　当控件的选择结果变化时，该函数会被调用。用户可以自己编写事件处理以响应变化。如果不需要动作，可以填 0。

支持的控件风格:

```
CB_STYLE_OUTLINE        // 显示边框
CB_STYLE_FILL           // 填充控件
CB_STYLE_TEXT           // 显示文字(pFont、pcText)
CB_STYLE_IMG            // 显示图片(pucImage)
CB_STYLE_TEXT_OPAQUE    // 绘制文字底色
CB_STYLE_SELECTED       // 控件已选中
```

常用函数(用于更改控件属性):

```
CheckBoxBoxSizeSet(pWidget, usSize)
CheckBoxCallbackSet(pWidget, pfnOnChg)
```

CheckBoxFillColorSet(pWidget, ulColor)

CheckBoxFillOff(pWidget)

CheckBoxFillOn(pWidget)

CheckBoxFontSet(pWidget, pFnt)

CheckBoxImageOff(pWidget)

CheckBoxImageOn(pWidget)

CheckBoxImageSet(pWidget, pImg)

CheckBoxOutlineColorSet(pWidget, ulColor)

CheckBoxOutlineOff(pWidget)

CheckBoxOutlineOn(pWidget)

CheckBoxTextColorSet(pWidget, ulColor)

CheckBoxTextOff(pWidget)

CheckBoxTextOn(pWidget)

CheckBoxTextOpaqueOff(pWidget)

CheckBoxTextOpaqueOn(pWidget)

CheckBoxTextSet(pWidget, pcTxt)

6.3.容器控件(Container)

容器控件通常用来将多个控件分组。最常见的用法是将多个单选按钮控件(Radio Button)组合在一起，实现单选的功能。在控件的顶部可以有一行标题。容器控件没有事件处理函数。

数据类型:

tContainerWidget

定义及参数:

```
#define Container(sName,        // 控件名
                  pParent,      // 父控件
                  pNext,        // 下一个控件
                  pChild,       // 子控件
                  pDisplay,     // 显示设备对象
                  lX,           // 控件 X 位置
                  lY,           // 控件 Y 位置
                  lWidth,       // 控件宽度
                  lHeight,      // 控件高度
                  ulStyle,      // 控件风格
                  ulFillColor,  // 填充颜色
                  ulOutlineColor, // 边框颜色
                  ulTextColor,  // 文字颜色
```

pFont, // 文字字体

pcText) // 显示的文字

支持的控件风格:

CTR _STYLE_OUTLINE // 显示边框

CTR _STYLE_FILL // 填充控件

CTR _STYLE_TEXT // 显示文字(pFont、pcText)

CTR _STYLE_TEXT_OPAQUE // 绘制文字底色

CTR_STYLE_TEXT_CENTER // 标题文字水平居中对齐

常用函数(用于更改控件属性):

ContainerFillColorSet(pWidget, ulColor)

ContainerFillOff(pWidget)

ContainerFillOn(pWidget)

ContainerFontSet(pWidget, pFnt)

ContainerOutlineColorSet(pWidget, ulColor)

ContainerOutlineOff(pWidget)

ContainerOutlineOn(pWidget)

ContainerTextCenterOff(pWidget)

ContainerTextCenterOn(pWidget)

ContainerTextColorSet(pWidget, ulColor)

ContainerTextOff(pWidget)

ContainerTextOn(pWidget)

ContainerTextOpaqueOff(pWidget)

ContainerTextOpaqueOn(pWidget)

ContainerTextSet(pWidget, pcTxt)

6.4.图形按钮控件(Image Button)

图形按钮提供了一个由用户自己控制外观的按钮控件。每个控件最多可以使用三幅图片实现效果：未按下时的背景图片、按下时的背景图片、按键响应图片(KeyCap Image)。未按下与按下时，背景图片会跟随状态改变；按键响应图片若未被隐藏，未按下时正常显示，按下后会发生位置偏移(偏移距离可调)。之所以这么设计，是因为有时会大量用到背景一样的按钮，复用背景按键可以节约内存和程序空间。

数据类型:

tImageButtonWidget

定义及参数:

```
#define ImageButton(sName,          // 控件名
                    pParent,        // 父控件
                    pNext,          // 下一个控件
                    pChild,         // 子控件
                    pDisplay,       // 显示设备对象
                    lX,             // 控件 X 位置
                    lY,             // 控件 Y 位置
                    lWidth,         // 控件宽度
                    lHeight,        // 控件高度
                    ulStyle,        // 控件风格
                    ulForeColor,    // 前景颜色(仅 1bpp 格式图片有效)
```

```
    ulPressColor,        // 按下颜色(仅 1bpp 格式图片有效)
    ulBackColor,         // 背景颜色(仅 1bpp 格式图片有效)
    pFont,               // 文字字体
    pcText,              // 显示的文字
    pucImage,            // 未按下状态背景图
    pucPressImage,       // 按下状态背景图
    pucKeycapImage,      // 按键响应图
    sXOff,               // 响应图按下时 X 偏移量
    sYOff,               // 响应图按下时 Y 偏移量
    usAutoRepeatDelay,   // 持续按下自动重复启动前延迟
    usAutoRepeatRate,    // 持续按下自动重复频率
    pfnOnClick)          // 点击事件处理函数
```

事件处理:

pfnOnClick 当按钮被点击时，该函数会被调用。若不需要使用该回调函数，可以填 0。

支持的控件风格:

```
IB_STYLE_TEXT          // 显示文字(pFont、pcText)
IB_STYLE_FILL          // 填充背景色
IB_STYLE_KEYCAP_OFF    // 不显示按键响应图片(KeyCap Image)
```

IB_STYLE_IMAGE_OFF // 不显示背景图片

IB_STYLE_AUTO_REPEAT // 持续按下时，使用自动重复功能

IB_STYLE_RELEASE_NOTIFY // 在按钮松开时才调用 pfnOnClick。不使用此选项时，按下就会调用 pfnOnClick

常用函数(用于更改控件属性):

ImageButtonAutoRepeatDelaySet(pWidget, usDelay)

ImageButtonAutoRepeatOff(pWidget)

ImageButtonAutoRepeatOn(pWidget)

ImageButtonAutoRepeatRateSet(pWidget, usRate)

ImageButtonBackgroundColorSet(pWidget, ulColor)

ImageButtonCallbackSet(pWidget, pfnOnClik)

ImageButtonFillColorSet(pWidget, ulColor)

ImageButtonFillOff(pWidget)

ImageButtonFillOn(pWidget)

ImageButtonForegroundColorSet(pWidget, ulColor)

ImageButtonImageKeycapSet(pWidget, pImg)

ImageButtonImageOff(pWidget)

ImageButtonImageOn(pWidget)

ImageButtonImagePressedSet(pWidget, pImg)

ImageButtonImageSet(pWidget, pImg)

ImageButtonKeycapOff(pWidget)

ImageButtonKeycapOffsetSet(pWidget, sX, sY)

ImageButtonKeycapOn(pWidget)

ImageButtonPressedColorSet(pWidget, ulColor)

ImageButtonTextOff(pWidget)

ImageButtonTextOn(pWidget)

ImageButtonTextSet(pWidget, pcTxt)

6.5.列表框控件(ListBox)

列表框可以提供一列文字供用户选择其中一个。通过使用 ListBoxTextAdd 可以向列表框增加新的内容。列表框控件也支持一种 WRAP 模式，可以让新加入的内容永远在最上端，当总的内容数超过最大容量时，最旧的内容会被删除以加入新的。

数据类型:

tListBoxWidget

定义及参数:

```
#define ListBox(sName,          // 控件名
                pParent,        // 父控件
                pNext,          // 下一个控件
                pChild,         // 子控件
                pDisplay,       // 显示设备对象
                lX,             // 控件 X 位置
                lY,             // 控件 Y 位置
                lWidth,         // 控件宽度
                lHeight,        // 控件高度
                ulStyle,        // 控件风格
                ulBgColor,      // 内容容器底色
                ulSelBgColor,   // 被选内容底色
```

```
    ulTextColor,        // 文字颜色
    ulSelTextColor,     // 被选文字颜色
    ulOutlineColor,     // 控件边框颜色
    pFont,              // 文字字体
    pcText,             // 显示的文字
    usMaxEntries,       // 最多内容条数
    usPopulatedEntries,// 当前实际内容条数
    pfnOnChange)        // 选择内容改变事件的处理函数
```

事件处理:

pfnOnChange 当列表框选择内容改变时，该函数会被调用。若不需要使用该回调函数，可以填 0。

支持的控件风格:

```
LISTBOX_STYLE_OUTLINE     // 显示控件边框
LISTBOX_STYLE_LOCKED      // 忽略用户输入，只显示内容
LISTBOX_STYLE_WRAP        // 使用 WRAP 模式
```

常用函数(用于更改控件属性):

```
ListBoxBackgroundColorSet(pWidget, ulColor)
ListBoxCallbackSet(pWidget, pfnCallback)
ListBoxClear(pWidget)
```

ListBoxFontSet(pWidget, pFnt)

ListBoxLock(pWidget)

ListBoxOutlineColorSet(pWidget, ulColor)

ListBoxOutlineOff(pWidget)

ListBoxOutlineOn(pWidget)

ListBoxSelectedBackgroundColorSet(pWidget, ulColor)

ListBoxSelectedTextColorSet(pWidget, ulColor)

ListBoxSelectionGet(pWidget)

ListBoxSelectionSet(pWidget, sSel)

ListBoxTextColorSet(pWidget, ulColor)

ListBoxTextSet(pWidget, pcTxt, ulIndex)

ListBoxUnlock(pWidget)

ListBoxWrapDisable(pWidget)

ListBoxWrapEnable(pWidget)

6.6. 按钮控件(Push Button)

按钮控件提供了和图形按钮控件类似的功能。只是主要通过颜色及文字表现文字效果。使用起来比图形按钮更简单。按钮控件主要有圆形和方形两种。

数据类型:

tPushButtonWidget

圆形按钮定义及参数:

```
#define CircularButton (sName,                  // 控件名
                        pParent,                // 父控件
                        pNext,                  // 下一个控件
                        pChild,                 // 子控件
                        pDisplay,               // 显示设备对象
                        lX,                     // 控件 X 位置
                        lY,                     // 控件 Y 位置
                        lR,                     // 按钮半径
                        ulStyle,                // 按钮风格
                        ulFillColor,            // 按钮填充色
                        ulPressFillColor,       // 按钮按下时填充色
                        ulOutlineColor,         // 边框颜色
                        ulTextColor,            // 文字颜色
```

```
        pFont,              // 字体
        pcText,             // 按钮文字
        pucImage,           // 按钮图片
        pucPressImage,      // 按钮按下图片
        usAutoRepeatDelay,  // 持续按下自动重复启动前延迟
        usAutoRepeatRate,   // 持续按下自动重复频率
        pfnOnClick)         // 点击按钮事件处理函数
```

方形按钮定义及参数:

```
#define RectangularButton(sName,        // 控件名
            pParent,        // 父控件
            pNext,          // 下一个控件
            pChild,         // 子控件
            pDisplay,       // 显示设备对象
            lX,             // 按钮 X 位置
            lY,             // 按钮 Y 位置
            lWidth,         // 按钮宽度
            lHeight,        // 按钮高度
            ulStyle,        // 按钮风格
```

```
    ulFillColor,            // 按钮填充色
    ulPressFillColor,       // 按钮按下时填充色
    ulOutlineColor,         // 边框颜色
    ulTextColor,            // 文字颜色
    pFont,                  // 字体
    pcText,                 // 按钮文字
    pucImage,               // 按钮图片
    pucPressImage,          // 按钮按下图片
    usAutoRepeatDelay,// 持续按下自动重复启动前延迟
    usAutoRepeatRate,       // 持续按下自动重复频率
    pfnOnClick)             // 点击按钮事件处理函数
```

事件处理:

pfnOnClick 当按钮被点击时，该函数会被调用。若不需要使用该回调函数，可以填 0。

支持的控件风格:

```
PB_STYLE_AUTO_REPEAT    // 持续按下时，使用自动重复功能
PB_STYLE_FILL           // 填充背景色
PB_STYLE_IMG            // 显示图片
PB_STYLE_OUTLINE        // 显示边框
```

PB_STYLE_PRESSED // 按钮已按下指示

PB_STYLE_RELEASE_NOTIFY // 在按钮松开时才调用 pfnOnClick。不使用此选项时，按下就会调用 pfnOnClick

PB_STYLE_TEXT //显示文字(pFont、pcText)

PB_STYLE_TEXT_OPAQUE // 绘制文字底色

常用函数(用于更改控件属性):

PushButtonAutoRepeatDelaySet(pWidget, usDelay)

PushButtonAutoRepeatOff(pWidget)

PushButtonAutoRepeatOn(pWidget)

PushButtonAutoRepeatRateSet(pWidget, usRate)

PushButtonCallbackSet(pWidget, pfnOnClik)

PushButtonFillColorPressedSet(pWidget, ulColor)

PushButtonFillColorSet(pWidget, ulColor)

PushButtonFillOff(pWidget)

PushButtonFillOn(pWidget)

PushButtonFontSet(pWidget, pFnt)

PushButtonImageOff(pWidget)

PushButtonImageOn(pWidget)

PushButtonImagePressedSet(pWidget, pImg)

PushButtonImageSet(pWidget, pImg)

PushButtonOutlineColorSet(pWidget, ulColor)

PushButtonOutlineOff(pWidget)

PushButtonOutlineOn(pWidget)

PushButtonTextColorSet(pWidget, ulColor)

PushButtonTextOff(pWidget)

PushButtonTextOn(pWidget)

PushButtonTextOpaqueOff(pWidget)

PushButtonTextOpaqueOn(pWidget)

PushButtonTextSet(pWidget, pcTxt)

6.7.单选按钮控件(Radio Button)

单选按钮控件是一个圆形的框，当选中时，框中会出现圆点；未选中时，圆点消失。单选按钮可以组合起来实现多选一的功能，比方说设定速度的高、中、低等。

组合单选按钮的方法，是将多个单选按钮作为同一节点的子节点。容器控件(Container)是作为父控件(Parent)的不错选择。同节点下的单选按钮会自动成组，实现多选一的功能。

数据类型:

tRadioButtonWidget

定义及参数:

```
#define RadioButton(sName,        // 控件名
                    pParent,      // 父控件
                    pNext,        // 下一个控件
                    pChild,       // 子控件
                    pDisplay,     // 显示设备对象
                    lX,           // 控件 X 位置
                    lY,           // 控件 Y 位置
                    lWidth,       // 控件宽度
                    lHeight,      // 控件高度
                    ulStyle,      // 控件风格
                    usCircleSize, // 单选圆形框尺寸
```

```
ulFillColor,        // 填充色
ulOutlineColor,     // 边框颜色
ulTextColor,        // 文字颜色
pFont,              // 文字字体
pcText,             // 显示文字
pucImage,           // 显示图片
pfnOnChange)        // 选择改变事件处理函数
```

事件处理:

pfnOnChange　当选择的内容改变时，该函数会被调用。若不需要使用该回调函数，可填 0。

支持的控件风格:

```
RB_STYLE_FILL           // 填充控件
RB_STYLE_IMG            // 绘制图片
RB_STYLE_OUTLINE        // 显示控件边框
RB_STYLE_SELECTED       // 控件被选中指示
RB_STYLE_TEXT           // 显示文字
RB_STYLE_TEXT_OPAQUE    // 填充文字底色
```

常用函数(用于更改控件属性):

RadioButtonCallbackSet(pWidget, pfnOnChg)

RadioButtonCircleSizeSet(pWidget, usSize)

RadioButtonFillColorSet(pWidget, ulColor)

RadioButtonFillOff(pWidget)

RadioButtonFillOn(pWidget)

RadioButtonFontSet(pWidget, pFnt)

RadioButtonImageOff(pWidget)

RadioButtonImageOn(pWidget)

RadioButtonImageSet(pWidget, pImg)

RadioButtonOutlineColorSet(pWidget, ulColor)

RadioButtonOutlineOff(pWidget)

RadioButtonOutlineOn(pWidget)

RadioButtonTextColorSet(pWidget, ulColor)

RadioButtonTextOff(pWidget)

RadioButtonTextOn(pWidget)

RadioButtonTextOpaqueOff(pWidget)

RadioButtonTextOpaqueOn(pWidget)

RadioButtonTextSet(pWidget, pcTxt)

6.8.拖滑/进度条控件 (Slider)

拖滑/进度条控件为条状，可以垂直也可以水平放置。控件可以用来显示，或由用户选择指定范围内的一个值。也就是说，用户设置范围及当前值后，控件可以自动用图形方式描绘百分比。

数据类型:

tSliderWidget

定义及参数:

```
#define Slider(sName,                   // 控件名
               pParent,                 // 父控件
               pNext,                   // 下一个控件
               pChild,                  // 子控件
               pDisplay,                // 显示设备对象
               lX,                      // 控件 X 位置
               lY,                      // 控件 Y 位置
               lWidth,                  // 控件宽度
               lHeight,                 // 控件高度
               lMin,                    // 数据范围 – 最小值
               lMax,                    // 数据范围 – 最大值
               lValue,                  // 数据初始值
               ulStyle,                 // 控件风格
```

```
    ulFillColor,                // 填充颜色
    ulBackgroundFillColor,      // 填充底色(未达到值部分颜色)
    ulOutlineColor,             // 边框颜色
    ulTextColor,                // 文字颜色
    ulBackgroundTextColor,      // 未达到值部分文字颜色
    pFont,                      // 文字字体
    pcText,                     // 显示文字
    pucImage,                   // 显示图片
    pucBackgroundImage,             // 未达到值部分图片
    pfnOnChange)                    // 值变化事件处理函数
```

事件处理:

pfnOnChange　当滑动条值改变时，该函数会被调用。若不需要使用该回调函数，可以填 0。

支持的控件风格:

```
SL_STYLE_BACKG_FILL             // 填充背景色
SL_STYLE_BACKG_IMG              // 显示背景图片
SL_STYLE_BACKG_TEXT             // 显示文字
SL_STYLE_BACKG_TEXT_OPAQUE      // 显示文字底色
SL_STYLE_FILL                   // 填充控件
```

SL_STYLE_IMG // 显示图片

SL_STYLE_LOCKED // 锁定控件，用户不可更改其值

SL_STYLE_OUTLINE // 绘制边框

SL_STYLE_TEXT // 显示文字

SL_STYLE_TEXT_OPAQUE // 填充文字底色

SL_STYLE_VERTICAL // 垂直放置控件

常用函数(用于更改控件属性):

SliderBackgroundFillOff(pWidget)

SliderBackgroundFillOn(pWidget)

SliderBackgroundImageOff(pWidget)

SliderBackgroundImageOn(pWidget)

SliderBackgroundImageSet(pWidget, pImg)

SliderBackgroundTextColorSet(pWidget, ulColor)

SliderBackgroundTextOff(pWidget)

SliderBackgroundTextOn(pWidget)

SliderBackgroundTextOpaqueOff(pWidget)

SliderBackgroundTextOpaqueOn(pWidget)

SliderCallbackSet(pWidget, pfnCallback)

SliderFillColorBackgroundedSet(pWidget, ulColor)

SliderFillColorSet(pWidget, ulColor)

SliderFillOff(pWidget)

SliderFillOn(pWidget)

SliderFontSet(pWidget, pFnt)

SliderImageOff(pWidget)

SliderImageOn(pWidget)

SliderImageSet(pWidget, pImg)

SliderLock(pWidget)

SliderOutlineColorSet(pWidget, ulColor)

SliderOutlineOff(pWidget)

SliderOutlineOn(pWidget)

SliderRangeSet(pWidget, lMinimum, lMaximum)

SliderTextColorSet(pWidget, ulColor)

SliderTextOff(pWidget)

SliderTextOn(pWidget)

SliderTextOpaqueOff(pWidget)

SliderTextOpaqueOn(pWidget)

SliderTextSet(pWidget, pcTxt)

SliderUnlock(pWidget)

SliderValueSet(pWidget, lVal)

SliderVerticalSet(pWidget, bVertical)